Taming Hosted Hypervisors with (Mostly) Deprivileged Execution

Chiachih Wu [†], Zhi Wang [*], Xuxian Jiang [†]

[†]Department of Computer Science
North Carolina State University
cwu10@ncsu.edu, jiang@cs.ncsu.edu

[*]Department of Computer Science
Florida State University
zwang@cs.fsu.edu

Abstract

Recent years have witnessed increased adoption of hosted hypervisors in virtualized computer systems. By non-intrusively extending commodity OSs, hosted hypervisors can effectively take advantage of a variety of mature and stable features as well as the existing broad user base of commodity OSs. However, virtualizing a computer system is still a rather complex task. As a result, existing hosted hypervisors typically have a large code base (e.g., 33.6K SLOC for KVM), which inevitably introduces exploitable software bugs. Unfortunately, any compromised hosted hypervisor can immediately jeopardize the host system and subsequently affect all running guests in the same physical machine.

In this paper, we present a system that aims to dramatically reduce the exposed attack surface of a hosted hypervisor by deprivileging its execution to user mode. In essence, by decoupling the hypervisor code from the host OS and deprivileging its execution, our system demotes the hypervisor mostly as a user-level library, which not only substantially reduces the attack surface (with a much smaller TCB), but also brings additional benefits in allowing for better development and debugging as well as concurrent execution of multiple hypervisors in the same physical machine. To evaluate its effectiveness, we have developed a proof-of-concept prototype that successfully deprivileges $\sim$ 93.2% of the loadable KVM module code base in user mode while only adding a small TCB (2.3K SLOC) to the host OS kernel. Additional evaluation results with a number of benchmark programs further demonstrate its practicality and efficiency.

1 Introduction

Based on recent advances on hardware virtualization (e.g., Intel VT [19] and AMD SVM [1]), hosted hypervisors non-intrusively extend the underlying host operating systems (OSs) and greatly facilitate the adoption of virtualization. For example, KVM [22] is implemented as a loadable kernel module that can be conveniently installed and launched on a commodity host system without re-installing the host system. Moreover, hosted hypervisors can readily benefit from a variety of functionalities as well as latest hardware support implemented in commodity OSs. As a result, hosted hypervisors have been increasingly adopted in today's virtualization-based computer systems [32].

Unfortunately, virtualizing a computer system with a hosted hypervisor is still a complex and daunting task. Despite the advances from hardware virtualization and the leverage of various functionality in host OS kernels, a hosted hypervisor remains a privileged driver that has a large code base with a potentially wide attack surface. For instance, the KVM kernel module alone contains $33.6K$ source lines of code (SLOC) that should be a part of trusted computing base (TCB). Moreover, within the current code base, several components – inherent to its design and implementation – are rather complex. Examples include the convoluted memory virtualization and guest instruction emulation. These components occupy half of its code base and are often the home to various exploitable vulnerabilities.

Using the popular hosted hypervisors – KVM and VMware Workstation – as examples, if we examine the National Vulnerability Database (NVD) [35], there are more than 24 security vulnerabilities reported in KVM and 49 in VMware Workstation in the last three years. Some of these vulnerabilities have been publicly demonstrated to "facilitate" the escape from a confined but potentially subverted (or even malicious) VM to completely compromise the hypervisor and then take over the host OS [24, 31]. Evidently, having a compromised hosted hypervisor is not just a hypothetical possibility, but a serious reality.

Moreover, once a hypervisor is compromised, the attacker can further take over all the guests it hosts, which could lead to not only disrupting hosted services, but also leaking potentially confidential data contained within guest VMs. It has been reported that the data confidentiality and auditability problem is a main obstacle for the continued growth and wide adoption of cloud computing [2]. Consequently, there is a pressing need to develop innovative solutions to protect the host system and running guest VMs from a compromised (hosted) hypervisor.

To address the above need, researchers have explored various approaches. For example, systems have been proposed to formally verify small micro-kernels (e.g., seL4 [23]) so that they do not contain certain software vulnerabilities. Others (e.g., HyperSafe [49]) admit the presence of exploitable software bugs in hypervisors, but develop new techniques to protect the runtime hypervisor integrity. Additional systems are also developed to re-visit (bare-metal) hypervisor design by proposing new architectures so that the hypervisor TCB can be minimized [30, 43]. However, these systems typically require a new bare-metal hypervisor design such that their applicability to commodity hosted hypervisors remains to be shown.

In another different vein, a number of systems have been proposed to isolate buggy or untrusted device drivers such as [7, 16, 39, 42, 53]. However, it is unclear how they can be applied to protect hosted hypervisors. In particular, they do not address host-guest mode switches and hardware-based memory virtualization (e.g., EPT [19]), which are unique and essential to hosted hypervisors. HyperLock [51] similarly creates a separate address space in host OS kernel so that the execution of KVM as a loadable module can be isolated. However, it still runs in privileged mode and requires additional complex techniques to avoid possible misuse of privileged code.

In this paper, we present *DeHype*, a system that applies the least privilege principle to hosted hypervisors so that the attack surface can be dramatically reduced. Specifically, by deprivileging the execution of (most) hypervisor code in user mode, we can not only reduce the exposed attack surface, but also protect the host system even in the presence of a compromised hypervisor.[1] However, challenges exist to deprivilege hosted hypervisor execution. In particular, hosted hypervisors are typically tightly coupled with the host OSs. Accordingly, we propose a *dependency decoupling* technique to break the tight dependency of hosted hypervisors on host OSs. In other words, the related kernel interfaces leveraged by hosted hypervisors are abstracted and provided at the user space. As a result, the related functionalities such as memory management and signal handling could be re-provisioned to the hypervisor without the help of the host OS. Moreover, to allow for hardware virtualization support (e.g. Intel VT-x [19]), there are certain instructions that cannot be deprivileged. To accommodate them, we define a minimal subset of privileged hypervisor code into an OS extension, called *HypeLet*. When the (deprivileged) hypervisor demands to issue a privileged instruction, it traps to the HypeLet by system calls and executes the related instruction in privileged mode. In addition, as hardware support for memory virtualization such as EPT [19] requires mapping virtual addresses into physical addresses, when DeHype deprivileges the related memory virtualization functionality to user mode, we accordingly propose another technique called *memory rebasing* for efficient translation in user mode.

We have developed a proof-of-concept prototype to deprivilege the popular hypervisor KVM (version kvm-2.6.32.28). Specifically, our prototype runs $\sim 93.2\%$ of the loadable KVM module code base in user mode while adding a small TCB (2.3K SLOC) to the host OS kernel. By decoupling the hypervisor code from the host OS and deprivileging its execution, our system essentially demotes the hypervisor as a user-level library (e.g., together with the original companion program – QEMU [3]). This brings additional benefits for its development, extension, and maintenance. For example, since it runs as a user mode process, we can use various feature-rich tools (e.g. GDB [18] and Valgrind [47]) to facilitate its development and debugging. Moreover, the DeHype design naturally supports running multiple (deprivileged) hypervisors independently on the same host and also opens new opportunities in readily applying recent "out-of-VM" monitoring methods or security mechanisms (e.g., VMwatcher [20] and Ether [13]). The evaluation with a number of benchmark programs show that our system is effective and lightweight (with a performance overhead of less than 6%).

The rest of the paper is organized as follows: In Section 2, we present the overall system design, followed by its implementation in Section 3. After that, we report the evaluation results in Section 4. We then discuss possible improvements in Section 3. Finally, we describe related work in Section 6 and conclude the paper in Section 7.

2 System Design

By effectively deprivileging the execution of hosted hypervisors, we aim to significantly reduce the attack surface possibly exposed from them. To elaborate our design, we use the popular KVM hypervisor as the example. Specifically, KVM is an open-source host hypervisor that has been integrated into mainstream Linux kernel. It is implemented as a loadable kernel module, which once loaded extends the host OS to make use of hardware virtualization support. Each KVM-based guest has a user-mode companion program called QEMU. It facilitates bootstrapping guest machines and emulating certain hardware devices (e.g., network cards) by directly interacting with KVM via system calls. For instance, the companion QEMU program may

[1] Although the hosted hypervisor includes the host OS in its TCB, we greatly narrow down the interface exposed by the host OS to untrusted (guest) code.

