HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated In-VM Management

Yangchun Fu Junyuan Zeng Zhiqiang Lin
The University of Texas at Dallas
{firstname.lastname}@utdallas.edu

Abstract

To direct the operation of a computer, we often use a shell, a user interface that provides accesses to the OS kernel services. Traditionally, shells are designed atop an OS kernel. In this paper, we show that a shell can also be designed below an OS. More specifically, we present HYPERSHELL, a practical hypervisor layer guest OS shell that has all of the functionality of a traditional shell, but offers better automation, uniformity and centralized management. This will be particularly useful for cloud and data center providers to manage the running VMs in a large scale. To overcome the semantic gap challenge, we introduce a reverse system call abstraction, and we show that this abstraction can significantly relieve the painful process of developing software below an OS. More importantly, we also show that this abstraction can be implemented transparently. As such, many of the legacy guest OS management utilities can be directly reused in HYPERSHELL without any modification. Our evaluation with over one hundred management utilities demonstrates that HYPERSHELL has 2.73X slowdown on average compared to their native in-VM execution, and has less than 5% overhead to the guest OS kernel.

1 Introduction

With the increasing use of cloud computing and data centers today, there is a pressing need to manage a guest operating system (OS) directly from the hypervisor layer. For instance, when migrating a virtual machine (VM) from one place to another, we would like to directly configure its IP address without logging into the system (if that is possible), similarly for firewall rule update; when there is a malicious process detected, we would like to directly kill it at hypervisor layer, similarly for malicious kernel modules; when there is a need to scan viruses, we would like to uniformly scan viruses for all of the running VMs regardless of who owns and manages the VM, whether the VMs might be using an unknown file system, or whether the file systems might be encrypted.

However, if we use a traditional OS shell, a user interface that is automatically executed when a user successfully logs in a computer, this would first require an administrator's password. But, hypervisor providers may not (always) have the administrator's password for each VM, and even when they do have the passwords, it is painful to maintain them considering today large cloud providers usually run millions of VMs (e.g., there were over one million VMs running in Skytap cloud as of January 2012 [7]). Second, it would also require the installation of the management utilities inside each guest OS. Whenever there are updates for these utilities, it is painstaking to update all of them in each VM. Therefore, the presence of a hypervisor layer shell (HYPERSHELL for brevity) for all guest OS would allow cloud providers to have an automated, uniformed, and centralized service for in-VM management.

Unfortunately, such a layer below shell is challenging to implement because of the semantic gap [9]. Specifically, the semantic gap exists since at the hypervisor layer we have access only to the zeros and ones of the hardware level state of a VM—namely its CPU registers and physical memory. But what a hypervisor layer program wants is the semantic information about the guest OS, such as the running processes, opened files, live network connections, host names, and IP addresses. Therefore, a layer below management program must reconstruct the guest OS abstractions in order to obtain meaningful information. A typical approach to do so is to traverse the kernel data structure, but such an approach often requires a significant amount of manual effort.

To advance the state-of-the-art, we introduce a new abstraction called Reverse System Call (R-syscall in short) to bridge the semantic gap for hypervisor layer programs that will be executed in our HYPERSHELL. Unlike traditional system calls that serve as the interface for application programs from a layer below, R-syscall serves as the interface in a reverse direction from a layer up (with a way similar to an upcall [11]). While hypervisor programmers can use our R-syscall abstraction to develop new guest OS management utilities, to largely reuse the existing legacy software (e.g., ps/lsmod/netstat/ls/cp) we make the system call interface of R-syscall transparent to the legacy software, resulting in no modification when using them in HYPERSHELL. In addition, we also make HYPERSHELL transparent to the guest OS, and we do not modify any guest OS code. All of our design and implementation is done at the hypervisor layer.

We have implemented HYPERSHELL. We show that by using the abstraction of R-syscall, we can quickly have a large number of hypervisor layer guest OS management utilities by reusing the existing legacy software (due to its transparency). In our current evaluation, we have tested with over 100 common system administrative utilities. All of them can be correctly executed in HYPERSHELL. The average performance overhead for these utilities is 2.73X slowdown compared to their native in-VM execution. Both micro and macro benchmark evaluation shows that HYPERSHELL has very small overhead (less than 5%) to the guest OS kernel.

In short, this paper makes the following contributions:

- We present HYPERSHELL, a new hypervisor layer shell for automated guest OS management, without using any user accounts from a guest OS.
- We introduce an R-syscall abstraction that allows hypervisor programmers to develop guest OS management utilities without worrying about the semantic gap. Its transparency feature also directly allows many of the legacy utilities to be executed in HYPERSHELL without any modification.
- We have implemented the whole system. We show that HYPERSHELL is practical, and can be used for timely, uniformed, and centralized guest OS management, especially for private cloud.

2 Background and Overview

Challenges. HYPERSHELL aims at executing guest OS management utilities at the hypervisor layer with the same effect as executing them inside an OS. To this end, we are facing two major challenges:

- **How to bridge the semantic gap**. In HYPERSHELL, guest OS management utilities execute below an OS kernel. However, for OS below software, there are no OS abstractions. For example, there is no pid, no FILE, and no socket. Therefore, we have to reconstruct these abstractions such that the utility software understands the guest OS states and can perform the management.
- **How to develop the utilities**. Suppose we have a perfect approach to bridging the semantic gap, we still have to develop the guest OS management software. Should we develop the software from scratch, or can we reuse any legacy (binary or source) code? Ideally, we would like to reuse the existing binary code as there are already lots of OS management utilities, and we show that this approach is feasible.

```
 1. execve("/bin/hostname", ["hostname"], ...) = 0
 2. brk(0)                                     = 0x8113000
 3. access("/etc/ld.so.nohwcap", F_OK)         = -1 ENOENT
 4. mmap2(NULL, 8192, ..., -1, 0) = 0xb7795000
...
36. uname({sys="Linux", node="debian", ...}) = 0
...
40. write(1, "debian\n", 7)                    = 7
41. exit_group(0)
```

(a) System call trace of command "hostname"

```
  c103c305 <sys_uname>:
  1. 0xc103c420  push   %ebx
  2. 0xc103c421  mov    $0xc137ad34,%eax
  3. 0xc103c426  call   0xc125ee10
 ...
                // get the current task structure
 19. 0xc103c430  mov    %fs:0xc13f9454,%eax
                 // point to current->nsproxy
 20. 0xc103c436  mov    0x2c4(%eax),%eax
                 // point to current->nsproxy->uts_ns
 21. 0xc103c43c  mov    0x4(%eax),%edx
 22. 0xc103c43f  mov    0x8(%esp),%eax
                 // point to current->nsproxy->uts_ns->name
 23. 0xc103c443  add    $0x4,%edx
                 // copy to user space buffer
 24. 0xc103c446  call  copy_to_user
```

(b) Disassembled instructions for system call sys_uname

Figure 1: System call trace of utility hostname and one of its sys_uname implementation.

Key Insights. Before describing how we solve these challenges, we would like to first revisit how an in-VM management utility executes. Suppose we want to know the host name of a running OS, we can use utility software such as hostname to fulfill this task. In particular, as illustrated in Fig.1(a), it will execute 41 system calls (syscall for short) in Linux kernel 2.6.32.8, our testing guest kernel. Among these syscalls, sys_uname is the one that really returns the host name. Also, as shown in Fig.1(b), this syscall will traverse the current task structure and dereference the field current->nsproxy->uts_ns->name to eventually retrieve the machine name.

