ExOShim: Preventing Memory Disclosure using Execute-Only Kernel Code

Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor
Thayer School of Engineering at Dartmouth College TR15-001
{scott.l.brookes.th, robert.m.denz.th, martin.osterloh, stephen.taylor}@dartmouth.edu

ABSTRACT

Information leakage and memory disclosure are major threats to the security in modern computer systems. If an attacker is able to obtain the binary-code of an application, it is possible to reverse-engineer the source-code, uncover vulnerabilities, craft exploits, and patch together code-segments to produce code-reuse attacks. These issues are particularly concerning when the application is an operating system because they open the door to privilege-escalation and exploitation techniques that provide kernel-level access. This paper describes ExOShim: a 325-line, lightweight "shim" layer, using Intel's commodity virtualization features, that prevents memory disclosures by rendering all kernel code *execute-only*. This technology, when combined with non-deterministic refresh and load-time diversity, prevents disclosure of kernel code on time-scales that facilitate kernel-level exploit development. The proof-of-concept prototype described here has been demonstrated on a 64-bit microkernel. It is evaluated using metrics that quantify its code size and complexity, associated run-time performance costs, and its effectiveness in thwarting information leakage. ExOShim provides complete execute-only protection for kernel code at a runtime-performance overhead of only 0.86%. The concepts are general and can also be applied to render application code execute-only.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection – *Access Control.*

General Terms

Security

Keywords

protection mechanisms, execute-only memory, memory disclosure exploits, information leaks, code reuse attacks, virtualization, access control, MULTIC protections

1. INTRODUCTION

Memory disclosures in modern computer systems provide attackers a valuable method of discovering vulnerabilities. These same disclosures enable an attacker to reverse-engineer the source code and eventually craft zero-day exploits. Often, these exploits are realized as code-reuse attacks such as Return-Oriented Programming (ROP). Information leakage from the kernel is of particular concern because a compromised kernel gives an adversary access to all system resources. To mitigate this threat, ExOShim enables *hardware-enforced* execute-only memory access control primitives on all kernel code pages. This eliminates the capability of an attacker to dynamically disassemble kernel code stored in memory. The specific threat model that ExOShim aims to disrupt is described below.

1.1 Threat Model

ExOShim is intended to operate as part of a defense in depth strategy to increase attacker workload and mitigate Advanced Persistent Threats (APTs). The associated threat model is illustrated in Figure 1. An attacker may take several coordinated steps including surveillance to determine if a vulnerability exists [32], use of an appropriate exploit or other access method [32], privilege escalation [12], removing exploit artifacts, and hiding behavior [23]. Surveillance may involve obtaining a copy of the binary code and using reverse engineering [14,15] or fuzzing [20] to facilitate a broad range of attack vectors including but not limited to return oriented programming [9]. The implant then persists for a time sufficient enough to carry out some malicious effect, obtain useful information, or propagate intrusion to other systems.

Unlike the time to execute an exploit, the time spent in surveillance and persistence may range from minutes to months or even years depending upon the intended effect. Moreover, the presence of an intrusion may never be detected by network defenses but instead may be recognized only indirectly due to either a deviation from expected behavior, or may be derived from other sources.

ExOShim increases attacker workload in this model in three specific ways. First, it makes privilege escalation more challenging by inhibiting the attacker's efforts to locate a vulnerability in the kernel. Subsequently, it inhibits patching the kernel with an implant. Finally, even if the kernel is compromised, ExOShim hampers the ability to extract the kernel binary for reverse-engineering.

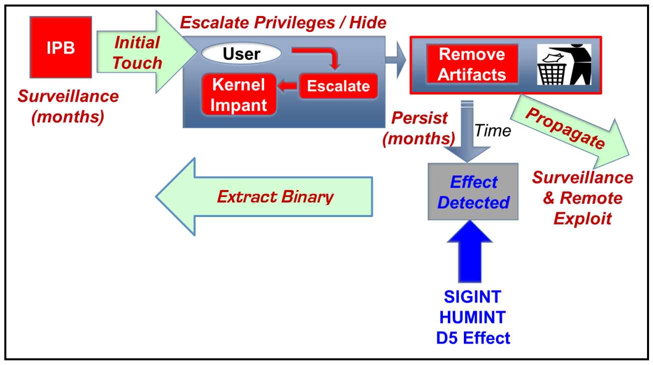

Figure 1. ExOShim Threat Model

2. BACKGROUND

A variety of techniques have been proposed to mitigate memory disclosure using code randomization. A minimal level of entropy is required to protect against brute-force attacks [43]. Address Space Layout Randomization (ASLR) [46] was an early version of this concept intended to defeat return-to-libc [42] and ROP [7,8,40]. Unfortunately, ASLR only requires the attacker to discover base addresses to craft a re-usable exploit. A variety of alternative methods have subsequently been introduced. These include randomization at development time [2,21], compile time [24,26,27,28,29,30,31], load time [5,28,30,48] and run time [13,28]. In general, these techniques serve to ensure that no two running instances of the same binary code contain the same exploitable address, which throttles vulnerability amplification and continually invalidates surveillance [30,38].

Just-In-Time ROP (JIT-ROP) improves upon ROP by dynamically locating kernel memory code gadgets [6]. In the presence of a memory disclosure, JIT-ROP bypasses the protections offered by many software diversification techniques [11]. Direct memory disclosure also allows an attacker to begin the process of dynamically reverse engineering the operating system binary code [7,36,37].

Direct memory disclosures can be eliminated if kernel code is marked *execute-only* as in MULTICS. Intel's 32-bit virtual memory mechanism provided execute-only access control primitives with segmentation [47]. Unfortunately, although more flexible in many ways, the current Intel virtual memory model, based on paging, does not support execute-only memory protection [25]. While side channel attacks have been developed to exploit this fact [17], these too can be mitigated through execute-only protection of code.

2.1 The Kernel and Virtual Memory

Although modules might be added or removed at runtime and virtual addresses to kernel functionality may change, kernel code is generally loaded once and only once. As a result, after initial bootstrapping, its location in physical memory does not change. The kernel therefore represents high-value target that, when coupled with its persistent nature, makes it an important resource to protect.

