Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection

Yangchun Fu
Department of Computer Science
The University of Texas at Dallas
yangchun.fu@utdallas.edu

Zhiqiang Lin
Department of Computer Science
The University of Texas at Dallas
zhiqiang.lin@utdallas.edu

***Abstract*—It is generally believed to be a tedious, time-consuming, and error-prone process to develop a virtual machine introspection (VMI) tool manually because of the semantic gap. Recent advances in Virtuoso show that we can largely *narrow* the semantic gap. But it still cannot completely automate the VMI tool generation. In this paper, we present VMST, an entirely new technique that can automatically bridge the semantic gap and generate the VMI tools. The key idea is that, through system wide instruction monitoring, we can automatically identify the introspection related data and redirect these data accesses to the in-guest kernel memory. VMST offers a number of new features and capabilities. Particularly, it automatically enables an in-guest inspection program to become an introspection program. We have tested VMST over 15 commonly used utilities on top of 20 different Linux kernels. The experimental results show that our technique is general (largely OS-agnostic), and it introduces 9.3X overhead on average for the introspected program compared to the native non-redirected one.** [1]

I. Introduction

Motivation Virtual Machine Introspection (VMI) [1] pulls the in-guest OS state to the outside virtual machine monitor (VMM), thereby offering an additional layer of isolation and opening new opportunities for security, reliability, and administration [2]. For example, in recent years we have witnessed a widespread adoption of VMI in intrusion detection (e.g., [1], [3], [4]), malware analysis (e.g., [5], [6]), process monitoring (e.g., [7]), and memory forensics (e.g, [8]).

However, when performing the introspection, we often have to interpret the in-guest hardware-layer state such as processors, physical memory, and devices at the outside VMM layer. Such interpretation typically requires detailed, up-to-date knowledge of the internal OS kernel workings. For example, to introspect the `pid` of a running process in a Linux kernel, one has to traverse the corresponding `task_struct` to fetch its `pid` field. Acquiring such knowledge is often tedious and time-consuming even for an open source OS. For a closed source OS, one may have to reverse engineer the internal kernel workings for the introspection, which may be error-prone.

The difficulty in interpreting the low level bits and bytes into a high level semantic state of an in-guest OS is called the semantic gap [9]. An early solution [1] to bridging the semantic gap leverages the Linux `crash` utility (a kernel dump analysis tool), but this approach requires the kernel to be recompiled with the debugging symbols. The other approach involves locating, traversing, and interpreting known structures of the in-guest memory. While the latter approach has been widely adopted (e.g., [5], [10], [11]), it relies on a manual effort to locate the in-guest kernel data and develop the in-guest semantic-equivalent code to introspect. Moreover, such a manual process has to be repeated for different kernels, which may suffer from frequent changes due to the new releases or patches. Furthermore, it may also introduce vulnerabilities for attackers to evade these hand-built introspection tools [12], [13].

To ease the burden of the manual process and develop more secure VMI tools, most recently Dolan-Gavitt et al. [13] presented Virtuoso, a system for *automatically* generating introspection programs *with minimum human effort*. The key idea of Virtuoso is to create introspection programs from the traces of the in-guest trusted programs. More specifically, given an introspection functionality (e.g., list all processes), Virtuoso will train and trace the system wide execution of the in-guest programs (e.g., `ps`) by *an expert*, automatically identify the instructions necessary in accomplishing this functionality, and finally generate the introspection code that reproduces the same behavior of the in-guest programs.

While Virtuoso has made a large step in narrowing the semantic gap, as acknowledged by the authors [13], one of its fundamental limitations is that it is only able, due to the nature of dynamic analysis, to reproduce introspection code that has been executed and trained. Meanwhile, it is still not fully automated and requires the intervention from a human expert. Thus, in this paper we present VM-Space Traveler (VMST for short), a new system to *automatically* bridge the semantic gap and generate the VMI tools.

Key ideas and techniques The key insight of our technique is that a program $\mathcal{P}(x)$ is often composed of *code* $\mathcal{P}$ and *data* x; for the same program, $\mathcal{P}$ is usually identical across different machines, and the only difference is the run-time consumed data x. Normally, for a machine A, its $\mathcal{P}$ always consumes

[1] Authors are in alphabetic order.

the x in A. Thus, if we can make $\mathcal{P}$ (suppose an inspection program such as `ps`) in A transparently consume the data y in B (i.e., without the awareness that y comes from B), then we automatically generate an introspection program $\mathcal{P}'$ such that $\mathcal{P}'(x)$=$\mathcal{P}(y)$.

However, it turns out to be a challenging task to enable $\mathcal{P}'(x)$ using $\mathcal{P}(y)$ when y is kernel data. Note that *an introspection program usually inspects kernel data*. We cannot simply *redirect*[2] all kernel memory access because kernel code may get redirected too, but in-guest kernel code is usually untrusted. Meanwhile, not all kernel data can be redirected. For example, an interrupt handler expects to read some hardware states, but after the redirection it may receive an inconsistent state leading to kernel panics. Also, data in the kernel stack cannot be redirected, otherwise kernel control flow will be disrupted. As such, we have to identify where the redirectable data is and only redirect introspection related data. To this end, we have developed a number of *OS-agnostic* enabling techniques, including *syscall execution context identification*, *redirectable data identification*, and *kernel data redirection* at the VMM layer.

Usage scenarios VMST is designed with transparency to the guest OS in mind, and it has achieved nearly full transparency against an in-guest OS kernel. For example, without any modification, VMST directly supports a number of most recent released Linux kernels. Note that in this paper, we mainly focus on the in-guest Linux/UNIX OS. Meanwhile, when using VMST for introspection, for a particular OS, end-users will only need to install the corresponding trusted version of the guest OS in the VM shipped with our VMST, and attach (or mount) the in-guest memory. The in-guest memory could be a live memory for VMI or a memory snapshot for memory forensic analysis. Subsequently, end-users can use a variety of OS utilities (e.g., `ps`, `lsmod`) to inspect the state of the in-guest OS.

New features VMST offers a number of new features and capabilities. In particular, (1) it enables the automatic generation of secure introspection tools. Such security is achieved by the nature of VMI [1] and the technique of our automatic tool generation. Similar to Virtuoso [13], our VMI-tools are literally generated from the trusted OS code as well as the widely used and tested utilities without any modification, and hence our VMI tools are more secure than many other manually created ones. (2) Our tools are also more reliable than tools generated from Virtuoso, because Virtuoso cannot guarantee the *path coverage* in their training, yet we have retained all the code. (3) Meanwhile, we directly generate a large volume of VMI tools for free, but Virtuoso has to train each program one by one to get the new VMI tools. (4) In addition, VMST allows the user-level programmers to develop new user-level programs *natively* to monitor system status for the introspection. (5) Further, it also allows the kernel-level programmers to develop native device drivers for inspecting the kernel states for the introspection. In short, *VMST automatically enables an in-guest legacy inspection program to become an introspection program, without any involvement from end-users and developers*.

We have implemented our VMST, and tested it over a variety of Linux distributions including Ubuntu, Redhat, Debian, and OpenSuse, with 20 different kernel versions to evaluate the generality of our approach. Our experimental results show that (1) redirecting the kernel data access is entirely feasible for automatic VMI tool generation, (2) such an approach directly supports many of the inspection utilities in the guest OS, (3) allows end-users to easily develop new VMI tools by reusing the native APIs, or the native device drives (or kernel modules), and (4) finally it introduces reasonable performance overhead (with 9.3X on average) for the introspection applications.

Contributions In summary, this paper makes the following contributions.

- We introduce a new binary code reuse technique. Unlike existing techniques that extract the code outside [13]–[15], we retain the code in original form but dynamically instrument the code and redirect the data access to achieve our goal.
- We show such code reuse is truly feasible in the VMI domain and demonstrate that end-users can automatically get a variety of VMI tools without any knowledge of the OS kernel internals.
- We devise a set of novel OS-agnostic enabling techniques, including *syscall execution context identification* (§III), *redirectable data identification* (§IV), and *kernel data redirection* (§V), to achieve nearly the full transparency of our techniques against an OS kernel.
- We have implemented these techniques into our prototype VMST (§VI), which automatically bridges the semantic gap in VMI. Meanwhile, we have applied it to automatically generate a number of VMI tools for the Linux OS as shown in our experiment (§VII).

II. Problem Statement and System Overview

A. Problem Statement

Observations The goal of our VMST is to bridge the semantic-gap and enable automated VMI tool generation. The basic observation is that many introspection tools are mainly used to *query* the guest-OS state, e.g., listing all the running processes, opened files, installed drivers, and connected sockets. In fact, these program-logics $\mathcal{P}$ have been shipped in an OS kernel with the corresponding user level utilities. Thus, instead of building an introspection tool $\mathcal{P}'$ from scratch, we can actually reuse the user level as well as OS kernel code $\mathcal{P}$ to automatically implement $\mathcal{P}'$.

