HYBRID-BRIDGE: Efficiently Bridging the Semantic Gap in Virtual Machine Introspection via Decoupled Execution and Training Memoization

Alireza Saberi
The University of Texas at Dallas
saberi.alireza@utdallas.edu

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

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

***Abstract*—Recent advances show that it is possible to reuse the legacy binary code to bridge the semantic gap in virtual machine introspection (VMI). However, existing such VMI solutions often have high performance overhead (up to hundreds of times slowdown), which significantly hinders their practicality especially for cloud providers who wish to perform real-time monitoring of the virtual machine states. As such, this paper presents HYBRID-BRIDGE, a new system that uses an efficient *decoupled execution* and *training memoization* approach to automatically bridge the semantic gap. The key idea is to combine the strengths of both offline training based approach and online kernel data redirection based approach, with a novel training data memoization and fall back mechanism at hypervisor layer that decouples the expensive Taint Analysis Engine (TAE) from the execution of hardware-based virtualization and moves the TAE to software-based virtualization. The experimental results show that HYBRID-BRIDGE substantially improves the performance overhead of existing binary code reuse based VMI solutions with at least one order of magnitude for many of the tested benchmark tools including `ps`, `netstat`, and `lsmod`.**

I. INTRODUCTION

Virtual machine monitor (i.e., hypervisor) [23] has provided many new opportunities for guest OS administration (e.g., consolidation, encapsulation, and migration), better security and reliability [10]. One popular application is the virtual machine introspection (VMI) [22] that pulls the guest OS states and inspects them at hypervisor layer. Because of such higher trustworthiness and stealthiness compared to running inspection software inside a guest OS, VMI has been an appealing alternative for many traditional in-VM based security applications, as demonstrated in recent malware analysis [31], [13], [43], [44], kernel rootkit defense [46], [28], and memory forensics [20], [16].

However, it is non-trivial to develop introspection software at hypervisor layer. When developing software inside an OS, programmers often have rich semantic abstractions such as system calls or APIs (e.g., `getpid`) to inspect kernel states. However, there are no such abstractions for guest OS at hypervisor layer, but rather the zeros and ones of raw memory data. Consequently, developers must reconstruct the guest OS abstractions from the raw data at hypervisor layer. Such reconstruction is often called to bridge the semantic gap [10], which is challenging and has been the road block for all of out-of-VM solutions for years.

An intuitive and widely adopted approach to bridging the semantic gap is to walk through kernel data structures to locate and interpret the kernel data of interest (e.g., [45], [43], [31], [5], [9]). However, this approach must rely on the kernel data structure knowledge, such as the layout of kernel object, and resolve the points-to relations among data structures [9], [12], which tends to be very expensive for OS kernels. Moreover, if there is any kernel update or a need to support different kernels, this process has to be repeated.

Fortunately, since there already exist many native inspection programs (e.g., `ps`, `netstat`, `lsmod`) inside an OS, if we can directly reuse the legacy binary code of these programs at hypervisor layer, we would not need the above data structure based approach. Based on this insight, recently VIRTUOSO [15] and VMST [19] were proposed towards automatically bridging the semantic gap in VMI by reusing the legacy binary code.

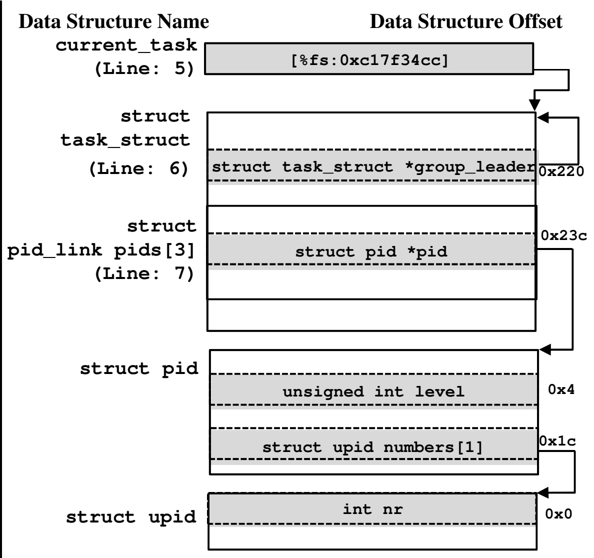
Specifically, VIRTUOSO [15] leverages a training-based, whole system dynamic slicing technique to identify the relevant x86 instructions that query the internal state of a guest OS (e.g., the relevant instructions involved by `ps` command). In the second step, VIRTUOSO extracts the identified sequence of these x86 instructions and lifts them up to a micro operation instruction set [18]. Finally it translates these micro operation code to Python code to eventually produce the introspection tool that can be used for VMI. However, VIRTUOSO suffers nearly 140X slowdown on average compared to native execution according to our experimental result in §VII.

In contrast, VMST [19] uses an online kernel data redirection approach that redirects kernel data access under the execution context of system calls of interest (e.g., `getpid` system call when retrieving a pid is needed) *without any training* to automatically bridge the semantic gap [20]. In particular, VMST leverages a taint tracking component [41] to identify the kernel data which should be redirected during an introspection process. However, this taint tracking component, implemented on top of a VM emulator QEMU [18], often contributes at least 10X slowdown without considering the emulator overhead. If we also consider the emulator overhead,

NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/doi-info-to-be-provided-later

```
<sys_getpid>:
<task_tgid_vnr>:
1:  c10583e0: push   %ebp
2:  c10583e1: mov    %esp,%ebp
3:  c10583e3: push   %ebx
4:  c10583e4: sub    $0x14,%esp

// Accessing Global Variable: struct task_strut current_task
5:  c10583e7: mov    %fs:0xc17f34cc,%ebx
        c10583ea: R_386_32   current_task

// Accessing struct task_struct: current_task->group_leader
6:  c10583fe: mov    0x220(%ebx),%eax

// Accessing struct pid: current_task->group_leader->pids[0]->pid
7:  c1058404: mov    0x23c(%eax),%eax

8:  c105840a: call   c1065660 <pid_vnr>
9:  c105840f: add    $0x14,%esp
```

(a)



(b)

Fig. 1: Code Snippet of System Call `sys_getpid` and the Corresponding Data Structures in Linux Kernel 2.6.32.8.

VMST would have up to hundreds of times performance slowdown.

As a result, the huge performance overhead of these existing solutions significantly hinders their practicality, especially for critical users such as cloud providers who wish to perform real-time monitoring of VM states at large scale. Therefore, in this paper we present HYBRID-BRIDGE, a hybrid approach that combines the strengths of both VIRTUOSO (from the perspective of offline training) and VMST (from the perspective of online taint analysis [41] and kernel data redirection [19]). At a high level, HYBRID-BRIDGE uses an *online* memoization [39] approach that caches the trained meta-data in an online fashion for a hardware-virtualization based VM (e.g., KVM [33]) to execute the native inspection command such as `ps,lsmod,netstat`, and a *decoupled execution* approach that decouples the expensive taint analysis from the execution engine, with an online *fall back* mechanism at hypervisor layer to remedy the coverage issue when the meta-data is incomplete. With such a design, our experimental results show that HYBRID-BRIDGE achieves one order of magnitude faster performance than that of similar systems such as VIRTUOSO and VMST.

More specifically, HYBRID-BRIDGE decouples the expensive online dynamic taint analysis from hardware-based virtualization through online memoization of the meta-data, and we call this execution component FAST-BRIDGE. However, we still need a component to perform the slow taint analysis and precisely tell those redirectable instructions (which are part of the meta-data), and this is done by the second component we call SLOW-BRIDGE. Therefore, HYBRID-BRIDGE is a combination of SLOW-BRIDGE, which extracts the meta-data using the online kernel data redirection approach from a software virtualization-based VM (e.g., QEMU [18]), and FAST-BRIDGE, a fast hardware virtualization-based execution engine via memoization of the trained meta-data from SLOW-BRIDGE. End users will only need to execute the native inspection utilities in FAST-BRIDGE to perform VMI, and SLOW-BRIDGE will be automatically invoked by the underlying hypervisor.