In general, the kernel provides applications with an interface to the hardware; every application needs to access the functionality of the kernel in order to run on the processor. In most modern operating systems the kernel is mapped directly into the address space of every process. The virtual memory abstraction allows for each of these "instances" of the kernel to resolve to a single copy of the kernel binary, stored in physical memory. On modern Intel 64-bit processors, the virtual memory abstraction is achieved by traversing a 4-level deep page table [25] as shown in Figure 2.

The entries in this paging structure contain bits that describe the permissions required to access all memory associated with the entry. These bits include a no-execute (NX) bit, a supervisor bit (U/S) and a write bit (R/W). Unfortunately, read permissions are implied as long as the privilege level requirement is satisfied. This means that although the virtual memory abstraction does provide some access control primitives it is not possible to directly mark memory as execute-only in the page tables.

2.2 Virtualization and the EPT

Intel's virtualization extensions (VT-x), provided in modern commodity processors, allow a hypervisor to abstractly represent the hardware environment. This is most commonly used to allow multiple operating systems to operate concurrently on a single processor.

Extended Page Tables (EPTs) are part of the hardware abstraction provided by virtualization. These tables provide a similar abstraction to the virtual memory abstraction discussed previously, with one major difference: while the kernel's page tables translate from virtual to physical addresses, the hypervisor's EPT translates from *guest physical* to real physical addresses. When virtualized, the kernel's page tables resolve to a physical address that is valid in the virtualized image of memory. The EPT is a hardware-enforced link between the virtualized interface to system memory and the actual system memory.

There are a few other differences between the EPT and the normal page table that are relevant. EPT entries provide unique read (R), write (W) and execute (X) bits that are enforced by hardware [25]. This makes it possible to mark pages directly as execute-only. Additionally, the EPT entries can only be managed by a hypervisor, not by a kernel. If an attacker compromises the kernel, the normal virtual memory abstraction cannot provide any security; however, the attacker would have to compromise the hypervisor in order to manipulate the EPT.

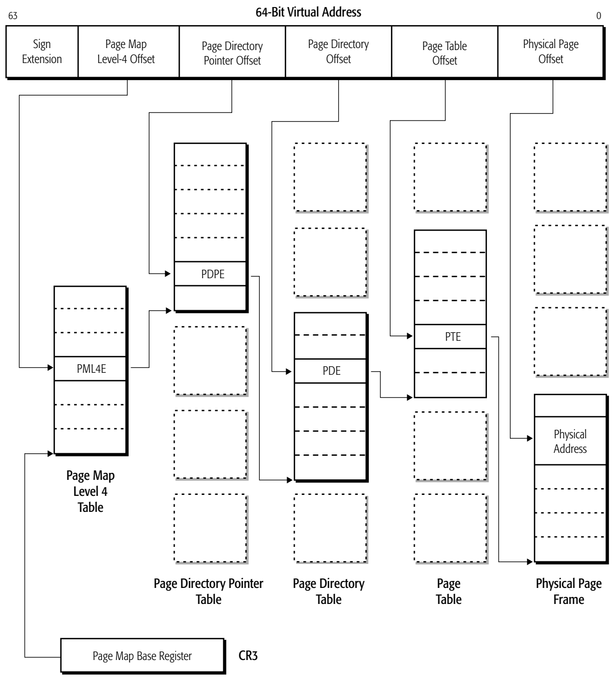

Figure 2. x86 Virtual Memory Translation [1]

3. OVERVIEW

ExOShim is a lightweight hypervisor that reclaims the MULTICS-style hardware-enforced execute-only memory protection using modern Intel virtualization technologies. It was designed as a "shim" that can be inserted beneath a running kernel [16]. A conscious effort was made to minimize its size to open the door for formal reasoning [33] and reduce the likelihood of introducing vulnerabilities [39]. The on demand insertion process allows ExOShim to enable execute-only memory for a running system using Extended Page Tables (EPTs).

3.1 Assumptions

ExOShim assumes that a trusted boot process is undertaken by the operating system it will be deployed on [35]. The security implications associated with the race to call VMXON [41] are well known and must be addressed within the trusted boot process. ExOShim currently assumes that it is the only hypervisor on the system [18] and makes no attempt to verify the absence of another hypervisor. The requirement that data sections in the kernel ELF binary are readable mandates that attention be paid during the compilation, linking, and loading of the kernel to distinguish between code and data. Upon the completion of the loading process, it is assumed that no code section will share any page in memory with a data section.

3.2 ExOShim

The design of ExOShim is illustated in Figure 3. Recall that the "shim" exists *only* to provide full access control to physical memory. In particular it exists to enforce execute-only protection on all kernel code pages.

ExOShim builds an *identity map* associated with all of physical memory using the EPT structure. In other words, the output from the translation process is identical to the input for all possible inputs. However, for kernel code pages only, both the R and W bits are cleared and only the X bit is asserted. This will cause a fault on any read or write access to the physical memory on which the kernel code resides. Most other mappings are marked with the most liberal permissions, allowing the kernel to enforce further protections using the normal paging mechanism. Pages used by ExOShim itself are marked in a unique fashion that is discussed in Section 4.2.

In order for ExOShim to build the correct EPT structure, it must have access to the addresses where kernel code is loaded. This would be a challenge if ExOShim were booted onto the hardware first, initialized itself, and then allowed the kernel to bootstrap itself in the virtualized environment it creates. Although this is the conventional way to think about virtualization, this method requires that ExOShim initialize its EPT without knowing where the kernel will eventually load code. Unfortunately, detecting which pages hold kernel code from a hypervisor is a non-trivial forensics task. Instead ExOshim uses a method developed by Embleton et al [16] that allows a hypervisor to install itself under a running kernel.

During bootstrapping, the kernel creates a single user-level ExOShim process. When scheduled, this process builds an appropriate EPT and queries the kernel to locate where its code has been loaded. After this initialization, it turns on the virtualization features of the Intel processor and launches a *virtual machine* whose state matches the state of the kernel prior to scheduling the ExOShim process. The result is that the kernel continues execution as a virtual machine but with the protections enforced by the underlying EPT.