If we implement the same hostname utility and execute it in HYPERSHELL, and if we use a manual approach to bridging the semantic gap, we have to traverse the data structure again, in the same way as how sys_uname does. Since the only interface for user level programs to request OS kernel services is through syscall, and the execution of a syscall is often trusted, then why not let hypervisor programs directly use the syscall abstractions provided by the guest OS? As such, we do not have to develop any code to reconstruct the guest OS abstractions. This is one of the key insights of designing HYPERSHELL.

Another key insight is that not all the syscalls should be executed inside the guest OS. One example is the write syscall that prints the "host name" to the screen. If we execute it inside the guest OS, we would not be able to observe the output from HYPERSHELL. Therefore, we introduce an R-syscall abstraction that is used by hypervisor programmers to annotate the syscalls that need to be redirected and executed inside the guest OS.

In addition, while hypervisor programmers can use our R-syscall abstraction to develop new software to manage

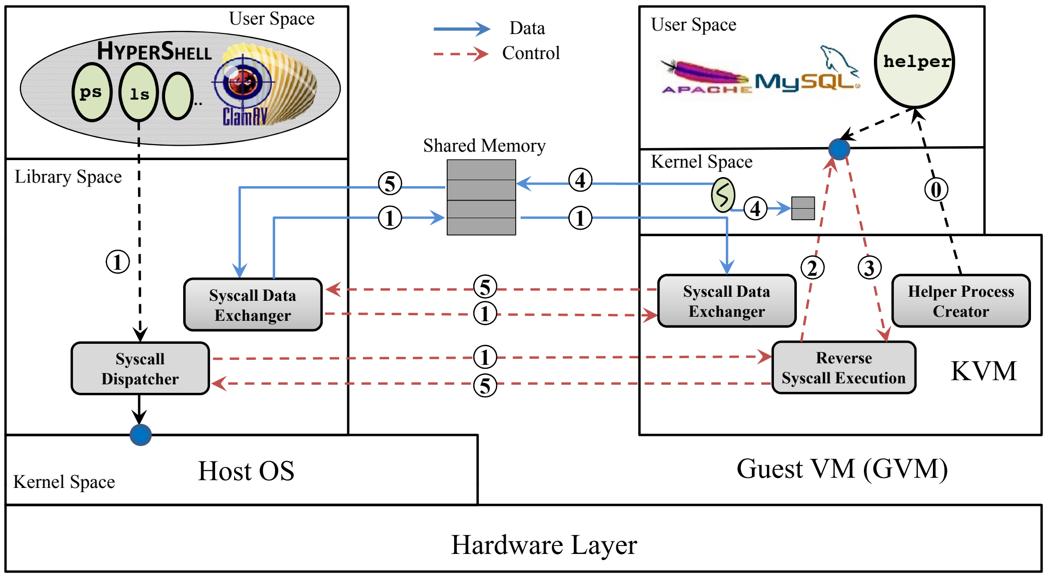

Figure 2: An Overview of the HYPERSHELL Design.

the guest OS, there are already lots of legacy utilities running inside a VM for the same purposes. For instance, there are over hundreds of tools in core-utility, util-linux, and net-tools for Linux OS. If we can make our R-syscall transparent to the legacy software, then there is no need to annotate the R-syscall and we can directly execute the legacy software in HYPERSHELL. For instance, in `hostname` example, only the `uname` syscall needs this abstraction. We can thus hook the execution of the syscall and use a transparent policy to determine whether a given syscall is an R-syscall.

Scope and Assumptions. As HYPERSHELL is executed at the hypervisor layer and will also invoke the syscalls from the guest OS, we assume everything below is trusted. This includes the guest OS kernel, host OS and the hypervisor code. Ensuring the hypervisor and guest kernel integrity is an independent problem and out of scope of this paper. In fact, recently there have been many efforts aiming at ensuring the guest kernel and hypervisor integrity (e.g., SecVisor [33] and HyperSafe [37]). Also note that HYPERSHELL is designed mainly for automated guest OS management and not for security. While it could defeat certain attacks such as guest user level viruses, it cannot defend against any guest kernel level attacks.

To make our discussion more focused, we assume a guest OS running with a 32-bit Linux kernel atop x86 architecture. For the hypervisor, we focus on the design and implementation of HYPERSHELL using KVM [4].

Overview. An overview of our HYPERSHELL is presented in Fig. 2. For a KVM based virtualization system, there are two kinds of OSes: one is the guest OS that is executed atop a KVM hypervisor, and the other is the host OS that hides the underlying hardware resources and provides the virtualized resources to KVM. The goal of HYPERSHELL is to execute the guest OS management utilities from the host OS to manage the guest OS. To this end, there are five key components: two located inside the library space of the host OS, and three located at the hypervisor layer of the GVM.

To use HYPERSHELL, assume hypervisor managers use `ls` (or other utilities such as `ps` or `hostname`) to list the guest files in a given directory. To get started, they will launch `ls` in our host OS. The real execution of `ls` will be divided into a master process that is executed inside the host OS, and a helper process that is executed in the GVM. Only when an R-syscall gets executed will we forward the execution of this syscall to a helper process in the GVM and map the execution result (e.g., the directory entries) back such that `ls` can continue its execution in the master process. There are five key steps involved during the execution of an R-syscall:

- **Step** ①: Right after a syscall enters the library space in the host OS, our Syscall Dispatcher intercepts it. If it is not an R-syscall, it directly traps to the host OS kernel for the execution. Otherwise it fetches the syscall number and arguments, and invokes our Syscall Data Exchanger at the host OS side which communicates with its peers at the GVM side with the detailed syscall execution information. Next, our master process gets paused and will be resumed at **Step** ⑤ when the redirected R-syscall finishes the execution. At the GVM side, according to each specific syscall specification, the Syscall Data Exchanger will set up the corresponding memory state for the to-be-executed R-syscall.

- **Step** ②: Our Reverse Syscall Execution will wait until the helper process traps to kernel. The helper process is created at **Step** ⓪ right after the execution of the management utilities in HYPERSHELL, or can be executed as a daemon depending on the settings.

- **Step** ③: Our Reverse Syscall Execution directly injects the execution of the R-syscall with the corresponding arguments and memory mapping, and makes the R-syscall be executed under the helper process kernel context. Note that such an R-syscall injection and execution mechanism works similar to function call injection from debuggers but with a more powerful capability because of the layer below control from hypervisor.

- **Step** ④: During the execution of the R-syscall, if there is any kernel state update to the guest OS, this syscall will directly update the kernel memory as usual (e.g., `sysctl` that changes the kernel configuration). If there is any user space update (such as the `buffer` in `read` syscall), it directly updates to the shared memory created by the Syscall Data Exchanger in **Step** ①.

- **Step** ⑤: Right after the execution of the syscall exit of the R-syscall, we notify Syscall Dispatcher and Syscall Data Exchanger at the host OS side. We also copy the data from the shared memory to the user space of the master process, if the R-syscall has any memory update. We also resume the execution of the master process and directly return to its user space for continued execution. Regarding the helper process state in the GVM: if the master process terminates, it will also be terminated (in non-daemon mode); otherwise, it will keep executing `int3`[1] such that our Reverse Syscall Execution can always take control of the helper process from the hypervisor layer.