HYBRID-BRIDGE does not have the path coverage issues as VIRTUOSO because it contains a fall back mechanism that works similarly to the OS page fault handler. That is, whenever there is a missing meta-data, HYBRID-BRIDGE will suspend the execution of FAST-BRIDGE and fall back to SLOW-BRIDGE to identify the missing meta-data for the executing instructions. After SLOW-BRIDGE identifies the missing meta-data, it will update and memoize the trained meta-data, and dynamically patch the kernel instructions in FAST-BRIDGE and resume its execution. Therefore, HYBRID-BRIDGE executes the instructions natively in FAST-BRIDGE most of the time. Only when the trained meta-data is incomplete, it falls back to the SLOW-BRIDGE. These VM-level fall-back, memoization, and synchronization can be realized thanks to the powerful control from hypervisor.

In short, this paper makes the following contributions:

- We present a novel *decoupled execution* scheme that decouples an expensive online taint analysis engine from hardware-based virtualization to achieve efficient VMI.
- We also propose a novel *training memoization* that caches the trained meta-data from software virtualization (e.g., QEMU) to avoid the recomputation of redirectable instruction identification.
- All these techniques are transparent to the user level inspection programs as well as end users. Orchestrated by hypervisor, these techniques together substantially improve the performance of the existing VMI solutions by one order of magnitude.
- We demonstrate it is practical to have a hybrid approach to bridging the ideas of two different VMI

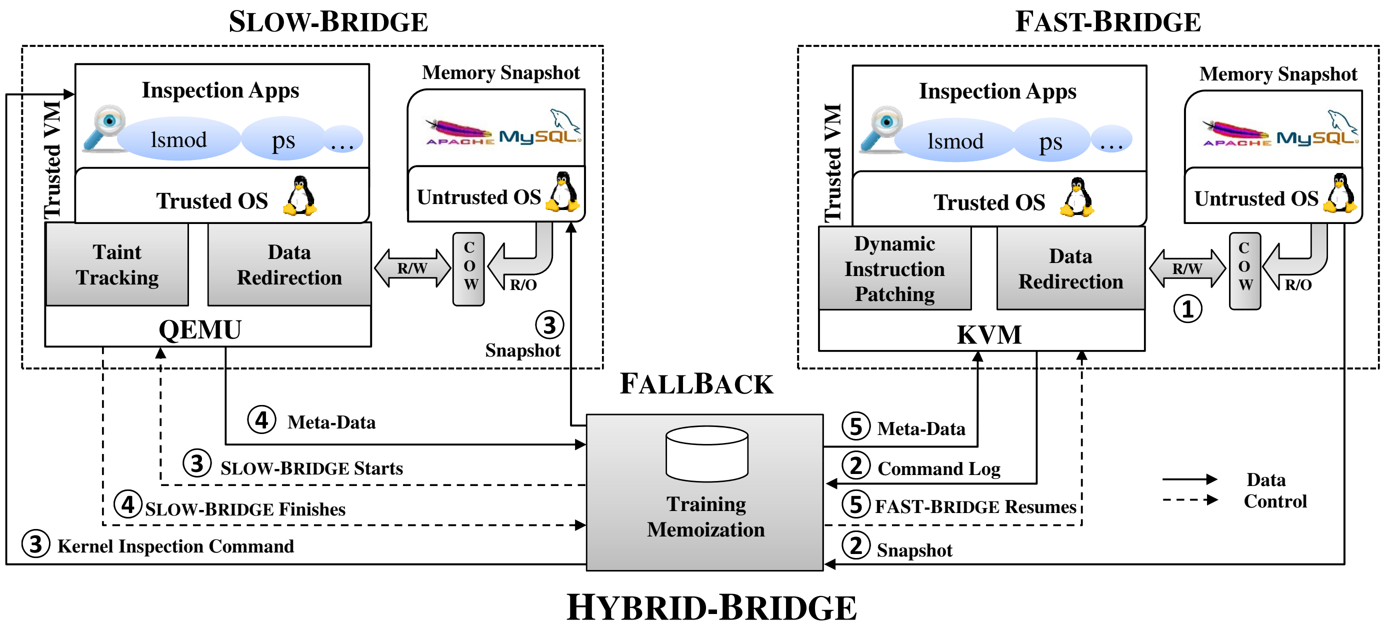


Fig. 2: An overview of HYBRID-BRIDGE.

solutions (VIRTUOSO and VMST), to improve the performance of VMST without suffering the path coverage issues of VIRTUOSO.

The rest of the paper is organized as follows: in §II, we give an overview of HYBRID-BRIDGE. Then, we provide the detailed design of each component of HYBRID-BRIDGE, namely FAST-BRIDGE, SLOW-BRIDGE, and FALLBACK from §III to §V. In §VI, we share the implementation details. In §VII, we present our evaluation result. We discuss the limitation and future work in §VIII, and review related work in §IX. Finally, §X concludes.

II. Background and Overview

Observation. Similar to VMST [19], the main goal of HYBRID-BRIDGE is to enable native inspection utilities (e.g., `ps`, `lsmod`) to transparently investigate a remote system out-of-VM. This goal is achieved by forwarding special kernel data from a remote system (i.e., untrusted VM) to a local system (i.e., trusted VM).

We use a simple inspection program, `GetPid`, to illustrate the basic idea behind HYBRID-BRIDGE. `GetPid` invokes the `sys_getpid` system call to retrieve a running process's ID. Fig. 1 (a) shows a code snippet of `sys_getpid` of Linux kernel 2.6.32.8. In particular, `sys_getpid` kicks off by accessing `current_task`, a global pointer which points to the current running task at line 5, then dereferences the `group_leader` field to access the *group leader task structure* at line 6. Next, it dereferences the pointer to `group_leader` of `task_struct` at line 7 to access the `pid` field. Note that the real PID value is stored in `int nr` field of `struct upid`. For the sake of brevity we only show the partial code and the data structures accessed during `sys_getpid` illustrated in Fig.1.

It is important to notice that all of these data structures are accessed by dereferencing a global variable, `current_task`, and traversing the subsequent data structures. This observation, as first discovered by VMST [19], lays one of the foundations of HYBRID-BRIDGE; namely, by fetching specific kernel global variables (e.g. `current_task`) and all of their derived data structures from the OS kernel of a remote VM, a commodity inspection tool can automatically achieve introspection capabilities. We refer to this technique as *data redirection*.

System Overview. At a high level, HYBRID-BRIDGE enables inspection tools in a trusted VM to investigate an untrusted system memory using native system calls and APIs as if they are investigating the trusted VM. HYBRID-BRIDGE achieves this goal by using data redirection (or forwarding) at kernel level. As shown in Fig. 2, there are three key components inside HYBRID-BRIDGE: SLOW-BRIDGE, FAST-BRIDGE and FALLBACK. SLOW-BRIDGE and FAST-BRIDGE are both capable of redirecting kernel data and enable commodity inspection tools to investigate the untrusted system memory. The main difference, as indicated by their names, is the lower performance overhead in FAST-BRIDGE compared to SLOW-BRIDGE.

Given an introspection tool T, as illustrated in Fig. 2, HYBRID-BRIDGE executes it in FAST-BRIDGE. With the *Meta-Data* provided by SLOW-BRIDGE and memoized by FALLBACK, FAST-BRIDGE enables T to investigate untrusted system memory with low overhead. In case that the *Meta-Data* is not rich enough to guide FAST-BRIDGE, FAST-BRIDGE will suspend its VM execution, and request the trusted VM inside SLOW-BRIDGE to execute T with the same untrusted memory snapshot as input through FALLBACK component. Similar to VMST, SLOW-BRIDGE monitors the execution of T and uses a *taint analysis* engine to infer the data redirection policy for each instruction. These inferred information, being part of the *Meta-Data*, are shared with FAST-BRIDGE. As soon as FAST-BRIDGE receives the *Meta-Data* from SLOW-BRIDGE, it resumes the execution of T.