If compromised, ExOShim would provide a dangerous security hole. For this reason, ExOShim is implemented in a manner that ensures it cannot be reconfigured or turned off. Intel's virtualization extensions allow a hypervisor to respond to events in the guest. However, ExOShim explicitly prevents any event eligible to be handled by the kernel from triggering an exit to the hypervisor. Additionally, any event that must trap to the hypervisor (such as a violation of the EPT permissions) results in an immediate assumption of compromise of the kernel. Depending on the goals of the system, this could trigger a reboot, a halt, a logging feature, or any number of other responses. It explicitly does not, however, return control to the kernel. In other

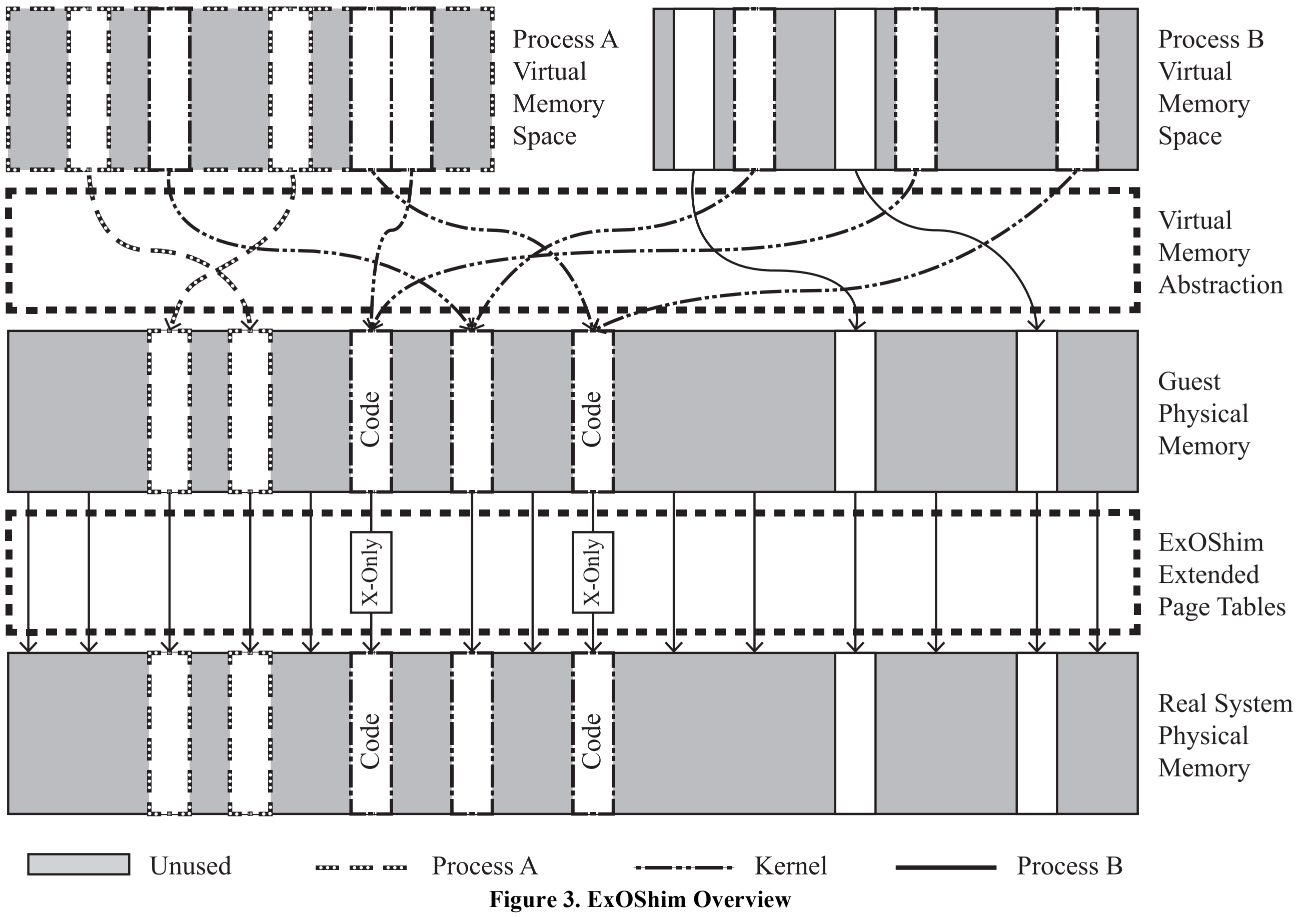
Process A Virtual Memory Space
Process B Virtual Memory Space
Virtual Memory Abstraction
Guest Physical Memory
Code
Code
ExOShim Extended Page Tables
X-Only
X-Only
Real System Physical Memory
Code
Code
Unused
Process A
Kernel
Process B

Figure 3. ExOShim Overview

words, ExOShim does not take any input from the kernel and consequently presents a hardened target without a viable attack vector.

Although it does not accept input of any kind from the kernel after initialization, ExOShim also enforces additional protection measures. When a kernel is booted on top of a hypervisor, it is presented, at initialization time, with a modified image of system memory that does not include the hypervisor code and data. Since ExOShim is loaded only after the kernel has already been initialized, it exists in memory that has already been observed by the kernel. As such, the kernel could alter the memory used by ExOShim after initialization. To mitigate this vulnerability, ExOShim marks all pages containing its own code or data as non-readable, non-writeable, *and* non-executable in the EPT. Thereby ensuring that access to ExOShim code or data is denied. Memory marked as completely untouchable in the EPT includes: the ExOShim initialization and error handling code, the ExOShim's call stack, structures necessary for initializing the virtual machine context, memory on which the EPT is stored, and the paging structures of the ExOShim kernel process. Any attempt to read, write, or execute any memory related to ExOShim after its initialization will be denied.

Obviously, the protections on memory exerted by ExOShim could be used to deny service to the kernel. However, any memory access that would cause ExOShim to be invoked already implies a dangerous security flaw in the operating system: For example, under normal operation, the kernel should not be reading its own code. Another method that ExOShim could be invoked is via an attempt to access the memory being used by ExOShim itself. Since the kernel initializes the ExOShim process, all memory used by ExOShim is marked as "taken" in the kernels bookkeeping. Therefore, there is no scenario in which normal operation of the system would result in ExOShim detecting a violation of its own access control primitives. In other words, any scenario in which ExOShim is invoked is a clear sign of compromise, and service *ought to be* denied to the kernel in the interests of security.