[2]"Redirect" means intercept and direct the access to the data traveled from other machines, which also actually leads to the name of our VM Space Traveler.

```
 1 execve("./getpid", ["./getpid"], [/* 38 vars */]) = 0
 2 brk(0)                                = 0x83b8000
 3 access("/etc/ld.so.nohwcap", F_OK)    = -1 ENOENT
 4 mmap2(NULL, 8192, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x4001d000
 5 access("/etc/ld.so.preload", R_OK)    = -1 ENOENT
 6 open("/etc/ld.so.cache", O_RDONLY)    = 3
 7 fstat64(3, {st_mode=S_IFREG|0644, st_size=50205, ...}) = 0
 8 mmap2(NULL, 50205, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001f000
 9 close(3)                              = 0
10 access("/etc/ld.so.nohwcap", F_OK)    = -1 ENOENT
11 open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
12 read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\340g\1"..., 512) = 512
13 fstat64(3, {st_mode=S_IFREG|0755, st_size=1425800, ...}) = 0
14 mmap2(NULL, 1431152, PROT_READ|PROT_EXEC, ..., 0) = 0x4002c000
15 mmap2(0x40184000, 12288, PROT_READ|PROT_WRITE, ..., 0x158) = 0x40184000
16 mmap2(0x40187000, 9840, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x40187000
17 close(3)                              = 0
18 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x4018a000
19 set_thread_area({entry_number:-1 -> 6,...}) = 0
20 mprotect(0x40184000, 8192, PROT_READ)  = 0
21 mprotect(0x4001b000, 4096, PROT_READ)  = 0
22 munmap(0x4001f000, 50205)              = 0
23 getpid()                               = 13849
24 fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
25 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x4001f000
26 write(1, "pid=13849\n", 10)           = 10
27 exit_group(0)                         = ?
```

Figure 1. System level behavior (in terms of syscall trace) of a typical user level `getpid` program.

Let us take a specific example to better understand our observation. Without introspection, normally when we run a utility program to inspect an OS state, e.g., get a current process ID (`getpid`) from a Linux kernel, as shown in Fig. 1, the OS kernel will execute a series of system calls such as create a new process (`execve`), set the end of the data segment (`brk`), check (`access`) the library (e.g., `ld.so.nohwcap`), `map` the standard shared library (`open`, `fstat`, `map`, `map2`), execute the `getpid` system call (syscall for short), output the result (`write`), and exit the process (`exit_group`).

With introspection, we can see that *in order to fully reuse the OS as well as user level program code $\mathcal{P}$, we should redirect the data read that is only related to the desired introspection functionality*. In our `getpid` example, it should be the data x within the `getpid` system call. For data in user space and other irrelevant kernel space, there should be no redirection and we should keep both kernel and other user processes running correctly.

Problem Definition As such, the central problem in our system is how to (1) automatically identify the introspection execution context, (2) automatically identify the data x in kernel code that is related to the introspection, (3) automatically redirect x, and (4) keep all the processes running, at the VMM layer.

Challenges However, this is a challenging task in reality since the OS kernel is designed to manage computer hardware resources (e.g., memory, disk, I/O) and provide common services (i.e., syscalls) for application software, it has a very complicated control flow and data access.

In particular, the kernel usually contains many routines for resource (e.g., page tables) management, interrupt and exception handling (e.g., timer, keyboard, page fault), context switch, and syscall service. When serving a system call, an interrupt could occur, a page fault (an exception) could occur, and a context switch could occur. Obviously, we do not want to redirect the kernel data access in context switches, page fault handlers, or interrupt service routines. Also, we do not want to redirect the data access in the execution context of any other processes.

Data access may be *code* reads or *data* reads. One of the advantages for VMI is that attackers usually cannot modify the introspection code. Thus, we do not want to load any code from an untrusted guest and we have to differentiate kernel code and data. Also, data could be in kernel global, heap, or stack regions. Obviously, we cannot redirect the kernel stack read, otherwise it will lead to a kernel crash (because of control data such as return addresses in the stack). We have to thus identify where the redirectable data is. Moreover, when redirecting the data, we have to perform the virtual to physical address translation. Otherwise it will not be able to find the correct in-guest memory data. Further, we have to perform copy on write (COW) of the redirected data to ensure there is no side effect of the in-guest memory.

B. Scope and Assumptions

Being OS-agnostic as much as possible is one of our design goals. That is, when designing our VMST, we should use the general knowledge from the OS design principles [16], [17], and we would avoid, if possible, hard coding any specific kernel addresses (otherwise it would be too kernel specific). However, we cannot blindly support all OSes because we rely on some OS knowledge – particularly the system call interfaces and conversions (i.e., the syscall number, argument, return value), which are OS-gnostic. Therefore, in this paper, we focus on Linux/UNIX OS, on top of the widely used x86 architecture.

Meanwhile, we assume our own VMM can intercept the system-wide instructions, because we need to dynamically instrument the instructions and redirect the data access if instructions are introspection related.

In addition, similar to Virtuoso [13], we also assume end-users have a trusted in-guest OS copy. The trusted copy will be installed in our VMST, and executed along with the utilities to provide the introspection. The reason is that without the trusted copy, when we redirect the introspection data, it will lead to a wrong in-guest memory address.

C. System Overview

An overview of our VMST is presented in Fig. 2. For any untrusted OS running in a product-VM, suppose end-users want to perform its introspection. They only need to install the corresponding trusted version of the in-guest OS on top of our own secure-VM (shipped in our VMST) and invoke the commonly used standard utility programs *without any modification*. They do not have to perform any manual effort to understand (or reverse engineer) the OS kernel and write the introspection program. Meanwhile, if they do want to customize an introspection program, they can develop these programs

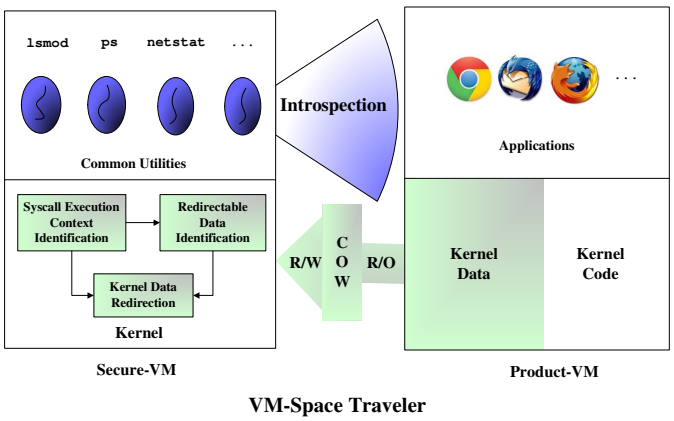

Figure 2. An architecture overview of our VMST.

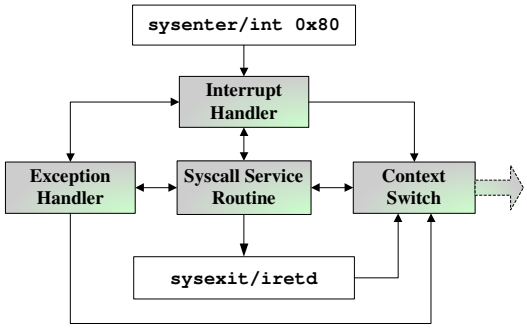

Figure 3. Typical kernel control flow when serving a system call.

natively (e.g., invoking native APIs/system calls) without worrying about any OS kernel internals. Note that the product-VM and our secure-VM in Fig. 2 can be completely different, and VMST is only bounded with our own secure-VM and is transparent to the guest product-VM, which can be a VM running on top of XEN/KVM/VMWare/VirtualBox/VirtualPC/QEMU, or even be a physical machine as long as we have its memory access.

There are three key techniques in our VMST: (1) *syscall execution context identification*, (2) *redirectable data identification*, and (3) *kernel data redirection*. *Syscall execution context identification* is used to identify only the system call execution context relevant to the introspection, and ensure the *kernel data redirection* only redirects the data x in the context of interest. *Redirectable data identification* pinpoints only the x (and its closure) that needs to be redirected under the context informed by the *syscall execution context identification*. *Kernel data redirection* performs the final redirection of x. Copy-on-write (COW) will be performed if there is any data write on x. In the next three sections, we present the detailed design of each technique of our VMST.

III. SysCall Execution Context Identification

Because of the complicated kernel control flow, we have to first identify the precise system call execution context, in which we perform the redirection for the necessary system call. When an introspection program is running, there are two spaces: user space and kernel space. In the x86 architecture, each process (and kernel thread, which essentially is a process) has a unique CR3 value (to locate the page directory). We could hence isolate the corresponding kernel as well as user space by inspecting the CR3 value.

Then the question is how to acquire the right CR3 value of the monitored process, given only the name of an introspected process. Note that our secure-VM needs to be transparent to the in-guest OS, and we should not traverse any specific `task_struct` to get the process name field even though we could. This turns out to be a challenging task, but before describing our solution let us first walk through what we could do at the VMM layer.

It is trivial to identify the syscall entry point. In Linux, user level programs invoke `int 0x80` or `sysenter` instructions to enter the kernel space. Therefore, by intercepting these two instructions at the VMM layer, it suffices to identify the beginning of a syscall execution context. However, the real difficulty is identifying the exit point of a system call. A naive approach may directly intercept the `sysexit` or `iret` instruction to determine the exit point. However, this approach would not work because of interrupt and exception handling as well as the possibility of a context switch happening during the execution of a system call.