As a concrete example, assume end users use `ps` to perform the introspection of a memory snapshot from an untrused OS. As illustrated in Fig. 2, if the *Meta-Data* is sufficient (provided in step ⑤), there will be no FALLBACK and FAST-BRIDGE executes normally as in step ①. Otherwise, FAST-BRIDGE will be suspended, and FALLBACK will be invoked (step ②) along with the snapshot of the guest VM and the command log (that is `ps`). Next in step ③, SLOW-BRIDGE will be started with the guest snapshot and the inspection command (namely `ps` in this case) to produce the missing *Meta-Data*. After SLOW-BRIDGE finishes (step ④), it will send the *Meta-Data* for *training memoization* and inform the FALLBACK to resume the execution of FAST-BRIDGE with the new *Meta-Data* (step ⑤). Step ② to ⑤ will be repeated whenever the *Meta-Data* is missing in FAST-BRIDGE. Except FAST-BRIDGE, the SLOW-BRIDGE and FALLBACK components are both invisible to end users.

HYBRID-BRIDGE requires that both trusted VMs in FAST-BRIDGE and SLOW-BRIDGE deploy the same OS version as the untrusted VMs. The specific OS version can be identified through guest OS fingerprinting techniques (e.g., [48], [24]). In order to efficiently bridge the semantic gap and turn the commodity monitoring tools into introspection tools, HYBRID-BRIDGE faces two new challenges: *(1) how to pass the control flow to the hypervisor and to orchestrate* FAST-BRIDGE, SLOW-BRIDGE, *and* FALLBACK in a seamless way, and *(2) how to identify both the data and instructions that should be redirected*. We will present how these two challenges are addressed by FAST-BRIDGE and SLOW-BRIDGE in §III and §IV, respectively.

Threat Model. HYBRID-BRIDGE shares the same threat model with both VIRTUOSO and VMST; namely, it defeats directly those attacks that tamper with the in-guest native inspection software and the guest kernel code (though facing more attack vectors than VIRTUOSO as discussed in §VIII). Note that there are three type of VMs involved in HYBRID-BRIDGE: a guest VM that runs guest OS for a particular application (e.g., a web or database service), a secure VM that runs in FAST-BRIDGE, and another secure VM that runs in SLOW-BRIDGE. We distinguish between trusted and untrusted VMs. The VM directly faced by attackers is the guest VM and we call it untrusted VM. The other two VMs are maintained by hypervisor and are invisible to attackers and we call them trusted VMs. While HYBRID-BRIDGE can guarantee there is no untrusted code redirected from untrusted VM to the local trusted VM, it will not defend against those attacks that subvert the hypervisors through other means (e.g., exploiting hypervisor vulnerabilities).

Also note that in the rest of the paper, we refer the trusted VMs or secure VMs as those (1) maintained by cloud providers, (2) installed with clean OS (the same version with the guest OS), and (3) invisible to attackers. This can be achieved because cloud providers can physically isolate HYBRID-BRIDGE with guest VMs. For untrusted VMs, they could be any type of product VMs (including KVM/Xen/HyperV, etc.) that offer services to cloud users.

III. FAST-BRIDGE

FAST-BRIDGE is designed with fast performance in mind and runs in hardware-based virtualization (e.g., KVM) to offer a VMI solution. It is built based on the key insight that each kernel instruction executed in a specific system call invocation S shows a consistent *data redirectable* behavior for all invocations of S (which forms the basis of the memoization [39]). For example, `sys_getpid` in Linux kernel 2.6.32.8 has 14 instructions that need to be redirected by FAST-BRIDGE. These 14 instructions that will always touch the *redirectable* data are called redirectable instructions.

To this end, FAST-BRIDGE needs to address two challenges:

- **Performing the data redirection**. For example, for these 14 instructions in `sys_getpid`, FAST-BRIDGE needs to redirect their memory access from untrusted VM to trusted VM. While there is no dynamic binary instrumentation engine in KVM, FAST-BRIDGE is still capable of redirecting the data access for these instructions at hypervisor layer transparently. This capability is achieved by manipulating the physical to virtual address translation and dynamic code patching.
- **Identifying the redirectable instructions**. To identify the redirectable instruction, it often requires a taint analysis engine [19], which is heavy and slow. Therefore, we propose the *decoupling* of the dynamic taint tracking engine, the primary contributor to the performance overhead of VMST, from FAST-BRIDGE and implant it into SLOW-BRIDGE. As a result, SLOW-BRIDGE executes the expensive taint analysis and provides the list of *redirectable instructions* for FAST-BRIDGE to bridge the semantic gap efficiently.

FAST-BRIDGE is depicted in the right hand side of Fig. 2. In this section, we present the detailed design of FAST-BRIDGE.

A. Variable Redirectability

A *redirectable variable* is defined as the data in a kernel data structure that is accessed by inspection tools to reveal the system status. These data are *redirectable* because if a monitoring tool in a secure VM is fed with *redirectable* data from untrusted VM, it will report the internal status of untrusted VM as if for the secure VM.

The most intuitive way to identify *redirectable variables* is by monitoring the behavior of introspection tools. As discussed in §II, an introspection tool usually starts an investigation by first accessing specific kernel global variables and then follows them to traverse the kernel internal data structures. These specific global variables and internal data structures, traversed through pointer dereferences, would belong to *redirectable variables*. We will describe how SLOW-BRIDGE uses a taint tracking engine to identify *redirectable variables* in greater details in §IV-B.

B. Instruction Redirectability

An instruction that accesses *redirectable variable* is defined as *redirectable instruction*. In general, kernel instructions are divided into six categories based on how they interact with the *redirectable variables*. Since SLOW-BRIDGE contains a taint analysis engine, it is able to categorize the instructions. The details on how SLOW-BRIDGE categorizes them are presented in §IV-C. In the following, we describe what these categories are and why we have them:

1) ***Redirectable***: An instruction whose operand always accesses *redirectable variables* is called *redirectable instruction*. Instructions at line 5, 6 and 7 in Fig. 1 (a) are the samples of such instructions, and the corresponding *redirectable variables* for these instructions are depicted in Fig. 1 (b). FAST-BRIDGE forwards all the memory access of *redirectable* instructions to the untrusted memory snapshot from the secure VM.

2) ***Non-Redirectable***: An instruction that never interacts with *redirectable variables* is categorized as *non-redirectable*. For example, instructions at line 1, 3 and 8 in Fig. 1 (a) fall into this *non-redirectable instruction* category. FAST-BRIDGE confines these instructions to the memory of the local secure VM only.

3) ***Semi-Redirectable***: *Semi-Redirectable* instructions have two memory references, and they copy data values between *redirectable variables* and *non-redirectable variables*. For instance, `push[%fs:0xc17f34cc]` is a sample of such an instruction, because this `push` instruction reads a global *redirectable variable* (of interest) and copies it to the stack which is *non-redirectable*.

   FAST-BRIDGE forwards the *redirectable variable* memory access to the untrusted memory snapshot and confines the *non-redirectable* memory access to the local secure VM. For `push[%fs:0xc17f34cc]`, FAST-BRIDGE reads the global variable (a *redirectable variable*) from the untrusted memory snapshot and saves it on top of the secure VM's stack that is *non-redirectable*.

4) ***Bi-Redirectable***: If an instruction shows both *redirectable* and *non-redirectable* behavior in different execution context, it is labeled as *bi-redirectable*. For example, function `strlen`, which returns the length of a string, can be invoked to return the length of either *redirectable* or *non-redirectable* strings in different kernel kernel execution context.

   As such, for each invocation of a *bi-bedirectable* instruction, FAST-BRIDGE must determine whether to redirect the data access (e.g., the argument of `strlen`) to untrusted memory snapshot or confine it to the local secure VM based on the execution context, which is defined as the kernel code path from the system call entry to the point of the *bi-redirectable* instruction execution.

   One of the key observations in HYBRID-BRIDGE is that for a specific execution context, a *bi-redirectable* instruction always shows the same redirection policy. (Otherwise the program behavior is non-deterministic). Introspection program is deterministic: given the same snapshot, it should always give the same output. Therefore, we can determine the correct data redirection policy of a *bi-redirectable* instruction based on its execution context. To this end, HYBRID-BRIDGE first trains the data redirection policy for each *bi-bedirectable* instruction (using SLOW-BRIDGE), and then memoizes the same data redirection policy in the next execution of the same kernel code path in FAST-BRIDGE.