4. IMPLEMENTATION

Since ExOShim is an operating system feature that utilizes virtualization, its implementation will by highly dependent on the kernel and processor on which it is used. However, all implementations must tackle the same core tasks: providing a context in which the ExOShim initialization routine is to execute, building an EPT structure, enabling virtualization, and resuming kernel execution.

4.1 Providing ExOShim a Context

ExOShim does not require special consideration by the kernel beyond that required by any other process. All that is required is that it executes with supervisor privileges and in a unique virtual address space.

In order to use the Intel virtualization instructions without causing faults, the processor must be operating in *ring 0*, privileged mode. This could be accomplished by loading ExOShim as a binary but allowing it special privileges beyond a typical user-space process. Some operating systems provide the notion of a kernel module to accomplish this type of task. Alternatively, ExOShim could be implemented directly in the kernel. As long as the ExOShim functionality is not located on pages shared by other memory (as stated in the assumptions in Section 3.1) and a unique virtual address space is created before jumping to the ExOShim initialization routine, it can be compiled into the kernel. For the proof-of-concept prototype described by this paper, all ExOShim functionality was compiled into the kernel directly.

It is important to understand why it is necessary to provide a unique virtual address space for ExOShim. Some functionality must be provided to handle an event in which a memory access breaks the primitives of ExOShim. This functionality is invoked by hardware in the event of an EPT violation. A set of page tables provide access to this functionality through the virtual memory abstraction. A unique set of page tables for ExOShim allows for the frames of memory where these page tables are stored to be marked as non-readable, non-writeable, and non-executable in the EPT. This prohibits the kernel from manipulating the paging structures that will be used to find the fault-handling functionality if ExOShim is invoked.

4.2 Building the EPT

The Intel manuals clearly show how to format memory to build a four-level EPT. The task is simplified by the fact that ExOShim's EPT is simply an identity map of all available physical memory. The kernel should therefore return the correct amount of physical memory when queried appropriately.

The challenge here for ExOShim is in determining the permissions to apply to each entry in the EPT. The default permissions are read, write, and execute. The most liberal permissions in the EPT allow for the kernel to continue with its own memory management and access control. Enabling caching by default is also important for performance reasons. However, there are three cases where the permissions will differ. These are kernel code frames, ExOShim frames, and volatile (uncacheable) frames.

4.2.1 Kernel Code Frames

As stated in the assumptions shown in Section 3.1, kernel code is loaded on frames of memory that contain *only* kernel code. Each such frame should be marked in the EPT as non-readable, non-writeable, but executable. This is the entire premise of ExOShim.

The kernel must provide a mechanism with which ExOShim can locate all frames of memory that contain kernel code. In the case of the prototype described here, kernel code frames are identified by a unique configuration of a process' page table; kernel code frames are marked as present, global (for caching purposes), executable, non-writeable, and supervisor-mode only. No frame on the system can be marked with these permissions and *not* be a kernel code frame, and every kernel code frame is marked with these permissions. With access to a paging structure for any process into which the kernel is fully mapped, ExOShim can thereby infer which frames contain kernel code. Subsequently, ExOShim can mark all corresponding frames in the EPT as execute-only.

4.2.2 ExOShim Frames

ExOShim is designed so that an attacker cannot disable, modify, or hijack its protection mechanism after initialization – no matter how deeply the rest of the system is compromised. In order to achieve this, ExOShim does not take input after initialization time and, more importantly, marks all ExOShim memory as non-readable, non-writeable, and non-executable in the EPT.

Care is needed to thoroughly and systematically mark all memory needed by ExOShim with no permissions in the EPT. All of the frames supporting each of the following parts of ExOShim need to be protected in order to prevent an attacker from tampering with it.

- *ExOShim Code* – Preventing the kernel from accessing ExOShim code is possible only when all its code is loaded onto frames with no other data. This is not strictly necessary to prevent an attacker from compromising ExOShim: Having access to the code might allow an attacker to attempt to install a second virtualized monitor. However, since ExOShim has already been instantiated, this operation would fail and ExOShim would consequently enter its fault handling routine.
- *ExOShim Call Stack* – When a fault occurs and ExOShim is invoked, it needs a stack to operate. The address of this stack is loaded by the ExOShim initialization routine. If the stack were not protected, an attacker could store a ROP style payload on the stack, cause an ExOShim fault, and run arbitrary code without the protection offered by ExOShim. By disabling kernel access to the ExOShim stack using the EPT, this attack vector is closed to the attacker.
- *ExOShim Data* – There is certain data that ExOShim needs to maintain during its initialization routine. This includes the list of kernel, ExOShim, and uncacheable frames that need special handling and virtualization-specific structures as discussed in Section 4.3. If an adversary could tamper with this data, it would be possible to compromise the state of ExOShim. To prevent this, the data is marked as inaccessible in the EPT. However, this cannot be achieved if the data is stored in the kernel's heap. In order to guarantee that no ExOShim data is stored in the heap, the prototype unmaps the heap from the ExOShim context on entry to its initialization routine. As a result, when ExOShim needs memory, it must use the underlying kernel memory management layer to request new frames. These are then marked appropriately in the EPT.
- *The EPT* – Perhaps the most important memory that the kernel must be prevented from accessing is the EPT tables themselves. These are simply frames of memory that the hardware traverses in order to uncover the permissions associated with a particular frame. If an attacker could modify these, it would be possible to disable or hijack ExOShim. Therefore, all frames used for the construction of the EPT structures must be marked in the EPT as untouchable. Any attempt to read, write, or execute any of the memory in the EPT structure will cause ExOShim to interrupt the kernel.
- *ExOShim Paging Structures* – Finally, the paging structures that define the ExOShim context itself must be set with null permissions in the EPT. When an EPT violation occurs, the hardware switches into the non-virtualized context of the ExOShim and directs control to the pre-defined fault handler. If the attacker could modify the page tables that define this context, it would be possible to disable or hijack ExOShim. Marking all of the paging structures that define ExOShim ensures that in the event of a fault, the code that is executed is exactly what was intended at initialization time.

