Isolating Commodity Hosted Hypervisors with HyperLock

Zhi Wang Chiachih Wu Michael Grace Xuxian Jiang

Department of Computer Science
North Carolina State University
{zhi_wang, cwu10, mcgrace}@ncsu.edu jiang@cs.ncsu.edu

Abstract

Hosted hypervisors (e.g., KVM) are being widely deployed. One key reason is that they can effectively take advantage of the mature features and broad user bases of commodity operating systems. However, they are not immune to exploitable software bugs. Particularly, due to the close integration with the host and the unique presence underneath guest virtual machines, a hosted hypervisor – if compromised – can also jeopardize the host system and completely take over all guests in the same physical machine.

In this paper, we present HyperLock, a systematic approach to strictly isolate privileged, but potentially vulnerable, hosted hypervisors from compromising the host OSs. Specifically, we provide a secure *hypervisor isolation runtime* with its own separated address space and a restricted instruction set for safe execution. In addition, we propose another technique, i.e., *hypervisor shadowing*, to efficiently create a separate shadow hypervisor and pair it with each guest so that a compromised hypervisor can affect *only* the paired guest, not others. We have built a proof-of-concept HyperLock prototype to confine the popular KVM hypervisor on Linux. Our results show that HyperLock has a much smaller (12%) trusted computing base (TCB) than the original KVM. Moreover, our system completely removes QEMU, the companion user program of KVM (with $> 531K$ SLOC), from the TCB. The security experiments and performance measurements also demonstrated the practicality and effectiveness of our approach.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Security kernels

General Terms Design, Security

Keywords Virtualization, Hypervisor, KVM, Isolation

1. Introduction

Recent years have witnessed the accelerated adoption of hosted or Type-II hypervisors (e.g., KVM [19]). Compared to bare-metal or Type-I hypervisors (e.g., Xen [5]) that run directly on the hardware, hosted hypervisors typically run within a conventional operating system (OS) and rely on this "host" OS to manage most system resources. By doing so, hosted hypervisors can immediately benefit from various key features of the host OS that are mature and stable, including abundant and timely hardware support, advanced memory management, efficient process scheduling, and so forth. Moreover, a hosted hypervisor extends the host OS non-intrusively as a loadable kernel module, which is arguably much easier to install and maintain than a bare-metal hypervisor. Due to these unique benefits, hosted hypervisors are increasingly being adopted in today's virtualization systems [28].

From another perspective, despite recent advances in hardware virtualization (such as Intel VT [18]), virtualizing a computer system is still a complex task. For example, a commodity hosted hypervisor, such as KVM, typically involves a convoluted shadow paging mechanism to virtualize the guest memory (including emulating five different modes of operation in x86: paging disabled, paging with 2, 3, or 4 levels of page tables, and hardware assisted memory virtualization, including EPT/NPT [18]). For performance reasons, many hypervisors also support an "*out-of-sync*" (OOS) shadow paging scheme that synchronizes the shadow page table with the guest only when absolutely necessary. In addition, hosted hypervisors still suffer from a large attack surface, because they take many untrusted guest virtual machine (VM) states as input. For example, shadow paging needs to read guest page tables for synchronization, while instruction emulation – another complicated component of a hypervisor – involves fetching guest instructions for interpretation and execution.

Due to inherent high complexity and broad attack surface, contemporary hosted hypervisors are not immune to serious security vulnerabilities. A recent study of the National Vulnerability Database (NVD) [25] indicates that there were 24 security vulnerabilities found in KVM and 49 in VMware Workstation over the last three years. These vulnerabilities

EuroSys'12, April 10–13, 2012, Bern, Switzerland.

can be potentially exploited to execute arbitrary code with the highest privilege, putting the whole host system at risk. In fact, successful attacks against both KVM [12] and VMware Workstation [21] have been publicly demonstrated to escape from a guest VM and directly attack the host OS. Worse, a compromised hypervisor can also easily take over all the other guests, leading to disruption of hosted services or stealing of sensitive information. This can have a devastating effect in the scenario where a single physical machine may host multiple VMs from different organizations (e.g., in a cloud setting). In light of the above threats, there is a pressing need to secure these hosted hypervisors and protect the host system – as well as the other guest VMs – from a compromised hypervisor.

To address the need, researchers have explored a number of approaches. For example, seL4 [20] takes a formal approach to verify that a small micro-kernel ($\sim$8.7K source lines of code or SLOC) is secure including the absence of certain software vulnerabilities (e.g., buffer overruns and NULL pointer references). However, it is not scalable or still incomplete in accommodating commodity hypervisors (e.g., KVM) that have a much larger code base and support various complex x86 CPU/chipset features [30]. HyperSafe [39] enables self-protection for bare-metal hypervisors to enforce their control flow integrity. Unfortunately, as admitted in [39], the proposed approach cannot be applied for hosted hypervisors due to different design choices in the commodity host OS (e.g. frequent page table updates) from bare-metal hypervisors. Other approaches [32, 42] take a layer-below method to isolate untrusted device drivers, which is also not applicable because hosted hypervisors already run at the lowest level on the system.

In this paper, we present HyperLock, a system that is able to establish a tight security boundary to isolate hosted hypervisors. Specifically, we encapsulate the execution of a hosted hypervisor with a secure *hypervisor isolation runtime*, which has its own separate address space and a reduced instruction set for safe execution. By doing so, the host system is not accessible to the hypervisor. Instead, it must go through a well-defined interface, which is mediated and sufficiently narrowed-down by HyperLock to block any unexpected side-effects. Moreover, we further propose a *hypervisor shadowing* technique, which can efficiently create a separate shadow hypervisor for (and pair it with) each guest so that a compromised hypervisor can affect *only* the paired guest, not others. By exploiting recent memory de-duplication techniques, these shadow hypervisors can be created without incurring additional resource overhead.

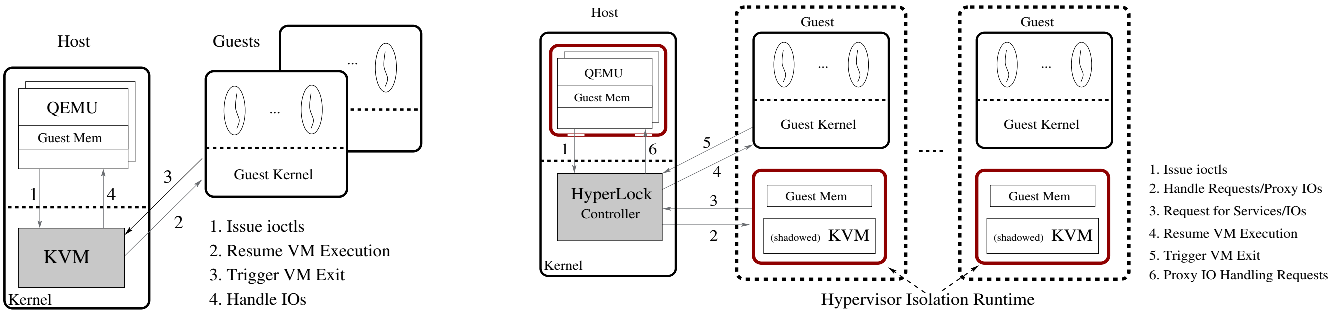
We have implemented a proof-of-concept HyperLock prototype for the popular KVM hypervisor (version kvm-2.6.36.1 and qemu-0.14.0). Our experience shows that HyperLock can be implemented with a small code base ($\sim$4K SLOC). We demonstrate its effectiveness and practicality by performing additional security analysis and performance measurement. To summarize, this paper makes the following contributions:

- To address the imperative need to confine commodity hosted hypervisors, we propose a secure *hypervisor isolation runtime* with a dedicated address space and a reduced instruction set to strictly confine their execution. To the best of our knowledge, the proposed hypervisor isolation runtime is among the first to isolate hosted hypervisors and protect the host OSs from being jeopardized by them.
- To effectively prevent a compromised hosted hypervisor from taking over all guests in the same physical machine, we propose another key technique, i.e., *hypervisor shadowing*, to create a guest-specific shadow hypervisor without additional resource overhead. By doing so, we ensure that a compromised hypervisor will only affect the corresponding guest, *not* others.
- We have developed a HyperLock prototype and used it to protect the popular KVM hypervisor. Our prototype introduces a very small TCB ($\sim$4K SLOC) to the current OS kernel while completely removing the original KVM code ($\sim$33.6K SLOC) as well as the companion QEMU program ($\sim$531K SLOC) from the TCB of the host OS. The security analysis and evaluation with standard benchmark programs shows that our prototype is not only effective, but also lightweight ($< 5\%$ performance slowdown).

The rest of the paper is structured as follows: we first present the design of HyperLock with a focus on the KVM hypervisor in Section 2. After that, we discuss the implementation and evaluation of HyperLock in Sections 3, and 4, respectively. Issues and possible improvements are discussed in Section 5. Finally, we describe related work in Section 6 and conclude the paper in Section 7.

2. Design

Before presenting our system design, we first briefly review the basic architecture of existing hosted hypervisors and the associated threat model. For simplicity, we use KVM as the representative example throughout the paper. As a popular, open-source hypervisor, KVM is incredibly simple to deploy and run. It can be dynamically loaded as a kernel module on Linux and once loaded, it instantly extends the host OS with virtualization support (based on hardware virtualization extensions such as Intel VT [18]). KVM uses a companion user program, i.e., a QEMU variant, to cooperatively emulate hardware devices for a guest (e.g., hard disks). In Figure 1(a), we show the main execution flow of a KVM-powered guest, which involves close interaction between KVM and QEMU. For example, when QEMU issues an `ioctl` command (e.g. `KVM_RUN` – arrow 1) to KVM, KVM proceeds by switching into the guest mode for the VM execution (arrow 2), which means the guest code



(a) Traditional KVM Execution Flow

(b) "HyperLocked" KVM Execution Flow

Figure 1. Traditional KVM Execution vs. "HyperLocked" KVM Execution

can run natively on the CPU. The guest mode continues until certain events (e.g., an I/O port access – arrow 3) happen to cause a *VM exit* back to KVM. Based on the VM exit reason, KVM may directly resolve it or delegate it to QEMU (arrow 4). After the VM exit is resolved, the guest can now enter the guest mode again for native VM execution (arrow 2).

In this work, we assume an adversary model where an attacker can successfully subvert the underlying hosted hypervisor from a malicious guest. To do that, the attacker can choose to exploit vulnerabilities either in the hypervisor itself (e.g., memory corruption in KVM) or in its companion user-level program (e.g., vulnerabilities in device emulation). As part of the exploitation process, the attacker may attempt to execute arbitrary malicious code in the compromised hypervisor or the user-level program [8]. In other words, by assuming the presence of exploitable software vulnerabilities in hosted hypervisors, we aim to deal with the threats from an untrusted guest so that the hosted hypervisor, even compromised, cannot take over the host and other guests.

Figure 1(b) illustrates the high-level architecture of HyperLock. To isolate hosted hypervisors, it has two key components, i.e., a *hypervisor isolation runtime* and a *hyperlock controller*. With these two components, unlike the traditional case of directly running KVM in the host OS (Figure 1(a)), HyperLock confines the KVM execution in a secure hypervisor isolation runtime with its own separate address space, where only unambiguous instructions from a reduced instruction set will be allowed to execute. Further, one isolation runtime is bound to one particular guest. That is, every guest logically has its own separate copy of the hypervisor code and data. Therefore, guests are completely isolated from each other. To confine KVM's user-level companion program, i.e., QEMU, HyperLock limits its system call interface and available resources with system call interposition. By doing so, HyperLock can mediate the runtime interaction between QEMU and KVM by acting as a proxy to forward commands from QEMU to KVM or relay requests from KVM on the opposite direction. The controller is also designed to provide runtime services to KVM as some tasks cannot be entrusted or delegated to KVM.

2.1 Hypervisor Isolation Runtime

Our first component – the hypervisor isolation runtime – is designed to safely isolate or confine the privileged hosted hypervisor so that even it is compromised, our host can still be protected. Isolation is achieved through two main mechanisms: *memory access control* and *instruction access control*. In the following, we describe each mechanism in detail.

2.1.1 Memory Access Control

To confine the privileged KVM module, we create a separate address space based on the CPU paging mechanism. Before permitting KVM to run, HyperLock switches to the KVM-specific address space by loading the CR3 register with the corresponding page table base address. The KVM-specific page table is maintained by HyperLock and cannot be changed by KVM because we map it read-only inside the KVM address space to facilitate the guest page table update (Section 3.3). We stress that it is critical to make the KVM page table unmodifiable to KVM. Otherwise, a compromised KVM can take advantage of it to access or modify the host OS memory and corrupt the whole system. In our design, we further enforce W⊕X [2] in the KVM address space. That is, there is no memory page in the isolated KVM address space that is simultaneously executable and writable. Also, as there is no legitimate need for KVM to execute any of the guest code, the whole guest memory is marked as non-executable in the KVM address space. In fact, within this address space, only the KVM module contains the executable code (one exception is the trampoline code we introduced to switch the address space from KVM back to the host – Section 2.1.2). By doing so, HyperLock guarantees that *no code inside the isolation runtime can alter its memory layout or change the memory protection settings*.

Within the isolation runtime, KVM can directly read from or write to the guest memory as usual. For performance reasons, it is important to allow KVM access to the guest memory. Specifically, if an I/O instruction executed by the guest is trapped and emulated by KVM, it requires several guest memory accesses for I/O emulation: It has to first traverse the guest page table to convert the guest virtual address to its corresponding guest physical address (which can further be converted to a host address usable by KVM); After that, KVM can then read or write the guest memory again to actually emulate the I/O operation. Because physical memory for the guest is linearly mapped inside the isolation runtime (starting at address zero), guest physical addresses can be directly used by KVM to access guest memory without further conversion.

There also exists a trampoline code inside the isolation runtime to switch the context back to the host, which will be discussed in the second isolation mechanism (Section 2.1.2).

2.1.2 Instruction Access Control

In addition to a separate address space for the hosted KVM hypervisor, we further restrict the instructions that will be allowed to execute within it. This is possible because KVM does not contain any dynamic code, and our memory access control removes the possibility of introducing any new additional code in the isolation runtime. However, challenges arise from the need of executing privileged instructions (of hardware virtualization extension) in the KVM module. For example, KVM needs to execute `VMWRITE`, a privileged instruction that updates the VM control structure (or VMCS [18]). Though we could potentially replace these `VMWRITE` instructions with functions to enlist help from HyperLock, the performance overhand could be prohibitively high due to context switching between the isolation runtime and the host kernel.