5) ***Neutral***: Instructions in this category do not reference memory. Instructions at line 2 and 4 of Fig. 1 (a) are labelled as *neutral* instructions. Since these instructions do not access memory, FAST-BRIDGE does not impose any memory restriction with them.

6) ***Unknown***: All the instructions that are not categorized in any of above categories are called *unknown*. This category is crucial for the synchronization and training data memoization between FAST-BRIDGE and SLOW-BRIDGE. Specifically, just before an *unknown* instruction gets executed, FAST-BRIDGE passes the control to FALLBACK component to ask SLOW-BRIDGE to provide detailed instruction categorization information for the same snapshot. §V will describe the fall-back mechanism in greater details.

C. Data Redirection Using Dynamic Patching

Observation. Having identified the redirectable instructions, we must inform the CPU and let it redirect the data access from secure-VM to untrusted VM for these instructions. We could possibly use static kernel binary rewriting, but this approach faces serious challenges such as accurate disassembly and sophisticated kernel control flow analysis [49]. Then an appealing alternative would be to use dynamic binary instrumentation through emulation based virtualization such as QEMU [3], but this approach suffers from high performance overhead [19]. In contrast, we would like to run hardware assisted virtualization such as KVM and thus we must exploit new approaches.

Fortunately, we have a new observation and we propose hijacking the virtual to physical address translation to achieve data redirection in FAST-BRIDGE. In general, CPU accesses data using their virtual addresses and the memory management unit (MMU) is responsible to translate the virtual address to physical address using page tables. By manipulating page table entries, we are able to make a virtual address translate to a different physical address. Therefore, FAST-BRIDGE *can redirect a memory access by manipulating the page table in a way that a redirectable virtual address is translated to the physical address of untrusted memory snapshot*. FAST-BRIDGE chooses this novel approach because it neither requires any static binary rewriting of kernel code, nor suffers from high overhead as of dynamic binary instrumentation. To the best of our knowledge, we are the first to propose such a technique for transparent data redirection as an alternative to static binary code rewriting or dynamic binary instrumentation.

Our Approach. More specifically, after loading an untrusted memory snapshot, FAST-BRIDGE controls data redirection by manipulating the physical page number in page tables. In order to redirect memory access for a *redirectable variable* v, FAST-BRIDGE updates the physical page number of the page containing v with a physical page number of a page in untrusted snapshot which contains the same variable v. Then FAST-BRIDGE flushes the TLB. From now on, any memory

Line Number	Instruction Type	Original Code Page	Non-Redirectable Code Page	Redirectable Code Page
1	NR	`<sys_getpid>:` `<task_tgid_vnr>:` `c10583e0: push %ebp`	`push %ebp`	<u>`int 3`</u>
2	N	`c10583e1: mov %esp,%ebp`	`mov %esp,%ebp`	`mov %esp,%ebp`
3	NR	`c10583e3: push %ebx`	`push %ebx`	<u>`int 3`</u>
4	N	`c10583e4: sub $0x14,%esp`	`sub $0x14,%esp`	`$0x14,%esp`
5	R	`c10583e7: mov %fs:0xc17f34cc,%ebx` <u>*`c10583ea: R_386_32 current_task`*</u>	<u>`int 3`</u> VMexit	`mov %fs:0xc17f34cc,%ebx` <u>*`c10583ea: R_386_32 current_task`*</u>
6	R	`c10583fe: mov 0x220(%ebx),%eax`	<u>`int 3`</u>	`mov 0x220(%ebx),%eax`
7	R	`c1058404: mov 0x23c(%eax),%eax`	<u>`int 3`</u>	`mov 0x23c(%eax),%eax`
8	NR	`c105840a: call c1065660 <pid_vnr>`	`call c1065660 <pid_vnr>` VMexit	<u>`int 3`</u>
9	N	`c105840f: add $0x14,%esp`	`add $0x14,%esp`	`$0x14,%esp`
10	NR	`c1058412: pop %ebx`	`pop %ebx`	<u>`int 3`</u>
11	NR	`c1058413: pop %ebp`	`pop %ebp`	<u>`int 3`</u>
12 …	NR	`c1058414: ret` `...`	`ret` `...`	<u>`int 3`</u>
36	U	`c106551a: xor %eax,%eax`	<u>`int 3`</u>	<u>`int 3`</u>
37	U	`c106551c: add $0x1c,%esp`	<u>`int 3`</u>	<u>`int 3`</u>

Instruction Types:
R: Redirectable
NR: Non-Redirectable
N: Neutral
U: Unknown

(a) (b) (c)

TABLE I: A Code Snippet of `sys_getpid` and the Corresponding Patched Code for Non-Redirectable and Redirectable Page

access to v is redirected to untrusted memory snapshot because all the virtual to physical address translations for variable v points to the desired physical page in untrusted snapshot. FAST-BRIDGE employs a similar technique to confine the memory access within the secure VM.

Note that changing the page table for each single instruction will introduce performance overhead, and in fact FAST-BRIDGE can avoid most of the overhead due to the instruction locality. In particular, usually instructions with similar type are located beside each other (this can be witnessed from Table I) and FAST-BRIDGE leverages this feature to avoid frequent page table updates for each instruction and set the page table once for all the adjacent instructions of the same type. FAST-BRIDGE uses code patching described below to inform KVM when the page table should be updated.

Dynamic Code Patching. As mentioned before, FAST-BRIDGE switches the data redirection policy by manipulating the page tables. An important question popping up is "*how* FAST-BRIDGE *informs KVM it is time to change the data redirection policy*". In our design, FAST-BRIDGE employs `int3`, a common technique used by debuggers to set up a break point. FAST-BRIDGE overwrites the first instruction that has a different redirection policy from the previous instructions by `int3`. In this way, when an `int3` is executed, KVM catches the software trap and knows this is the time to change the data redirection policy by manipulating the page table.

For instance, instructions in line 1-4 of `sys_getpid` in column (a) of Table I are *non-redirectable* or *neutral* and they are executed with no data redirection policy. But instruction at line 5 is *redirectable* and has a different data redirection policy from previous instructions and thus FAST-BRIDGE patches instruction at line 5 as shown in column (b) of Table I. Next, when FAST-BRIDGE executes the code page as of column (b) of Table I, `int3` at line 5 will cause a software trap and notify the KVM to change the data redirection policy.

The next question is "*what should happen to the instructions that are overwritten by `int3`?*" FAST-BRIDGE actually makes several copies of the kernel code to make sure kernel control flow would not be affected in spite of the dynamic code patching. More precisely, FAST-BRIDGE makes four copies of kernel code pages namely *redirectable* code page, *non-redirectable* code page, *semi-redirectable* code page and *bi-redirectable* code page. Each code page has some part of original kernel code page as well as `int3` patches for the *unknown* instructions if there is any.

FAST-BRIDGE constructs *non-redirectable* code page by copying all the *non-redirectable* and *neutral* kernel instructions and patch all the remaining instructions (*redirectable*, *semi-redirectable*, *bi-redirectable*, and *unknown* instructions) with `int3`. The column (b) in Table I depicts a *non-redirectable* code page which is derived from a code snippet of `sys_getpid` in column (a) of Table I. Instructions in lines 5, 6 and 7 in column (b) of Table I are patched because they are *redirectable* and instructions at lines 36 and 37 are patched since they are *unknown* instructions.

More specifically, FAST-BRIDGE constructs each code page with kernel instruction of the corresponding category as well as the kernel *neutral* instructions. The rest instructions of that category are all patched with `int3` to make sure KVM always takes control and changes the data redirection for different categories of instructions. FAST-BRIDGE constructs *redirectable, semi-redirectable* and *bi-redirectable* code pages by following this rule. For example, column (c) of Table I shows *redirectable* code page which contains *redirectable* and *neutral* instructions.