4.2.3 Uncacheable Frames

The abundance of internal processor cache memory improves performance of all system tasks. The Memory Management Unit (MMU) manages what memory addresses are contained in and outside of the cache internal to the processor boundary. This is especially important when traversing the four level paging structures of the virtual memory or the EPT system. Having these structures internal to cache reduces page translation times for the MMU. However, there are certain system memory pages contained in these structures that must never be maintained in the cache. If these pages are cached, read operations may return stale or incorrect information about the state of the page. The particular pages that must be considered will vary on each kernel being protected. Those described here serve as a guide to the types of pages that must be considered. For ExOShim on the prototype kernel used in this paper, pages that must be marked uncacheable in the creation of the identity mapped EPT are concerned with the VGA driver, Interrupts, and Memory Registers associated with the Network Card.

The prototype kernel uses legacy Basic Input/Output System support for its VGA driver. This is a block of low memory, which when written into, determines what characters, color, and screen position to place information on the monitor. If this memory is cached, when the processor accesses the VGA memory, the MMU will not traverse the EPT, but access the internal cache. When the cache page is accessed the last character or configuration setting will be returned to the processor, which will result in an undefined opcode exception. When these pages are marked uncacheable the MMU traverses the EPT and is forced to go to external memory to find the latest information contained in that memory page.

For interrupt handling the Advanced Programmable Interrupt Controller (APIC) and Input/Output APIC (IOAPIC) are used. The processor determines the type of interrupt that occurs by reading memory mapped registers on pages that are written to by these controllers. These pages refer to internal control logic rather than memory. However, when caching is enabled, just as in the VGA memory, the processor interprets these pages as actual memory. Thus, instead of fetching information internal to the processor the MMU attempts to access information contained on these pages, which results in a page fault. Disabling caching of these pages forces the processor to read its internal control structures, as the interrupt subsystem requires.

For the network card to perform Direct Memory Access (DMA), it also makes use of memory-mapped registers called Memory Mapped Input/Output (MMIO) registers. The network card reports to the PCI subsystem what virtual memory pages it requires for MMIO registers. The prototype kernel writes to these virtual memory pages to program the network card for DMA. However, for the network card to be aware of these changes it must poll these registers through the actual physical memory. However, if the pages are contained in the cache, the card will never receive updated programming from the kernel on how to perform DMA. Forcing the pages associated with MMIO registers to be uncacheable allows the network card to receive the necessary information for DMA.

4.3 Starting Virtualization

Virtualization as seen on modern systems is described in terms of Type-1 or Type-2 hypervisors. A Type-1 hypervisor runs directly on top of the hardware and acts as an abstraction layer between the guest and the hardware. Type-2 hypervisors run on top of an operating system and use system hooks into the operating system kernel to enable the running of guest operating systems. ExOShim is unique in that it must insert itself as a Type-1 hypervisor after the kernel has already started running. This, and the requirement to protect kernel memory, requires some special handling while enabling virtualization features.

4.3.1 Required Data Structures
Intel processors support a wide range of virtualization features and settings. The programming of these is accomplished through the Virtual Machine Control Structure (VMCS). To support any level of guest abstraction the VMCS contains over one hundred configurable fields. These fields can be subdivided into Execution Control, Exit Control, Entry Control, Host-state, Guest-State, and Exit Information fields. ExOShim programs the control fields with only those settings required to enable EPT and exit on catastrophic failure. It does not use Exit Information fields as they are populated on *vmexit*; ExOShim simply halts execution if a *vmexit* occurs. How ExOShim uses the Host-State (hypervisor) fields and the Guest-State fields requires more detailed consideration.

4.3.2 Host VMCS State
The Host-State fields in the VMCS define the context of the hypervisor when control is transferred from the guest to hypervisor during a *vmexit*. For ExOShim, they contain the information needed to run an exit handler upon hypervisor reentry. In a Type-1 hypervisor the CR registers, selectors, Descriptor Tables, and stack are setup by the hypervisor itself and then programed into the Host-State fields. ExOShim's unique status "between" Type-1 and Type-2 is advantageous in this case. The kernel has already initialized these fields. Thus, they can be read from memory and loaded into the Host-State fields. There is only one unique Host-state field: the location of the exit handler to call when a *vmexit* occurs. This is a simple field to initialize, the address of the appropriate exit handler function is written directly into the appropriate Host-State field.

4.3.3 Guest VMCS State
Typically, this is the initial state needed by a kernel to begin operation. In this case, the kernel is already operating. Instead of launching another new kernel, the goal of ExOShim is to let the existing kernel resume as if ExOShim had never been initialized.

If ExOShim were a normal process, it would *yield* to the next process. This involves saving and suspending the kernel context of the yielding process, and restoring the context of the next process to be scheduled. ExOShim aims to emulate this procedure; the guest instruction pointer (RIP), stack pointer (RSP) and paging structures (CR3) are extracted from the next process' saved state, accessed by invoking the kernel scheduler. By invoking the scheduler and setting the guest state to match that of the next process to run, ExOShim enables the kernel to carry on undisturbed when it launches the virtual machine.

Additional challenges associated with the Guest-State fields are the Global Descriptor Table (GDT) and Task State Segment (TSS). Unlike the Host-State fields, the guest requires significant detail regarding the information contained in the GDT pertaining to access rights and segmentation. ExOShim must restore the state found in the kernel exactly. If a single bit in this field does not match the running guest state, the guest cannot resume operation. For Type-1 and Type-2 hypervisors the guest can populate these fields for the hypervisor. The nature of ExOShim's insertion process makes this impossible. Fortunately, the GDT containing this information can be read from memory using Intel provided assembly code. From there it can be dissected and loaded into the necessary Guest-State fields. The TSS information for guest operation is contained at a set offset inside of the GDT.

4.3.4 Launching the VM (Resuming the Kernel)
To return operation to the kernel from ExOShim, a *vmlaunch* instruction must be executed to change the processors operating context. Upon execution of this instruction, the Transition Lookaside Buffer (TLB) is flushed to prevent accidental, or unauthorized, guest and hypervisor sharing. The processor then reads the data stored in the VMCS fields to ensure they are in a legal configuration. Upon passing these checks the processor then resumes operation of the guest context, which is the next process to be scheduled by the kernel.