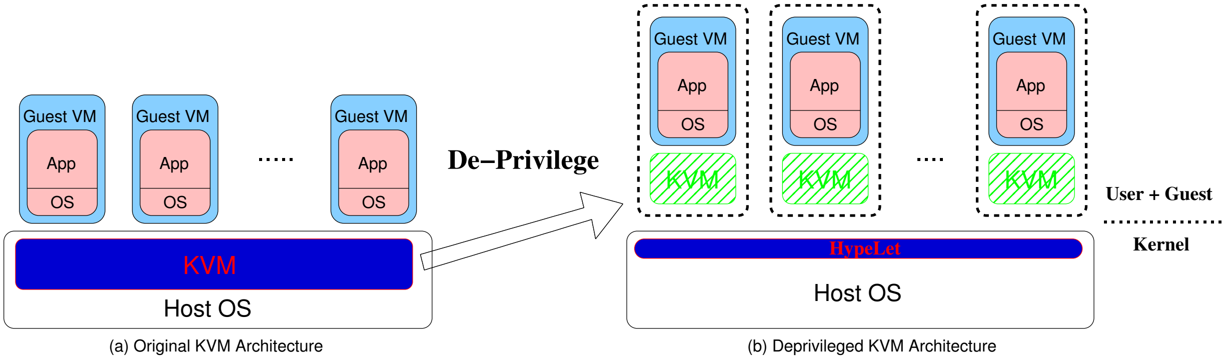


Figure 1. An overview of DeHype to deprivilege hosted hypervisor execution

issue an `ioctl` command, say `KVM_RUN`, to KVM to perform a host-to-guest world switch. By design, each guest VM is paired with an instance of the user-mode QEMU program while sharing the same privileged KVM hypervisor instance with other guest VMs.

With DeHype, we decompose the KVM into two parts: the deprivileged KVM hypervisor running in user mode and a minimal loadable kernel module called *HypeLet* running in kernel mode. The deprivileged KVM essentially runs as a user-level library that provides necessary functionalities to interact with HypeLet. In our current design, we naturally integrate the deprivileged KVM into its user-mode companion program QEMU. By doing so, when QEMU issues an `ioctl` command to KVM, the deprivileged hypervisor receives it as a user-mode function call and then processes it locally. If the processing involves certain privileged code that cannot be deprivileged, it relays the request to HypeLet through a system call. As a result, if a host runs multiple VMs, each VM is paired with its own instance of deprivileged KVM and the original QEMU instance while sharing the same *HypeLet* OS extension. In Figure 1, we show the comparison between the original KVM and the deprivileged KVM. In the rest of this section, we describe our system in detail with a focus on key challenges and related solutions.

2.1 Dependency Decoupling

To deprivilege a hosted hypervisor, our first challenge is to delineate the tight dependency between the hosted hypervisor and the host OS for decoupling. Particularly, KVM intensively leverages several key functionalities implemented in the host OS. For example, KVM allocates kernel memory based on the default slab allocator [6] provided by Linux kernel. Also, the scheduling API, `cond_resched`, is invoked to relinquish the processor such as when the hypervisor is pending for certain inputs or events. Accordingly, we need to supply those related functionalities to the hypervisor in user space.

Our approach starts from performing a breakdown of the KVM hypervisor. By decomposing it into multiple components, we gain necessary insights and take different ways to deprivilege them. Specifically, there are a few components that involve little or no interaction with the host OS and thus can be largely moved into user space in a straightforward manner. One such example is guest instruction emulation component in KVM. Although the component itself is rather complex and will be invoked to interpret and execute certain guest instructions, its interaction with the host OS is minimal and can be largely deprivileged to user mode.

Meanwhile, there also exist certain components that may rely on host OS for their functionalities. A representative example is the kernel memory management that depends on the host OS kernel by utilizing known kernel APIs for memory allocation and deallocation. To deprivilege it, we need to provide a user-mode counterpart. In certain cases where a privileged operation may be involved, a user-mode replacement may not be sufficient and it becomes necessary to split the functionality into two parts: one in user mode and the other in kernel mode. As the user-mode part is deprivileged, there is a need to minimize the kernel-mode part, which eventually becomes part of HypeLet. One example is the guest memory virtualization where basic operations on updating guest page tables may be performed in user mode but critical ones on instantiating or putting them into effect should be performed in kernel space only.

Last but not least, there also exist certain components in KVM that may not be demoted to user space. For example, kernel-side event handling and notification as well as hardware virtualization support (e.g., Intel VT-x [19]) will remain inside the host OS kernel and become part of HypeLet. We highlight that HypeLet should contain only the privileged hypervisor code that simply cannot be executed in user space. Being part of TCB, it is desirable to keep HypeLet minimal. In our current prototype, it mainly contains those privileged instructions introduced for hardware virtualization support (e.g. Intel VT-x [19]). When the deprivileged hypervisor running in user mode demands to issue such a privileged instruction, it traps to the HypeLet by a system call, which then executes the corresponding instruction in the privileged kernel space. In addition, other than those privileged instructions, there also exist kernel-side routines in HypeLet to facilitate the inquiries from the deprivileged hypervisor. For example, a `MAP_HVA_TO_PFN` service is provided to translate the host virtual address to the related physical memory frame, which is needed to deprivilege hardware-assisted memory virtualization (Section 2.2).

To further restrict the deprivileged KVM and user-mode QEMU, we also limit the exposed system call interface and available resources with system call interposition. By doing so, we can effectively mediate the runtime interaction from deprivileged KVM (and QEMU) with HypeLet. As the system call interposition mechanism is a well-studied topic, we omit the details in the paper.

2.2 Memory Rebasing

Our next challenge is to efficiently support hardware-assisted memory virtualization such as Intel's EPT [19]. Specifically, with hardware-assisted memory virtualization, a hosted hypervisor requires to directly manage memory pages in physical address space so that those addresses stored in the nested page tables can be accessed by guest VMs. In the original KVM design as a loadable kernel module, it can simply enjoy feature-rich APIs in the host OS kernel to perform the translation between virtual and physical address spaces. However, once deprivileged, it poses challenges in two main aspects: First, a memory page allocated by a user-level program may be paged out at runtime. Second, a user-level program does not have the mapping information for virtual-to-physical translation.

In our current prototype, we solve these problems by allocating pinned memory blocks in Linux kernel and mapping them to user space. Specifically, through HypeLet, we pre-allocate a contiguous pinned memory block for each hypervisor. The pre-allocated memory block is then mapped to user space through the `mmap` system call so that the (deprivileged) hypervisor can access and use it to build the memory pool for its internal memory management (Section 2.1). By passing the base address of the pre-allocated memory to it, the hypervisor though running in user mode can still obtain the necessary mapping to translate a host virtual address of the memory chunk allocated from its memory pool into a physical address. Accordingly, we propose a *memory rebasing* technique that allows for simply calculating the offset from the memory pool in virtual space and adding it to the base of the pre-allocated block in physical space. Since the memory pool mapped from a pinned memory block is allocated in kernel, we can ensure that any memory page allocated from the pool is always *present*. Therefore, the hypervisor can safely assign those memory pages into the nested page tables with the corresponding physical addresses.[2]

In essence, by applying the *memory rebasing* mechanism, we can allow the deprivileged hypervisor to maintain nested page tables (NPTs) in user mode. With that, these NPTs become the interface for guest VMs to access actual physical memory pages. It has a caveat though: if the hypervisor is compromised, despite the fact that it runs in user mode, a guest VM might still be able to access memory beyond the permitted range. In other words, these NPTs may be exploited to subvert the host OS. Fortunately, as NPTs are only used in guest mode, we can postpone all NPT updates (requested by the hypervisor) until the next *VM entry* occurs. Since each single *VM entry* is handled by the privileged HypeLet, we can apply a sanity check to ensure only memory pages that belong to the hypervisor or the guest VM are eligible to be mapped (right before HypeLet updates NPTs for actual use).

In our prototype, when the user-mode hypervisor is about to update an NPT entry, the entry address and the value to be stored are recorded in a buffer, which is later batch-processed until the hypervisor traps to the HypeLet. During the sanity check, if a malicious address is identified in the buffer, HypeLet simply suspends its execution of the affected hypervisor and the guest VM. By doing so, a compromised hypervisor cannot access those memory pages that belong to other guest VMs or the host OS.

2.3 Optimizations

When compared with the original KVM running in kernel mode, a deprivileged one needs to trap to HypeLet for privileged operations. This naturally introduces a system call latency and potentially becomes a source of performance overhead. In our prototype, we monitor the bootstrap process of a guest VM to understand the number of traps (to HypeLet) caused by the privileged instructions executed within each `KVM_RUN` session. Our results show that thousands of privileged instructions are executed within most `KVM_RUN` sessions when the guest VM is booting up. As an example, we observe $195,187$ privileged instructions executed within a particular `KVM_RUN` session. If we naively invoke a system call for each privileged instruction, it would translate to $195,187$ system calls for the particular `KVM_RUN` session.