3 Host OS Side Design

3.1 Syscall Dispatcher

The key idea of HYPERSHELL in bridging the semantic gap is to selectively redirect and execute a syscall in the guest OS. (The selected one is called an R-syscall). As shown in Fig.1(a), not all the syscalls belong to R-syscalls. Therefore, the first step in our Syscall Dispatcher design is to systematically examine all of the Linux syscalls and define our reverse execution policy for each syscall.

Syscall Execution Policy. In our testing guest kernel Linux 2.6.32.8, there are 336 syscalls in total. Among them, we find that technically, nearly all of them can be redirected to execute in a guest OS. However, for process creation (e.g., `execve`, `fork`, `exit_group`), dynamic loading (e.g., `open`, `stat`, `read` when loading a shared library), memory allocation (e.g., `brk`, `mmap2`), and

[1] An interrupt that is often used by debuggers to set up break points.

The Syscall Trace of "`cp /etc/shadow /outside/shadow`"	Host OS	GVM
`execve("/bin/cp",["cp","/etc/shadow","/tmp/shadow"],…= 0`	✓	
`brk(0)                                   = 0x8824000`	✓	
`access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT`	✓	
…	✓	
`stat64("/etc/shadow",{st_mode=S_IFREG|0640,st_size=713, ...})=0`		✓
`stat64("/outside/shadow", 0xbf9bad78)        = -1 ENOENT`	✓	
`open("/etc/shadow", O_RDONLY|O_LARGEFILE) = 0`		✓
`fstat64(0, {st_mode=S_IFREG|0640, st_size=713, ...}) = 0`		✓
`open("/outside/shadow", O_WRONLY|O_CREAT|…|O_LARGEFILE, 0640)=3`	✓	
`fstat64(3, {st_mode=S_IFREG|0640, st_size=0, ...}) = 0`	✓	
`read(0, "root::15799:0:99999:7:::\ndaemon:"..., 32768) = 713`		✓
`write(3, "root::15799:0:99999:7:::\ndaemon:"..., 713) = 713`	✓	
`read(0, "", 32768)                       = 0`		✓
`close(0)`		✓
`close(3)`	✓	

Table 1: Syscalls in `cp` with different execution policy.

screen output (e.g., `write`), we would like them to be executed in the master process created in our HYPERSHELL.

Unfortunately, for the rest syscalls, it is also not always clear which syscalls need to be executed in the helper process. For instance, as shown in Table 1, suppose we want to copy `/etc/shadow` from the guest OS to the host OS; in this case, some of the file system related syscalls (e.g., `open/stat64/read/close`) are executed in the GVM, and some (e.g., `open/stat64/write/close`) are executed in HYPERSHELL. Even though we could leave the solution to hypervisor programmers, where they would specify which syscall needs to be executed in the master process or helper process, we would prefer to make an automated policy for these syscalls in order to allow for transparent reuse of the legacy binary code.

In general, syscalls are relatively independent of each other (e.g., `getpid` will just return a process ID, and `uname` will just return the host name). After having examined all of the 336 syscalls, we realize that the syscalls that have connections are often file system and `socket` related (e.g.,`open/stat64/read/write/close`), and these syscalls have dependences with the file descriptors. For instance, as illustrated in Table 1, if we can differentiate the file descriptor from the GVM and the host OS automatically, we can then transparently execute the existing legacy utility in HYPERSHELL without any modification.

Intuitively, we would use dynamic taint analysis [29] to differentiate the file descriptors that are accessed inside the GVM or the host OS. However, such a design would require instruction level instrumentation, which is often very slow. In fact, our earlier design adopted such a taint analysis approach by running HYPERSHELL in an emulator. Surprisingly, we have a new observation and we can actually eliminate the expensive dynamic taint analysis.

In particular, as a file descriptor is just an index (a 32-bit unsigned integer) to the opened files (and network `socket`) inside the OS kernel for each process, it has a limited maximum value (due to the resource constraints). In our testing Linux kernel, it is 1023 (which means a process can only open 1024 files at the same time). Also, it is extremely rare to perform data arithmetic operations on a file descriptor. Therefore, we can in fact add a distinctive value (e.g., 4096 or 8192) to the file descriptor returned by the GVM. Whenever such a descriptor is used by the GVM again, we subtract our added value. As such, we can differentiate whether a file descriptor is from the host OS or the GVM by simply looking at its value.

Whether a file descriptor should be returned from the GVM or the host OS depends on the semantics of `open`. Specifically, if it is opening the guest OS files (we can differentiate this based the parameters, and internally we add a prefix associated with the guest files), it is executed in the GVM; otherwise it is executed in the host OS. For instance, we know "/etc/shadow" is in the GVM, and "/outside/shadow" is in the host OS while executing "`cp /etc/shadow /outside/shadow`". Similarly, we can also infer the files involved in "`cp -R <src> <dst>`" by their names and their opening mode.

Syscalls in Dynamic Loader. To intercept the syscall, we use dynamic library interposition [13] (a technique that has been widely used in many applications such as LibSafe [36]). Interestingly, we notice that the syscalls executed in dynamic loader cannot be trapped by our library interposition. Therefore, syscalls executed while loading a dynamic library (e.g., `access/open/stat64/read/close`) will not be checked against our policy, and they will be executed directly on the host OS side, which is exactly what we want.

Summary. By default, the majority of the syscalls will be treated as redirectable and they will be executed in the GVM, except process execution and memory management related syscalls that will be executed in the host OS. All file system and network connection related syscalls will be checked against the file descriptor. Whether a file descriptor needs to be checked is determined by the semantics of the corresponding file operations.

3.2 Syscall Data Exchanger

Since we need to make an R-syscall executed in the GVM, we must inform the GVM with the corresponding context and also update the corresponding memory state at the host OS side to reflect the R-syscall's execution. Our Syscall Data Exchanger is designed for this goal.

Specifically, right after an R-syscall enters the library space (**Step** ①), we will retrieve the syscall arguments (e.g., the buffer address and size information) based on the corresponding syscall's specification. Then, we will inform its peer (to be discussed in §4.3) to prepare for the necessary arguments at the GVM side. Once an R-syscall finishes the execution (**Step** ⑤), we will pull the data back from the GVM to the host OS. All of these operations are quite straightforward.

4 Guest VM Side Design

4.1 Helper Process Creator

An R-syscall must be executed under a certain process execution context in the GVM. While we could hijack an existing process to execute an R-syscall, such an approach is too intrusive to the hijacked process. Therefore, we choose to create a helper process dedicated to executing our R-syscall in the guest OS. Regarding the permission of this helper process, it should have the highest privilege; otherwise an R-syscall may fail due to certain permissions. Also, it would terminate when the master process terminates (to minimize the impacts to the guest OS workloads). To have better performance while executing the management utilities in HYPERSHELL, we can also have an option of creating a daemon process as the creation of a helper process takes additional time. There are only three instructions for this helper process as shown below:

```
00000001   cd 80         int 0x80
                    _loop:
00000003   cc            int 3
00000004   eb fd         jmp  _loop
```

Basically, it keeps executing `int3` (i.e., `while(1) int3`) with a prefix of `int 0x80`. We will explain why we use such an instruction sequence in §4.2.