As we mentioned earlier, each of the four kernel code pages has a special data redirection policy and FAST-BRIDGE overwrites the instruction whose data redirection policy does not match with the code page policy with an `int3`. Such a simple technique notifies KVM the right moment to change the data redirection policy. An important advantage of using four different kernel code pages embedding with `int3` is that FAST-BRIDGE *preserves the original kernel control flow as what it should be, and changes the data redirection policy without the need of any sophisticated kernel control flow analysis*.

Considering the virtual address of instructions in Table I (b) and Table I (c), we notice that FAST-BRIDGE maps the *redirectable* and *non-redirectable* code pages at the same virtual address to preserve the kernel control flow. In other words each time FAST-BRIDGE changes the data redirection, it also re-maps the appropriate kernel code page. Next, we describe how FAST-BRIDGE uses Algorithm 1 at the right moment to map appropriate code page for each data redirection policy.

The Control Transfers of the Patched Code. While instructions in FAST-BRIDGE have six different categories, control flow will be as usual for *neutral* instructions. As such, in the following we focus on the other five categories and describe how FAST-BRIDGE uses Algorithm 1 to choose the appropriate kernel code page and map it for each different category during the kernel instruction execution:

1) ***Non-Redirectable***: As described earlier, *non-redirectable* instructions should be restricted to the secure VM memory. Line 7 of of Algorithm 1 restricts the memory access to the local secure VM by using the original secure VM's page table and the *non-redirectable* code page through manipulating the page table entires. Table I (b) shows a *non-redirectable* code page which is mapped to the same virtual address as original kernel code page in Table I (a). We can see that the original program semantics is still preserved. For instance, instructions in lines 1-4 of *non-redirectable* code page in Table I (b) would be executed just like the original kernel code page but instruction 5, `int3`, would cause a software trap and a *VM exit*. KVM then looks up the data redirection policy for this instruction and finds out that instruction 5 is a *redirectable* instruction by querying the memoized *Meta-Data*, and consequently redirectable code page will be mapped and executed.

Algorithm 1: $SetPageTable(MD, pc, stack)$: Construct Page Table to Enforce Data Redirection based on Instruction Type

```
Input: Meta-Data $MD$ shared by SLOW-BRIDGE, Program
       Counter of instruction under investigation $pc$ and
       Kernel stack of secure VM $stack$
Output: Return Appropriate $PageTable$ to Enforce Data
        Redirection for instruction located at address $pc$
1  Instruction_Type $it \leftarrow MD.InstructionType[pc]$;
2  if $IsBi\text{-}Redirectable(it)$ then
3      CallSiteChains $CSCs \leftarrow MD.CallSiteChain[pc]$;
4      $it \leftarrow Match\&FindType(stack, pc, CSCs)$
5  switch $it$ do
6      case Non-Redirectable:
7          return [Original_Secure_VM_PageTable
                + Non-Redirectable_Code_Page];
8      case Redirectable:
9          return [Untrusted_Snapshot_PageTable
                + Redirectable_Code_Page];
10     case Semi-Redirectable:
11         return [Untrusted_Snapshot_PageTable
                + Original_Secure_VM_PageTable[stack]
                + Semi-Redirectable_Code_Page];
12     case Unknown:
13         call FALLBACK
```

2) ***Redirectable***: All the data access for *redirectable* instructions should be forwarded to untrusted memory snapshot and the *redirectable* code page should be mapped instead of the original kernel code. To this end, line 9 of Algorithm 1 manipulates the page table to point to the untrusted VM snapshot. During virtual to physical address translation in secure VM using manipulated page table entires, virtual addresses of secure VM are translated into physical addresses of untrusted memory snapshot. In this way secure VM (i.e., our KVM) can access the snapshot memory of the untrusted VM transparently.

   Line 9 of Algorithm 1 also maps *redirectable* kernel code page. Table I (c) illustrates the *redirectable* kernel code page of `sys_getpid` shown in Table I (a). Using Algorithm 1, KVM changes the page table and maps the *redirectable* code page, then instructions 5-7 are executed while accessing the untrusted memory snapshot. Instruction 8 of *redirectable* code page, `int3`, informs KVM to change the data redirection policy to *non-redirectable* by raising a software trap.

3) ***Semi-Redirectable***: Based on *semi-redirectable* instruction definition, these instructions are allowed to reference non-redirectable data (i.e., stack) in trusted VM and the redirectable data of untrusted VM. Line 11 of Algorithm 1 manipulates page table to map the memory of untrusted VM snapshot, trusted VM kernel stack and *semi-redirectable* code page.

4) ***Bi-Redirection***: For *bi-redirection* instructions, whether they are *redirectable* depends on the execution context. Ideally, we should use the kernel code execution path to precisely represent the execution context. To that end, we

have to instrument all the branch and call instructions to track the path, which is very expensive and contradicts our design. Therefore, we use a lightweight approach to approximate the kernel execution path.

Specifically, we use a combination of the instruction address (i.e., the PC) and the *Call-Site-Chain* (CSC), which is defined as a concatenation of all return addresses on the kernel stack, as the representation of a unique execution context. SLOW-BRIDGE provides a set of CSC that are stored in the *Meta-Data* for each *bi-redirection* instruction bi. While this approximation is less precise, our experimental results (§VII) reveal that for each bi, the CSC and PC uniquely distinguishes the execution context. If it happens that CSC and PC are not sufficient to distinguish the correct execution context, then HYBRID-BRIDGE will fail and we have to warn the user, though we have not encountered such a case. Note that HYBRID-BRIDGE is able to detect this by simply checking the *Meta-Data*.

In particular, before a bi gets executed, as illustrated in line 3 of Algorithm 1 FAST-BRIDGE retrieves the $CSCs$ for current instruction pointed by PC from the *Meta-Data*. Line 4 of Algorithm 1 then matches $CSCs$ with current kernel stack. If any of the $CSCs$ matches with current stack then FAST-BRIDGE picks the correct data redirection policy between *redirectable* or *non-redirectable* and resumes the execution. Otherwise, the instruction type would be *unknown* and FALLBACK is invoked to find the appropriate policy.

To retrieve the CSC from the current kernel stack, FAST-BRIDGE reads each specific return address from the offset location information provided by the memoized *Meta-Data*. This offset location of a return address inside a stack frame is acquired by SLOW-BRIDGE through dynamic binary instrumentation. In other words, we do not actually need guest kernel to be compiled with stack frame pointer.

5) ***Unknown***: If an *unknown* instruction (e.g., line 36 in Table I) gets executed, KVM catches the software trap and queries Algorithm 1. Since FAST-BRIDGE dose not know the corresponding data redirection policy for *unknown* instructions, lines 13 and 14 of Algorithm 1 falls back to SLOW-BRIDGE to find out the correct data redirection policy. This is the only moment when SLOW-BRIDGE will get invoked in HYBRID-BRIDGE.

IV. SLOW-BRIDGE

SLOW-BRIDGE, as depicted in the left hand side of Fig. 2, consists of (1) a trusted VM that is installed with the same version of the guest OS kernel as of FAST-BRIDGE, and (2) an untrusted guest OS memory snapshot forwarded by FALLBACK from FAST-BRIDGE. SLOW-BRIDGE provides two important services for FAST-BRIDGE:

- **Instruction Type Inference.** As discussed in §III-B, instructions are classified into six different categories, and the classification is done by SLOW-BRIDGE.
- **Fall Back Mechanism.** When FAST-BRIDGE faces a new code path and does not know the appropriate data redirection policy, SLOW-BRIDGE provides a vital fall back mechanism to deal with this issue.