To minimize the performance overhead, we propose a cache-based batch-processing mechanism to reduce the

[2] For simplicity, our current prototype assumes that the hypervisor makes static physical memory allocation in its initialization phase. However, it could be readily extended to support dynamic physical memory allocation (e.g., by maintaining multiple pinned memory blocks and associated base addresses).

number of unnecessary system calls. In particular, by profiling the runtime behavior of deprivileged KVM (another benefit from running it in user mode), we notice that most system calls are triggered by the instructions to access various fields in the virtual-machine control structure (VMCS). Also, we notice that it is not necessary to make those VMCS fields always synchronized. In fact, while running in host mode, as far as these fields are updated before the next guest-to-host world switch, we can ensure the correctness of guest execution. Based on the above observations, we maintain a cached VMCS copy in user mode for the deprivileged hypervisor to access without invoking any system calls. The cached copy will be synchronized to the real one (maintained in kernel) on demand when there is a need to issue a world switch.

Beside the cache-based VMCS optimization, our system also implements another optimization that is related to another frequently invoked privileged service in HypeLet, i.e., MAP_HVA_TO_PFN. This privileged service fulfills the queries to translate a host virtual address into the corresponding physical frame number. Different from the previous memory rebasing mechanism, this service could be used to translate memory pages allocated by the QEMU, which are not from the hypervisor's memory pool. Although these memory pages are not managed by the hypervisor, it still needs the physical address to handle related NPT faults. We notice that the mapping of these memory pages is always consistent throughout the QEMU lifetime, we can therefore cache the mappings that are already queried inside the hypervisor to reduce the number of system call traps into HypeLet.

3 Implementation

We have implemented a proof-of-concept prototype to deprivilege the KVM execution (version 2.6.32.28). Our current prototype is developed on a Dell desktop (with the Intel Core^{TM} i7 860 CPU and 3GB memory) running Ubuntu 11.10 and Linux kernel 2.6.32.28. Next we present our prototype in more details.

3.1 Dependency Decoupling

To deprivilege the KVM execution, our prototype abstracts the host OS interface that is being used by KVM and provides a similar one in user mode. Specifically, our prototype provides a slab-based memory allocator in user mode to fulfill the need of allocating and releasing memory to satisfy KVM needs. But different from the default memory allocator in Linux kernel that prepares its memory pool in boot-up time with the pre-defined kernel heap, our version of memory allocator can be flexibly configured to set its heap to an arbitrary memory block in user space, which becomes one key step to enable the memory rebasing mechanism (Section 2.2).

Table 1. Ten Privileged Services in DeHype

Name	Function Description
VMREAD	read VMCS fields
VMWRITE	write VMCS fields
GUEST_RUN	perform host-to-guest world switches
GUEST_RUN_POST	perform guest-to-host world switches
RDMSR	read MSR registers
WRMSR	write MSR registers
INVVPID	invalidate TLB mappings based on VPID
INVEPT	invalidate EPT mappings
INIT_VCPU	initialize vCPU
MAP_HVA_TO_PFN	translate host virtual address to physical frame

Our prototype also provides necessary function routines to emulate original kernel memory access APIs. For example, `virt_to_page` has been widely used in KVM to translate a virtual address to the corresponding memory frame. As the deprivileged hypervisor allocates memory pages from an internal memory allocator, the original memory accesses cannot be directly used but need to be adjusted for conforming to a different memory layout of the memory heap. Moreover, our prototype also leverages the default support in GLIBC [17] for a variety of issues, such as handling signals, performing process scheduling-related operations, and invoking system calls to trigger the privileged HypeLet services. As these library routines are ready-to-use, we found integrating them together with the deprivileged KVM hypervisor is a rather straightforward process.

As mentioned earlier, there also exist some privileged instructions that cannot be demoted to user space. To accommodate them, our prototype introduces HypeLet to support a minimal set of privileged hypervisor code that can be invoked from the deprivileged KVM. In Table 1, we show those privileged services being supported in HypeLet. In total, there are 10 privileged services. Six of them, i.e., VMREAD, VMWRITE, GUEST_RUN, GUEST_RUN_POST, INVVPID, INVEPT, are services for executing privileged instructions that are introduced for hardware virtualization support. INIT_VCPU is another service that basically initializes essential data structures for a virtualized guest VM, including vCPU. RDMSR and WRMSR are two other services to access model-specific registers with privileged instructions. Our profiling results indicate that RDMSR and WRMSR are mainly used in the VM initialization phase and do not frequently occur in normal hypervisor execution. The last service, MAP_HVA_TO_PFN, does not contain any privileged instruction but is included to answer requests (from the deprivileged KVM) about the mapping from a host virtual address to its physical address. Since the hypervisor requires the mapping to handle possible NPT faults, MAP_HVA_TO_PFN is a frequently requested service that should be optimized (Section 2.3).

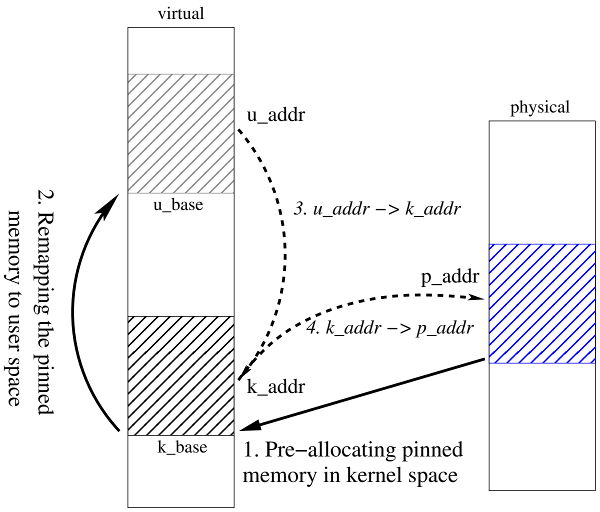

virtual
physical
u_addr
u_base
3. u_addr −> k_addr
p_addr
4. k_addr −> p_addr
k_addr
k_base
1. Pre−allocating pinned memory in kernel space
2. Remapping the pinned memory to user space

Figure 2. The memory management in DeHype. The solid lines mark the ways to generate the memory blocks in different address spaces while the dotted lines mark the translation between memory address spaces.

3.2 Memory Rebasing

With deprivileged KVM, the support of hardware-assisted memory virtualization poses unique challenges. Unlike prior software based approaches that require the hypervisor to frequently update the shadow page tables, the hardware-assisted memory virtualization enables the guest to maintain guest page tables (GPTs) while the hypervisor maintains nested page tables (NPTs) to regulate the translation from guest physical addresses to host physical addresses. To maintain NPTs, the hypervisor requires allocating memory pages and storing the associated physical addresses into NPTs for proper translation. For the traditional KVM as a loadable kernel module, allocating new memory pages and translating their virtual addresses into physical addresses are relatively straightforward. However, with DeHype, the deprivileged hypervisor runs in user mode and does not have the knowledge of the physical addressing space. Moreover, the deprivileged hypervisor cannot prevent the host OS kernel from paging out the memory pages it allocated.

To address these problems, our prototype implements a *memory rebasing* mechanism to facilitate the deprivileged hypervisor to maintain NPTs correctly. In essence, our solution (shown in Figure 2) involves allocating pinned memory pages in kernel space and then remapping them to user space. Specifically, in the initialization phase (line 1), we have the HypeLet pre-allocate a pinned memory block (base address: `k_base`) for each hypervisor.[3] With a simple driver interface implemented in HypeLet, we can allow the user-mode hypervisor to remap the pinned memory block to user space. In particular, a `mmap` call effectively translates `k_base` to `u_base` – so that the pinned memory block based at `k_base` in kernel memory can be accessed by `u_base` in user space (line 2). After that, the `mmap`'ed memory block combined with the (`k_base`, `u_base`) can be used to build the memory pool for the deprivileged hypervisor's memory allocator in user space. By doing so, we can guarantee that each memory page allocated from the pool can be efficiently translated to physical address space with our scheme.

As an example, suppose the hypervisor allocates a new NPT table for NPT violation handling. Whenever an NPT violation occurs, a memory page (located at `u_addr`) is allocated from the memory pool for filling the page table entry. To do that, we need to locate the corresponding physical address, namely `p_addr`. As the mapping of a userspace address to physical address cannot be conveniently retrieved, we choose to use the corresponding kernel space address, namely `k_addr`, and rely on the `virt_to_phys(x)` function, which in our x86-32 Linux-based prototype is a simple calculation, i.e., `(x) - PAGE_OFFSET`, to perform the translation. Further, because `u_addr` is allocated from the memory pool based at `u_base` that has a corresponding kernel space address `k_base`, we can simply calculate `k_addr` by `u_addr - u_base + k_base` (line 3). With that, we can further calculate `p_addr` by `virt_to_phys` (line 4) and use it to update the NPT entry.