Existing hardware does not allow granting privileges to individual instructions. Therefore, while still running the KVM code at the highest privilege, there is a need to prevent this privilege from being misused. Specifically, our instruction access control scheme guarantees that *no privileged instructions other than explicitly-allowed ones can be executed within the isolation runtime*. In our prototype, we permit only two privileged instructions, i.e., `VMREAD` and `VMWRITE`, for direct execution while re-writing other privileged instructions to rely on the trusted supporting routines in HyperLock (Section 3.3). Note that these two permitted privileged instructions could be executed frequently (e.g., tens of times per VM exit) and it is thus critical to execute them directly to avoid unnecessary context-switching overhead. Moreover, to avoid the highest privilege from being abused, we need to prune the KVM instructions to remove any other "unexpected" privileged ones. Specifically, due to x86's variable length instruction set, it is still possible to uncover "new" or unintended instructions, including privileged ones (e.g., by fetching or interpreting the same memory stream from different offsets [8]). To remove these unintended instructions, we enforce the same instruction alignment rules as in PittSFIeld [23] and Native Client (NaCl) [43] to allow for unambiguous, reliable disassembly of KVM instructions. Specifically, in our prototype, the KVM code is organized and instrumented into equal-length fragments (32 bytes). As a result, no instruction can overlap the fragment boundary. Also, computed (or indirect) control transfers are instrumented so that they can only transfer to fragment boundaries. These two properties ensure that all the instructions that are executable inside the isolation runtime are known at compile time [43]. With that, we can then scan the instrumented code to verify that KVM can only contain the two explicitly-allowed privileged instructions, and not any other privileged ones.

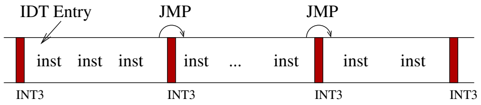


Figure 2. Trampoline Code Layout: Each code fragment starts with a one-byte `INT3` and ends with a short `JMP`, which skips over the next `INT3` of the following code fragment. The starting address of the trampoline code is loaded into an IDT entry as the interrupt handler.

In addition to effectively restricting the instructions allowed to execute within the isolation runtime, our scheme also provides a way to safely return back to the host kernel. This is needed as KVM is now strictly confined in its own address space and will enlist HyperLock for the tasks that cannot be delegated to itself. To achieve that, we design a trampoline that will safely load the `CR3` register with the host kernel page table base address. For isolation purposes, the trampoline code also needs to switch a number of critical machine registers, including x86 segment descriptor table (`GDT`/`LDT`), interrupt descriptor table (`IDT`), and task state segment (`TSS`) [18], which means that a number of critical state registers need to be accessible to KVM. Fortunately, from the trampoline's perspective, these registers are static across the context switches. Therefore, we simply collect them in a separate memory page, mark it read-only to KVM, and make it available to the trampoline code. In other words, as long as we stay inside the isolation runtime, this critical state becomes write-protected.

Because critical hardware state is updated by the trampoline code, we take a step further by ensuring the atomicity of its execution, thus preventing the partial loading of hardware state. Specifically, we ensure that the trampoline code can be entered only from a single entry point inside the isolation runtime, and its execution cannot be interrupted. In our prototype, KVM has to issue a software interrupt (using the `INT` instruction) to execute the trampoline code and exit the isolation runtime. Hardware interrupts are automatically disabled by the hardware to run the interrupt handler (i.e., the trampoline code) and will not be enabled until it has returned to the host. The handler for this software interrupt is the entry point to the trampoline code. To further make sure it is the only entry point, we need to foil any attempts that jump to the middle of the trampoline code. In our design, we arrange the trampoline akin to the service runtime call in NaCl for this purpose (Figure 2). Specifically, we put a single byte `INT3` instruction at the beginning of each code fragment in the trampoline. If executed, this `INT3` instruction will immediately cause a debug exception. As mentioned earlier, HyperLock enforces instruction alignment rules for any code running inside KVM. This guarantees that indirect control flow transfers (that may be controlled by the attacker) can only jump to code fragment boundaries. By putting an `INT3` instruction at these locations, HyperLock can immediately catch any attempts to subvert the trampoline code (since there is no legitimate code in the original KVM to call the trampoline). On the other hand, legitimate invocation of the trampoline code will not be interrupted by `INT3` because a short jump is placed at the end of each code fragment to skip over them. As such, we can effectively ensure a single entry to the trampoline code and the atomicity of its execution.

2.2 HyperLock Controller

Our second component is designed to accomplish three tasks complementary to the first one. Specifically, the first task is to achieve complete guest isolation by duplicating a KVM hypervisor (running inside an isolation runtime) for each guest. Traditionally, a compromised KVM hypervisor immediately brings down all the running guests. By duplicating the hypervisor for each guest and blocking inter-hypervisor communication, we can ensure that a compromised KVM can only take over one guest, not all of them. However, instead of simply duplicating all the hypervisor code and data, which unnecessarily increases memory footprint of our system, we propose a hypervisor shadowing technique by assigning each guest a shadow copy. The shadow copy is virtually duplicated to segregate the hypervisor instances; there is only a single physical copy. This is possible because all the shadow copies share identical (static) hypervisor code, which means we can apply classic copy-on-write or recent memory de-duplication techniques [1] to maintain a single physical copy, thus avoiding additional memory consumption overhead. Each shadow copy still runs within a hypervisor isolation runtime and can legitimately access the memory space of one and only one guest within its own address space.

The second task is to act as a proxy connecting QEMU and the isolated KVM. On one hand (arrow 2 in Figure 1(b)), it accepts `ioctl` commands from QEMU (e.g. `CREATE_VM`, `CREATE_VCPU`, and `KVM_RUN`) and passes them to KVM via remote procedure calls (RPCs). Our system maintains the same `ioctl` interface and thus supports the same companion QEMU program without any modification. On the other hand (arrow 3 in Figure 1(b)), HyperLock provides runtime services for tasks that either require interaction with the host OS (e.g. to allocate memory for the guest), or that cannot be safely entrusted to KVM (e.g. to update shadow page tables). Because HyperLock relies on the host OS to implement these runtime services, it is critical to understand the possible impact should KVM be compromised or these services be misused. To proactively mitigate these consequences, our prototype defines a narrow interface that exposes only five well-defined services, which are sufficient to support commodity OSs (including both Linux and Windows XP) as VMs. These five services include (1) `map_gfn_to_pfn` to convert a guest physical page number (`gfn`) to the physical page number (`pfn`) of its backing memory, (2) `update_spt/npt` to batch-update the shadow page table (`spt`) or the nested page table (`npt`), (3) `read_msr` to read x86 machine-specific registers (MSRs), (4) `write_msr` to write MSRs, and (5) `enter_guest` to switch the guest execution into guest mode. In our prototype, we scrutinize possible arguments to these services and block any unexpected values. Furthermore, resources allocated by HyperLock on behalf of each guest will be accounted to that guest to foil any attempts to deplete or misuse resources.

The third task is to reduce the exposed system call interface to the user-level companion program, i.e., QEMU. Specifically, we manually obtain the list of system calls that will be used in QEMU and then define a stand-alone system call table for it. This system call table is populated with only those allowed entries to prevent QEMU from being abused. In addition, we also limit the allowed parameters for each system call and deny anomalous ones. As this technique has been well studied [16, 27], we omit the details here.