E	TV[E]	Comments
c	0	Constants are always untainted
`esp`	1	Stack pointer is always tainted
R	TV[R]	Taint value of register or memory R
$R := R'$	$TV[R] := TV[R']$	
$R := *(R')$	$TV[R] := TV[*R']$	$*$ is the dereference operator
$(*R) := R'$	$TV[*R] := TV[R']$	
$R'\ op\ R''$	$TV[R'] \| TV[R'']$	*op* represents a binary arithmetic or bitwise operation
$op\ R'$	$TV[R']$	*op* represents a unary arithmetic or bitwise operation

TABLE II: Taint Propagation Rules

At a high level, SLOW-BRIDGE works as follows: when an inspection tool inside the trusted VM of SLOW-BRIDGE invokes a system call, it will then identify the system call of interest (§IV-A), pinpoint the redirectable variables (§IV-B), infer the corresponding redirectable instruction types (§IV-C), perform data redirection (§IV-D), and share the *Meta-Data* with FAST-BRIDGE. In the following, we present how SLOW-BRIDGE works regarding these behaviors.

A. Detecting the System Calls of Interest

SLOW-BRIDGE is interested in systems calls that reveal the internal states of an OS. In terms of the identification of the system call interest, SLOW-BRIDGE has no difference compared to VMST. Specifically, SLOW-BRIDGE is interested in two types of system calls: (1) state query system calls (e.g. `getpid`) and (2) file system related system calls which inspect the kernel by reading the `proc` files. SLOW-BRIDGE follows a similar approach to VMST and inspects 14 file system and 28 state query system calls (c.f., [19]).

B. Redirectable Variables Identification

Redirectable variables, described in §III-A, are *kernel data* accessed by inspection tools to reveal the system status. There are two approaches to identify *redirectable variables*. The first approach follows a typical introspection footsteps by reading interesting kernel global variables which are exported in `System.map`. Following the global variables, introspection tools reach out to kernel data structures in heap and extract system status. Finding relevant set of global variables for each system call is a challenging task especially considering the fact that this list has to be tailored for different versions of OSes.

As such, the second approach focuses on *non-redirectable variables* and redirects the rest of kernel data in the specific system call execution context. A simple definition of *non-redirectable variable* is all variables derived from kernel stack pointer (i.e., `esp`) which are tied to the local trusted system. SLOW-BRIDGE follows the second approach and embodies a *taint analysis* engine to find all the data derived from `esp`. Note that this approach has been proposed in VMST [19] and SLOW-BRIDGE has no technical contribution regarding this.

The taint analysis engine maps each register and memory variable to a boolean value called taint value (`TV`). All `TVs` are initialized to zero except the taint value of `esp` (`TV[esp]`) which is set to one. The initial taint values indicates that at the start of a system call, `esp` is the only data that is considered *non-redirectable*. A concise description of the rules for taint propagation is presented in Table II, though more detailed rules and design can be found in [19]. All the access to variable `R` with `TV[R]` equal to zero is redirected to the untrusted memory snapshot. If `TV[R]` is equal to one, FAST-BRIDGE would use the local value of variable `R` from trusted VM.

C. Inferring Instruction Redirectability

Tracking *redirectable variables* also enables SLOW-BRIDGE to infer kernel instruction's data redirection types based on their interaction with the *redirectable variables*. To this end, SLOW-BRIDGE logs every instructions executed along with all the memory references in the context of a monitored system call. SLOW-BRIDGE then traverses the log file to infer each instruction into one of the instruction category mentioned in §III-B. More specifically, SLOW-BRIDGE uses the following rules to infer the instruction redirection type:

1) ***Redirectable***: If all execution records of an instruction always accesses the *redirectable variables*, this instruction is categorized as *redirectable instruction*.

2) ***Non-redirectable***: If all execution records of an instruction always accesses *non-redirectable variables*, this instruction is a *non-redirectable instruction*.

3) ***Semi-Redirectable***: If an instructions access two variables, one *redirectable* and the other *non-redirectable* in a single record, this instruction is called *semi-redirectable*.

4) ***Bi-Redirection***: If there are several execution records showing that an instruction accesses *redirectable* and *non-redirectable variables* always in different execution context, then this instruction is categorized as *bi-redirectable*.

   Note that having taint tracking engine, SLOW-BRIDGE infers whether *bi-redirectable instruction* is referencing *redirectable* or *non-redirectable variable* in each execution. But FAST-BRIDGE needs a mechanism to differentiate between different invocations of *bi-redirectable instructions*, and it relies on *Meta-Data* information provided by SLOW-BRIDGE to enforce correct data redirection for each execution of *bi-redirectable instruction*.

   In particular, before each *bi-redirectable instruction* gets executed, SLOW-BRIDGE extracts the value of all return addresses on the stack as well as their location offsets with respect to the base address of the stack, and stores them in the *Meta-Data*. The return value is used to form the Call-Site-Chain (CSC) as a signature in the training data, and the offset list is to facilitate FAST-BRIDGE retrieving these return addresses at run-time in the FAST-BRIDGE kernel stack.

5) ***Neutral***: An instruction with no record of memory access in the log is categorized as *neutral instruction*.

6) ***Unknown***: All the instructions which are not executed in the context of a system call are labelled as *unknown* instructions, which is crucial for FALLBACK to take over the control and invoke SLOW-BRIDGE to infer the instruction redirection type.

D. Data Redirection

SLOW-BRIDGE enables the trusted VM to access the untrusted memory snapshot transparently by forwarding all the access of *redirectable variable* to the untrusted snapshot memory. Unlike in FAST-BRIDGE which uses a page manipulation technique to redirect the data, SLOW-BRIDGE uses memory emulation at VMM level. More details on how to use emulation-based VM for data redirection can be found in VMST [19].

V. FALLBACK

The key component to connect FAST-BRIDGE and SLOW-BRIDGE is the FALLBACK, which is shown in the middle of Fig. 2. Since FAST-BRIDGE uses the *Meta-Data* provided by SLOW-BRIDGE through dynamic analysis, there might exist instructions that have not been trained by SLOW-BRIDGE and we call them *unknown* instructions. At a high level, if FAST-BRIDGE faces an *unknown instruction* (ui in short) during the execution, it suspends its execution and falls back to SLOW-BRIDGE through FALLBACK.

The rationale for such an OS page fault style fall back mechanism is based on the observation that if FALLBACK passes the same untrusted memory snapshot and introspection command to SLOW-BRIDGE, then the trusted VM (i.e., the QEMU emulator) in SLOW-BRIDGE would invoke the same command and eventually execute the same ui. Because we run the same code to examine the same state of the untrusted memory snapshot, the program should follow the same path and finally touch the same ui in both trusted VMs of FAST-BRIDGE and SLOW-BRIDGE (the deterministic property of the introspection program).

In order to execute the same introspection command in trusted VM inside SLOW-BRIDGE, there are several approaches: one is to use network communication to connect the trusted VM from hypervisor and invoke the command, the other is to use a process implanting approach [25] to inject the introspection process in trusted VM or use the in-VM assisted approach that installs certain agent inside trusted VM to invoke the command. After the introspection command finishes the execution in the trusted VM, SLOW-BRIDGE will update the *Meta-Data*, which is implemented using a hash table for the memoization, and then inform FALLBACK to resume the execution of trusted VM in FAST-BRIDGE for further introspection.

VI. IMPLEMENTATION

We have developed a proof-of-concept prototype of HYBRID-BRIDGE. Basically, we instrument KVM [33] to implement FAST-BRIDGE and FALLBACK component, and modify VMST [19] to implement SLOW-BRIDGE. Specifically:

FAST-BRIDGE. FAST-BRIDGE provides three main functionalities and they are implemented in the following way:

- **Guest VM system call interception**: To activate the data redirection policy on the system calls of

App. Name	Description	Neutral	Non-Red.	Red.	Semi-Red.	Bi-Red.	Syntax Equal	Semantics Equal
getpid	Displays the current process pid	32	28	10	1	0	✗	✓
gettime	Reports the current time	31	17	1	1	0	✗	✓
hostname	Shows the system's host name	92	53	26	1	0	✓	✓
uname	Prints system information	92	53	26	1	0	✓	✓
arp	Manipulates the system ARP cache	4649	3383	1852	55	34	✓	✓
uptime	Tells how long the system has been running	2339	1781	908	24	0	✗	✓
free	Displays amount of free and used memory	2497	1958	987	28	0	✗	✓
lsmod	shows the status of modules in kernel	2418	1752	923	26	19	✓	✓
netstat	Prints network connections statistics	2884	2020	1106	31	7	✓	✓
vmstat	Reports virtual memory statistics	2865	2432	1086	32	0	✗	✓
iostat	Reports CPU statistics	3472	2793	1299	30	26	✗	✓
dmesg	Prints the kernel ring buffer	106	54	13	2	0	✓	✓
mpstat	Reports processors related statistics	3219	2650	1205	44	21	✗	✓
ps	Displays a list of active processes	5181	4185	1825	51	14	✗	✓
pidstat	Reports statistics for Linux tasks	4325	3678	1630	44	28	✗	✓

TABLE III: Correctness Evaluation Result of HYBRID-BRIDGE and the Statistics of the number of each Instruction Types.

interest, FAST-BRIDGE needs to intercept all the guest VM system calls. We implement the system call interception feature atop a recent KVM based system call interception system Nitro [47].