To securely update NPT entries (Section 2.2), each deprivileged hypervisor instance saves the pairs of address and value to be updated into a local buffer for batch processing. Note that the NPT consists of four levels of page tables. If the hypervisor needs to update an entry in the level-1 table (the lowest level), the parent table or level-2 as well as all the ancestor tables – level-3 and level-4 – need to be traversed before reaching the level-1 table. Since our hypervisor runs in the user mode and is prohibited from performing NPT updates, there are no actual NPTs for traversal from the hypervisor standpoint. To accommodate that, we choose to construct pseudo NPTs.

Specifically, when an NPT violation occurs, the hypervisor allocates two memory pages, page P from the `mmap`'ed memory pool and page P' from the process' heap while a hash table is used for bookkeeping the relationship. The hypervisor will use P to update the real NPT and P' to update the pseudo NPT. In particular, as illustrated in Figure 3, we first initialize a root-level or level-4 pseudo page table R'. The NPT traversals are redirected to the pseudo page table

[3] In the kernel configuration, `CONFIG_FORCE_MAX_ZONEORDER` can be adjusted for allocating a larger-sized block.

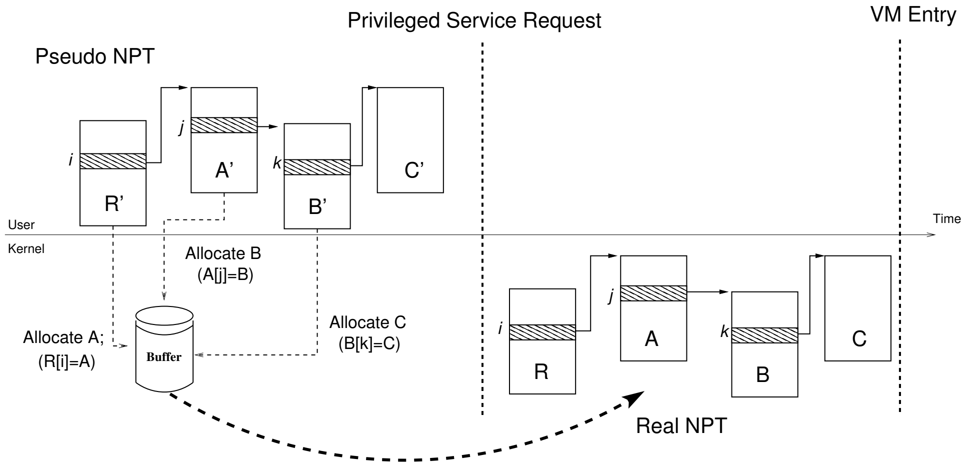


Figure 3. An example of constructing pseudo NPTs for the deprivileged hypervisor to traverse.

and the updates go to the real root-level table R. When the first NPT violation occurs, all NPTs except the root-level one are empty. We then allocate a page A' to modify some entry, say i, of R'. At the same time, we also allocate a page A and issue an update to the ith entry in R. Therefore, a further update on the jth entry of the level-3 table A' could be done by (1) finding A' from R', (2) allocating two page B' and B, (3) book keeping the two pages on the hash table, (4) updating the jth entry of A' with B', and (5) adding a record of updating the jth entry of A with B. For further updates to an existing entry on the pseudo NPT (e.g. flushing page B'), the corresponding log for the page B on the real NPT could be obtained with the help of the hash table. As a result, the hypervisor can traverse the pseudo NPT and generate accurate records for updating the real NPT.

Our pseudo NPT design is similar to the traditional shadow paging but differs in two aspects: First, pseudo NPT only shadows the NPT tables while shadow paging needs to mirror a much larger number of guest page tables; Second, our scheme batch-updates the real NPT tables thus incurs less performance overhead than shadow paging, which is required to trap on the guest's updates to their page tables and synchronize the updates to the real page tables. Our experiments show that pseudo NPT enables the hypervisor to securely manage NPT with a small performance overhead. Although pseudo NPT introduces additional memory overhead, it is necessary to secure NPT updates as we assume that the hypervisor is untrusted.

3.3 Optimizations

As elaborated in Section 2.3, our system design requires a system call to invoke any privileged service in HypeLet, which could introduce extra performance overhead. To mitigate that, we provide a cache-based batch processing mechanism to reduce the number of unnecessary system calls. In particular, our prototyping experience shows that around 90% of invoked privileged instructions are related to accessing the virtual machine control structure (VMCS). Therefore, our prototype aims to reduce the overhead from the large number of VMCS accesses.

To elaborate our implementation, we briefly review how VMCS is accessed in a virtualized system. For each guest VM, the hypervisor needs to allocate memory to initialize the corresponding VMCS. Before the guest launches and between each of its guest-mode runs, two privileged instructions, `VMREAD` and `VMWRITE`, will be executed to access VMCS (for the purpose of either monitoring or controlling the behavior of the guest VM). Throughout the running period in guest mode, the guest VM execution indirectly affects related VMCS fields that can be later retrieved by hypervisor when it switches back to host mode (e.g., triggered by a `VMEXIT`)

In our implementation, we maintain a VMCS copy in user mode so that `VMREAD` calls can simply be redirected to read update-to-date results from the cache without issuing any system call. To avoid synchronizing the large VMCS structure (over 140 fields inside), we profile the KVM execution to locate the top 28 most frequently accessed VMCS fields and save them in the cache. By caching those 28 fields, we found that we can effectively reduce 99.86% of the extra system calls caused by `VMREAD` requests. For `VMWRITE`, we apply the similar caching scheme. By choosing to save the 8 frequently `VMWRITE`'ed VMCS fields, we can reduce 98.28% of extra system calls caused by `VMWRITE` requests. In total, our prototype caches 31 VMCS fields[4] and achieves a good balance between the synchronization cost and the system call latency. The detailed list of cached fields is shown in Table 2.

[4] There are five overlapping VMCS fields common in cached `VMREAD` and `VMWRITE` fields.

Table 2. Cached VMCS Fields

VMREAD	
GUEST_INTERRUPTIBILITY_INFO	GUEST_CS_BASE
IDT_VECTORING_INFO_FIELD	GUEST_ES_BASE
GUEST_PHYSICAL_ADDRESS_HIGH	GUEST_CR3
VM_EXIT_INTR_INFO	GUEST_RFLAGS
GUEST_PHYSICAL_ADDRESS	VM_EXIT_REASON
VM_EXIT_INSTRUCTION_LEN	GUEST_CR4
EXIT_QUALIFICATION	GUEST_DS_BASE
CPU_BASED_VM_EXEC_CONTROL	GUEST_RSP
GUEST_CS_SELECTOR	GUEST_RIP
GUEST_CS_AR_BYTES	GUEST_CR0
GUEST_PDPTR0_HIGH	GUEST_PDPTR0
GUEST_PDPTR1_HIGH	GUEST_PDPTR1
GUEST_PDPTR2_HIGH	GUEST_PDPTR2
GUEST_PDPTR3_HIGH	GUEST_PDPTR3
VMWRITE	
GUEST_RFLAGS	GUEST_RSP
CPU_BASED_VM_EXEC_CONTROL	GUEST_RIP
VM_ENTRY_INTR_INFO_FIELD	EPT_POINTER
EPT_POINTER_HIGH	GUEST_CR3

Further, in order to maintain the same hardware protection scheme of VMCS, we have a *dirty* bit associated with each cached VMWRITE field. When a VMWRITE is requested by the deprivileged hypervisor in user mode for updating a cached VMWRITE field, the dirty bit would be set. On the other hand, if a cached VMWRITE field is somehow written via other ways (e.g., MOV) instead of VMWRITE, the dirty bit would not be set and the content would not be flushed to the hardware. To avoid potential attacks that overwrite a dirty cached VMWRITE field, we also store the hash value of the legitimate VMWRITE'd value in a separate array. Therefore, we can invalidate the illegal cache fields while performing synchronization.

3.4 Lessons Learned

In this subsection, we share additional experiences or frustrations we learned when implementing the prototype. The first one is about missing interrupt events in our earlier unsuccessful prototype. In particular, QEMU issues the `KVM_RUN ioctl` command to enter guest mode. If there is no event occurred (e.g., a pending interrupt), the main thread keeps doing *VM entry* and *VM exit* in a loop. When QEMU is about to inject an interrupt to the guest, it signals the main thread (by `pthread_kill`) so that the main thread knows that it needs to exit the loop and returns to QEMU for interrupt handling (by checking the existence of pending signal after each *VM exit*). In the current KVM code base, it sets the signal masks to ensure that specific signals are allowed to be delivered only when the main thread is in its `KVM_RUN` session to kernel. More specifically, a kernel API `sigprocmask` is used in the entry point of the `KVM_RUN ioctl` to allow only *SIG_IPI* and *SIGBUS* to be delivered. Before returning back to QEMU, KVM restores the signal mask so that those signals would not be delivered when the main thread is running in userspace.