As illustrated in Fig. 3, at a high level, when serving a system call, an interrupt could occur and kernel control flow may go to the interrupt handler. An exception such as a page fault (when a system call routine accesses some unmapped memory region of the process) could also occur. Also, at the system call exit point or during the service of a system call, a context switch could occur (e.g., a process has used over its time slice). A context switch could also occur in the interrupt and exception handler.

Fortunately, since our secure-VM virtualizes all hardware resources (e.g., through emulation), we could easily observe and control these hardware states including the interrupt and timer at the VMM layer, as long as we can keep our own introspection process and kernel running correctly. More specifically, we use the following approaches to handle interrupt, exception, and context switch.

Interrupt and Exception Handling Generally, there are two kinds of interrupts: synchronous interrupts generated by the CPU while executing instructions, and asynchronous interrupts generated by the other hardware devices at arbitrary times. In the x86 architecture, synchronous interrupts are designated as exceptions, and asynchronous interrupts as interrupts.

When an interrupt occurs (if it is not disabled), whether it is an exception or a hardware interrupt, it will first issue an interrupt vector number in the hardware interrupt controller. This

controller will pick up the corresponding interrupt handler, to which the kernel control flow will transfer. By monitoring this controller and tracking the interrupt number, we can differentiate system calls (`int 0x80`) and other interrupt or exception handlers, and we can track the beginning of an interrupt service.

In our design, right before the interrupt handler gets executed (not the system call), we will set a global flag to indicate data in the current execution context is not redirectable (as the kernel control path will be in the interrupt context). Also, as an interrupt always ends with an `iret` instruction, we are able to track the end of an interrupt. However, the interrupt could be nested. That is, when serving an interrupt, the kernel could suspend the execution of the current interrupt in response to a higher priority interrupt. Therefore, we use a stack data structure to track the execution status of the interrupt handler. In particular, we use a `counter` to simulate the stack. Whenever an interrupt other than a system call occurs, we increase the `counter`; when an `iret` instruction executes, we decrease the `counter`. If the `counter` becomes zero, it means all the interrupt service has finished. Note that the `counter` is only updated when the execution context is within the introspection process, and initially it is zero.

Another possible design is to track the next program counter (PC) in the system call routine to determine the end of an interrupt, since after an interrupt handler finishes, it will transfer the kernel control flow back to the system call routine (the next PC). However, we observe that such a simple design will have a problem. For example, in Linux kernel 2.6.32-rc8 (the working kernel we used during the design of VMST), the system call routine will call the `cond_reschedule` function to determine whether a context switch is needed (in particular checking the `_TIF_NEED_RESCHED` flag in the kernel stack), and it is also called in the interrupt and exception handler routine [16]. If an interrupt occurs during the execution of `cond_schedule` in the system call context, this approach will mistakenly identify the end of an interrupt handler. Therefore, using kernel call stack and PC together could be a feasible approach as they precisely define the execution context.

The stack-based approach above is able to determine the interrupt handler context, or more specifically, the top half of an interrupt. However, one may worry about how we identify the bottom half of an interrupt as most UNIX systems, including Linux, divide the work of processing interrupts into two parts or halves [16]. Fortunately, the execution of the bottom half of an interrupt is usually bounded with a working queue and will be scheduled by a context switch that is discussed below.

Context Switch Controlling Context switches are one of the key techniques to allow multiple processes to share a single CPU. Basically, it is a procedure of storing and restoring the context state (including CPU registers, CR3, etc.) of a process (or a kernel thread) such that execution can be resumed from the same point at a later time [16], [17].

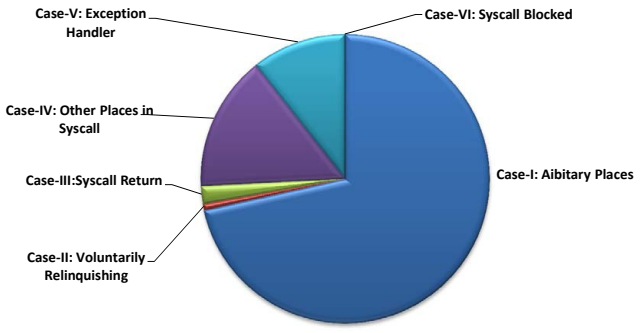

Figure 4. Statistics of context switch when running `ps`

A context switch could occur in a variety of cases in Linux/UNIX including:

- Case-I: arbitrary places, when an asynchronous interrupt happens (could be timer) and the process has used its CPU time slice (preempted);
- Case-II: when a process voluntarily relinquishes their time in the CPU (e.g., invoking `sleep`, `waitpid` or `exit` system call);
- Case-III: when a system call is about to return;
- Case-IV: other system call subroutine places (besides system call return point), in which the kernel pro-actively checks whether a context switch is needed;
- Case-V: in exception (e.g., page fault) handler; or
- Case-VI: when a system call gets blocked

During our design, we profiled one execution of the `ps` command, and the statistics of where a context switch happens is reported in Fig. 4. Among these six cases, four (Case-I, Case-III, Case-IV, and Case-V that account for 99.3% in our profile) are triggered due to the time slice. Case-II (0.7% because of the `exit` syscall) is not of concern because the entry of the `sleep` or `waitpid` system call can be detected and the redirection in these system calls' execution context, including any other possible context switches, can be disabled. After context switching to other process, it will switch back to these system calls execution context and we are able to detect it by just looking at the CR3. Also, an introspection program typically will not invoke the blocking-mode system calls (Case-VI). Meanwhile Case-V can be detected by our exception handler. Therefore, one of the key ideas of VMST is that as long as we can keep the running introspection process always owning the CPU, we can prevent context switch happening until the monitored process exits, or we can allow context switch as long as we can pro-actively detect it (such as the case of `sleep` system call). Note that at VMM layer we own the hardware and we can modify the timer such that the process will not feel it has gone beyond its time slice.

Also, Virtuoso mentioned an approach to disable the context switch by running the training programs with a higher priority (e.g., using `start /realtime` on Windows and `chrt` on

Linux) [13]. However, this approach alone will not work in VMST as the determination of a context switch will still be executed, and the data access in the determination context will hence be redirected (that is not desirable). Also, it might not always be true that a user-level process can get the highest priority.

Acquiring the right CR3 Now we are ready to describe how we acquire the right CR3 when only given a to-be-executed process name. Notice in Fig. 1, when a process is executed, it will first call the `execve` system call. By inspecting the value in `ebx` at the moment when this system call gets executed, we are able to determine the process name. However, the value of CR3 when executing this system call belongs to its parent process. During the execution of this system call, it will release almost all resources belonging to the old process (`flush_old_exec`) and update the CR3, which is the right moment to acquire the CR3 for our monitored process. Therefore, by monitoring the update of CR3 (a `mov` instruction) in `execve` syscall context, we are able to get our desired value because there is no other CR3 update since we disabled context switching.

There is also an alternative approach to monitor all CR3 (essentially `pgd`) values from the boot of our secure-VM and detect the newly used CR3 since a new CR3 certainly belongs to a new process, as the above design is a bit OS-gnostic to Linux kernel (e.g., what happens if other OSes' "`execve`" does not update the CR3?). For this approach, we need to track the life time of a `pgd`. Our instrumentation is to maintain a map between the CR3 and the process. Whenever a process dies (detected through such as `exit_group` system call as noticed in Fig. 1), we remove its CR3 from our map. As such, we are able to determine whether a given CR3 belongs to a new process.

By tracking the interrupt service routine and disabling the timer for context switches, our *syscall context identification* is able to largely identify the system call execution context of the monitored process. However, it still does not fully isolate all the syscall service routines. For example: the `cond_schedule` function will be called in many places to determine whether a context switch is needed, including all of the system call exit points. Obviously, we will redirect the data access of this function if we do not have any other techniques to remedy this. We cannot white list this specific function (OS-gnostic, though that is a viable option). Fortunately, our second technique, *redirectable data identification*, solves this problem and will automatically tell the data in such a function is not redirectable.

IV. Redirectable Data Identification

The goal of our *redirectable data identification* is to identify the kernel data x that can be redirected to the in-guest memory. Thus, we have to first determine what kind of data should be redirected. Normally, when writing an introspection program manually, we traverse the kernel memory starting from the global memory location (exported in the `system.map` file for instance) to reach other memory locations including the kernel heap by following pointers.

As such, one of the intuitive approaches would be to track and redirect the data x that is from kernel global variables and derived from global variables through pointer dereference and pointer arithmetic instructions. Note that at the instruction level, we can easily identify the kernel global variables, which are usually literal values after the kernel is compiled and identical for the same OS version for a given global address. By dynamically instrumenting each kernel instruction and checking whether there is any data transitively derived from global variables (a variant of widely used taint analysis [18]–[21]), we are able to identify them. Our early design naturally adopted this approach. However, we found another design that is simpler and can save more shadow memory space for the data flow tracking.