Then the challenge lies in how to select a high privilege process to `fork` the helper process. Since all Linux kernels have an `init` process with PID 1, one option is to traverse the `pid` field of the `task_struct` for each process. But such a design would make HYPERSHELL too OS-specific. Fortunately, since we are able to inject an R-syscall (discussed in §4.2), we are certainly able to inject `getpid` to inspect the return values. If it is 1, we can therefore infer that the current execution context is the `init` process, and we can then inject a `fork` syscall to create our helper process. Meanwhile, we will retrieve the child PID from the return value of `fork`, and then use `getpid` again to identity the helper process. Once we have identified it, we will pull its CR3 such that the hypervisor knows it is the `int3` that occurs in our helper process, not others (e.g., `gdb`) by looking at the CR3 value.

Consequently, we must design a mechanism to intercept the entry point and exit point of the syscall execution for each process in order to select the `init` process. Once we have created our helper process, we will not need this interception. We call the selection of `init` process *redirection initialization phase* (i.e., the RI-Phase that only occurs at **Step** ⓪) in the GVM. With hardware-assisted virtualization, we can rely on hardware mechanisms to intercept the execution of the syscall instructions. Ether [14], built atop the Xen hypervisor, leverages a page fault exception to capture syscall entry and syscall exit points. Nitro [31], based on the KVM hypervisor, leverages invalid segment exceptions to intercept the pair of `sysenter/int0x80` and `sysexit` syscalls for a single process. In our design, we extend Nitro to intercept the system wide syscall entry and exit pairs (for all processes).

4.2 Reverse Syscall Execution

After we have passed the RI-Phase, we are then ready to execute an R-syscall if there is any. Yet we have to solve two additional challenges: when and how to execute an R-syscall under our helper process context.

When to Inject a Syscall. At a given time, a process either executes in user space or kernel space. To trap to kernel, a process must use a syscall or interrupt (including exceptions). As an interrupt or exception can occur at arbitrary time, the OS must be designed in such a way that it is safe to trap to OS kernel and execute syscall or interrupt handler services at any time in user space. However, we cannot inject a syscall execution at arbitrary time in kernel space. This is because: (1) the injected syscall might make kernel state inconsistent. For instance, we might inject a syscall when the kernel is handling an interrupt, and there might be some synchronization primitives involved (e.g., `spin_lock`). After we inject a new syscall, if this syscall execution also happens to lock some data or release certain locks, it may cause inconsistency among these locks. (2) Similarly, we might make the non-interruptible code interruptible. For instance, if the kernel is executing the `cli` code block and has not executed `sti` yet, and if we inject a new syscall, this may make the non-interruptible code interruptible. (3) We might also overflow the kernel stack of a running process if it already has a large amount of data.

Therefore, to inject the execution of a syscall, we use the approach that right before entering the kernel space (e.g., `sysenter/int0x80`), or right after exiting to the user space of a running process, we will save the current execution context (namely all the CPU registers), and then execute the injected syscall (such as our `getpid` case in §4.1).

Regarding our helper process, we have a slightly different strategy to inject the R-syscall. In particular, when the `int3` traps to hypervisor, we change the current user level EIP (pointing to `cc` at this moment) to EIP-2, which points to "`int 0x80`"; meanwhile, we prepare for the necessary arguments such as setting up the corresponding registers. Then when control returns to the user space of the helper process, it will automatically execute the syscall we prepared for because we have changed its EIP. The use of `int3` is to make the control flow of the helper process trap to the hypervisor. There are also alternative approaches such as using a `cpuid` instruction.

How to Execute an R-syscall. To execute an R-syscall, we have to set up the syscall arguments and map the memory that will be used during the R-syscall execution. This is done by our Syscall Data Exchanger (§4.3) at **Step** ①. After that, the syscall will be executed as usual in the GVM. If there is any memory update to the user space, it will directly (**Step** ④) update to the shared memory that is allocated by our Syscall Data Exchanger. For kernel space, it directly updates the guest kernel. Once an R-syscall finishes, we inform the Syscall Dispatcher at **Step** ⑤, and push the updated memory back to the master process. At the GVM side, the helper process continues its execution of `int3`. When the master process exits, we terminate the helper process if it is not executed in the daemon mode.

4.3 Syscall Data Exchanger

As discussed in §3.2, we need to pass the corresponding syscall parameters to the GVM. Also, we need to map the data back to the host OS if there is any memory update. The Syscall Data Exchanger at the GVM side is exactly designed to achieve these goals.

One issue we have to solve is the virtual address relocation. This is because the same virtual addresses used by the host OS may not be available for the helper process in the GVM, and we have to relocate the virtual addresses used in the syscalls of the master process to the available addresses of the helper process. To this end, before the execution of the first R-syscall, we will first allocate a large buffer (as a cache) with a default size of 64K bytes by injecting a `mmap` syscall and recording the mapped virtual address of this buffer, denoted as V_g, and its size, denoted as S_g. (Certainly, the guest OS will automatically `munmap` this allocated space once the helper process terminates.) Then whenever there is an R-syscall (e.g., `read`) that has an argument with virtual address V_h and size S_h, we will use V_g as the buffer starting address instead of V_h, and if S_h is greater than S_g, we will inject `mmap` to map more caches.

Also, to avoid too many data transmissions between the host OS and the GVM, we allocate a shared memory between them. Right after the execution of the `mmap` syscall to allocate new pages for the redirected syscall, in the hypervisor layer we map the pages of the shared memory to the virtual address of the `mmap` returned page by traversing the page tables (rooted by the captured CR3) of the helper process, such that we do not have to perform an additional memory copy from the GVM to the shared memory. To prevent being swapped by the guest OS, we inject `mlock` syscall to lock the `mmap` allocated memory.

5 Evaluation

We have developed a proof-of-concept prototype of HYPERSHELL with 3,700 lines of C code. The implementation is scattered across both the host OS side, which is atop Linux kernel 3.0.0-31, and the KVM side. While we have used KVM to build HYPERSHELL, we believe our