Our earlier prototype intercepts `KVM_RUN ioctl` from the deprivileged hypervisor and handles it in user mode with real `ioctl`s issued for privileged instructions. If the signal mask is set as the original KVM, *SIG_IPI* and *SIGBUS* would be delivered even when the `KVM_RUN` is handled in user mode. Therefore, after each *VM exit*, the signal pending condition would not be accurate since some signals are now delivered in user mode. This is the culprit why our earlier prototype misses interrupt events and fails to maintain accurate system time. To solve this problem, we shrink the allowed signal delivery window to each `ioctl` handler of `VMLAUNCH/VMRESUME` instruction. Since KVM checks the signal pending condition after *VM exit*, it would not affect the QEMU by sending signals but keeps the signal pending condition until the next *VM entry*. This mechanism ensures our system to have a similar interrupt injection frequency as the original KVM architecture has.

Another implementation detail is related to a privileged instruction – `VMPTRLD`. This instruction is used to load the guest states before switching to guest mode when the hypervisor is handling the `KVM_RUN` request. In many cases, especially when the guest is running a CPU intensive workload, a `VMPTRLD` could be followed by multiple runs of (`VMRESUME,VMEXIT`). The reason is that it does not need to handle those *VM exits* in QEMU. Instead, the hypervisor handles the *VM exit* and continues the guest's execution by another `VMRESUME`. However, in some extreme cases such as running an IO intensive workload in the guest, most *VM exits* need to be handled in QEMU (e.g. IO instructions). Since `VMPTRLD` and `VMRESUME` are executed as separate system calls in our system, it requires at least one more system call than the original KVM to handle a single `KVM_RUN` request. If the time running in guest mode is extremely short (e.g., the guest is frequently interrupted by IO accesses), the extra system call latency could introduce significant overheads. Notice that the guest states are only used in guest mode, we can then postpone the `VMPTRLD` instruction until the first `VMRESUME` instruction to eliminate the extra system call.

4 Evaluation

In this section, we evaluate our system by first analyzing security and other related benefits from DeHype and then measuring the performance overhead of our prototype with several standard benchmarks.

4.1 Security Benefits

Reducing the attack surface In this work, we assume host hypervisors, either before or after being deprivileged, contain software vulnerabilities that might be exploited by attackers. Accordingly, the traditional "VM escape" attack from a compromised or malicious VM to the hypervisor will still happen in our system. Fortunately, thanks to the deprivileged execution, potential damages that may be caused from such attacks are mostly limited to the hypervisor itself (i.e., including the QEMU process). In particular, with DeHype, all the interactions between the hypervisor and the guest VM occur in the user space. The host OS kernel instead is not directly accessible to a compromised hypervisor, but must be accessed through the system call interfaces exported by HypeLet, which is the only privileged component added by current hypervisor code base.

In our prototype, HypeLet contains $2.3K$ SLOC and defines 10 system calls in total. To further restrict the access to these system calls, our system adopts the known system call interposition technique (Section 2.1) to mediate their access and block the default system call interface in host OS kernel from being accessible (that has more than 300 system calls in recent 3.2 Linux kernels). As a result, our system effectively reduces the previously exposed wide attack surface to these 10 system calls. Moreover, the added TCB by KVM is reduced from $33.6K$ to $2.3K$ – a $\sim 93.2\%$ reduction.

It is worth mentioning that in DeHype, each guest is paired with its own deprivileged hypervisor. The hypervisor keeps the guest's states in pre-allocated memory pages mapped exclusively in its address space. Therefore, it can only access its own guest; other guests are strictly isolated in other processes and not accessible by default. This has the additional benefit of DeHype by protecting other unrelated guest VMs from the compromised hypervisor.

Testing real-world vulnerabilities To illustrate DeHype's security benefits, we explain how real world vulnerabilities from NVD [35] could be mitigated by our system. In the following, we elaborate three of them. The first one we examined is CVE-2009-4031, a vulnerability that is caused by interpreting wrong-size instructions (with too many bytes) in KVM's guest (x86) instruction emulation. This vulnerability can be exploited by the guest to launch a denial-of-service attack against the host OS kernel. Since DeHype performs instruction emulation in the user space, its exploitation, even successful, is strictly confined within a user-space process. Thus our system effectively mitigates such attack.

The second vulnerability we examine is CVE-2010-0435, which allows the guest kernel to cause a NULL pointer dereference in KVM as some function pointers in its Intel-VT support are uninitialized.[5] Because KVM is originally running in privileged mode, this vulnerability can be exploited to crash the host OS. In DeHype, the vulnerability could still be exploited to crash the hypervisor. However, only the hypervisor that is paired with the malicious guest will be affected. With the isolation provided by process boundary, other hypervisors and the host OS are still not affected. This test case is a good example to show the difference from other related work [15, 29, 51], which leverage software fault isolation techniques to confine memory corruption bugs. Specifically, the difference is that DeHype enables the isolation from hardware (i.e., page tables) instead of rather complex software-based fault isolation techniques.

The third vulnerability is CVE-2010-3881, a vulnerability in KVM that leaks kernel data to user space when certain data structures are copied to the user land but without clearing the paddings. A QEMU process could potentially obtain sensitive information from the kernel stack. In DeHype, such "system call" would be intercepted and handled in the user space as a function call. Therefore, the leaked information would only come from the stack of the hypervisor paired with that QEMU process, not from the kernel or other guest VMs.

4.2 Other Benefits

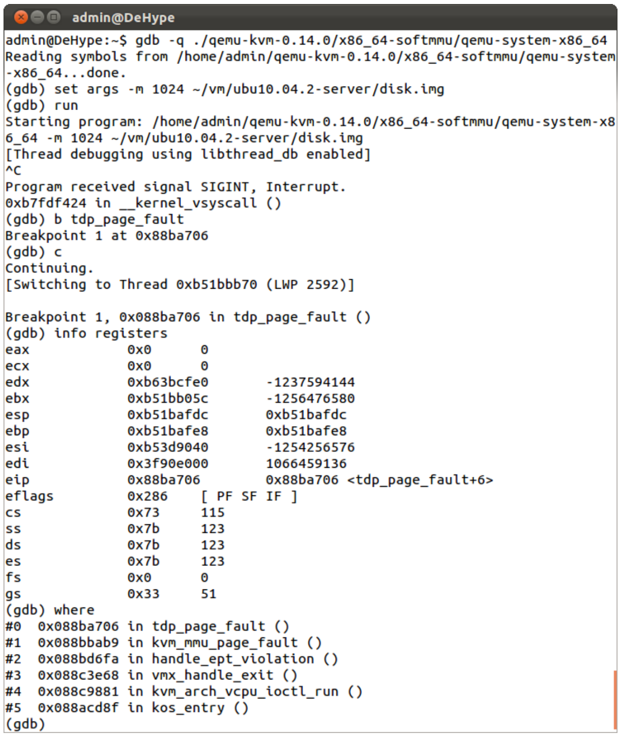
By moving the hypervisor to the user space, DeHype also enables some unique benefits and opportunities. In this section, we elaborate two of them.

Facilitating hypervisor development In DeHype, the hypervisor is deprivileged to the user space. This makes it possible to develop and debug the hypervisor with tools such as GDB that are convenient and familiar to most programmers. For example, when developing our prototype, we used GDB to debug its `pseudo NPT` component (Figure 3 – Section 3), which is one of the most complicated components in the system.

In Figure 4, we show one debug session with GDB. In this session, we set up a breakpoint at the `tdp_page_fault` function, the NPT fault handler in KVM. When the KVM execution hits the breakpoint, we can further display the stack trace with the `where` command, or use the `step/stepi` command to single step the code and examine changes in machine registers and memory contents (e.g., `pseudo NPT` table) after each step. We can also use the `continue` command to resume the execution until the next NPT fault to monitor how the `pseudo NPT` table is built up. During our development, we also used Valgrind [47], a dynamic instrumentation tool, to detect memory leaks in our prototype (Figure 5).

To understand the distribution of modifications in new KVM releases that may be related to DeHype, we manually examined three recent releases of KVM (2.6.32, 2.6.33, and 2.6.34) and attributed each change to either HypeLet or the deprivileged hypervisor. Specifically, we reviewed changes in the `arch/x86/kvm` and `virt/kvm` directories of the Linux kernel which contain the main body

[5] Note that these function pointers are part of the internal data structure of KVM. The guest kernel may trigger NULL pointer dereference by tricking the KVM to emulate some specific instructions instead of crafting the pointers for other purposes (e.g., running shellcode to access privileged KVM system call interfaces).