- **Data redirection**: As described in §III-C, FAST-BRIDGE manipulates guest OS page table to achieve a transparent data redirection.
- **Finding the exact time to change the data redirection policy**: As mentioned earlier, FAST-BRIDGE changes the data redirection policy only when the instruction type of the next-to-be-executed instruction is different with the current one. To this end, FAST-BRIDGE needs an efficient mechanism to notify when the current data redirection policy should be changed. As described in §III-C, FAST-BRIDGE uses a software trap technique to notify KVM to change the data redirection policy. In particular, FAST-BRIDGE employs Exception Bitmap, a 32-bit VM-Execution control filed that contains one bit for each exception. If the forth bit of the Exception Bitmap is set, then an `int3` execution in guest VM will cause a *VMExit*. Using this technique KVM is notified to take the control and change the data redirection policy accordingly. In total, we added 3.5K LOC to implement FAST-BRIDGE in KVM code base.

SLOW-BRIDGE. We reused our prior VMST code base (especially the taint analysis component) to implement SLOW-BRIDGE. Additionally, we developed over 1K LOC atop VMST to infer the instruction's data redirection type (described in §IV-C) and memoizes it in the *Meta-Data*.

FALLBACK. We did not adopt the process implanting or in-VM agent assisted approach to implement FALLBACK, and instead we use a network communication approach. In particular, in order to run the same introspection command in trusted VM inside SLOW-BRIDGE, FALLBACK dynamically creates a shell which uses `ssh` to invoke the command through `system` API such that FALLBACK can precisely know when the command finishes. Also, this `ssh` shell did not introduce any side effect for our introspection purpose regarding the untrusted memory snapshot. Also, it is straightforward to implement the logic for parsing the command log, managing the *Meta-Data*, and controlling the VM states. In total, we developed 300 LOC for FALLBACK.

VII. EVALUATION

In this section, we present our evaluation results. We took 15 native inspection tools to examine the correctness of HYBRID-BRIDGE, and we report this set of experiment in §VII-A. Then we evaluate the performance overhead of HYBRID-BRIDGE, and compare it with both VIRTUOSO and VMST in §VII-B. Note that we have the access to the source code of VIRTUOSO (as it is public open [14]) as well as our own VMST source code.

We run VMST, VIRTUOSO and HYBRID-BRIDGE on a box with Intel Core i7 and 8GB of physical memory to collect the performance results. Ubuntu 12.04 (kernel 2.6.37) and Debian 6.04 (kernel 2.6.32.8) were our host and guest OS, respectively.

A. *Correctness*

To evaluate the correctness of HYBRID-BRIDGE, we use a cross-view comparison approach as in VMST. Specifically, we first execute the native inspection tools shown in the first column of Table III on an untrusted VM and save their outputs. Then we take a memory snapshot of the untrusted VM and use HYBRID-BRIDGE to execute the same set of inspections tools inside the trusted VM and compare the two outputs.

The eighth column of Table III shows that six inspection tools have exactly the same output for this two rounds of execution. The manual investigation of the remaining nine tools shows that the slight differences in outputs are due to the timing. For example `date` and `uptime` have different output because there is a time difference between running them on the untrusted OS and taking the snapshot. If we consider this time difference then the output are similar. Another example is `ps` which also has a small difference in output. The `ps` command in untrusted OS shows itself in the list of processes but when we take the snapshot right after `ps` execution, `ps` is not running anymore thus the output of HYBRID-BRIDGE shows one process less compared to untrusted OS output. The last column of Table III shows that considering timing differences the output of all 15 tools are semantically equivalent.

In addition, Table III also presents the statistics of the different instruction types categorized by SLOW-BRIDGE during

App. Name	KVM (sec.)	VMST (sec.)	HYBRID-BRIDGE w/o any *Meta-Data* (sec.)	HYBRID-BRIDGE w/ Full *Meta-Data* (sec.) (i.e. FAST-BRIDGE)	HYBRID-BRIDGE *#VMExit*	Speedup FAST-BRIDGE vs. VMST	Slowdown FAST-BRIDGE vs. KVM
getpid	0.004	0.423	1.976	0.005	2	84.60X	1.25X
gettime	0.004	0.392	1.985	0.005	4	78.40X	1.25X
hostname	0.004	0.488	2.199	0.005	10	97.60X	1.25X
uname	0.003	0.389	2.211	0.005	10	77.80X	1.66X
arp	0.086	0.739	2.360	0.094	1852	7.86X	1.09X
uptime	0.005	0.591	1.810	0.012	1892	49.25X	2.40X
free	0.007	0.627	2.755	0.017	3927	36.88X	2.42X
lsmod	0.018	1.034	2.329	0.048	11875	21.54X	2.66X
netstat	0.014	1.454	1.719	0.107	23165	13.59X	7.64X
vmstat	0.007	2.195	4.186	0.109	86578	20.13X	15.57X
iostat	0.01	2.323	5.047	0.120	97390	19.35X	12.00X
dmesg	0.155	8.622	4.845	0.295	11663	29.22X	1.90X
mpstat	0.008	1.635	4.460	0.153	124525	10.68X	19.12X
ps	0.009	6.623	10.047	0.481	418124	13.76X	53.44X
pidstat	0.016	8.095	12.585	0.598	490713	13.53X	37.37X

TABLE IV: Performance of each component of HYBRID-BRIDGE and its comparison with VMST.

the execution of each command. These are shown from the third to seventh columns. An interesting observation can be drawn from these statistics is that *semi-redirectable* and *bi-redirectable* instructions tend to be rare compared to other instruction categories, and the majority of the instructions are either neutral, or non-redirectable.

Also, note that HYBRID-BRIDGE does not have a direct correspondence with the size of the user level program, and all of our instrumentation execution occurs at kernel level for the system call of interest, which is the primary factor for the scalability of our system. For instance, the first three programs in Table III have less monitored instructions even though their user level code size is as big as others. In our experiment, `ps` command has the largest number of trapped instructions according to Table III. More specifically, we dynamically observed over four million instruction execution at kernel side, which is in total $10,244$ unique instructions according to sum of the third column to the seventh of Table III.

B. Performance Evaluation

HYBRID-BRIDGE is designed to significantly improve the performance of existing VMI solutions. In this subsection, we present how and why HYBRID-BRIDGE advances the state-of-the-art and meets our design goals.

Table IV shows the execution time of inspection tools tested in §VII-A. The second and fifth columns of Table IV display the execution time of inspection tools on a vanilla KVM and FAST-BRIDGE, respectively. Comparing these two columns reveals that FAST-BRIDGE has on average 10X slowdown compared to the vanilla KVM. Fig. 3 illustrates the details of the performance evaluation for each inspection tool in FAST-BRIDGE compared to KVM.

The fourth column of Table IV displays the execution time of inspection tools in SLOW-BRIDGE. Taint analysis engine and the full emulation architecture of QEMU are the two main contributors to 150X slowdown of SLOW-BRIDGE compared to FAST-BRIDGE.