3. Implementation

We have implemented a HyperLock prototype to isolate the KVM hypervisor (version 2.6.36.1 with $\sim 33.6K$ SLOC) and QEMU (version 0.14.0 with $> 531K$ SLOC). Our prototype runs on Linux/x86 and has $\sim 4.1K$ SLOC. Specifically, our prototype contains 862 lines of C code for the hypervisor isolation runtime and 270 lines of assembly code for the trampoline that manages the context switches between the host and KVM. The five runtime services take 569 SLOC. The remaining code ($\sim 2.3K$ SLOC) is primarily helper routines to manage the host state, confine QEMU, and support its interaction with KVM. Our current prototype is implemented and evaluated based on a Dell machine (with an `Intel Core i7 920` CPU and 3GB memory) running `Ubuntu 10.04 LTS` and a Linux `2.6.32.31` kernel. In the rest of this section, we present details about our prototype based on the Intel VT [18] hardware virtualization extension. Note that our prototype is implemented on the 32-bit x86 architecture. As we will explain in the paper, new features of the 64-bit x86 architecture (e.g., the interrupt stack table) actually make the implementation less challenging than on the 32-bit architecture.

3.1 Memory Access Control

HyperLock confines the KVM memory access by creating a separate paging-based address space. Within this address space, there are three components: KVM itself, guest memory, and the trampoline code for host and KVM context switches. Among these three components, the memory layout for KVM and our trampoline code do not change after initialization, while the page table entries (PTEs) for guest memory have to be updated on demand (because the guest memory layout and mapping might be changed frequently when the guest is running). To set up these PTEs, KVM needs to notify HyperLock (via the `map_gfn_to_pfn` service), which then checks whether a page of memory can be successfully allocated for the guest. If it can, HyperLock fills in the corresponding PTE. Otherwise, it returns failure back to KVM. Notice that from the host OS's perspective, the guest is just a normal process, i.e., the QEMU process. Therefore, the guest memory may be swapped out or in by the host OS when under certain memory pressure. To accommodate that, HyperLock needs to synchronize the KVM page table when such events happen. In our prototype, we register an MMU notifier [11] with the host kernel in order to receive notifications of these events. Upon every notification, our prototype will update the affected page table entries in the notification handler and further forward these events to KVM so that KVM can synchronize the SPT/NPT for the guest.

HyperLock's paging based memory access control is relatively straightforward to implement. However, there is one subtlety related to TLB (translation lookaside buffer) in the x86 paging mechanism. To illustrate, TLB is known as a fast cache of virtual to physical address mapping; if a mapping is already cached in TLB, CPU directly returns the mapping without bothering to traverse page tables again to translate it. Also, reloading the `CR3` register flushes all TLB entries *except* those for global pages [18], which are being used by Linux to retain TLB entries for kernel memory during the task switching. However, global pages could lead to serious security vulnerabilities in HyperLock. More specifically, because of these global pages, memory mappings for the host kernel will remain in the TLB cache even after switching to the isolation runtime for KVM, which means KVM can exploit the stale cache to access the mapped host OS kernel memory or instructions. As such, the host kernel's memory would be exposed to the untrusted KVM code. In our prototype, we had to disable the global page support in CPU by clearing the `PGE` bit in the `CR4` register before entering the isolation runtime for KVM execution. By doing so, we can ensure that CPU flushes all TLB entries when switching to the KVM address space, thus making host kernel pages inaccessible to KVM. Although disabling the global page support leads to more frequent TLB reloading for kernel memory, the Linux kernel's use of large pages ($2MB$) for kernel memory relieves some performance overhead. In our prototype, we also considered using the VPID (virtual-processor identifier) feature [18] of Intel-VT. However, we found that the feature cannot be used to avoid disabling global page support, because it is always set to zero in the non-guest mode.

3.2 Instruction Access Control

In addition to memory access control, HyperLock also confines the available instructions inside the isolation runtime. Specifically, we first enforce instruction alignment [23, 43] on the KVM and our trampoline code by compiling them through the Native Client (NaCl) compiler, a customized gcc compiler developed by Google. With the help of instruction alignment, we can then reliably disassemble available code inside the isolation runtime with the assurance of no unintended instructions. As mentioned earlier, our prototype blocks all privileged instructions that can be executed inside the isolation runtime, except `VMREAD` and `VMWRITE` for performance reasons. To remove disallowed instructions, we further create a small script to scan the (reliably-disassembled) instructions of KVM and replace every privileged instruction (except `VMREAD` and `VMWRITE`) with a call to the corresponding runtime service.

Based on Intel VT, each guest is associated with a VMCS memory page that contains 148 fields to control the behavior of both the host and the guest. These fields can be roughly divided into four categories: *host state*, *VM execution control*, *guest state*, and *VM exit info*. Generally speaking, the first two categories need to be handled by trusted code because they can critically affect the host behavior. For example, `HOST_RIP` specifies the instruction CPU will return to after a VM exit; and `EPT_POINTER` stores the address of EPT/NPT table for the guest. In our prototype, we directly handle them outside the isolation runtime. Our development experience indicates that KVM handles these VMCS fields in a rather simple way: Most of these fields involve just loading the host state directly into its corresponding VMCS field, and will never change after the initial setup. Fields belong to the latter two categories can be safely delegated to untrusted KVM since they reflect the guest VM's state. For example, `VM_EXIT_REASON` gives the reason that caused the VM exit; and `GUEST_CS_SELECTOR` contains the current CS segment selector for the guest. Unlike host state and VM execution control fields, these fields are frequently retrieved and updated by KVM during each VM exit. For performance reasons, we would like to grant KVM direct access to guest state and VM exit information while preventing it from touching any other fields relating to host state or VM execution control. That is also the reason why our prototype makes an exception for the `VMREAD` and `VMWRITE` instructions.

To avoid these two instructions from being misused, our prototype takes the following precautions: First, we prevent KVM from directly accessing VMCS memory. Specifically, Intel VT requires that *physical address* of VMCS must

```
movl      $0xc00195d7,     %eax
movl      $GUEST_EIP,      %edx
vmwrite   %eax,            %edx
```

Figure 3. A VMWRITE macro-instruction that writes 0xc00195d7 into the GUEST_EIP VMCS field.

be loaded to CPU before software can access its fields with VMREAD and VMWRITE. However, nothing prevents attackers from directly overwriting its fields if the VMCS is virtually mapped in the KVM address space. As such, HyperLock allocates VMCS for the guest outside of the isolation runtime, and loads its physical address to CPU before entering the KVM. The VMCS structure itself is not mapped inside the KVM address space, therefore attackers cannot manipulate its fields by directly changing the VMCS. Meanwhile, the CPU has no problem executing the VMREAD and VMWRITE instructions because it uses a *physical address* to access the VMCS. Second, both the VMREAD and VMWRITE instructions take a VMCS field index as a parameter. Our prototype ensures that only fields belonging to guest state and VM exit information can be passed to them. More specifically, we define two macro-instructions (similar to nacljmp in NaCl [43]) for VMREAD and VMWRITE as shown in Figure 3. Each macro-instruction first fetches the hard-coded field index from KVM's code section (which is read-only, because it is protected by $W \oplus X$) into a register, then passes the register directly to VMREAD or VMWRITE. Further, we verify that the macro-instructions (each 17 bytes long) cannot overlap a fragment boundary (32 bytes) to block attackers from jumping into the middle of macro-instructions. There is a subtlety here: if an attacker is able to interrupt the CPU right before a VMWRITE, he might maliciously modify the register content saved by the interrupt handler. When the interrupt handler returns, registers are restored and the malicious field index gets used by VMWRITE. HyperLock avoids this problem because the trampoline handles the interrupt context, so it is not accessible to KVM. Finally, our script to scan KVM's assembly code also makes sure that only fields pertaining to guest state and VM exit information can be passed to these two macro-instructions.