Process	**S**	$B(ms)$	$D(ms)$	$T(X)$	date	✗	0.11	0.12	1.09	mkdir	✓	0.10	0.19	1.90
ps	✗	1.33	5.42	4.08	w	✗	0.95	6.62	6.97	mkfifo	✓	0.10	0.19	1.90
pidstat	✗	1.95	7.56	3.88	hostname	✓	0.04	0.06	1.50	mknod	✓	0.10	0.19	1.90
nice	✓	0.07	0.11	1.57	groups	✓	0.21	0.62	2.95	mv	✓	0.15	0.31	2.07
getpid	✓	0.01	0.02	2.00	hostid	✓	0.16	0.56	3.50	rm	✓	0.08	0.15	1.88
mpstat	✗	0.29	0.66	2.28	locale	✓	0.09	0.17	1.89	od	✓	0.12	0.35	2.92
pstree	✗	0.69	6.03	8.74	getconf	✓	0.09	0.34	3.78	cat	✓	0.07	0.18	2.57
chrt	✓	0.11	0.16	1.45	**System Utils**	**S**	$B(ms)$	$D(ms)$	$T(X)$	link	✓	0.07	0.13	1.86
renice	✓	0.11	0.18	1.64	uptime	✗	0.07	0.47	6.71	comm	✓	0.08	0.22	2.75
top	✗	504.92	510.85	1.01	sysctl	✓	8.5	42.72	5.03	shred	✗	0.72	0.92	1.28
nproc	✓	0.07	0.26	3.71	arch	✓	0.07	0.11	1.57	truncate	✓	0.07	0.26	3.71
sleep	✓	1.27	1.28	1.01	dmesg	✓	0.38	0.51	1.34	head	✓	0.07	0.15	2.14
pgrep	✓	0.89	4.72	5.30	lscpu	✓	0.26	1.21	4.65	vdir	✓	0.63	3.95	6.27
pkill	✓	0.87	4.33	4.98	mcookie	✗	0.29	0.49	1.69	nl	✓	0.08	0.17	2.13
snice	✓	0.17	0.65	3.82	**Disk/Devices**	**S**	$B(ms)$	$D(ms)$	$T(X)$	tail	✓	0.08	0.20	2.50
echo	✓	0.07	0.09	1.29	blkid	✓	0.14	0.61	4.36	namei	✓	0.07	0.13	1.86
pwdx	✓	0.05	0.07	1.40	badblocks	✓	0.35	0.44	1.26	whereis	✓	2.05	4.86	2.37
pmap	✓	0.16	0.36	2.25	lspci	✓	31.40	36.52	1.16	stat	✓	0.27	0.78	2.89
kill	✓	0.01	0.04	4.00	iostat	✓	0.45	1.04	2.31	readlink	✓	0.07	0.12	1.71
killall	✓	0.62	3.03	4.89	du	✓	0.11	0.53	4.82	unlink	✓	0.07	0.13	1.86
Memory	**S**	$B(ms)$	$D(ms)$	$T(X)$	df	✓	0.16	0.35	2.19	cut	✓	0.08	0.17	2.13
free	✗	0.04	0.08	2.00	**Filesystem**	**S**	$B(ms)$	$D(ms)$	$T(X)$	dir	✓	0.07	0.20	2.86
vmstat	✗	0.19	0.33	1.74	sync	✓	8.07	6.53	0.81	mktemp	✓	0.09	0.18	2.00
slabtop	✗	0.22	0.36	1.64	getcap	✓	0.04	0.08	2.00	rmdir	✓	0.07	0.13	1.86
Modules	**S**	$B(ms)$	$D(ms)$	$T(X)$	lsof	✓	3.31	6.12	1.85	ptx	✓	0.12	0.45	3.75
rmmod	✓	0.51	3.14	6.16	pwd	✓	0.07	0.11	1.57	chcon	✓	0.06	0.12	2.00
modinfo	✓	0.48	1.54	3.21	**Files**	**S**	$B(ms)$	$D(ms)$	$T(X)$	**Network**	**S**	$B(ms)$	$D(ms)$	$T(X)$
lsmod	✓	0.10	0.17	1.70	chgrp	✓	0.19	0.47	2.47	ifconfig	✗	0.32	1.15	3.59
Environment	**S**	$B(ms)$	$D(ms)$	$T(X)$	chmod	✓	0.07	0.14	2.00	ip	✓	0.10	0.20	2.00
who	✓	0.14	0.72	5.14	chown	✓	0.19	0.47	2.47	route	✓	138.65	150.32	1.08
env	✓	0.07	0.11	1.57	cp	✓	0.11	0.27	2.45	ipmaddr	✓	0.13	0.34	2.62
printenv	✓	0.07	0.1	1.43	uniq	✓	0.09	0.35	3.89	iptunnel	✓	0.09	0.29	3.22
whoami	✓	0.19	0.45	2.37	file	✓	0.87	1.72	1.98	nameif	✓	0.10	0.21	2.10
stty	✓	0.11	0.46	4.18	find	✓	0.20	0.58	2.90	netstat	✗	0.25	0.37	1.48
users	✓	0.09	0.53	5.89	grep	✓	0.35	2.14	6.11	arp	✓	0.14	0.24	1.71
uname	✓	0.09	0.11	1.22	ln	✓	0.08	0.14	1.75	ping	✗	15.02	18.2	1.21
id	✓	0.26	0.85	3.27	ls	✓	0.14	0.27	1.93	Avg.	-	7.27	8.45	2.73

Table 2: Evaluation Result of the Tested Utility Software. *S* stands for whether there is any Syntax-difference, $B(ms)$ stands for the average time of the base execution, $D(ms)$ stands for the average execution time of the utility in HYPERSHELL when using the daemon mode in GVM, and $T(X)$ stands for the result of D/B (i.e., the times).

design can be applied to other types of hypervisors such as Vmware, Xen and VirtualBox.

In this section, we present our evaluation results. All of our experiments were carried out on a host machine configured with an Intel Core i7 CPU with 8G memory and running with Ubuntu 12.04 using Linux kernel 3.0.0-31; the guest OS is Debian 6.04 with kernel 2.6.32.8.

5.1 Effectiveness

Benchmark Software. Recall the goal of HYPERSHELL is to enable the execution of native management utilities at the hypervisor layer to manage a guest OS, and also enable the fast development of these software by using the R-syscall abstraction. Since the software development with HYPERSHELL is very simple (a hypervisor programmer just needs to annotate the syscall and inform HYPERSHELL which one is an R-syscall), we skip this evaluation. In the following, we describe how we automatically execute the native utilities in HYPERSHELL to transparently manage a guest OS.

Today, there are a large number of administrative utilities to manage an OS. To test HYPERSHELL, we systematically examined all of the utilities (in total 198) from six packages including core-utility, util-linux, procps, module-init-tools, sysstat, and net-tools, and eventually we selected 101 utilities, as presented in Table 2, though technically we can execute all of them. The selection criteria is the following: if a utility is all user level program (e.g., hash computation such as `md5sum`), or not so system management related (e.g., `tr`), or can be executed in alternative way (e.g., `poweroff`, `halt`), or not supported by the kernel any more (e.g., `rarp`), we ignore them.

Experimental Result. Without any surprise, through our automated system call reverse execution policy, all of these utilities can be successfully executed in HYPERSHELL. To verify the correctness of these utilities, we use a cross-view comparison approach in a similar way when we tested our prior systems such as VMST [16, 17] and EXTERIOR [18]. Basically, to test a given utility such as `ps`, we first execute it inside the GVM and save the output, which is called the in-VM view; then we execute it inside HYPERSHELL to manage the GVM and also save the output, which is called the out-of-VM view. Then we compare the syntax (through `diff`) and semantics (with a manual verification) of the in-VM and out-of-VM views, which leads to the two sets of effectiveness test results: one is the syntax comparison, and the other is the semantic (i.e, the meaning) comparison.

We notice that while there are 16 utilities that have syntax differences (as shown in the *S* column in Table 2), all other utilities have the same screen output. A further

Tested Item	Native-KVM	GVM-RI-Phase	Slowdown (%)	GVM-RE-Phase	Slowdown (%)
stat (μs)	0.39	2.28	82.89	0.41	4.88
fork proc (μs)	47.20	147.26	67.95	47.54	0.72
exec proc (μs)	158.20	480.00	67.04	161.30	1.92
sh proc (μs)	384.90	1088.10	64.63	386.30	0.36
ctxsw (μs)	0.59	1.23	52.03	0.73	19.18
10K File Create (μs)	17.80	40.67	56.23	17.96	0.89
10K File Delete (μs)	4.64	7.16	35.20	4.65	0.22
Bcopy (MB/s)	5689.17	5647.71	0.73	5605.40	1.47
Rand mem (ns)	72.20	72.65	0.62	73.24	1.42
Mem read (MB/s)	10150.00	10000.00	1.48	10000.00	1.48
Mem write (MB/s)	8567.70	8543.00	0.29	8540.40	0.32

Table 3: Micro-benchmark Test Result of GVM.

investigation shows that the syntax differences among them is actually caused due to the different location (host OS vs. GVM) and timing of performing our in-VM and out-of-VM experiment. Regarding the semantics, we notice that all of the utilities have *the same semantics* as the original in-VM programs through our manual verification.