Since it is a boolean function to determine whether some data is redirectable, instead of tracking all the redirectable data, we can track which data is unredirectable. Certainly, it is the data dereferenced from stack variables or derived from them because some kernel stack variables manage the kernel control path, and they vary from machine to machine even for an identical OS at a different time.

Though our redirectable data identification is a variant of taint analysis, there are still significant differences. Below we sketch our design and highlight them.

Shadow Memory Similar to all taint analysis, we need a shadow memory to store the taint bits for memory and CPU registers. We keep taint information for both memory and registers at byte granularity by using only one bit to indicate whether they are redirectable (with value 1) or not (with value 0).

However, we have to use three pieces of shadow memory, shadow $\mathcal{S}$ and shadow $\mathcal{V}$ for the memory data and shadow $\mathcal{R}$ for registers. $\mathcal{S}$ is used to track the unredirectable stack addresses, and $\mathcal{V}$ and $\mathcal{R}$ are used to track whether the value stored in the corresponding memory addresses or registers when used as a memory address needs to be redirected. Considering our working example shown in Fig. 5, if we only have $\mathcal{S}$, for the instruction at line 17 `mov 0xc(%ebp),%ecx`, we will move the taint bit 0 to `ecx`. When the kernel subsequently dereferences a memory address pointed to by `ecx`, we will not redirect it. However, we should redirect it as this address is actually a global memory address. Therefore, we will keep taint information for both the stack address and its value because of pointers.

Taint Source Right before the introspection process enters the first monitored system call, we initialize the taint bits for the shadow state. For $\mathcal{R}$, all are initialized with 0, as the parameters passed from the user space are local to our secure-VM. For $\mathcal{S}$ and $\mathcal{V}$, the taint bits are allocated and updated on demand

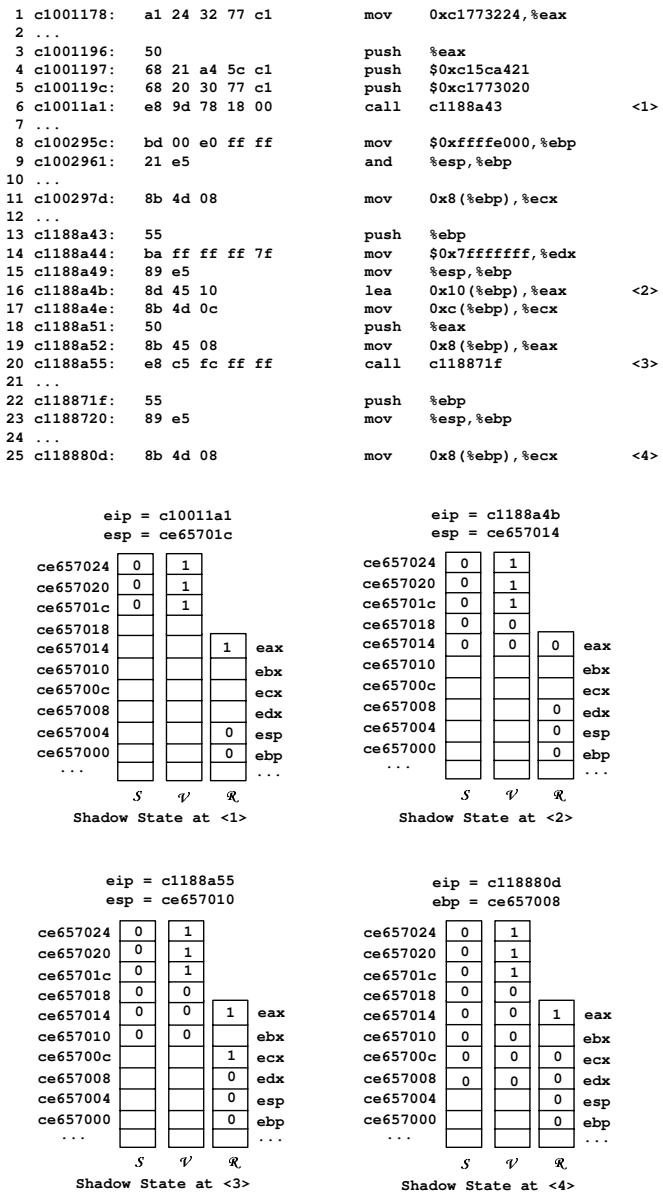

Figure 5. Shadow memory state of our working example code.

when the kernel uses the corresponding addresses.

The taint bit for the `esp` register is always 0. When loading a global memory address (a literal value that falls into kernel memory address space), the taint information for the corresponding register or memory will be set to 1. Some special instructions (e.g., `xor eax,eax`, `sub eax,eax`) will reset register value, and consequently we will set their taint bits with 0.

Propagation Policy The propagation policy determines how we update the shadow state. Similar to all other taint analyses, based on the instruction semantics, we update it. However, as we have three pieces of shadow memory (i.e., $\mathcal{S}$, $\mathcal{V}$, and $\mathcal{R}$), we have significantly different policies.

In particular, for $\mathcal{S}$, we always update its shadow bit with 0 whenever we encounter a stack address. We can regard $\mathcal{S}$ as a bookkeeping of all the exercised stack address. Later on, when dereferencing a memory address, we will query $\mathcal{S}$ about whether we have seen such an address before. *The taint-bit value in $\mathcal{S}$ (which is always 0) is not involved in any data propagation.*

In fact, we could eliminate $\mathcal{S}$ because, in practice, nearly all the stack addresses (involved in an x86 instruction) are computed (directly or indirectly) from `esp`. For example, as shown in the last two instructions (line 23-25) of our working example, we can easily infer `0x8(%ebp)` is a stack address, and we do not need to query any $\mathcal{S}$. The main reason we keep it is to make sure that the stack address will not be redirected. For example, it may have an instruction that actually has a stack address but does not use `esp` (or its derivation such as `ebp`) in certain context for address computation. In this case, our system will fail, though we have not encountered such a case in practice.

For $\mathcal{V}$ and $\mathcal{R}$, we use the following policies.

- **Data Movement Instruction** For one directional data movement $A \rightarrow B$, such as `mov/movsb/movsd`, `push`, and `pop`, we update the corresponding $\mathcal{R}(B)$ or $\mathcal{V}(B)$, with the taint bit in $\mathcal{R}(A)$ or $\mathcal{V}(A)$. For data exchange instructions $A \leftrightarrow B$, such as `xchg`, `xadd` (add and exchange), we update shadow state for both operand. Note `lea` may be a special case of "data movement". It does not exactly load any data from memory, but it may load a memory address. Therefore, we will check if the source operand generates a stack address, and if so we will update $\mathcal{V}$ or $\mathcal{R}$ of the destination operand with 0. For example, at line 16, it loads a stack address to `eax`, and we will consequently update $\mathcal{R}$(`eax`) with 0.
- **Data Arithmetic Instruction** As usual, for data arithmetic instructions such as `add`, `sub`, `or`, we should update shadow state by ORing the taint bit of the two operands. However, this is only true for operands that are both global and heap addresses as well as their propagations. Note that if one of the operands in these instructions is a literal value but not within kernel address space, there is no need to update any shadow state. If either of the operands is stack address related, we will update the taint bit with 0. Considering the instructions in line 8-11, we will first taint `ebp` with 1 as `0xffffe000` is a literal and within kernel address space; at line 9 when we execute `and %esp,%ebp`, because the taint bit for `esp` is always 0, we will get the new taint bit for `ebp` as 1; next at line 11 when we dereference memory `0x8(%ebp)`, we will redirect it, which is wrong. Therefore, the stack address will hijack the normal propagation policy and clear the operand taint.
- **Other Instructions** A large body of instructions do not contribute to any taint propagation, such as `nop`, `jmp`,

jcc, test, etc. For them, we will only check whether any memory address involved in these instructions needs to be redirected.

One may wonder whether we could get rid of this complicated data flow analysis scheme, given that our secure-VM already intercepts each instruction and knows the stack base address when entering the kernel; it should be straightforward to check if a memory access is on the secure-VM kernel stack. However, there is a big issue that an in-guest kernel heap address could also be in the range of the secure-VM stack address, because there is no explicit boundary between kernel stack and kernel heap. In fact, a process' kernel stack is dynamically allocated when a process is created. As such, we have to track the flow; otherwise, we cannot differentiate whether a given address is in a secure-VM kernel stack or in-guest heap.

V. Kernel Data Redirection

Having been able to identify the syscall execution context and pinpoint the data x that needs to be redirected, in this section we present how we redirect the kernel memory access. As discussed in §II that not all system calls need to be redirected, we first describe our syscall redirection policy in §V-A. Next we discuss how we perform the virtual to physical address translation including COW handling in §V-B. Finally, we present our redirection algorithm in §V-C.

A. System Call Redirection Policy

Back in the syscall trace of our getpid process (Fig. 1), we noticed our syscall redirection policy has to be fine-grained. That is, based on the semantics of each system call, we decide whether we will redirect the data access during the execution. As such, we have to *systematically* examine all the system calls. This has been widely studied in security literature, especially in intrusion detection (e.g., [12], [22], [23]).