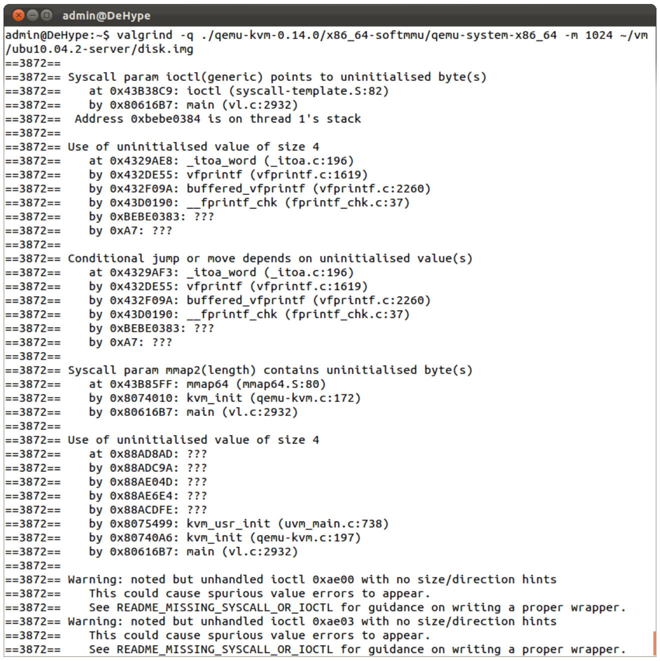
```
admin@DeHype:~$ gdb -q ./qemu-kvm-0.14.0/x86_64-softmmu/qemu-system-x86_64
Reading symbols from /home/admin/qemu-kvm-0.14.0/x86_64-softmmu/qemu-system
-x86_64...done.
(gdb) set args -m 1024 ~/vm/ubu10.04.2-server/disk.img
(gdb) run
Starting program: /home/admin/qemu-kvm-0.14.0/x86_64-softmmu/qemu-system-x8
6_64 -m 1024 ~/vm/ubu10.04.2-server/disk.img
[Thread debugging using libthread_db enabled]
^C
Program received signal SIGINT, Interrupt.
0xb7fdf424 in __kernel_vsyscall ()
(gdb) b tdp_page_fault
Breakpoint 1 at 0x88ba706
(gdb) c
Continuing.
[Switching to Thread 0xb51bbb70 (LWP 2592)]

Breakpoint 1, 0x088ba706 in tdp_page_fault ()
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0xb63bcfe0       -1237594144
ebx            0xb51bb05c       -1256476580
esp            0xb51bafdc       0xb51bafdc
ebp            0xb51bafe8       0xb51bafe8
esi            0xb53d9040       -1254256576
edi            0x3f90e000       1066459136
eip            0x88ba706        0x88ba706 <tdp_page_fault+6>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) where
#0  0x088ba706 in tdp_page_fault ()
#1  0x088bbab9 in kvm_mmu_page_fault ()
#2  0x088bd6fa in handle_ept_violation ()
#3  0x088c3e68 in vmx_handle_exit ()
#4  0x088c9881 in kvm_arch_vcpu_ioctl_run ()
#5  0x088acd8f in kos_entry ()
(gdb)
```

Figure 4. A GDB session that debugs KVM code with the environment familiar to most programmers.

of KVM. According to our examinations, 71.7% changes in KVM-2.6.33 (vs. KVM-2.6.32) and 60.9% changes in KVM-2.6.34 (vs. KVM-2.6.33) can be confined in the user space. With DeHype, their development can benefit significantly from the abundant user-space debugging tools. While the results still show 28.3% changes in KVM-2.6.33 (or 39.1% in KVM-2.6.34) may impact DeHype, this is largely because current KVM development freely uses the large body of host OS kernel APIs without restriction. Once the interface between the deprivileged KVM and HypeLet is defined, we found these changes can be dramatically reduced in HypeLet.

Running multiple hypervisors DeHype also naturally allows for multiple mutually isolated hypervisors to concurrently run on the same host and each may have different security features (e.g., in different versions). To illustrate this, we executed two deprivileged KVM hypervisors on our test machine: one has the secure NPT updating feature enabled, while the other has the feature disabled. A guest is then created for each hypervisor. Since both hypervisors share the same HypeLet, we successfully check all NPT updates issued by the guest running on the hypervisor with the feature turned on while the updates of the other guest are handled by the hypervisor itself.



```
admin@DeHype:~$ valgrind -q ./qemu-kvm-0.14.0/x86_64-softmmu/qemu-system-x86_64 -m 1024 ~/vm
/ubu10.04.2-server/disk.img
==3872==
==3872== Syscall param ioctl(generic) points to uninitialised byte(s)
==3872==    at 0x43B38C9: ioctl (syscall-template.S:82)
==3872==    by 0x80616B7: main (vl.c:2932)
==3872==  Address 0xbebe0384 is on thread 1's stack
==3872==
==3872== Use of uninitialised value of size 4
==3872==    at 0x4329AE8: _itoa_word (_itoa.c:196)
==3872==    by 0x432DE55: vfprintf (vfprintf.c:1619)
==3872==    by 0x432F09A: buffered_vfprintf (vfprintf.c:2260)
==3872==    by 0x43D0190: __fprintf_chk (fprintf_chk.c:37)
==3872==    by 0xBEBE0383: ???
==3872==    by 0xA7: ???
==3872==
==3872== Conditional jump or move depends on uninitialised value(s)
==3872==    at 0x4329AF3: _itoa_word (_itoa.c:196)
==3872==    by 0x432DE55: vfprintf (vfprintf.c:1619)
==3872==    by 0x432F09A: buffered_vfprintf (vfprintf.c:2260)
==3872==    by 0x43D0190: __fprintf_chk (fprintf_chk.c:37)
==3872==    by 0xBEBE0383: ???
==3872==    by 0xA7: ???
==3872==
==3872== Syscall param mmap2(length) contains uninitialised byte(s)
==3872==    at 0x43B85FF: mmap64 (mmap64.S:80)
==3872==    by 0x8074010: kvm_init (qemu-kvm.c:172)
==3872==    by 0x80616B7: main (vl.c:2932)
==3872==
==3872== Use of uninitialised value of size 4
==3872==    at 0x88AD8AD: ???
==3872==    by 0x88ADC9A: ???
==3872==    by 0x88AE04D: ???
==3872==    by 0x88AE6E4: ???
==3872==    by 0x88ACDFE: ???
==3872==    by 0x8075499: kvm_usr_init (uvm_main.c:738)
==3872==    by 0x80740A6: kvm_init (qemu-kvm.c:197)
==3872==    by 0x80616B7: main (vl.c:2932)
==3872==
==3872== Warning: noted but unhandled ioctl 0xae00 with no size/direction hints
==3872==    This could cause spurious value errors to appear.
==3872==    See README_MISSING_SYSCALL_OR_IOCTL for guidance on writing a proper wrapper.
==3872== Warning: noted but unhandled ioctl 0xae03 with no size/direction hints
==3872==    This could cause spurious value errors to appear.
==3872==    See README_MISSING_SYSCALL_OR_IOCTL for guidance on writing a proper wrapper.
```

Figure 5. A Valgrind session that checks possible KVM memory leaks.

This unique capability of DeHype can be potentially leveraged in several different settings. For example, we can apply certain security services such as virtual machine introspection ([39, 20]) to monitor the execution of some guests in a host, while running other guests with the normal hypervisor. Moreover, when a new vulnerability is reported and fixed in the deprivileged hypervisor, we can live-migrate all the guests in a host one-by-one to the patched hypervisor at runtime. Under the original KVM, we need to migrate all the guests to another machine altogether, patch the hypervisor, and migrate them back again.

4.3 Performance

To evaluate the performance overhead introduced by DeHype, we install a number of standard benchmark programs such as SPEC CPU2006 [40] and Bonnie++ (a file system benchmark) [5]. In addition, we use two application benchmarks to decompress and compile Linux kernel. We measured the elapsed time in the guest with the `time` command. Our test platform is a Dell OptiPlex^{TM} 980 desktop with a 2.80GHz Intel Core^{TM} i7 860 CPU and 3G memory. The host runs a default installation of Ubuntu 11.10 desktop with the 2.6.32.31 Linux kernel. The guest runs Ubuntu 10.04.2 LTS server. Table 3 summarizes the software packages and configurations in our experiments.