Testing w/ More Guest Kernels. Working at syscall level allows HYPERSHELL with less constraint and wider applicability because of the POSIX compatibility. For instance, we can now use a single host OS to manage a large number of syscall-compatible OSes. To validate this, we selected five other recently released Linux kernels of versions 2.6.32, 2.6.38, 3.0.10, 3.2.0, and 3.4.0, and executed them in our GVM. Our benchmark utilities were all correctly executed with these kernels.

5.2 Performance Overhead

When executing a program in HYPERSHELL, there are two processes to fulfill the execution: the master process executed in the host OS, and the helper process executed in the guest OS. Consequently, we have to measure two sets of performance. One is how slow an end-user would feel when executing a utility in HYPERSHELL. The other is the impact with respect to the guest OS kernel due to our syscall capturing and helper process execution at the GVM. Below we report these two types of overhead.

5.2.1 Performance Impact to the Native Utilities

With different settings of the helper process (daemon or non-daemon), we could also have two sets of performance overhead for the utility software. However, the performance differences for these two settings mainly come from the creation of the helper process, which is almost a constant factor (the time interval between the two scheduled executions of the `init` process). Our evaluation shows that every 5 seconds, the `init` process will be scheduled. Therefore, it leads to the creation of a helper process with maximum 5 seconds, the worst case delay if we want to use a non-daemon helper process to execute the R-syscall. All other latency is the same compared to the daemon mode execution. Therefore, in the following, we present our result with the daemon mode execution of the helper process.

Again, we used these 101 utilities in effectiveness tests to measure this overhead. Specifically, we executed the utilities each with 100 times and computed their average. First, we ran all of them in a native-KVM and got the average execution time for each of them as the base. This result is presented in the *B*-column of Table 2. Then we collected the average run time of these utilities in HYPERSHELL with a daemon helper process in the GVM. This result is presented in the *D*-column. We computed the overhead of this test with the base one, and we report them in the *T*-column. We compare with the execution running in native-KVM instead of native host OS because we are comparing our out-of-VM approach with an in-VM approach. We notice that on average, with a daemon mode helper process, HYPERSHELL has 2.73X slowdown compared to the executions running in a native-KVM. This overhead mainly comes from the data exchange and synchronization between the host OS and the GVM during the R-syscall execution.

5.2.2 Performance Impact to the GVM

The performance impact to the GVM also falls into two scenarios: one is the system wide sysenter/sysexit interception that is used to capture the `init` process (recall we name it the RI-Phase), and the other is the R-syscall execution that occurs in the helper process (we call this RE-Phase). These two phases inevitably introduce performance penalty to the running workloads/processes at GVM. Note that if the GVM is neither running in RI nor RE-Phase, there is no performance overhead. To quantify the overhead from these two scenarios, we used standard benchmark programs (e.g., LMBench, and ApacheBench) that are used in other work (e.g., [37, 39]) to measure the runtime overhead of the guest OS execution at both micro and macro level for these two phases.

Also, according to the result from Table 2, the execution of the RE-phase is very short (on average 8.45 milliseconds). In addition, our RI will never be executed if the helper process has created. Therefore, we have to create an environment to keep executing RI and RE such that we can measure the impact to the long running benchmark programs. That is, we will keep polling the `init` process to measure the impact from the RI-Phase, and keep executing the `int3` loop for the helper process to measure the impact from the RE-Phase. These results are the worst case performance impact to running processes in the GVM.

Micro-benchmarks. To evaluate the primitive level performance slowdown, we used LMBench suites. In particular, we focused on the overhead of the `stat` syscall, process creation (fork proc), process execution (exec proc), C library function (sh proc), context switches (`ctxsw`), memory-related operations (e.g., bcopy, Mem read, Mem Write), and IO-related operations (e.g., 10k File Create, and 10K File Delete).

The detailed result is presented in Table 3. The RI-Phase tends to have large overhead on tests which contain syscalls, as we intercept the system-wide syscall entry and exit points. While we do not intercept context switches, our system still has large overhead on the `ctxsw` test. The reason is that LMBench tests the time of context switches on a number of processes. And these process are connected using pipe. Therefore, the measurement still contains syscalls. In contrast, during the RE-Phase, our syscall interception is only within the helper process, and it has significantly less overhead except the `ctxsw` case with similar reasons in the RI-Phase.

Macro Benchmarks. We used four real world workloads to quantify the performance slowdown at the macro level. In particular, we decompressed a source tarball of Linux 2.6.32.8 using bzip, and then compiled the kernel using kbuild. We recorded the process time. In the test of Apache, we used ApacheBench [1] to issue 100,000 requests for a 4k-byte file from a client machine and got the throughput (#request/s). For memcached [5], we recorded the time of processing 1,000 requests.

The performance overhead is presented in Table 4. For the RI-Phase, the overhead comes from the `VMexit` of trapping syscall entry and exit. Hence, the workloads that have large portions of IO operation will incur large overhead, e.g., as in Kbuild, Apache, and memcached. The worst case is memcached which is also sensitive to IO-latency. In contrast, computation intensive workloads have small overhead (as in the bzip case). Regarding the RE-Phase, all the workloads have small overhead because our system only introduces a user mode `int3` loop. The `VMexit` only occurs in the helper process execution context. The only side effect is that the helper process takes some CPU time slices from them.

5.3 Case Studies

Once we have enabled the execution of native utilities in HYPERSHELL to manage the guest OS, many new use cases would appear. For instance, we can now kill malicious processes, remove malicious drivers, change the guest IP address, update the firewall rules, etc., directly from the hypervisor layer. In the following, we demonstrate an interesting use case of our system—full disk encryption (FDE) protected virus scanning from the hypervisor.

Today, because of the privacy and data-breach concerns, a growing practice for outsourced VMs is to deploy FDE. Unfortunately, this has brought challenges for disk introspection, forensics, and management. With HYPERSHELL, we can actually use off-the-shelf anti-virus software from the host OS to transparently scan files in the guest OS even though the GVM disk might have been encrypted by FDE.

Benchmark Program	Native-KVM	GVM-RI-Phase	Slowdown (%)	GVM-RE-Phase	Slowdown (%)
bzip (s)	16.83	18.35	8.28	17.04	1.23
kbuild (s)	1799.00	2270.25	20.76	1889.97	4.81
memcached (s)	1.57	3.11	49.52	1.64	4.27
Apache (#request/s)	1104.60	904.12	18.15	1065.28	3.56

Table 4: Macro-benchmark Test Result of GVM.

To validate this, we installed `dm-crypt` [3], a transparent FDE subsystem in the Linux kernel (since version 2.6) in our GVM. Under a `test` user home directory, we copied a large volume of files including the source code of Linux-2.6.32.8, gcc, glibc, QEMU, Apache, and Lmbench, as well as two viruses from `offensivecomputing.com`, resulting in a total number of 101,415 files adding up to 1336.09 megabytes in size. In the host OS, we installed ClamAV-0.98 [2] and used it (in particular its `clamscan`) to scan the files in `/home/test` in the GVM. We tried two different approaches in this testing:

- The first is to directly allow `clamscan` running in HYPERSHELL to scan the files in the GVM by redirecting the R-syscall, and in this case it took 188.35 seconds to scan the entire 1336.09 megabytes of files and find the two viruses.
- The second is to copy (i.e., `cp`) the files in `/home/test` to our host OS, and then scan them natively. In this case, it took 59 seconds to copy these files, with another 120.91 seconds scanning them, resulting in a total of 179.91 seconds.