From this point forward ExOShim will only run again in one of three cases: The first is an unlikely catastrophic failure in the guest's interrupt system, which will cause a triple fault and *vmexit*. Without a hypervisor, the processor would hard freeze in this case. The second case occurs if the guest attempts to read or write a protected execute-only kernel code page. This will result in a *vmexit* associated with an EPT violation. Finally, if the guest attempts to read, write, or execute a protected ExoShim page; this will again result in a *vmexit* due to an EPT violation.

In the event that any of these conditions occur, the ExOShim *vmexit* handler is invoked. In the prototype, this routine executes a processor halt instruction immediately. However, the handler could also preform other actions such as restart the kernel from a known gold-standard or log the failure.

5. EVALUATION
In this section, the complexity of the ExOShim prototype is evaluated by counting its lines of code; it is well known that the number of vulnerabilities is directly proportional to lines of code [39]. The prototype's performance is examined by comparing the cycles consumed by the processor on a standard test suite, running on the same kernel, both with and without ExOShim. Finally, a qualitative discussion of the threats that ExOShim seeks to combat explores the protections offered by the prototype.

5.1 Prototype Complexity
ExOShim is implemented in terms of only two functions: an initialization/installation function and the exit handler function. The initialization function requires some memory management code, a handful of routines to detect the 3 types of special memory by the EPT as discussed in Section 4.2, and initialization for the VMCS structure. All told, the initialization routine comprises 290 lines of code. The simple handler that deals with terminating execution if the kernel violates any of the EPT permissions set forth adds only 35 lines of code. However, these routines rest upon kernel functions already present in the prototype 64-bit microkernel presented in [38]. As a point of comparison regarding the size of attack surcface, the popular Xen hypervisor is approximately 150,000 LOC while ExOShim totals 325 LOC.

5.2 Performance
A performance benchmark suite packaged with [38] was run on a Dell Optiplex 9010 workstation with 4 GB of RAM and an Intel core i7-3770k quad-core 3.5 GHz processor with an 8 MB cache. The test suite consists of a series of tests that exercise different parts of the kernel. This primarily consists of memory intensive queue, heap, hashmap, and inter-process communication tests. Network tests were not included in the test suite in order to eliminate non-deterministic network based latencies from hiding the effects of ExOShim on performance.

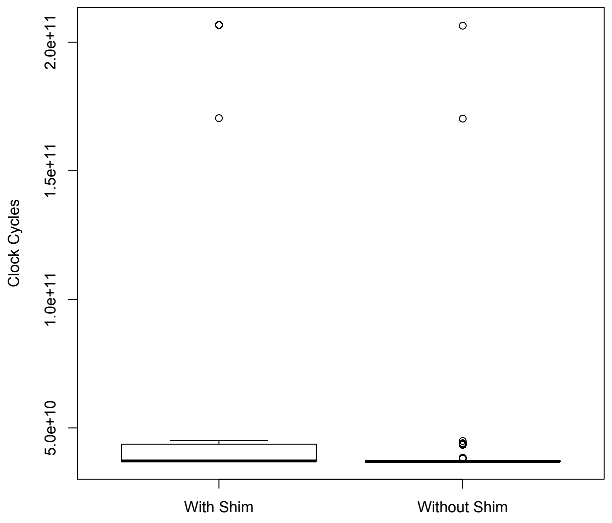

Figure 4. Performance Benchmark Tests

The memory-intensive test suite was run 425 times on the same machine – once with ExOShim installed and once without. In each instance of each test case, the clock cycles needed to complete all tests in the suite was recorded. The results of these tests are shown as boxplots in Figure 4. Note that the extreme outliers in Figure 4 hide detail from the boxplots; Figure 5 shows a closer view of the boxplots, hiding the extreme outliers.

The overall performance cost of ExOShim is estimated by comparing the average cycle count for a full test suite iteration without ExOShim to that with ExOShim. A summary of the test results can be found in Table 1, shown both with and without the outliers.

Table 1. Performance Benchmark Test Results

Test Case	Without ExOShim		With ExOShim	
Outliers Included	No	Yes	No	Yes
Max (x10^{10})	4.492	20.64	4.511	20.67
Min (x10^{10})	3.686	3.686	3.706	3.706
Mean (x10^{10})	3.850	3.921	3.883	4.111
Performance Cost (%)	n/a	n/a	0.86	4.85

Even with the outliers included there is only a 4.85% performance cost for using ExOShim to make the kernel code execute-only. However, the outliers appear both with and without ExOShim. Clearly, they represent an unanticipated artifact of the test suite or kernel. Without the outliers, the performance overhead drops to only 0.86%. This performance cost is a startling testament to modern caching technology. Without ExOShim, every memory access uses a 4-step page table translation mechanism (if not found in the TLB). With the EPT from ExOShim, however, each of those 4 steps must go through its own 4-step EPT walk. To accomplish this additional level of indirection with such little performance cost is truly impressive.

5.3 Security Implications

The security benefits of using ExOShim are clear: it substantially increases attacker workload. Without read permission on kernel code pages, traditional memory disclosure bugs at the kernel level are obsolete. This mitigates an entire range of information leakage-based attacks, including various forms of ROP. An execute-only kernel binary also inhibits most reverse engineering techniques, which rely on reading code.

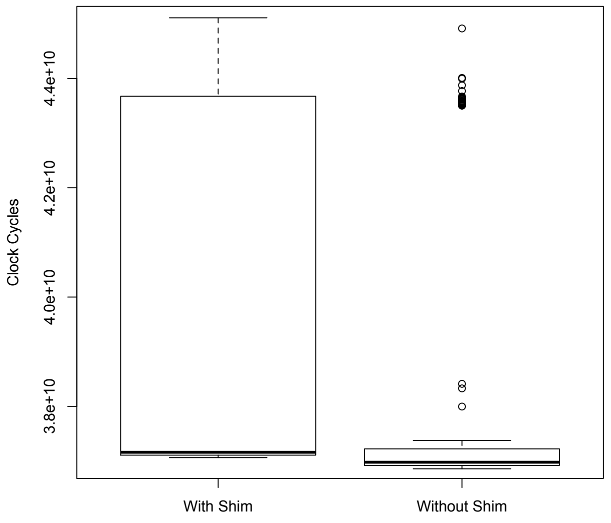

Figure 5. Performance Benchmark Tests - No Outliers

Even with ExOShim, attackers can still learn some facts regarding the layout of the address space. For example, it is possible to read return addresses off of the stack and function pointers stored in data to learn about the location of code. This has been used to construct JIT-ROP attacks [13]. However, a system that employs fine-grained function and block level diversity [28] devalues this information. With thorough code randomization, an attacker must read the code in order to know exactly where gadgets are located; finding the entry point to a function is not sufficient.