Table 3. Software Packages used in Our Evaluation

Software Package	Version	Configuration
Benchmarks		
SPEC CPU2006	v1.0.1	reportable int
Bonnie++	1.03e	bonnie++ -f -n 256
linux kernel	2.6.39.2	make defconfig vmlinux
Host/Guest Installation		
Ubuntu Desktop	11.10	default
Ubuntu Server	10.04.2 LTS	default

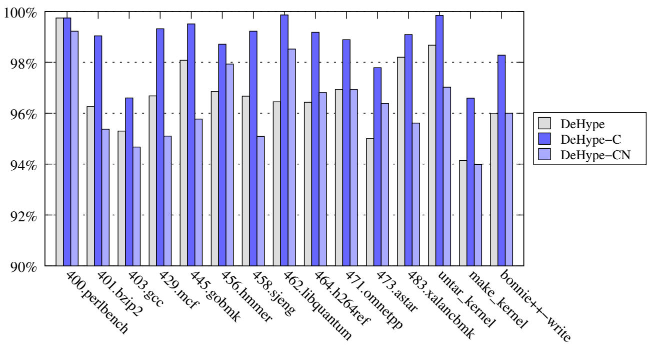


Figure 6. Relative Performance of DeHype

Figure 6 shows the relative performance of running the benchmarks. The first 12 groups of bars present the relative performance of DeHype running the integer benchmarks of SPEC CPU2006 compared with the vanilla KVM while the last three groups present decompressing Linux kernel (`untar_kernel`), compiling Linux kernel (`make_kernel`), and the sequential output performance of Bonnie++. In each group, there are three different DeHype configurations. The `DeHype` bar denotes the vanilla DeHype system; `DeHype-C` reports the optimization benefits from cache-based batch-processing of certain VMCS fields (e.g., `VMREAD/VMWRITE`); while `DeHype-CN` indicates additional overhead by performing secure NPT updates (Section 3.2). As shown in the figure, the overall overhead introduced by DeHype is less than 6%. This overhead is inevitable since DeHype by design invokes more system calls than the original KVM.

5 Discussion

In this section, we re-examine our system design and implementation for possible improvements as well as explore new opportunities enabled by our approach. First, we assume an adversary model where attackers try to compromise the hypervisor from a guest VM. The privileged HypeLet and its host OS kernel are a part of the TCB. Although the total TCB (with the host OS kernel) may not be greatly reduced, our system still provides strong protection against malicious or compromised guests by securely confining the hypervisor in the user space. This is particularly true in the cloud environment where the highly constrained HypeLet is the main attack surface exposed to a guest VM. To improve the security level of our system, our prototype performs necessary sanity check on the new 10 system calls introduced by HypeLet to prevent bugs inside the user-level hypervisor from affecting the HypeLet (e.g., including explicit checks for NPT update – Section 2.2).

Second, our current prototype is still limited in pinning the guest memory. This limitation can be readily addressed by integrating the Linux MMU notifier [10]. Specifically, HypeLet registers a set of callback functions to the kernel's MMU notifier interface, which will notify HypeLet when important memory management events are about to happen. For example, when a "memory swapped out" event takes place, HypeLet will be notified and further reflect the event to the user-level hypervisor. The user-level hypervisor can decide whether to prevent (by marking the page as recently accessed in the `age_page` mmu notifier) or allow the page swapping according to whether the page is currently in-use or not, respectively. Other events can be similarly handled. By integrating the MMU notifier, we can avoid pinning the guest memory. Meanwhile, the performance of DeHype might be negatively affected slightly due to the overhead in managing these events.

Third, our current prototype is limited in not supporting all full-fledged KVM features. Notable ones are SMP and para-virtualized I/O (e.g., virtio [38]). To retrofit our prototype with their support, it is necessary to make a few adjustments that mainly involve additional engineering efforts. Specifically, to support SMP, HypeLet needs to be aware of the presence of multiple virtual CPUs in a guest so that it can schedule VCPUs to physical CPUs, and provide a mechanism (e.g., inter-processor interrupt [19]) for VCPUs to interrupt and synchronize with one another. The SMP support in the original KVM can be leveraged for this purpose and make the implementation likely straight-forward. To support para-virtualized I/O, we only need to migrate the `virtio` [38] virtual device in the original KVM from kernel space to user space. This will likely reduce the performance benefit of `virtio` because kernel functions used by `virtio` are not directly accessible and must be replaced by system calls. Still, para-virtualized I/O will perform better than emulated I/O (e.g., virtual Intel e1000 PCI network card in KVM) because it does not involve expensive I/O memory and I/O registers emulation.[6]

[6] We also point out that with the wide adoption of hardware virtualization, for obvious performance reasons [25], we choose our prototype in favor of hardware-assisted memory virtualization (i.e., NPT), instead of shadowing-based memory virtualization (i.e., SPT). However, we do not envision any technical challenges in supporting the software-based memory shadowing in our prototype.

From another perspective, the deprivileged hypervisor architecture as demonstrated in DeHype also introduces some unique capabilities or new opportunities. In particular, as the DeHyped KVM runs as a normal user-mode process, the system can be developed and debugged with the help of many existing tools that are familiar to most programmers. For example, we used GDB [18] to debug our prototype by setting breakpoints, inspecting variables, and executing the code in single-steps. We also used the dynamic instrument tool Valgrind [47] to detect possible memory leaks in KVM. This is a significant improvement over the kernel-level debugging, in which irregular control flow (e.g., interrupts, task switching, and asynchronous events) makes debugging highly challenging.

In addition, our architecture also naturally makes it feasible to run different versions of the KVM hypervisor (as user-level processes) on the same machine. This capability could be useful in several scenarios, for example, to balance performance and security: for virtual machines requiring higher level of security guarantee, we can use an instrumented hypervisor with dynamic information flow tracking [33] to detect attacks against the hypervisor. At the same time, we can use a normal hypervisor to manage other virtual machines for better performance. By enabling the suspend/resume support of KVM, virtual machines could be live-migrated between these hypervisors, making the performance-security trade-off dynamically configurable. We leave it as future work.

Moreover, our architecture can facilitate the design and implementation of a variety of virtualization-based security services (e.g., virtual machine introspection [20]). Some of these services might require modifications to the hypervisor code, which leads to concerns of increased TCB and new vulnerabilities. In DeHype, such changes will most likely be limited to the unprivileged user-mode hypervisor code. Vulnerabilities will still be confined in the process and mediated with the traditional system call interposition approaches (Section 2.1).

6 Related Work

Improving hypervisor security The first area of related work is recent systems that are developed to improve hypervisor security. For example, seL4 [23] is proposed to formally verify the absence of certain types of software vulnerabilities in a customized small hypervisor. Verve [56] mechanically verifies every instruction in the software stack so that the hypervisors running over it could also be verified to ensure type and memory safety. HyperSafe [49] instead admits the presence of exploitable software bugs in hypervisors but proposes solutions to protect the runtime (bare-metal) hypervisor integrity. Others re-architect the hypervisor design for a minimized TCB. Specifically, NOVA [43] implements a thin bare-metal hypervisor that moves the virtualization support to user level. Xoar [9] modifies the original Xen design by breaking the control VM into single-purpose *service VMs*. Xen disaggregation [30] decomposes Xen by moving the privileged domain builder into a minimal trusted compartment for trusted virtualization. Min-V [34] disables non-critical virtual devices by minimizing the codebase of the virtualization stack with the so-called delusional boot approach. By using formal verification, MinVisor [11] provides integrity guarantees. Notice that such efforts require a new design of bare-metal hypervisors. Their applicability and effectiveness remain to be demonstrated to protect hosted hypervisors (e.g., KVM) that run together with a commodity host OS.

From another perspective, NoHype [45] works in a controlled cloud setting by eliminating the bare-metal hypervisor after preparing the virtualization environment. Specifically, it strictly partitions the hardware resource among guest VMs so that there is no need for the guest VM to interact with the hypervisor during its execution. Due to the close interaction between a hosted hypervisor and the host OS, the NoHype approach cannot be applied for hosted hypervisor protection. In addition, DeHype transparently supports commodity OS kernels (e.g., Linux and Windows) while NoHype still requires minor modifications on the guest OS.

KVM-L4 [37] is a closely related system that enables a modified Linux kernel (i.e., L^{4}Linux with the KVM module loaded) to run in user mode over the customized L4/Fiasco microkernel. With that, in order for QEMU to interact with KVM, it has to go through the IPC mechanism implemented in the L4/Fiasco microkernel. In comparison, as KVM is largely demoted as a user-level library with DeHype, the interaction between QEMU and KVM is simply achieved with a user-mode function call – instead of expensive L4 IPC in KVM-L4. Also, DeHype naturally supports running multiple KVM instances on the same host while KVM-L4 requires starting a new L^{4}Linux to host another KVM instance on the same host.

HyperLock [51] is another closely related system that creates a separate address space in host OS kernel to confine the loadable KVM module execution. However, since it still executes in privileged mode, additional complex techniques still need to be proposed to prevent potential misuse of its privileged code (e.g., enforcing instruction alignment rules through the compiler). In comparison, by deprivileging the KVM execution to user mode, DeHype naturally leverages the user-kernel mode separation (or the process boundary) to protect the host system (or other unrelated guest VMs) from a compromised KVM.

User-mode Linux [12] is a system to run virtual Linux systems as applications of a normal Linux system. As such, the guest of UML is limited to the Linux while DeHype does not have such a limitation. On the other hand, UML can potentially be leveraged by DeHype for kernel function supports similar to SUD [7]. However, our prototype shows that a full-fledged Linux is not required as DeHype only relies on a small number of kernel functions that are simple to recreate in the user-space.