It is worth noting that very interestingly if we installed ClamAV inside the GVM and scanned these files, it would take 271.58 seconds. Therefore, by moving certain management software running into HYPERSHELL, it can in fact speedup certain computation (188.35 vs. 271.58) as shown in our `clamscan` case. There are two primary sources for this speedup: one is that there is no additional VMexit when processing the disk IO at the host OS side (i.e., IO in host OS is usually faster than guest OS), and the other is because there is no need for the decryption of the signature data base of ClamAV when running at host OS.

6 Limitations and Future Work

While HYPERSHELL offers better automation (e.g., no need of login), uniformity (e.g., all of the VM can be checked for anti-virus), and centralized management (e.g., using only one copy of the software running at a hypervisor to manage a large number VMs, and there is a need of only updating the copy at the hypervisor layer), it comes with price. In particular, it will circumvent all of the existing user login and system audit for each managed VM.

For instance, `syslog` in each individual VM will not be able to capture all the executed events inside the guest OS. To fix this, we need to add a new log record at the hypervisor layer for each activity executed in HYPERSHELL, such that the entire cloud can still be audited. One avenue of our future research will address this.

Second, as normal utility software does, HYPERSHELL requires the trust of the guest OS kernel as well as the `init` process. Consequently, it cannot be used for security critical applications, especially when the kernel has been compromised. Also, unlike introspection, which aims to achieve stealthiness, HYPERSHELL is not designed with this goal in mind, since its primary goal is to manage the guest OS (which definitely introduces footprints) from out-of-VM in the same way as we manage in-VM but in a more centralized and automated manner.

Third, our current prototype requires both OSes running in the host OS and the GVM to have compatible syscall interface. If a guest OS uses a randomized system call interface (e.g., RandSys [25]), it could thwart the execution of the management utilities at HYPERSHELL. In fact, we can design certain logic in our Syscall Dispatcher and Reverse Syscall Execution component to perform syscall translations even though the syscalls are not fully compatible or randomized (e.g., with different syscall number). We leave this as another future work. Again, we would like to emphasize that working at syscall boundary makes HYPERSHELL with less constraint when compared to other alternative approaches. For instance, it is possible to directly inject the shell command to the guest OS to achieve the same goal (e.g., configure the guest OS), or directly inject the file system updates. However, command-line interfaces or configuration file interfaces are less stable when compared to the syscall interface. That is why eventually it leads to our R-syscall based approach.

Finally, our Syscall Dispatcher uses dynamic library interposition, and it ignores the syscall policy checking in the dynamic loader. Therefore, static linked native utilities cannot be executed in HYPERSHELL. Also, if there is a different loader whose syscall can be captured by library interposition, we have to design new techniques to differentiate the syscall policy for these syscalls. One possible solution is to add the call stack context in our policy check. In addition, while most of our design is OS-agnostic, we currently only demonstrate HYPERSHELL with the Linux kernel and we would like to test with other OSes such as Microsoft Windows. We leave these in our other future efforts.

7 Related Work

Our work is related to the virtual machine introspection (VMI) [19, 26] and VM management in the cloud. In this section, we review and compare them with HYPERSHELL.

Being a layer below of the OS, virtual machine gives new opportunities for VMI, which inspects and analyzes both the user level program and OS kernel states outside the machine itself. However, the key challenge in VMI lies in how to bridge the semantic gap. Over the past decade, many approaches have been proposed to address this problem, and these approaches can be classified into: debugger assisted (e.g., [19]), manual kernel data structure traversal (e.g., [24]), kernel source code analysis and customization (e.g., [8, 23]), in-VM kernel module assisted (e.g., [30, 34]), and binary code or execution context reuse (e.g., [21, 15, 35, 22, 16, 17, 18]). In this section, we will not go through and compare with each of these existing techniques, but rather compare with the most related ones as presented in Table 5.

To narrow the semantic gap, VIRTUOSO [15] made a first step showing that we can actually reuse the legacy binary code to automatically create VMI tools with the assistance from a human expert. The key idea is to first train each in-VM program (e.g., `ps`) and then translate the trained traces (essentially slices) into an independent introspection program running at the hypervisor layer. Inspired by VIRTUOSO, VMST [16] shows a dual-VM based, online kernel data redirection approach that addresses the limitations from the training (i.e., code coverage issue). Its key idea is to reuse the execution context of an inspection process in an SVM; when a kernel instruction accesses the kernel data of introspection interest, it redirects the data from the GVM to the SVM. Built atop VMST, EXTERIOR [18] demonstrates that it is feasible to build an external shell to perform the out-of-VM guest OS writable operations (e.g., for configuration) [28]. Similar to VIRTUOSO, both VMST and EXTERIOR do not need to trust the guest-OS kernel.

While EXTERIOR [18] has made an early attempt of building a hypervisor layer shell, it has a lot of constraints and is far from practical. Specifically, it has to first perform the guest OS fingerprinting [20], and then use the exact same version of the guest OS running in an SVM to introspect the kernel state of a GVM. Second, it can suffer from various failures and shortfalls when an introspection related syscall uses kernel synchronization primitives [16, 18]. Third, it is built atop a binary code translation based VM (e.g., QEMU [6]), which often has 10X-40X performance slowdown compared to the native execution (though recently HybridBridge [32] has improved the performance with one order of magnitude). Finally, it is mainly for introspection and has very limited functionality (e.g., it ignores the disk data including the swapped memory).

Process Implanting (PI) [21] shows that we can inject a process running into a GVM by reusing an existing process context. At a high level, HYPERSHELL does share some similarity regarding the process injection. However, PI has only limited functionality. For instance, it cannot directly copy a file from inside to outside. It also cannot observe the output from native software, unless rewriting the utility with hypercall (a para-virtualization approach that is not transparent to the guest OS). In addition, it requires the recompilation of the injected programs with static linking; in contrast, HYPERSHELL is transparent to both utility software and the guest OS.

Designed for process monitoring, process out-grafting (POG) [35] relocates a suspect process from a GVM to an SVM, and then uses a trusted security tool (e.g., `strace`) in the SVM to monitor the behavior of the suspect process. Unlike HYPERSHELL, which selectively redirects the syscall based on a transparent policy, all the syscalls of the suspect process are redirected from the SVM to the GVM. Therefore, POG does not face the challenges as in HYPERSHELL to differentiate the syscall redirection policy. Meanwhile, all the applications supported by POG can certainly be supported by HYPERSHELL, but not vice versa.

Designed for high performance computing, GEARS [22] shows that we can push certain VMM level virtual services for a guest into the guest itself. Through such a way, we can reduce the implementation complexity (since there is no semantic gap for in-VM programs) and increase the performance. At a high level, while GEARS and HYPERSHELL shares some similarity of using syscall interception and code injection techniques, the substantial difference is that GEARS is not a binary code reuse based approach, and it is not transparent to the in-VM programs and requires programmer's efforts to (re)develop the new software.