An added security benefit of using ExOShim comes simply from turning on virtualization. The first entity to enable virtualization owns the processor [41]. Without a virtualized component, an operating system running on bare metal is vulnerable to VMM rootkits such as [16]. With ExOShim initialized, any attempt to install such a rootkit would invoke the ExOShim exception handler.

6. FUTURE WORK

The ExOShim prototype described and evaluated in this paper provides complete execute-only protection on all kernel code. However, there are many possible extensions of the current prototype that can be explored further.

The operating system used in this study consists of both a 64-bit hypervisor with an associated microkernel that share code extensively to reduce their attack surface [38]. The hypervisor is used to non-deterministically refresh the micro-kernel from a read-only gold-standard to deny persistence. Similarly, the micro-kernel non-deterministically refreshes user-space device drivers and services. Both the hypervisor and kernel employ compile- and load-time diversity to ensure that at every refresh results in loading a unique binary, sharing no exploitable addresses with other instances of the same binary; recall that this serves to *repeatedly* throttle vulnerability amplification and invalidate surveillance over time. The current prototype of ExOShim replaces the hypervisor but still employs diversity in loading the kernel randomly across the entire virtual address space. Fortunately, Intel virtualization technology allows *nested* *virtualization*. This would allow for the shim to initialize underneath the kernel, even if the kernel is already operating on top of another hypervisor. Support for this capability requires adding functionality to the underlying hypervisor, not to ExOShim.

Although the microkernel is a full 64-bit multicore implementation, ExOShim currently operates on a single core. To generalize the implementation will require initializing ExOShim once *per core*; each core must be virtualized independently. It would also be possible to place different permissions in the ExOShim EPTs with different cores assigned to specific tasks.

Currently, devices that support Direct Memory Access (DMA) are a known threat to the protection provided by ExOShim. These devices make memory requests *without* passing through the EPT translation provided by ExOShim. Intel's VT-d virtualization extensions are designed to eliminate this problem by providing a table structure similar to the EPT that defines a translation mechanism through with these devices must pass. In fact, the same memory can be used for both the EPT and the VT-d tables. Therefore, ExOShim can eliminate the DMA threat by enabling and configuring VT-d's DMA remapping functionality.

7. RELATED WORK

The idea of making machine operation instructions execute-only has become more active recently. The following subsections briefly introduce other state-of-the-art memory protections techniques and compare them with ExOShim.

7.1 Execute-Only-Memory

Similar concepts have been used in a variety of systems for protection of user code. For example, Readactor [11] is a kernel implementation that allows the administrator chose to mark user-space applications execute-only. ExOShim on the other hand, was specifically crafted with the goal of preventing ExOShim from being disabled or hijacked. Furthermore, ExOShim protects the operating system pages, not user-space applications.

Backes et al [3] have shown in Execute-no-Read (XnR) another way of achieving execute-only code pages. Though, unlike ExOShim it emulates execute-only paging. XnR uses a sliding window mechanism to keep the last *n* pages both readable and executable. All other pages are marked as non-present. Note that a page is readable while it is marked executable. Unfortunately, an attacker could craft an exploit once access is gained to the page. Once a gadget has been crafted, XnR could be tricked into jumping to malicious code pages and thus marking other pages as readable. ExOShim on the other hand, offers true execute-only memory enforced by the hardware. Utilizing Intel's EPTs, a page is marked execute-only and not readable. This prevents an attacker from obtaining information about running the programs code.

7.2 Software Diversity

Cohen [10] introduced the notion of diversity in software. Later on, Forrest et al [19] showed that diversifying the stack can mitigate stack-smashing attacks. Address Space Layout Permutation [46] is an improvement to the aforementioned ASLR. Kanter et al [28,29,30] have shown several additional approaches, including block level diversity through source-to-source transformation, load-time diversity by dispersing functions over virtual memory, and function replication to masquerade execution paths.

Oxymoron [4], introduced by Backes and Nürnberger randomizes the code layout on a 4 kB page granularity. Note that ExOShim utilizes load-time diversity on a much more fine granular level. In fact, the ELF standard and its section alignment requirements impose the only constraint in the diversity implementation. Though, it ought to be mentioned that a 4 kB granularity allows to share the code page between multiple protected applications running on the same system. Once this granularity is crossed on the lower bound, no two applications must share one code page. Using an internally hidden table, Oxymoron attempts to jump between code pages through segmentation and inter-page calls.

Gionta et al [22] proposed HideM. Here, the concept of a Split Transaction Lookaside Buffer (TLB) is being used. By utilizing a Split-TLB, Gionta is able to direct instruction fetches and data reads to different physical memory locations. An instruction fetch is a legitimate operation, as well as a *white listed* data read of embedded constants. Used by PaX [46] previously to implement W⊕X (write XOR execute), it now will most likely not work on any modern hardware system since modern processors use unified second level TLBs.

Table 2. Comparison of Memory Protection Techniques

		ExOShim	Readactor [11]	XnR [3]	HideM [22]
Prevents Information Leakage	Kernel	Yes	No	No	No
	User	No	Yes	Yes	Yes
Performance Cost		0.86%	6.4%	2.2%	1.49%
Persists through Kernel Compromise		Yes	No	No	No

7.3 Comparison

Table 2 illustrates and contrasts key features of memory protection mechanisms. ExOShim makes two contributions to the area of memory disclosure prevention. Firstly, it targets kernel code rather than user-level applications. The kernel manages all processes, hardware, and other system resources. Hence, finding vulnerability in the kernel will directly impact the security and stability of the entire system. Thus, ExOShim explicitly targets prevention of kernel information leakage.