System calls in general can be classified into the following categories according to a comprehensive study made by Sekar [24]: file access (e.g., open, read, write), network access (e.g., send/recv), message queues (e.g., msgctl), shared memory (e.g., shmat), file descriptor operations (e.g., dup, fcntl), time-related (e.g,. getitimer/setitimer), process control related (e.g., execve, brk, getpid), and other system-wide functionality related including accounting and quota (e.g., acct).

In our introspection settings, as we are interested in pulling the guest OS state outside, (1) system calls dealing with retrieving (i.e., get) the status of the system and (2) system calls related to file access are of particular interest. Specifically, our introspected system calls are summarized in Table I. We are interested in the file access related system call because of the proc files in Linux/UNIX. Note that the proc file system is a special file system which provides a more standardized way for dynamically accessing kernel state. Actually, standard, important utility programs such as ps, lsmod, netstat all read proc files to inspect the kernel status. Also, for disk files, there is no redirection (because VMI largely deals with memory), and we have to differentiate them by tracking the file descriptors. To this end, we maintain a file descriptor mapping whenever the introspected process opens a file, and by checking the parameters we differentiate whether the opened file is a proc file.

Category	System Calls
State Query	get(p|t|u|g|eu|eg|pp|pg|resu|resg)id getrusage, getrlimit,sgetmask, capget gettimeofday,getgroups,getpriority getitimer,get_kernel_syms, getdents getcwd,ugetrlimit,timer_gettime, timer_getoverrun,clock_gettime clock_getres,get_mempolicy, getcpu
File System	open, fstat, stat, lstat, statfs fstatfs, oldlstat, ustat, lseek, _llseek read, readlink, readv, readdir

Table I
Introspected System call in our VMST.

It is worthy to note that nearly all of our key techniques in VMST are OS-agnostic (our design goal). However, the system call redirection policy, as described, appears to be OS-gnostic. That is, it requires the specific knowledge of each syscall conversion and the semantics for a particular OS. As such, to support other systems such as Microsoft Windows, we need to scrutinize each Windows system call to determine whether they are redirectable. Once we have this knowledge, it is trivial to introspect them. For instance, as a proof-of-concept, we successfully introspected a Windows-XP (SP2) process ID by enabling our VMST redirecting the system call NtQueryInformationProcess (syscall number 0x9a) and disabling the stack redirection; meanwhile using the alternative approach to track the new CR3 value for the introspection process.

B. Virtual to Physical Address Translation

When dynamically instrumenting each kernel instruction, we are only able to observe the virtual address. If a given address is redirectable, we have to identify its in-guest physical address. That is, we have to perform the MMU level virtual to physical (V2P) address translation.

To this end, we design a shadow TLB (STLB) and shadow CR3 (SCR3) in our secure-VM, which will be used in our introspection process during address translation if we need to redirect a given address α. SCR3 is initialized with the guest CR3 value at the moment of introspection, and is used for kernel memory address *iff* α needs to be redirected, and similarly for STLB.

Meanwhile, as discussed earlier in §II, we have to perform COW at page level to avoid any side effect of the in-guest OS if there is a data write on the redirected data. This time, our design is to extend one of the reserved bits in page table entries to indicate whether a page is dirty (has been copied) and add one bit each to our software STLB entry. Note that this is one

Algorithm 1: Kernel data redirection

1: **Require**: **SysExecContext**(s) returns true if syscall s is executed in a system call execution context; **SysRedirect**(s) returns true if data access in s needs to redirected; **RedirectableDataTracking**(i) performs our redirectable data identification and flow tracking for instruction i; **MemoryAddress**(i) returns a set of memory addresses that need to be accessed by instruction i. **NotDirty**(α) queries STLB, or SCR3 and the page table to check if the physical page located by α is dirty. **V2P**(α) will translate the virtual address of α and get its physical address by querying STLB, or SCR3 and the page tables and update STLB if necessary.

```
2: DynamicInstInstrument(i):
3:     if SysExecContext(s):
4:         if SysRedirect(s):
5:             RedirectableDataTracking(i);
6:             for α in MemoryAddress(i):
7:                 if DataRead(α):
8:                     PA(α) ← V2P(α)
9:                     Load(PA(α))
10:                else:
11:                    if NotDirty(α):
12:                        CopyOnWritePage(α)
13:                        UpdatePageEntryInSTLB(α)
14:                    PA(α) ← V2P(α)
15:                    Store(PA(α))
```

of the advantages of instrumenting the VMM because we can add whatever we want in the emulated software such as our STLB. In addition, for the page table entry we just extend one of the reserved bits to achieve our goal. Certainly, we can also make a shadow page table and extend it with a dirty bit for page entry if there does not exist any reserved bit.

More specifically, before the start of an introspection process, STLB is initialized with zero. When a kernel address α needs to be redirected and it is a read (i.e., memory `load`) operation, we first check whether STLB misses, if not, we directly get the physical address $PA(\alpha)$ derived from STLB. Otherwise, we get its $PA(\alpha)$ of the in-guest physical memory by querying SCR3 and performing the three-layer address translation. At the same time, we fill our STLB for address α with the physical address of $PA(\alpha)$ such that future reference for the address sharing the same page of α can be quickly located (the essential idea of TLB). Also, the STLB entry only gets flushed *iff* its entries are full and we have to replace, because we only have one SCR3 value.

If there is a memory write on α, similar to read operation to check whether STLB hits, we also check whether the target page is dirty by querying the dirty bit in our STLB entry. If it is not dirty or STLB misses, we perform the three-layer address translation by querying SCR3 and the page tables, from which we further check whether the target page is dirty. If not (the first time data write on this page), we set the dirty bit of the target page table entry as well as our STLB entry, and perform a target page copy and redirect the future access of the original page to our new page. Otherwise, we just set the dirty bit in the STLB entry.

C. Directing the Access

After we have described all of the necessary enabling techniques, we now turn to the details of our final data redirection procedure. Specifically, as shown in Algorithm 1, for each kernel instruction i, we will check whether its execution is in a syscall execution context (line 3). If so, we check whether the data access in the current syscall context needs to be redirected (line 4). If not, there will be no instrumentation for i.

Next, we perform the redirectable data tracking for i (line 5) by updating our shadow state according to the instruction semantics. After that, for each memory address used in i (line 6), if it is a data read (line 7), we will invoke the `V2P` address translation function to get the corresponding address (line 8), and load the data (line 9). Otherwise (line 10), we will check whether the target page is dirty or not (line 11). If not, we will perform the COW operation (line 12) and update the page entry dirty bit, copying the page if necessary (line 13). After that, we get its physical address (line 14) and do the write operation (line 15).

From Algorithm 1, we could also notice that our *data redirection engine* (line 5 - 15) can work in any other kernel execution context as long as it can be informed. For instance, we could inspect and redirect the kernel data access in a particular kernel function, e.g., in a regular kernel module routine, or a user developed device driver routine. This is actually another benefit of our VMST. That is, it allows end-users to customarily inspect a specific chunk of kernel code. We will demonstrate this feature in our evaluation.

VI. Implementation

We have implemented our VMST based on a recent version of QEMU (0.15.50) [25], with over 7,400 lines of C/C++ code (LOC). In this section, we share the implementation details of interest, especially how we dynamically instrument each instruction in a recent QEMU, how we intercept the interrupt execution context, and how we manage the MMU with respect to our new STLB.

Dynamic Binary Instrumentation There are quite a few publicly available dynamic binary instrumentation frameworks built on top of QEMU (e.g., TEMU [26]). However, their implementations are scattered across the entire QEMU instruction translation, and our redirectable data identification can be implemented more simply. In particular, we take a more general and portable approach, and leverage the XED library [27] (which has been widely used in PIN tool [28]) for our dynamic instrumentation. Upon the execution of each instruction, we invoke XED decoder to disassemble it and dynamically jump to its specific instrumentation logic for performing our redirectable data tracking. The benefit is such an approach allows us to largely reuse our previous code base of PIN-based dynamic data flow analysis [29].

Interrupt Context Interception The beginning execution of an interrupt for the x86 architecture in QEMU is mainly processed in the function `do_interrupt_all`. We instrument this function to acquire the interrupt number, and determine whether it is a hardware or software interrupt. After the QEMU executes this function, it will pass the control flow to OS ker-

nel. The kernel then subsequently invokes the interrupt handler to process the specific interrupt. As discussed in §III, the interrupt handler returns using an `iret` instruction. Thus, by capturing the beginning and ending of an interrupt (the pair), we identify the interrupt execution context.

MMU Management with STLB In QEMU, MMU is emulated in `i386-softmmu` module for our x86 architecture. To implement our STLB, we largely mirrored and extended the original TLB handling code and data structures (e.g., `tlb_fill`, `tlb_set_page`, `tlb_table`). For `load` and `store`, QEMU actually differentiates code and data when translating the code (e.g., generating `ldub_code` for the instruction load). Therefore, we only instrument the data load and store helper functions in QEMU.

VII. Empirical Evaluation

We have performed an empirical evaluation of our VMST. In this section, we report our experimental results. We first tested its effectiveness in §VII-A with a number of commonly used utilities and natively developed programs on top of our testing kernel-2.6.32-rc8, followed by the performance overhead of these programs in §VII-B. Next, we show the generality of our system by testing with a diversity of other Linux kernels in §VII-C. Finally, we demonstrate its security applications in §VII-D. All of our experiments were carried out on an Intel Core i7 CPU with 8G memory machine using a ubuntu 11.04, Linux kernel 2.6.38-8.