The trampoline code for host and KVM context switches is also worth mentioning. To prevent KVM from monopolizing the CPU and to ensure a timely response to hardware interrupts, we need to enable interrupt delivery while KVM is running. Specifically, the trampoline code contains a handler for each exception or interrupt. The handler for a hardware interrupt first switches to the host OS and redirects control to the host OS's corresponding interrupt handler (defined in the host IDT table). Execution of the KVM will resume after host interrupt handler returns. The handler for an exception, which is caused by error conditions in the KVM, instead switches to the host OS and then immediately terminates the VM after dumping the KVM's state for auditing and debugging purposes. Under normal conditions, KVM should never cause exceptions, in particular, page faults: updates to the guest memory mapping by the host OS (e.g., paging out a block of memory) are synchronized to KVM through an MMU notifier [11]. When the need arises for KVM to access guest memory, it proactively calls the map_gfn_to_pfn service to read in the page, thus avoiding page faults.

At first glance, enabling interrupt delivery while KVM runs may only require setting up the IDT (Interrupt Descriptor Table). However, one quirk of the x86 architecture makes this more complicated than it should be: when an interrupt happens, the CPU will save the current state (such as EIP, ESP, and EFLAGS) to the stack so that it can resume the execution of the interrupted task. Because the sandbox is running at the highest privilege (ring 0), this state is saved to the current stack, which could then be manipulated by the attacker to launch a denial of service attack on the *32-bit* x86 platform. For example, a double fault will be triggered if the attacker manages to set the stack pointer ESP to an invalid (unmapped or write-protected) memory address and then write to the stack. The first write to the stack will cause a page fault. To handle the page fault, CPU tries to push more content to the *invalid* stack, which will lead to a second page fault. This time, CPU throws a double fault instead of more page faults. However, the stack pointer remains invalid for the double fault handler. Eventually, the CPU can only be recovered by power-cycling the machine. An astute reader may point out that we can switch to an interrupt task (thus a known-good stack) through task gate for an interrupt handler, and it has been used by Linux to handle double faults. Unfortunately, task gate cannot be securely deployed inside the isolation runtime where untrusted code runs. Specifically, to switch to an interrupt task, CPU uses a data structure called the TSS (task state segment) descriptor that specifies where to load CPU state information from, including CR3, EIP and ESP. Therefore, it is critical to write-protect the TSS descriptor. However, this structure cannot be write-protected in this case because CPU needs to change the descriptor's B (busy) bit *from zero to one* before switching to the interrupt task. Write-protecting the TSS descriptor will lead to another undesirable situation where the CPU can only be recovered by a hardware reset.

To accommodate that in 32-bit x86 architecture, our system always keeps a valid ESP to foil such attacks. Specifically, we allocate three continuous memory pages ($12KB$ total) at a *fixed* location in the KVM address space: the middle page is used for the stack itself (the same size as the stack in recent Linux kernels), while the top and bottom pages are used as overflow space. At the compiling time, we instrument the instructions that change ESP to maintain the stack location invariant by replacing the page number part (top 20 bits) of ESP to that of the pre-allocated stack (the middle page). This can be implemented with an AND

Runtime Service	SLOC
`map_gfn_to_pfn`	84
`update_spt/npt`	251
`read_msr`, `write_msr`	156
`enter_guest`	78

Table 1. The breakdown of HyperLock's runtime service implementation. `read_msr` and `write_msr` are implemented together. `enter_guest` includes 53 lines of inline assembly code and 25 lines of `C` code.

instruction (6 bytes) and an `OR` instruction (6 bytes). Both of them take a 4-byte constant and `ESP` as operands. As such, no scratch registers are required for this instrumentation. Moreover, to prevent the check from being circumvented, the check and its related instruction are kept in the same fragment (A similar inline software guard to prevent stack overflow was also explored in XFI [13].) HyperLock support for 64-bit x86 architecture will not suffer from the same issue because the CPU can be programmed to always switch to a known-good stack for interrupt handling with the help of a new feature called IST (interrupt stack table) [18].

3.3 Others

To properly isolate the KVM hypervisor, HyperLock also exposes a narrow interface to five well-defined runtime services to KVM. All these five services were implemented in a small number of lines of source code (Table 1). Among them, `update_spt/npt` is the most involved as it is directly related to the memory virtualization in KVM. For a concrete example, we use hardware assisted memory virtualization (`NPT`) to describe how `update_spt/npt` is implemented.

With `NPT` support, the CPU uses two page tables to translate a guest virtual address to the corresponding physical address: a guest page table (GPT) to convert a guest virtual address to a guest physical address, and a nested page table (NPT) to further convert a guest physical address to the (actual) physical address. The guest kernel has full control over GPT, while KVM is responsible for maintaining the NPT. Since NPT maps physical memory into the guest, only the trusted NPT should be loaded to CPU for guest address translation. In HyperLock, KVM still maintains its own NPT for the guest. However, this NPT is not used for address translation. Instead, HyperLock creates a mirror of the NPT outside the hypervisor isolation runtime to translate guest addresses. The NPT table and its mirror are synchronized via `update_npt` calls. In `update_npt`, HyperLock ensures that *only physical pages belonging to this guest will be mapped to the guest.* Noticing that the guest memory is mapped in the hypervisor isolation runtime with the (HyperLock-maintained) KVM page table, this guarantee can be efficiently achieved using the KVM page table. More specifically, KVM provides a guest physical page number and its memory protection attributes as parameters to `update_npt`. In this function, HyperLock traverses the KVM page table to find the physical page for this guest physical page and combine it with the memory protection attributes to update the corresponding page table entry in the NPT mirror. Moreover, the KVM page table is made available to KVM (by mapping it read-only in the KVM address space) so that KVM can use it to maintain its own NPT in the same way.

Overall, to isolate KVM with the proposed hypervisor isolation runtime, our prototype re-organizes KVM in a slightly different way. However, our modification to KVM is minor and focuses on three areas. First, the original `ioctl` based communication interface between KVM and QEMU is replaced by our RPC calls through HyperLock, which results in changing six `ioctl` functions in KVM (e.g., `kvm_dev_ioctl`, `kvm_vm_ioctl`, etc.). Second, we replace certain dangerous KVM calls with RPCs to runtime services, which results in changing eight functions in KVM. As an example, the original KVM's function (`__set_spte`) that writes directly to shadow page table entries is replaced by the `update_spt` service. Third, we also need to reduce one file, i.e., `vmx.c`, to avoid changing host state and VM execution control fields of VMCS in KVM. Instead, functions that access these two categories are moved to HyperLock while the rest stays the same. Our experience shows these changes (1) are mainly one-time effort as they essentially abstract the underlying interaction with hardware, which remains stable over the time, and (2) do not involve the bulk KVM code, which could undergo significant changes in future releases.