Furthermore, ExOShim is designed to persist despite any potential kernel compromise. ExOShim does not accept any input after its initialization and uses the EPT to protect all of its own code and data from any read, write, or execute access. As shown in table 2, Readactor, XnR, and HideM could all be hijacked or disabled in the event of a kernel compromise. Readactor provides an explicit mechanism for the kernel to configure or disable its execute-only protections. XnR uses a page fault handler to implement its execute-only memory protections; the kernel can change the page fault handler at any time, completely bypassing XnR. HideM also relies on kernel cooperation to provide the memory protection it offers. It requires the kernel to "prime" the split-TLB with different images of memory in the read-case versus the execute-case. ExOShim distinguishes itself from these approaches by acting specifically to survive through kernel compromise.

ExOShim denies all direct memory disclosures at the kernel level and configures itself to persist through kernel compromise while incurring a performance cost of only 0.86%.

8. CONCLUSION

This paper presented ExOShim, a novel technique in which an operating system deploys a thin "shim" virtualization layer that uses Intel's EPT hardware to provide execute-only protection to all kernel code. This prevents all direct memory disclosure scenarios. The attacker cannot read kernel code directly. Thus, an entire class of kernel-level attack vectors is denied by prohibiting the attacker from disassembling kernel code pages to find gadgets for any type of code reuse attack.

The method illustrated by ExOShim relies on kernel code being loaded onto unique pages. This can be performed at compile- or load-time, and can be combined with software diversity and randomization. When combined with function and block level diversity, even an indirect memory disclosure (such as reading a return address from the kernel stack) does not provide sufficient information to harvest gadgets. The ExOShim prototype discussed here operates in conjunction with compile- and load-time diversification techniques.

This work demonstrates that modern commodity hardware does support a hardware-enforced execute-only primitive that can be deployed to prevent kernel level exploits. It also shows that the flexibility of Intel's virtualization technologies allows the shim layer to protect itself from being disabled or hijacked by an attacker.

The 325-line ExOShim prototype denies an entire class of common attacks at the kernel level while significantly increasing the workload associated with surveillance, reverse engineering, and exploit development. Due to caching technologies in modern Intel processors, the ExOShim prototype here was measured to introduce a performance cost of only 0.86%.

9. NOTICE

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number: FA8750-11-2-0257.

10. REFERENCES

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming,* October 2013.

[2] A. Avizienis and L. Chen. On the Implementation of N-version Programming for Software Fault Tolerance during Execution. In *Proc. the First IEEE-CS International Computer Software and Applications Conference (COM PSAC 77), Chicago*, 1977.

[3] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1342–1353, New York, NY, USA, 2014. ACM.

[4] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 433–447, San Diego, CA, Aug. 2014. USENIX Association.

[5] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[6] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazi`eres, and D. Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 227–242, Washington, DC, USA, 2014. IEEE Computer Society.

[7] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.

[8] c0ntex. Bypassing non-executable-stack during exploitation using return-to-libc. http://www.open-security.org/texts/4.

[9] S. Checkoway and E. W. Felten. Can DREs provide long-lasting security? the case of return-oriented programming and the avc advantage. 2009.

[10] Cohen. Operating System Protection Through Program Evolution. *Comput. Secur.*, 12(6):565–584, Oct. 1993.

[11] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[12] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[14] C. Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2008.

[15] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2008.

[16] S. Embleton, S. Sparks, and C. Zou. Smm rootkits: A new breed of os independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, pages 11:1–11:12, New York, NY, USA, 2008. ACM.

[17] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. 2015.

[18] P. Ferrie. Attacks on more virtual machine emulators.

[19] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, pages 67–, Washington, DC, USA, 1997. IEEE Computer Society.

[20] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[21] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44(6):735–770, 2014.

[22] J. Gionta, W. Enck, and P. Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 325–336, New York, NY, USA, 2015. ACM.

[23] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[24] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[25] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, June 2014.

[26] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Diversifying the software stack using randomized nop insertion. In S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense II*, volume 100 of *Advances in Information Security*, pages 151–173. Springer New York, 2013.

[27] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-Generated Software Diversity. In S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense*, volume 54 of *Advances in Information Security*, pages 77–98. Springer New York, 2011.

[28] M. Kanter. *Enhancing Non-determinism in Operating Systems*. PhD thesis, Thayer School of Engineering at Dartmouth College, October 2013.

[29] M. Kanter and S. Taylor. Attack mitigation through diversity. In *Military Communications Conference, MILCOM 2013 - 2013 IEEE*, pages 1410–1415, Nov 2013.

[30] M. Kanter and S. Taylor. Diversity in cloud systems through runtime and compile-time relocation. In *Technologies for Homeland Security (HST), 2013 IEEE International Conference on*, pages 396–402, Nov 2013.

[31] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection Against Return-to-user Attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[32] D. Kennedy, J. O'Gorman, D. Kearns, and M. Aharoni. *Metasploit: The Penetration Tester's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[34] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[35] R. Lehtinen, D. Russell, and G. T. Gangemi. *Computer Security Basics*. O'Reilly Media, Inc., 2006.

[36] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. 2010.

[37] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh. Bird: Binary interpretation using runtime disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 358–370, Washington, DC, USA, 2006. IEEE Computer Society.

[38] C. Nichols. Bear - a Resilient Core for Distributed Systems. Master's thesis, Thayer School of Engineering at Dartmouth College, 2013.

[39] R. Pandey and V. Tiwari. Reliability issues in open source software. 2011.

[40] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15, 2012.

[41] J. Rutkowska and A. Tereshkin. Bluepilling the Xen Hypervisor. Black Hat USA, 2008.

[42] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[43] H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *CCS: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM Press, 2004.

[44] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[45] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference*

on Information Systems Security, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[46] PaX Team. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt, 2001.

[47] J . Turley. Taming the x86 beast, 2004.

[48] H. Xu and S. J. Chapin. Address-space Layout Randomization Using Code Islands. *J. Comput. Secur.*, 17(3):331–362, Aug. 2009.