A. Effectiveness

Automatic VMI Tool Generation Most VMI functionality can be achieved by running administrative utility programs. In our VMST, these utilities are automatically generated. In this experiment, we took 15 commonly used administrative utilities and tested them with options shown in the 1^{st} column of Table II.

To measure whether we get the correct result, we take a cross-view comparison approach on our testing kernel-2.6.32-rc8. Right before we take the snapshot, we run these commands and save their result to a file. Then we attach the snapshot and run these utilities in our trusted VM, and syntactically compare (`diff`) the two output files. Note that we did not install any rootkit in this experiment, and the security application is tested in §VII-D. Interestingly, as shown in the 3^{rd} column of Table II, 8 out of 15, including `ps`, and `date`, have *slight* syntax discrepancy, and all other commands returned syntax-equivalent (SyE) result.

Then we examined the reasons and found the root cause to be due to the timing when we took the snapshot. For `ps` command, our introspected version found one fewer process and this process is actually the `ps` itself when running in the guest. It did not exist in the snapshot. For all other commands such as `uptime` and `date`, the slight difference is also due to the timing field in the output (reflecting the time difference between running the command and taking the snapshot), but every time our introspected version will always output the same result, which precisely shows that we introspected the correct state of the in-guest OS and the state never changed. That is, our automatically generated introspection program returns the semantic-equivalent (SeE) result and we summarize this in the last column of Table II.

Utilities w/ options	Description	SyE	SeE
`ps -A`	Reports a snapshot of all processes	✗	✓
`lsmod`	Shows the status of modules	✓	✓
`lsof -c p`	Lists opened files by a process p	✓	✓
`ipcs`	Displays IPC facility status	✓	✓
`netstat -s`	Displays network statistics	✓	✓
`uptime`	Reports how long the system running	✗	✓
`ifconfig`	Reports network interface parameters	✓	✓
`uname -a`	Displays system information	✓	✓
`arp`	Displays ARP tables	✓	✓
`free`	Displays amount of free memory	✗	✓
`date`	Print the system date and time	✗	✓
`pidstat`	Reports statistics for Linux tasks	✗	✓
`mpstat`	Reports CPU related statistics	✗	✓
`iostat`	Displays I/O statistics	✗	✓
`vmstat`	Displays VM statistics	✗	✓

Table II
EVALUATION RESULT WITH THE COMMONLY USED UTILITIES WITHOUT ANY MODIFICATION.

Customized Program	LOC	Description	SyE	SeE
`ugetpid`	5	Reports the current process pid	✗	✓
`kps`	52	Reports all the processes	✗	✓
`klsmod`	65	Displays all the modules	✓	✓

Table III
EVALUATION RESULT WITH THE NATIVE CUSTOMIZED PROGRAMS.

Native Customized VMI Tool Development Sometimes, end-users may develop their own customized VMI tool by either invoking system calls or programming with the kernel directly. For example, a utility command may not be able to see some rootkits, and end-users could either write a native kernel module to retrieve the state, or write a user-level program but invoke special system calls (e.g., by developing a new system call). Recall that our system allows customized kernel code inspection as long as the end-user informs the execution context (§V-C).

To show such a scenario, we developed three programs. One is a very simple user-level `getpid` program (with only 5 LOC) to demonstrate that end-users can invoke system calls to inspect a kernel state, and the other two are kernel level programs (i.e., device drivers) that list all the running processes and installed kernel modules by traversing the `task_struct.tasks` and `module` list data structures. Note that it took us less than one hour in developing `kps` (with 52 LOC) and `klsmod` (with 65 LOC).

As presented in Table III, interestingly `ugetpid` will always return a constant `pid` value. This is because the `getpid`

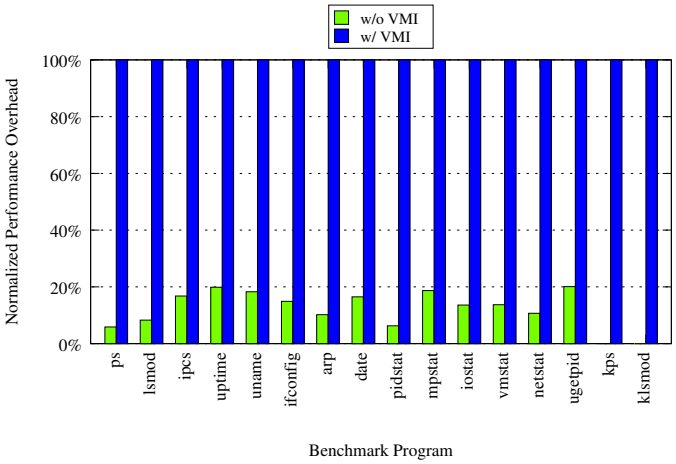

Figure 6. Normalized performance overhead when running VMST.

Linux Distribution	Kernel Version	Release Date	OS-agnostic	LOC
Redhat-9	2.4.20-31	11/28/2002	✗	53
Fedora-6	2.6.18-1.2798.fc6	10/14/2006	✗	53
Fedora-15	2.6.38.6-26.rc1.fc15	05/09/2011	✓	0
OpenSUSE-11.3	2.6.34-12-default	09/13/2010	✓	0
	2.6.35	08/10/2010	✓	0
OpenSUSE-11.4	2.6.37.1-1.2-default	02/17/2011	✓	0
	2.6.39.4	08/03/2011	✓	0
Debian 3.0	2.4.27-3	08/07/2004	✗	53
Debian 4.0	2.6.18-6	12/17/2006	✗	53
Debian 6.0	2.6.32-5	01/22/2010	✓	0
	2.6.32-rc8	02/09/2010	✓	0
Ubuntu-4.10	2.6.8.1-3	08/14/2004	✗	53
Ubuntu-5.10	2.6.12-9	08/29/2005	✗	53
Ubuntu-10.04	2.6.32.27	12/09/2010	✓	0
	2.6.33	03/15/2010	✓	0
	2.6.34	07/05/2010	✓	0
	2.6.36	11/22/2010	✓	0
	2.6.37.6	03/27/2010	✓	0
Ubuntu-11.04	2.6.38-8-generic	06/03/2011	✓	0
Ubuntu-11.10	3.0.0-12-generic	08/05/2011	✓	0

Table IV
OS-AGNOSTIC TESTING OF VMST.

system call by default is redirected to guest memory, and every time it will return the `pid` of the guest "current" process when taking the snapshot. Therefore, it becomes a constant value for a particular snapshot.

For `kps`, it traverses kernel `task_struct.tasks` list and outputs all the pids. Similar to our automatically generated `ps`, `kps` also has almost syntax-equivalent `pid`s with the user-level `ps` except there is one fewer pid of the `ps` itself. However, for `klsmod`, we extracted all kernel modules. The module list result is both syntax and semantic equivalent to the state when `lsmod` reads them from the `proc` file system. Also, note that when we run the two kernel modules, we will inform our secure-VM of the start and end addresses of `kps` and `klist`. To this end, we inserted two consecutive `cpuid` instructions, each at the beginning and the end of our kernel modules. Our secure-VM automatically detects this during the execution and senses the start and end address of the monitor context. In addition, this time the execution context monitoring is not based on any CR3 or system call, and we also controlled our secure-VM ensuring that there is no context switch during our module execution.

Summary The above two sets of experiments have demonstrated we can automatically generate the VMI tools, and also enable end-users to develop their own VMI tools natively. Also, the slightly different result of the two views does not mean we get the wrong result. Instead, all of our above experiments have faithfully introspected the guest OS and reported the precise state.

B. Performance Overhead

We also measured the performance overhead of the programs in Table II and Table III by running each of them 100 times and computing the average. The result is summarized in Fig. 6. We found our VMST introduces 9.3X overhead on average compared with the non-introspected process for the user-level introspection programs running in the VM. For the two kernel level modules, it introduces very large performance overhead up to 500X, but we have to emphasize that the absolute time cost is very short. For example, for `kps`, it only takes 0.06s to dump all the pids in the kernel.

There are a number of reasons why we have small performance overhead at the user level. First, all user level code runs normally without any instruction interception, data flow tracking, or redirection. Second, not all kernel level system calls get redirected, only the introspection related ones. Therefore, if a program frequently opens and reads `proc` files, it tends to have larger overhead. For instance, there is 20X overhead in `ps` and `pidstat`, and 30X for `lsof`. However, for kernel level modules, we have huge performance overhead. The primary reason is because there is no user-level code, and we intercepted each instruction of our kernel modules. Also, these kernel modules run too fast (almost negligible) and the 500X is an over approximation.

When compared with Virtuoso for the performance, as these two systems have entirely different software environment, it is unfair to compare the absolute time cost (for instance its `pslist` took around 6s to inspect all the running process in their experiment [13], and our `kps` only took 0.06s). Our 9.3X overhead on average is the one compared with the tool running in a VM, and we did not include the performance overhead incurred by the VM.