Most recently, concurrent to HYPERSHELL, ShadowExecution [38] also explores the concept of system call redirection and process injection. With a number of other security means such as process image protection (e.g., code and data integrity) and runtime execution protection (e.g., control flow integrity) of both guest OS kernel and the injected process, ShadowExecution shows that VMI tools can be built as in VMST. The difference compared to ShadowExecution is that HYPERSHELL is mainly designed for Cloud in-VM management, whereas ShadowExecution is mainly for security, though they both are based on the system call redirection concept.

Our work is also related to VM management in the cloud, such as VM cloning (e.g., [27]), VM migration (e.g., [10]), and VM replication (e.g., [12]). However, these management techniques treat each VM as a whole. In contrast, we aim to design programs to manage each guest OS at a fine grained level from our HYPERSHELL, much like the way we manage an OS in-VM.

8 Conclusion

We have presented the design, implementation, and evaluation of HYPERSHELL, a practical hypervisor layer shell for automated, uniformed, and centralized guest OS management. To overcome the semantic gap challenge, we introduce a reverse system call abstraction, and we show that this abstraction can be transparently implemented. Resulting from this, many of the legacy guest OS management utilities can be directly executed in HYPERSHELL. Our empirical evaluation with 101 native Linux utilities shows that we can use HYPERSHELL to manage a guest OS directly from the hypervisor layer without requiring any access to administrator's account. Regarding the performance, it has on average 2.73X slowdown for the tested utilities compared to their native in-VM execution, and less than 5% overhead to the guest OS kernel.

Systems	**Execution Context Reuse**	**wo/ Dual-VM Architecture**	**wo/ Identical Kernel**	**wo/ Trust to Guest Kernel**	**High Code Coverage**	**Fully Automated**	**Memory Introspection**	**Disk Introspection**	**Guest Management**	**Process Monitoring**
VIRTUOSO	✗	✓	✗	✓	✗	✗	✓	✗	✗	✗
VMST	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗
EXTERIOR	✓	✗	✗	✓	✓	✓	✓	✗	✓	✗
PI	✓	✓	✓	✗	✓	✓	✗	✗	✓	✗
POG	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓
GEARS	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓
HYPERSHELL	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓

Table 5: Comparison with the most related work.

Acknowledgment

We are grateful to Jon Howell and other anonymous reviewers for their insightful comments. We also thank Erick Bauman and Kenneth Miller for their valuable feedback on an early draft of this paper. This research was supported in part by a research gift from VMware Inc. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the VMware.

References

[1] Apachebench - apache http server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[2] Clamav, an open source (gpl) antivirus engine designed for detecting trojans, viruses, malware and other malicious threats. http://www.clamav.net/.

[3] dm-crypt, linux kernel device-mapper crypto target. http://code.google.com/p/cryptsetup/wiki/DMCrypt.

[4] Kernel based virtual machine. *http://www.linux-kvm.org*.

[5] memcached - a distributed memory object caching system. http://memcached.org/.

[6] QEMU: an open source processor emulator. *http://www.qemu.org/*.

[7] Skytap announces over one million virtual machines launched. http://www.skytap.com/news-events/press-releases/skytap-announces-over-one-million-virtual-machines-launched, January 2012.

[8] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)* (Chicago, IL, USA, 2009).

[9] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS'01)* (Napa, CA, USA, 2011).

[10] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)* (Boston, MA, USA, 2005).

[11] Clark, D. D. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles (SOSP'85)* (Orcas Island, Washington, USA, 1985).

[12] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)* (Berkeley, CA, USA, 2008).

[13] Curry, T. W. Profiling and tracing dynamic library usage via interposition. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1 (ATC'94)* (Boston, Massachusetts, USA, 1994).

[14] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)* (Alexandria, Virginia, USA, 2008).

[15] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32^{nd} IEEE Symposium on Security and Privacy (S&P'11)* (Oakland, CA, USA, 2011).

[16] Fu, Y., and Lin, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of 33^{rd} IEEE Symposium on Security and Privacy (S&P'12)* (San Francisco, CA, USA, 2012).

[17] Fu, Y., and Lin, Z. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. *ACM Trans. Inf. Syst. Secur. 16*, 2 (2013).

[18] Fu, Y., and Lin, Z. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments (VEE'13)* (Houston, TX, USA, 2013).

[19] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'03)* (San Diego, CA, USA, 2003).

[20] Gu, Y., Fu, Y., Prakash, A., Lin, Z., and Yin, H. Ossommelier: Memory-only operating system fingerprinting in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC'12)* (San Jose, CA, USA, 2012).

[21] Gu, Z., Deng, Z., Xu, D., and Jiang, X. Process implanting: A new active introspection framework for virtualization. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems (SRDS'11)* (Madrid, Spain, 2011).

[22] Hale, K. C., Xia, L., and Dinda, P. A. Shifting gears to enable guest-context virtual services. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC'12)* (San Jose, California, USA, 2012).

[23] Hofmann, O. S., Dunn, A. M., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with osck. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS'11)* (Newport Beach, California, USA, 2011).

[24] Jiang, X., Wang, X., and Xu, D. 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 (CCS'07)* (Alexandria, Virginia, USA, 2007).

[25] Jiang, X., Wangz, H. J., Xu, D., and Wang, Y.-M. Randsys: Thwarting code injection attacks with system service interface randomization. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07)* (Beijing, China, 2007).

[26] Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP'05)* (Brighton, United Kingdom, 2005).

[27] Lagar-Cavilla, H. A., Whitney, J. A., Scannell, A. M., Patchin, P., Rumble, S. M., de Lara, E., Brudno, M., and Satyanarayanan, M. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys'09)* (Nuremberg, Germany, 2009).

[28] Lin, Z. Toward guest os writable virtual machine introspection. *VMware Technical Journal 2*, 2 (2013).

[29] Newsome, J., and Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)* (San Diego, CA, USA, 2005).

[30] Payne, B. D., Carbone, M., Sharif, M. I., and Lee, W. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of 2008 IEEE Symposium on Security and Privacy (S&P'08)* (Oakland, CA, May 2008).

[31] Pfoh, J., Schneider, C., and Eckert, C. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, vol. 7038 of *Lecture Notes in Computer Science*. Springer, Nov. 2011, pp. 96–112.

[32] Saberi, A., Fu, Y., and Lin, Z. Hybrid-bridge: Efficiently bridging the semantic-gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)* (San Diego, CA, USA, 2014).

[33] Seshadri, A., Luk, M., Qu, N., and Perrig, A. SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'07)* (Stevenson, WA, USA, 2007).

[34] Sharif, M. I., 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 (CCS'09)* (Chicago, Illinois, USA, 2009).

[35] Srinivasan, D., Wang, Z., Jiang, X., and Xu, D. 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 (CCS'11)* (Chicago, Illinois, USA, 2011).

[36] Tsai, T. K., and Singh, N. Libsafe: Transparent system-wide protection against buffer overflow attacks. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)* (Washington, DC, USA, 2002).

[37] Wang, Z., and Jiang, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy (S&P'10)* (Okaland, CA, USA, 2010).

[38] Wu, R., Chen, P., Liu, P., and Mao, B. System call redirection: A practical approach to meeting real-world vmi needs. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN'14)* (Atlanta, Georgia, June 2014).

[39] Zhang, F., Chen, J., Chen, H., and Zang, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (Cascais, Portugal, 2011).