4. Evaluation

In this section, we present our evaluation results by first analyzing the security guarantees provided by HyperLock. After that, we report the performance overhead with several standard benchmarks.

4.1 Security Analysis

Based on our threat model (Section 2), an attacker starts from a compromised guest and aims to escape from HyperLock's isolation and further take over the host OS or control other guests (by exploiting vulnerabilities in KVM or QEMU). In HyperLock, we create a separate system call table for the QEMU process to constrain system calls available to it and validate their parameters. The security guarantee provided by such a system call interposition mechanism has been well studied [16, 27, 40]. In the following, we focus our analysis on the threats from a (compromised) KVM hypervisor when it aims to break out of HyperLock confinement.

Breaking Memory Access Control The first set of attacks aims to subvert memory protection in the isolation runtime to inject malicious code or modify important data structures, especially (read-only) control data in the trampoline. Since $W \oplus X$ is enforced in the isolation runtime, any attempt to directly overwrite their memory will immediately trigger a page fault and further cause the guest to be terminated by HyperLock. Having failed direct memory manipulation, the attacker may try to disable $W \oplus X$ protection by altering the KVM page table. Since the KVM page table is not directly changeable in the isolation runtime, the attacker has to leverage the trusted HyperLock code to manipulate the KVM page table. Fortunately, HyperLock will never change memory attributes for the (static) trampoline and KVM after initial setup, and the whole guest memory is marked as non-executable. Another possibility is for the attacker to trick HyperLock to map the host or HyperLock memory (e.g. host page table) into the isolation runtime or a malicious guest as (writable) guest memory. Sanity checks in the `map_gfn_to_pfn` and `update_spt/npt` service would prevent this from happening.

Subverting Instruction Access Control Since the hypervisor isolation runtime has the highest privilege, it is critical to prevent attackers from executing arbitrary privileged instructions. With the protection of $W \oplus X$ and instruction alignment, the attackers cannot inject code or uncover "new" instructions based on legitimate ones. Instead, they would have to target existing legitimate privileged instructions in the isolation runtime, for example, to maliciously modify host state or VM execution control fields in the VMCS by exploiting the field index parameter of the `VMWRITE` instruction (Figure 3). Notice that hard coding the field index (`GUEST_EIP` in Figure 3) and instruction alignment alone can *not* prevent misuse of the `VMWRITE` instruction. This is because that the second (fetching the field index into a register) and third instructions (executing `VMWRITE`) can be separated by an instruction fragment boundary. In other words, `VMWRITE` is the first instruction in an instruction fragment. With the capability to jump to any instruction fragment boundary under the instruction alignment rule, the attacker can directly jump to the `VMWRITE` instruction after loading the `edx` register with a malicious field index. This attack is prevented in HyperLock by ensuring that the three-instruction sequence (17 bytes) in Figure 3 cannot overlap any fragment boundary, thus ensuring `VMWRITE` and `VMREAD` always receive the fixed known-good field index parameters.

Another source of legitimate privileged instruction is the trampoline code (to load `CR3` etc). Similar to the `VMWRITE` instruction, it is necessary for HyperLock to prevent the attacker from jumping to the middle of the trampoline code. Unfortunately, the trampoline code (about $4K$ bytes) cannot fit in a single instruction fragment. In HyperLock, we set up a one-byte `INT3` instruction at each fragment boundary of the trampoline code to capture direct jumps to the middle of the trampoline. Moreover, the execution of the trampoline cannot be disrupted by KVM because interrupts are disabled by the hardware and will not be enabled until the trampoline has safely returned to the host.

The attacker may also try to perform a denial of service attack by corrupting the interrupt stack (Section 3.2). Such attempts will be foiled by the runtime check before the `ESP`-changing instructions. Similar to the `VMWRITE` instruction, the runtime check and its following instruction that modifies `ESP` cannot overlap the fragment boundary. Therefore, the runtime check cannot be bypassed by the attacker.

Name	Version	Configuration
Bonnie++	$1.03e$	`bonnie++ -f`
Kernel (61MB) build	2.6.32.39	`make defconfig;make`
SPEC CPU 2006	1.0.1	`reportable int`
Ubuntu desktop	10.04.2 LTS	Linux-2.6.32.31
Ubuntu server	10.04.2 LTS	Linux-2.6.32.28

Table 2. Software Packages used in Our Evaluation

Misusing HyperLock Services Another set of targets for the attacker is the five services provided to KVM by HyperLock. Since they can directly access the host OS, we have sanity checks in place to prevent these services from being misused. For example, we validate that only guest memory can be mapped by the `update_spt/npt` services, and `map_gfn_to_pfn` can never allocate more memory than that specified by the user when starting the VM. Moreover, as shown in Table 1, these services have a small code base (569 SLOC) and therefore can be thoroughly reviewed and verified to remove vulnerabilities.

Case Studies To better understand the protection provided by HyperLock, we examine several real-world vulnerabilities from NVD [25] and show how HyperLock could mitigate these threats. The first vulnerability we examined is CVE-2010-3881, a kernel-level bug in KVM in which data structures are copied to user space with padding and reserved fields uninitialized. This bug could potentially lead to leaking of sensitive content on the kernel stack. Under HyperLock, the host OS is not directly accessible to KVM and each guest is paired with its own KVM instance. Therefore, only data related to the guest itself could be leaked to it. The second vulnerability we examined is CVE-2010-0435, in which a malicious guest can crash the host by causing a NULL pointer dereference in KVM's x86 emulator. Under HyperLock, this vulnerability could be similarly exploited by the guest and trigger an exception. However, instead of crashing the host, HyperLock would terminate only the KVM instance paired with this guest (Section 3.2). The last vulnerability we examined is CVE-2011-4127, a bug related to device emulation in QEMU in which a guest can gain access to the data of other guests that reside on the same physical device due to insufficient checks of the SCSI ioctl commands. Under HyperLock, this vulnerability could be mitigated by system call introspection on QEMU.

4.2 Performance Evaluation

To evaluate the performance overhead caused by HyperLock, we test the guest performance with several standard benchmarks, including SPEC CPU 2006 [33], Bonnie++ (a

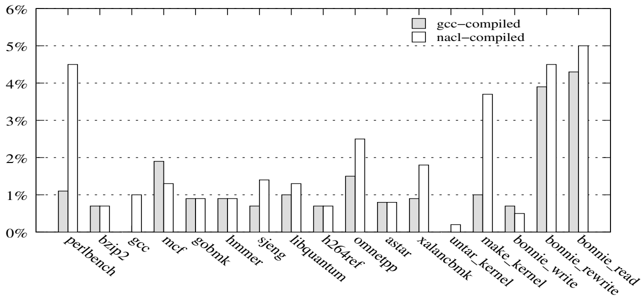


Figure 4. Normalized Overhead of HyperLock

file system performance benchmark) [9], and two application benchmarks (Linux kernel decompression and compilation). Our test platform is based on the Dell XPS studio desktop with a 2.67GHz Intel Core^{TM} i7 CPU and 3GB memory. The host runs a default installation of Ubuntu 10.04 LTS desktop with the $2.6.32.31$ kernel. The guest is based on the standard Ubuntu 10.04 LTS server edition. Table 2 lists the software packages and configurations used in our experiments. Among these benchmarks, SPEC CPU 2006, kernel decompression are CPU intensive tasks, while the other two tests (kernel compilation and Bonnie++) are more intensive in I/O accesses. For the kernel decompression and compilation test, we run the tests in the guest with the `time` command, and reported the sum of system and user time. As a result, these two experiments are based on the virtual time and due to the difficulty of keeping exact time in the guest OS [37], they could be less accurate. However, we did not observe clock drift during our experiments.