C. Generality

Next, we tested how general (OS-agnostic) our design is, regarding different OS kernels. We selected a wide range of Linux distributions, including Fedora, OpenSUSE, Debian, and Ubuntu (presented in the 1^{st} column of Table IV), with a variety of 20 different kernel versions (the 2^{nd} column). We tested these kernels whether running correctly or not, by using our benchmark programs in Table II and our `ugetpid`.

Rootkit Name	Attack Vector	Target Data Structure	Detected by				
			ps	kps	kps'	lsmod	klsmod
adore-ng-0.53	Patching system call table and kernel function pointers	task_struct	✓	✓	✓		
adore-ng-0.56	Patching kernel function pointers	task_struct	✓	✓	✓		
hide process-2.6	Patching task list pointer (DKOM)	task_struct	✗	✗	✓		
linuxfu-2.6	Patching task list pointer (DKOM)	task_struct	✗	✗	✓		
sucKIT1.3b	Patching system call table and kernel function pointers	task_struct	✓	✓	✓		
override	Patching system call table and kernel function pointers	task_struct	✓	✓	✓		
synapsys	Patching system call table and kernel function pointers	task_struct,module	✓	✓	✓	✓	✓
enyelkm-1.3	Patching system call table and kernel function pointers, and DKOM	task_struct,module	✓	✓	✓	✗	✗
modhide	Patching module list (DKOM)	module				✗	✗

Table V
ROOTKITS EXPERIMENT UNDER KERNEL-2.6.32-RC8

Note that the two kernel modules are not transparent to all kernels, and we did not test them. One of the basic metrics is to test whether our design presented from §III to §V is fully transparent to the guest OS (the 3^{rd} column) without any modification, and if it is not fully transparent (OS-gnostic), we measure the code size of our hard-coded part (the 4^{th} column) in order to make that kernel work.

Surprisingly, our design is truly transparent to Linux kernel starting from 2.6.20, and for the previous kernels we have to make just 53 LOC to support it. The reason why we have to introduce these 53 LOC to the old kernel is because of the way the kernel extracts the current process. More specifically, in the Linux kernel, each process has a `task_struct` which stores all the process execution and management information [16]. During a system call execution, the kernel itself usually first fetches the current `task` and then performs the specific system call for this `task`. Thus, when we perform our kernel data redirection, it is also crucial for us to find the current `task` of the in-guest OS, from which we could know about such things as where the file system (the `fs` field in `task_struct`) is and execute our introspection. For example, most of the inspection utilities open `proc` files, and we have to know where the file system gets stored in the in-guest OS.

Starting from kernel 2.6.20, Linux uses a global variable to store the current task, and in our secure-VM it automatically gets redirected. That explains why our system is fully transparent to the Linux kernel starting from 2.6.20. However, for the old versions, Linux acquires the current task from its kernel stack. In particular, each `task_struct` has a pointer pointing to `thread_info`, which is usually allocated in the bottom of a kernel stack. From `thread_info`, it has a pointer to `task_struct`. Therefore, each time for the kernel to fetch the current task, it first gets the `thread_info` by bit-masking `esp` with 8192 (note in our testing kernel, only 2.6.18-1.2798.fc6 has a 4k kernel stack by default), and then dereferencing the `task_struct` field in `thread_info`.

Since our kernel data redirection will not redirect the data derived from stack, we have to hard code these instructions to get the current `task_struct` with only 53 LOC. Note only kernel 2.6.18-1.2798.fc6 needs to change the stack size to 4K. All other kernels have an identical 53 LOC patch. Also, even though we introduce the hard coded patch, which thwarts our transparency to some extent, we emphasize that for *end-users and native VMI tool developers*, our system is fully transparent to them. Such a nature is similar to system software vendors. As long as we (like an OS vendor) hide all low-level details to end-users and native developers, our tool is a viable option and has met our transparency expectation.

D. Security Applications

VMST has many security applications. It can be naturally used in intrusion detection, malware analysis, and memory forensics. In the following, we demonstrate one of the particular applications – kernel malware (i.e., rootkit) detection, and show its distinctions. Meanwhile, we also briefly describe how to use it for memory forensics.

Kernel Rootkit Detection A kernel rootkit is a special kernel level malware that hides the presence of important kernel objects by either hijacking system calls or other kernel functions, or direct kernel object manipulation (DKOM). In this experiment, we selected 9 publicly available rootkits (shown in the 1^{st} column of Table V) from *packetstorm.com*, and test how our VMST detects them on our working kernel 2.6.32-rc8. Note that we have to slightly modify the source code of the outdated rootkits to make them work in the 2.6.32 kernel.

Most rootkits aim to hide `task_struct` or kernel `modules`. To detect them, we also take a cross view comparison approach. As shown in Table V, for rootkits that hijack kernel control flow by patching the system call table or other kernel hooks, VMST trivially detected them by running either the native `ps` or `kps`, because the OS kernel in our secure-VM is not contaminated.

Regarding rootkits that directly modify the kernel objects (i.e., the DKOM attack), normal VMI tools (or administrative inspection command) will not be able to detect them. However, our system enables end-users to develop kernel modules natively to inspect the kernel objects, and we are able to detect some of them but not all. Specifically, `ps` and `lsmod` command visit `proc` files to inspect the running processes and device drivers. The data in the `proc` files comes from the `task_struct` and `module` list. If rootkits removed the node from the two lists through DKOM, neither original `ps` and `lsmod`, nor our `kps` and `klsmod` is able to identify them.

As such, we have to develop new kernel modules to detect them. In particular, we developed a `kps'` (with 130 LOC) which will periodically traverse the `runqueue` inside the kernel, and compare with the tasks list (note that in this case we have to periodically take the memory snapshot or attach to the guest-memory lively). It is an opportunistic approach, namely, it detects the hidden DKOM process at the moment when it is in `runqueue` but not in tasks list. We attempted to identify all the `waitqueue` in the kernel, but they are scattered across many places and it is very hard to traverse them in a centralized way. Our `kps'` is able to detect "hide process-2.6" and "linuxfu-2.6" DKOM rootkits as shown in Table V. But for the last two module hidden rootkits, we cannot detect them through traversal-based approaches. One may have to use signature-based approaches (e.g., [30], [31]) to detect them. For instance, it has been shown that SigGraph can detect the last two rootkits [31].

Memory Forensics Our automatically generated VMI tools can also be used in memory forensics. The only requirement is that end-users need to provide a guest CR3 value and assign it to our SCR3 for the `V2P` translation when "mounting" the memory. If it is a hibernation file, they have to recover a CR3 or more generally recover a page directory (`pgd`). In fact, this is not a problem as (1) `pgd` has a strong two-layer SigGraph [31] signature, namely, each entry in `pgd` is either NULL or a pointer pointing to a page table entry (`pte`), and each entry in `pte` is either NULL or a pointer pointing to a page frame; (2) meanwhile, all the processes largely share the identical kernel address mapping except little process-specific private data. In addition, there are several other approaches to recovery a `pgd` in physical memory snapshot such as using kernel symbols [32]. Due to space limitation, we omitted the details on our memory forensics experiment.

VIII. Limitations and Future Work

While VMST has automatically bridged the semantic-gap in VMI, its current implementation has a number of limitations. In this section, we examine each limitation below and outline our future work.

The first limitation is that VMST is not entirely transparent to arbitrary OS kernels. It still binds with some particular OS kernel knowledge such as system call interface, interrupt handling and context switching, though such knowledge (particularly from the design of UNIX [17]) is general. For example, as demonstrated in our experiment, VMST directly supports a variety of Linux Kernels without any modifications. However, an entirely different OS may have different system-call interfaces and semantics. Obviously we cannot directly run VMST on an arbitrary OS. In other words, as illustrated step-by-step in our design, for a different OS other than the UNIX, we have to inspect how its kernel handles system calls, interrupts, context switches, and its specific semantics of system calls, etc.

But we do believe our design is general (OS-agnostic), and we suspect it should work for other kernels. For instance, as demonstrated in §V-A, with a slight modification of the "system call redirection policy" in our VMST, we can directly enable a `getpid` process in Windows to directly introspect the in-guest Windows OS, which experimentally proves that our interrupt handling, context switching controlling, redirectable data identification, and kernel data redirection are indeed OS-agnostics. Thus, one of our immediate future efforts is to make our system completely support other kernels such as Microsoft Windows.

Secondly, while most system calls on UNIX platform are synchronized, that is, during the execution of a system call it will block other executions except context switches until the system call finishes, there could be some system calls that are asynchronized ("non-blocking"). VMST does not support asynchronous system calls unless we can detect the precise execution context in which the kernel notifies the caller that a system call has completed. Fortunately, we have not encountered such asynchronous system calls in our VMI tool generation.

Thirdly, we only support the introspected tool examining the memory data. If a VMI tool needs to open an in-guest disk file, it will not be redirected in our current implementation, though end-users could directly copy these files outside and directly inspect them. However, we suspect it is possible to redirect disk data access. As file opening in Linux/UNIX is through the VFS (virtual file system) [16], we have already intercepted the `open` system call when we open the `proc` files. We are able to determine the disk file opening. The only issue is we have to locate the corresponding real disk loading data in an OS kernel. This is a non-trivial task. Supporting disk data redirection is another future work.