The Turtles project [4] enables nested virtualization support for KVM. Since the deprivileged hypervisor in our system to some extent *emulates* certain privileged instructions such as `VMREAD/VMWRITE` (Section 3.3), it has a similar role as an L_1 hypervisor. Therefore, our *VMCS caching* approach shares the idea of the *VMCS shadowing* they proposed. The mechanism of *Pseudo NPT* is also similar to the $EPT_{0 \rightarrow 2}$. However, the L_0 hypervisor in the Turtles project is a full-fledge hypervisor while HypeLet has a much smaller privileged code base which could be used to better secure the lowest level hypervisor.

Isolating untrusted device drivers The second area of related work includes systems that isolate device drivers from the host OS kernel. For example, Gateway [42], HUKO [53], and SIM [39] leverage a trustworthy hypervisor to isolate kernel device drivers or security monitors. Zhou et. al [58] builds a verifiable trusted path to ensure data transfers between devices and user programs with the leverage of a small hypervisor. In comparison, our goal here is to deprivilege the hosted hypervisor, which is assumed to be trusted in these systems. Inside the host OS kernel, Nooks [44] improves the OS reliability by isolating device drivers in the light-weight protection domain. By assuming the drivers to be faulty but not malicious, Nooks by design cannot handle malicious or compromised device drivers.

From another perspective, researchers also proposed solutions to isolate device drivers in user space. For example, L3 [26] enables user-level device drivers based on a microkernel architecture. SUD [7] executes existing drivers as untrusted user-level processes to prevent misbehaving drivers from crashing the rest of the system. MicroDrivers [16] splits drivers to a privileged kernel part and an unprivileged user part at the cost of increased performance overhead. RVM [52] executes device drivers with limited privilege in user space, where all the interactions between the driver and the device is constrained by the reference monitor built with a customized device safety specification.

When deprivileging the KVM execution, we share a similar motivation behind those efforts. However, a hosted hypervisor module is more than a traditional device driver and its deprivileged execution poses additional challenges. Particularly, a hosted hypervisor has a richer set of special privileged instructions to execute than a driver. As a result, the earlier approach such as the way IOMMU is being employed in SUD [7] may not be applicable to hypervisors. In addition, the host hypervisor differs from traditional device drivers with its unique host-guest world switching operations and the need for hardware-assisted memory virtualization. Their support requires new design and implementation considerations (Sections 2 and 3). Specifically, the VMCS caching and memory rebasing are unique in our DeHype system to allow for efficient deprivileged execution without sacrificing security.

Applying virtualization to host security The third area of related work is a long stream of research [8, 13, 14, 20, 21, 27, 28, 36, 41, 46, 48, 50, 54, 55, 57] that applies virtualization to address various host security issues. For example, Proxos [46] divides the existing system call interface between the untrusted commodity OS and a trusted private OS to protect security-sensitive data. Patagonix [27] can detect and identify covertly executing binaries in an OS-agnostic way by relying only on the hardware features and binary formats. Overshadow[8] protects the privacy and integrity of application data even if the OS is compromised by interposing transitions between the guest OS and an application to present a different view of application data to them. HookSafe [50] and Lares [36] protect kernel function pointers from being hijacked by rootkits. Lycosid [21] detects and identifies hidden processes using hypervisor support. Lockdown [48] partitions resources across time with a light-weight hypervisor to isolate the trusted and untrusted environments. Such systems all require a trusted hypervisor that is being addressed in this work as well as other systems in the first area of related work.

7 Conclusion

We have presented the design, implementation and evaluation of DeHype, a system to deprivilege hosted hypervisor execution to user mode. Specifically, by decoupling the hypervisor code from the host OS and deprivileging most of its execution, our system not only substantially reduces the attack surface for exploitation, but also brings additional benefits in allowing for better development and debugging as well as concurrent execution of multiple hypervisors in the same physical machine. We have implemented a DeHype prototype for the open source KVM hypervisor. The evaluation results show that our system successfully deprivileged 93.2% of the loadable KVM module code base to user mode while only adding a small TCB ($2.3K$ SLOC) to the host OS kernel. Additional experiments with a number of benchmark programs further demonstrate its practicality and efficiency.

Acknowledgments

We would like to thank our shepherd, Heng Yin, and the anonymous reviewers for their numerous, insightful comments that greatly helped improve the presentation of this paper. This work was supported in part by the US National Science Foundation (NSF) under Grants 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, September 2007.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4), April 2010.

[3] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, April 2005.

[4] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. HarEl, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010.

[5] Bonnie++. `http://www.coker.com.au/bonnie++/`.

[6] J. Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference - Volume 1*, June 1994.

[7] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, June 2010.

[8] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[9] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, October 2011.

[10] J. Corbet. Memory Management Notifiers. http://lwn.net/Articles/266320/.

[11] M. Dahlin, R. Johnson, R. Krug, M. McCoyd, S. Ray, and B. Young. Toward the Verification of a Simple Hypervisor. In *10th International Workshop on the ACL2 Theorem Prover and its Applications*, November 2011.

[12] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th annual Linux Showcase & Conference*, October 2000.

[13] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, October 2008.

[14] Ú. Erlingsson, T. Roeder, and T. Wobber. Virtual Environments for Unreliable Extensions. Technical Report MSR-TR-05-82, Microsoft Research, June 2005.

[15] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, November 2006.

[16] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The Design and Implementation of Microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[17] GLIBC. `http://www.gnu.org/software/libc/`.

[18] GDB: The GNU Project Debugger. `http://www.gnu.org/s/gdb/`.

[19] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, September 2010.

[20] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-based "Out-Of-the-Box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[21] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification using Lycosid. In *ACM International Conference on Virtual Execution Environments*, Seattle, Washington, March 2008.

[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium*, July 2007.

[23] 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 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[24] Cloudburst: A VMware Guest to Host Escape Story. `http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-SLIDES.pdf`.

[25] KVM. `http://www.linux-kvm.org/`.

[26] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two Years of Experience with a μ-Kernel Based OS. *Operating Systems Review*, 25(2), April 1991.

[27] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proceedings of the 17th Conference on Security Symposium*, July 2008.

[28] L. Litty and D. Lie. Patch Auditing in Infrastructure as a Service Clouds. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, March 2011.

[29] Y. Mao, H. Chen, D. Zhou, , X. Wang, N. Zeldovich, and M. F. Kaashoek. Software Fault Isolation with API Integrity and Multi-principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, October 2011.

[30] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security through Disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, March 2008.

[31] Virtunoid: Breaking out of KVM. `http://nelhage.com/talks/kvm-defcon-2011.pdf`.

[32] NetworkWorld. Red Hat's KVM Virtualization Proves Itself in IBM's Cloud. `http://www.networkworld.com/community/blog/red-hats-kvm-virtualization-proves-itself-ibm`.

[33] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[34] A. Nguyen, H. Raj, S. K. Rayanchu, S. Saroiu, and A. Wolman. Delusional Boot: Securing Cloud Hypervisors without Massive Re-engineering. In *Proceedings of the 7th ACM SIGOPS/EuroSys European Conference on Computer Systems*, April 2012.

[35] National Vulnerabilities Database. `http://nvd.nist.gov/`.

[36] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, May 2008.

[37] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases. In *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, April 2009.

[38] R. Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42(5), July 2008.

[39] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[40] SPEC CPU2006. `http://www.spec.org/cpu2006/`.

[41] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process Out-grafting: an Efficient "Out-of-VM" Approach for Fine-grained Process Execution Monitoring. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, October 2011.

[42] A. Srivastava and J. Giffin. Efficient Monitoring of Untrusted Kernel-Mode Execution. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, February 2011.

[43] U. Steinberg and B. Kauer. NOVA: a Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer systems*, April 2010.

[44] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[45] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, October 2011.

[46] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust between Applications and Operating Systems Configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, November 2006.

[47] Valgrind. `http://valgrind.org`.

[48] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig. Lockdown: Towards a Safe and Practical Architecture for Security Applications on Commodity Platforms. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST)*, June 2012.

[49] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, May 2010.

[50] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, October 2009.

[51] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM SIGOPS/EuroSys European Conference on Computer Systems*, April 2012.

[52] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device Driver Safety through a Reference Validation Mechanism. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, December 2008.

[53] X. Xiong, D. Tian, and P. Liu. Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, February 2011.

[54] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin. V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis. In *Proceedings of the Eighth Annual International Conference on Virtual Execution Environments*, March 2012.

[55] L.-K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[56] J. Yang and C. Hawblitzel. Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2010.

[57] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System Support for Derived Data Management. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, March 2010.

[58] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.