In our evaluation, we repeated the experiments with three different KVM configurations: the vanilla KVM, the gcc-compiled KVM under HyperLock, and the NaCl-compiled KVM under HyperLock. Comparing the performance of gcc- and NaCl-compiled KVM allows us to separate the effects of memory access control (switching between address spaces) and instruction access control (instruction alignment). As shown in Figure 4, the overall performance overhead introduced by HyperLock is less than 5%. Moreover, the gcc-compiled system has better performance than the NaCl-compiled system in most tests except mcf and Bonnie++ write. Generally speaking, instruction alignment will increase the binary size and reduce the performance because of additional code (mostly `NOP`s) inserted to align instructions. For example, the NaCl-compiled KVM in HyperLock contains $99,695$ instructions, 265% more than the gcc-compiled version's $37,553$ instructions. Also, 91.2% ($56,713$ out of $62,142$) of the added instructions are `NOP`s. Meanwhile, due to complex interaction of instruction alignment, instruction cache and TLB, the NaCl-compiled system may actually perform better than the gcc-compiled system [43].

To better understand the performance of HyperLock, we measured the latency of context switching between the host OS and hypervisor isolation runtime in HyperLock. Specifically, we created a null RPC in the isolation runtime that did nothing but returned directly back to the host, then called this function $1,000,000$ times from the host OS and calculated the average latency of round trip to the isolation runtime. Our results show that each round trip to the isolation runtime costs about 953 `ns`, or 45% of that to the guest mode ($2,115$ `ns`). The performance overhead of HyperLock is directly related to the frequency of context switches to the isolation runtime, which is further determined by the frequency of VM exits as illustrated by Figure 1 (at runtime, arrows 3, 4, 5, and 2 form the most active execution path). Advances in both hardware virtualization support (e.g., EPT [18]) and hypervisor software (e.g., para-virtualized devices [29]) have significantly reduced the number of necessary VM exits. For example, the average number of VM exits for the kernel compilation benchmark is $4,913$ per second. Also, the latency of address space switch in 64-bit x86 architecture could be significantly reduced by using a new CPU feature called process-context identifier (PCID) [18]. When PCID is enabled, the CPU tags each TLB entry with the current process-context id, thus rendering it unnecessary to flush all the TLB entries during an address space switch. However, we must still disable global page support (Section 3.1) because TLB entries for global pages are shared by all the address spaces even though PCID is enabled.

5. Discussion

In this section, we re-visit our system design and explore possible alternatives for either enhancement or justification. First, HyperLock confines KVM with its own paging-based memory space, *not* segmentation. Though segmentation could potentially provide another viable choice (especially in 32-bit x86 architecture), in HyperLock, we are in favor of paging for two reasons. (1) The 64-bit x86 architecture does not fully support segmentation [18]. Using paging can make HyperLock compatible with both the 32-bit and 64-bit x86 architectures. (2) Paging provides more flexible control over the layout and protection (e.g., *readable*, *writable*, or *executable*) of our isolation runtime. For example, our system maps the guest physical memory starting at address zero, which allows a guest physical address to be directly used by KVM to access guest memory without further address translation. While segmentation may limit memory access to a continuous range, unnecessary components (e.g., system libraries) could be loaded in the middle of this range and cannot be excluded from the segment.

Second, for performance reasons, HyperLock allows KVM to retain and execute two privileged instructions, i.e., `VMREAD` and `VMWRITE`. This design choice significantly affects the design of hypervisor isolation runtime for KVM confinement. Particularly, because KVM is still privileged, it is critical to prevent it from executing any unwanted privileged instructions, either intended or unintended [8].

Also, certain x86 architecture peculiarities complicate the design for safe context switches between the host and KVM, which will be invoked by the unsafe KVM. If it runs non-privileged, the design would be much simpler and straightforward. On the other hand, we may choose to run KVM at ring-3 by further replacing these two privileged instructions with runtime services. The use of runtime services is necessary because Intel VT mandates accessing the VMCS with the `VMREAD` and `VMWRITE` instructions as the format of the VMCS is not architecturally defined [18]. Thus, it is not feasible to simply map the VMCS in the KVM address space and use memory move instructions to access it. The overhead of frequent VMCS access through runtime services could potentially be reduced though pre-fetching VMCS reads and batch-processing VMCS writes. As such, this design and HyperLock offer different design trade-offs. We leave further consideration of the implications of this choice to future work.

Third, HyperLock enforces instruction alignment [23, 43] to prevent unintended instructions from being generated out of legitimate ones. Alternatively, we can enforce control-flow integrity (CFI) [3, 39] on KVM inside the isolation runtime to provide a stronger security guarantee, especially in eliminating recent return-oriented programming (ROP)-based code-reuse attacks [8]. However, one challenge behind CFI enforcement is the lack of an accurate and complete points-to analysis tool that could be readily applied to KVM. From another perspective, the lack of CFI does not weaken our security guarantee because by design the isolation runtime is not trusted and has been strictly confined within its own address space. In HyperLock, for ease of implementation, we choose to enforce the instruction alignment and combine it with other instruction/memory access control mechanisms to meet our design goal of isolating hosted hypervisors.

Finally, our current prototype defines a narrow interface that exposes five runtime services to support guests with virtual devices. A few more runtime services could be added to incorporate new functionality. For example, our current prototype does not support multi-core VMs, which could be accommodated by adding a new service to handle Inter-Processor Interrupts (IPIs). For the current lack of hardware pass-through support, we can develop a new service to request and release a PCI device on demand and accordingly manage `IOMMUs` [18] to enforce hardware isolation (e.g., to block DMA-based attacks). Further, a service to release guest memory back to the host OS might also be needed to support a balloon driver for cooperative memory management. However, as mentioned earlier, the number of added services should be kept to a minimum and scrutinized to avoid being abused (as they will be considered as part of the HyperLock TCB). Furthermore, our prototype implements `update_spt/npt` by mirroring the corresponding KVM data structure. This makes the prototype relatively simpler to implement by trading off extra memory. This additional memory could be reclaimed in future enhancements to the prototype by sharing the SPT/NPT between HyperLock (readable and writable) and KVM (read-only).