Fourthly, VMST cannot read the in-guest data that has been swapped out to disk. To our surprise, we have not encountered this situation in our experiments. Our explanation is that the kernel tends to swap user space memory instead of kernel, as kernel space is shared, and swapping in and out a kernel page may have to update all processes' page tables. Meanwhile, for the Linux/UNIX kernel, we actually confirmed that the kernel page never gets swapped out [16].

Finally, for other architectural issues such as using multi-core or multi-CPUs running the guest OS, our secure-VM (a single CPU) may encounter some issues when reading their memory, though usually the product-VM runs on a single CPU. Also, attacks targeting our secure-VM or DKSM-based [33] are out-of-scope of our current work. Another venue of future work will investigate how to address these problems.

IX. Related Work

Binary Code Reuse Recently, there is great attention towards binary code reuse for security analysis [13]–[15] or creation of malicious code [34]. In particular, BCR [14] made a systematic study of automated binary code reuse, and demonstrated its effectiveness in extracting encryption and decryption components from malware. Similarly, through dynamic slicing,

Inspector gadget [15] also focuses on extracting and reusing features inside malware programs. In a different application, our prior work [34] shows that we can also reuse the legal binary code to create stealthy trojans by directly patching benign software.

Most recently, Dolan-Gavitt et al. proposed Virtuoso [13], a technique for better VMI. The idea is to acquire traces of an inspection command (e.g., `ps`) on a clean guest OS through dynamic slicing. Such clean slices can be executed at the VMM layer to introspect the identical version of the guest OS that may be compromised. Our VMST is directly inspired and motivated by this technique.

Compared to all of the existing techniques, VMST distinguishes itself by its exploration of other settings of binary code reuse. Specifically, instead of extracting the code outside from binary (kernel or user-level code) for the reuse, we can still retain these pieces of code. Through automatically identifying the specific execution context, we can dynamically instrument the code and achieve our reuse.

Virtual Machine Introspection Due to the nature of strong isolation, VMI has largely been used in many security applications, including intrusion detection (e.g., [1], [3], [4]), malware analysis (e.g., [5], [6]), process monitoring (e.g., [7]), and memory forensics (e.g., [8]). Again, similar to Virtuoso, our VMST complements these works by enabling automated VMI tool generation.

Meanwhile, there is another work [35] aiming to automatically bridge the semantic gap in VMI. Their technique involves using a C interpreter facilitated by the OS kernel data structure information and XenAccess library [3] to interpret the introspection code. Such a technique is entirely different from VMST. For example, users have to write the introspection code running in their interpreter. In contrast, VMST directly uses the common utilities without any code development from users.

Also, for the most recent process out-grafting (POG) [7], although VMST shares a general idea of using a trusted VM to do the monitoring and redirection of "some" data during execution, VMST is still substantially different from this approach in a number of aspects. In particular, we have entirely different goals. POG aims to monitor an untrusted process, but we aim to inspect the whole OS. Meanwhile, POG only intercepts kernel execution at the system call granularity (which explains why they can implement it using KVM), whereas we have to monitor all the instructions. Consequently, their data redirection is only system call arguments and return values, whereas we have to automatically identify the redirectable data on-the-fly, and redirect only the introspection related data.

Dynamic Data Dependency Tracking VMST employs a generic technique of dynamic data dependency tracking (i.e., taint analysis) in determining the redirectable data. Such techniques have been widely investigated and there exists a large body of recent work in this area, such as data life time tracking (e.g., [18]), exploit detection (e.g., [19]), vulnerability fuzzing (e.g., [36], [37]), protocol reverse engineering (e.g., [38]–[41]), and malware analysis (e.g., [20], [21]).

Memory Forensics Technically, memory forensic analysis shares large similarity with VMI in that both techniques have to interpret and inspect memory. For example, forensic tools can actually facilitate VMI [42].

The basic memory forensic technique is object traversal and signature scanning. Thus, many existing techniques focus on how to build the object map (e.g., [43]) or generate robust signatures (e.g., [30], [31]). Again, VMST complements these techniques by offering a new set of automatically generated VMI-based tools [8] to analyze memory.

X. Conclusion

We have presented the design, implementation, and evaluation of VMST, a novel system to automatically bridge the semantic gap and generate VMI tools. The key idea is that through system wide instruction monitoring at VMM layer, we can automatically identify the introspection related kernel data and redirect their access to the in-guest OS memory (which could be directly attached, or from a snapshot). We showed that such an idea is practical and truly feasible by devising a number of OS-agnostic enabling techniques, including *syscall execution context identification*, *redirectable data identification*, and *kernel data redirection*, and implemented them. Our experimental results have demonstrated that VMST offers a number of new features and capabilities. Particularly, it automatically enables the in-guest inspection program to become an introspection program and largely relieve the procedure of developing customized VMI tools. Finally, we believe VMST has significantly removed the hurdles in virtualization-based security, including but not limited to VMI, malware analysis, and memory forensics, and will largely change their future daily practice.

Acknowledgment

We greatly appreciate our shepherd, Weidong Cui, for his invaluable feedback, and the anonymous reviewers for their insightful comments and suggestions. We are also grateful to Mitchell Adair, Mark Gabel, Guofei Gu, Scott Hand, Bhavani Thuraisingham, Vinod Yegneswaran, Rhonda Walls, Xinyuan Wang, and Zhi Wang for their constructive feedback on an early draft of this paper.

References

[1] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.

[2] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *IEEE Computer*, May 2005.

[3] B. D. Payne, M. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Proc. of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[4] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proc. of 2008 IEEE Symposium on Security and Privacy*, May 2008.

[5] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proc. of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, October 2007.

[6] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proc. of the 15th ACM conference on Computer and communications security (CCS'08)*, October 2008.

[7] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring," in *Proc. of the 18th ACM conference on Computer and communications security (CCS'11)*, October 2011.

[8] B. Hay and K. Nance, "Forensics examination of volatile system data using virtual introspection," *SIGOPS Operating System Review*, vol. 42, pp. 74–82, April 2008.

[9] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proc. of the 8th Workshop on HotOS*, 2001.

[10] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - A coprocessor-based kernel runtime integrity monitor," in *Proc. of the 13th USENIX Security Symposium*, August 2004.

[11] F. Baiardi and D. Sgandurra, "Building trustworthy intrusion detection through vm introspection," in *Proc. of the Third International Symposium on Information Assurance and Security*. 2007.

[12] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Proc. of Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.

[13] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. of 2011 IEEE Symposium on Security and Privacy*, May 2011.

[14] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proc. of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, February 2010.

[15] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, "Inspector gadget: Automated extraction of proprietary gadgets from malware binaries," in *Proc. of 2010 IEEE Security and Privacy*, May 2010.

[16] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[17] M. J. Bach, *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[18] J. Chow, B. Pfaff, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole-system simulation," in *Proc. of the 13th USENIX Security Symposium*, 2004.

[19] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)*, February 2005.

[20] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic spyware analysis," in *Proc. of the 2007 USENIX Annual Technical Conference (Usenix'07)*, June 2007.

[21] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[22] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[23] N. Provos, "Improving host security with system call policies," in *Proc. of the 12th USENIX Security Symposium*, August 2003.

[24] R. Sekar, "Classification and grouping of linux system calls," http://seclab.cs.sunysb.edu/sekar/papers/syscallclassif.htm.

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

[26] H. Yin and D. Song, "Temu: Binary code analysis via whole-system layered annotative execution," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3, January 2010.

[27] "Xed: X86 encoder decoder," http://www.pintool.org/docs/24110/Xed/html/.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. of ACM SIGPLAN PLDI 2005*.

[29] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proc. of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, February 2010.

[30] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proc. of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. October 2009.

[31] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Proc. of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, February 2011.

[32] A. Walters, "The Volatility framework: Volatile memory artifact extraction utility framework," https://www.volatilesystems.com/default/volatility

[33] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "Dksm: Subverting virtual machine introspection for fun and profit," in *Proc. of the 29th IEEE Symposium on Reliable Distributed Systems*, 2010.

[34] Z. Lin, X. Zhang, and D. Xu, "Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction," in *Proc. of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2010.

[35] H. Inoue, F. Adelstein, M. Donovan, , and S. Brueckner, "Automatically bridging the semantic gap using a c interpreter," in *Proc. of the 2011 Annual Symposium on Information Assurance*, June 2011.

[36] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*. October 2006.

[37] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proc. of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[38] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol format using dynamic binary analysis," in *Proc. of the 14th ACM Conference on Computer and and Communications Security (CCS'07)*, October 2007.

[39] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, October 2008.

[40] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proc. of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[41] G. Wondracek, P. Milani, C. Kruegel, and E. Kirda, "Automatic Network Protocol Analysis," in *Proc. of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[42] B. Dolan-Gavitt, B. Payne, and W. Lee, "Leveraging forensic tools for virtual machine introspection," *Technical Report; GT-CS-11-05*, 2011.

[43] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proc. of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, October 2009.