6. Related Work

Hypervisor Integrity The first area of related work is recent efforts in enforcing or measuring the hypervisor integrity. By applying formal verification, seL4 provides strong security guarantees that certain types of vulnerabilities can never exist in a micro-kernel. However, its application to protect commodity hypervisors that run on complex x86 hardware still remains to be demonstrated [30]. HyperSafe [39] enables the self-protection of bare-metal hypervisors by enforcing CFI. However, certain design choices in the host OS make it difficult to be applied to hosted hypervisors. For example, HyperSafe implicitly assumes infrequent updates to page tables in the supported bare-metal hypervisors, which is not the case in commodity OSs (and hosted hypervisors). Moreover, it is important for HyperLock to isolate memory writes from untrusted hypervisors, which cannot be achieved by CFI alone because CFI can only regulate the *control data* access. NoHype [36] removes the (type-I) hypervisor layer by leveraging the virtualization extension to processors and I/O devices. Due to tight coupling between the host OS and the (hosted) hypervisor, NoHype cannot be directly applied to protect hosted hypervisors. The Turtles project [6] and Graf et al. [17] implement nested virtualization support for KVM. However, in these systems, KVM, particularly the (lowest-level) L0 hypervisor, is still tightly integrated with the host (including the sharing of the same address space). As such, HyperLock can be used in these systems to better achieve the isolation of the lowest-level L0 hypervisor. From another perspective, HyperSentry [4] measures the hypervisor for integrity violations using the system management mode (SMM), which has a different goal from HyperLock.

There also exist related efforts in reducing the hypervisor TCB, often adopting the micro-kernel principles. For example, NOVA [34] applies the micro-kernel approach to build a bare-metal hypervisor. Xoar [10] also applies the approach to partition the control domain (of type-I hypervisors) into single-purpose components. Xen disaggregation [24] shrinks the TCB for Xen by moving the privileged domain builder to a minimal trusted component. KVM-L4 [26] extends a micro-kernel with CPU and memory virtualization to efficiently support virtual machines. Compared to these systems, HyperLock focuses on the isolation of *hosted* hypervisors by replacing the hypervisor's TCB in the host kernel with the smaller (12%) and simpler HyperLock code.

Device Driver Isolation The second area of related work includes systems that isolate faults or malicious behaviors in device drivers. For example, Nooks [35] improves OS reliability by isolating device drivers in the light-weight kernel protection domain. Nooks assumes the drivers to be faulty but not malicious. Accordingly, the Nooks sandbox by design lacks instruction access control and malicious drivers cannot be completely isolated by Nooks. A closely related system is SUD [7] that can securely confine malicious device drivers in the user space. SUD relies on IOMMU, transaction filtering in PCI express bridges, and IO permission bits in TSS to securely grant user space device drivers direct access to hardware. However, SUD cannot be applied to the hosted hypervisors such as KVM simply because that hardware virtualization extension (e.g., Intel VT) is not constrained by the IOMMU or other hardware mechanisms that SUD relies on. Microdrivers [15] reduces device driver's TCB in the kernel by slicing the driver into a privileged performance-critical kernel part and the remaining unprivileged user part. RVM [41] executes device drivers in the user space and uses a reference monitor to validate interactions between a driver and its corresponding device. In HyperLock, we securely confine the privileged KVM code in the hypervisor isolation runtime. Gateway [32], HUKO [42], and SIM [31] are systems that use a hypervisor (e.g., KVM) to isolate kernel device drivers or security monitors. We did not take this approach because otherwise we will face the recursive question of how to isolate the hypervisor that runs at the lowest level.

Software Fault Isolation The third area of related work is a series of prior efforts [3, 13, 14, 23, 38, 43] in implementing SFI to confine untrusted code in a host application. For example, PittSFIeld [23] and Native Client [43] both apply instruction alignment to enable the reliable disassembly of untrusted code. CFI [3] constrains runtime control flow to a statically determined control flow graph. Among them, Native Client is closely related but with a different application domain in leveraging user-level SFI to web plugins. As a kernel-level isolation environment, HyperLock needs to address challenges that arise from the needs of enforcing access control of privileged instructions and supporting x86 hardware architecture peculiarities (Section 3). XFI [13] and LXFI [22] are two closely related works. Based on CFI and data sandboxing, XFI combines inline software guards and a two-stack execution model to isolate system software. LXFI ensures API integrity and establishes module principals to partition and isolate device drivers. In comparison, HyperLock focuses on the secure isolation of hosted hypervisors and needs to address additional challenges that are unique to virtualization, such as how to prevent VMWRITE from being misused or how to securely support memory virtualization. To the best of our knowledge, HyperLock is the first system that has been designed and implemented to confine hosted hypervisors so that the host OS and other guests can be protected.

7. Conclusion

We have presented the design, implementation and evaluation of HyperLock, a system that establishes secure isolation of hosted hypervisors. Specifically, we confine the hypervisor execution in the isolation runtime with a separated address space and a constrained instruction set. Moreover, we create a logically separated hypervisor for each guest, thus ensuring a compromised hypervisor can only affect its own guest. We have implemented a prototype of HyperLock for the popular open source KVM hypervisor. The prototype is only 12% of KVM's code size, and further completely removes QEMU from the TCB (of the host and other guests). Security analysis and performance benchmarks show that HyperLock can efficiently provide the intended isolation.

Acknowledgments

We would like to deeply thank our shepherd, Hermann Härtig, 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 Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI) and 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 ARO and the NSF.

References

[1] Kernel Samepage Merging. http://lwn.net/Articles/330589/.

[2] W∧X. http://en.wikipedia.org/wiki/W∧X.

[3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (November 2005).

[4] AZAB, A. M., NING, P., WANG, Z., JIANG, X., ZHANG, X., AND SKALSKY, N. C. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (October 2010).

[5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T. L., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (October 2003).

[6] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAREL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (October 2010).

[7] BOYD-WICKIZER, S., AND ZELDOVICH, N. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the*

2010 USENIX Annual Technical Conference (June 2010).

[8] Buchanan, E., Roemer, R., Shacham, H., and Savage, S. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (October 2008).

[9] Coker, R. Bonnie++. http://www.coker.com.au/bonnie++/.

[10] Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P., and Warfield, A. 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).

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

[12] Elhage, N. Virtualization Under Attack: Breaking out of KVM. http://www.blackhat.com/html/bh-us-11/bh-us-11-briefings.html.

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

[14] Ford, B., and Cox, R. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of 2008 USENIX Annual Technical Conference* (June 2008).

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

[16] Garfinkel, T. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 20th Annual Network and Distributed Systems Security Symposium* (February 2003).

[17] Graf, A., and Roedel, J. Nesting the Virtualized World. Linux Plumbers Conference, September 2009.

[18] Intel. *Intel 64 and IA-32 Architectures Software Developerś Manual Volume 3: System Programming Guide, Part 1 and Part 2*, 2010.

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

[20] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009).

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

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

[23] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th conference on USENIX Security Symposium* (July 2006).

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

[25] National Vulnerability Database. http://nvd.nist.gov/.

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

[27] Provos, N. Improving Host Security with System Call Policies. In *Proceedings of the 12th Usenix Security Symposium* (August 2002).

[28] red hat. KVM: Kernel-based Virtual Machine. www.redhat.com/f/pdf/rhev/DOC-KVM.pdf.

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

[30] Rutkowska, J. On Formally Verified Microkernels (and on attacking them). http://theinvisiblethings.blogspot.com/2010/05/on-formally-verified-microkernels-and.html.

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

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

[33] Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006.

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

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

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

[37] VMware. Timekeeping in VMware Virtual Machines. http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf.

[38] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium On Operating System Principles* (December 1993).

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

[40] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium* (August 2010).

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

[42] Xiong, X., Tian, D., and Liu, P. 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).

[43] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Orm, T., Okasaka, S., Narula, N., Fullagar, N., and Inc, G. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (May 2009).