The third column of Table IV shows the running time for VMST. FAST-BRIDGE speedup compared to VMST is illustrated in the sixth column as well as in Fig. 4. It is

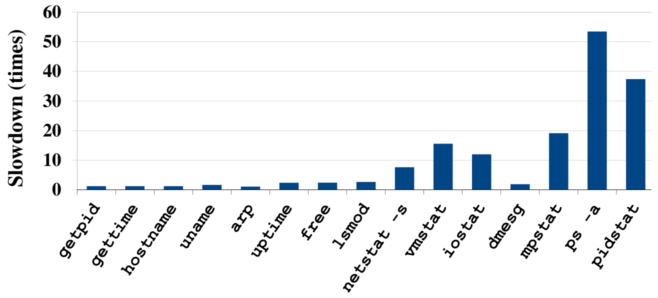


Fig. 3: FAST-BRIDGE Slowdown Compared to KVM.

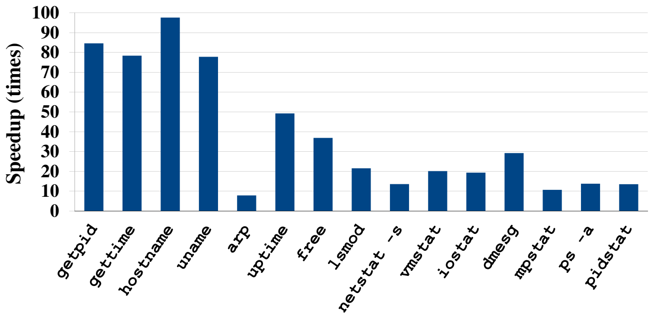


Fig. 4: FAST-BRIDGE Speedup Compared to VMST.

important to notice that FAST-BRIDGE on average has 38X speedup compared to VMST.

Speedup and Slowdown Gap. After examining the performance data, a natural question that pops up is *why there is a huge gap between speedup of inspection tools in Table IV?* The very same question should be also answered for slowdown gap between inspection tools. While there are several reasons to justify the speedup or slowdown gap, we believe the main contributor is the number of *VMExits*.

As we mentioned in §III-C FAST-BRIDGE notifies KVM to change the data redirection policy by using code patching technique. The software trap, raised by code patching, causes a *VMExit* and transfers the execution to the KVM, as illustrated in Table I. The sixth column of Table IV shows the number of

App. Name	Description	Native (sec.)	VIRTUOSO (sec.)	#X86 Inst. in VIRTUOSO	FAST-BRIDGE (sec.)	FAST-BRIDGE vs. VIRTUOSO
gettime	Tells current time of system	0.004	0.023	482	0.005	4.60X
getpid	Shows pid of current process	0.004	0.024	516	0.005	4.80X
tinyps	A compact version of PS	0.020	1.501	140843	0.064	23.45X
getprocname	Displays current Process Name	0.006	2.716	294797	0.132	20.57X

TABLE V: Performance comparison of FAST-BRIDGE and VIRTUOSO

VMExit during the corresponding inspection tools' execution. We also illustrate this fact in Fig. 3, which sorts the inspection tools based on the number of *VMExit*s. We can observe from Fig. 3 that as the number of *VMExits* increases from left to right, FAST-BRIDGE slowdown compared to vanilla KVM jumps from 25% to more than 20X. This trend clearly illustrates that *VMExit* is the main contributor to the FAST-BRIDGE overhead.

The FAST-BRIDGE speedup illustrated in Fig. 4 also indicates the negative effect of *VMExit* on FAST-BRIDGE. In particular, Fig.4 shows as the number of *VMExit* increases from left to right, the speedup factor drops dramatically. For example `getpid` achieved 84X speedup because it needs only two *VMExit*s but `ps` cannot achieved better than 13X speedup because it causes more than 418,000 *VMExit*s.

Comparison with VIRTUOSO. In addition, we use the four inspection tools, shipped in VIRTUOSO source code, to compare the performance of FAST-BRIDGE and VIRTUOSO. The detailed result is presented in Table V. We can see that FAST-BRIDGE achieves 4X-23X speed up (13X on average) compared to VIRTUOSO. The fifth column in Table V shows the number of x86 instructions extracted by VIRTUOSO for each tool. Considering the fifth and the last columns of Table V, we can see that as the size of inspection tool increases FAST-BRIDGE achieves a better speedup compared to VIRTUOSO.

We have verified that the two primary reasons of VIRTUOSO's slowdown are: (1) *micro operations* code explosion – the number of *micro operations* often increases by 3X to 4X, and (2) executing the translated *micro operations* in Pythons (which is very slow).

Number of Fall-Backs. HYBRID-BRIDGE outperforms VMST and VIRTUOSO if the inspection tools primarily get executed in FAST-BRIDGE. It is important to find out how many times FAST-BRIDGE have to fall back to SLOW-BRIDGE before it can execute an inspection tool completely in FAST-BRIDGE.

In order to answer this question we take five different snapshots of untrusted VM and execute these inspections tools using HYBRID-BRIDGE. As shown in Fig. 5, in the first round when no *Meta-Data* is available, all the tools fall back to SLOW-BRIDGE and they all have a very high overhead. Fig. 5 also shows that the first round of *Meta-Data* provides large enough instruction category information for FAST-BRIDGE: 11 out of 15 inspection tools with no more memoization (no more fall-back) to SLOW-BRIDGE.

The rest four inspection tools face new code paths in their second executions and fall back to SLOW-BRIDGE for the second time. After two rounds of execution on two different memory snapshots, according to Fig.5, FAST-BRIDGE is able to execute all the inspection tools on new memory snapshots without any support from SLOW-BRIDGE. In other words, after few runs all the inspection tools would be executed with a very low overhead in FAST-BRIDGE.

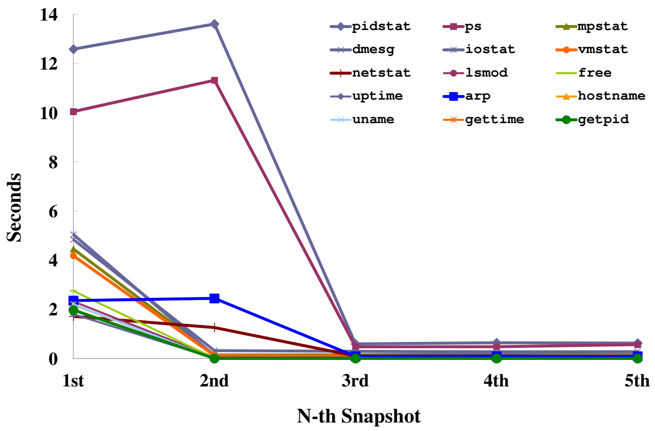


Fig. 5: Execution time of inspection tools in HYBRID-BRIDGE with five different memory snapshots

VIII. LIMITATIONS AND FUTURE WORK

Homogeneity of Guest OS Kernel. As discussed in §II, HYBRID-BRIDGE requires that both trusted VMs in FAST-BRIDGE and SLOW-BRIDGE deploy the same OS version as the untrusted VMs. Note that we only require the same version of guest OS kernel, and do not require the same set of kernel modules. For instance, `lsmod` can certainly return different sets of running kernel modules for different running instances, because end users might have different customizations for kernel modules.

Memory-only Introspections. Similar to VIRTUOSO and VMST, HYBRID-BRIDGE supports introspection tools that investigate only memory but not on files in the disk. It might be an option to directly mount disk file and inspect it. But for encrypted file system, we have to seek other techniques. We leave the introspection of disk files in future work.

Also, if a memory page is swapped out, HYBRID-BRIDGE, including VMST and VIRTUOSO cannot perform the introspection on these pages. However, we may argue that OSes usually tend not to swap out the kernel pages since they are shared between applications. In fact, kernel memory pages are never swapped out in Linux kernel [6].

Attacking HYBRID-BRIDGE. Since HYBRID-BRIDGE is built atop KVM and QEMU, any individual successful exploits against KVM or QEMU might be able to compromise HYBRID-BRIDGE, if our infrastructure is not completely isolated from attackers. Moreover, it might appear to be possible to launch a returned-oriented programming (ROP) attack, or other control flow hijack attacks against our trusted VM by manipulating the non-executable data in the untrusted VM kernel because HYBRID-BRIDGE consumes data from untrusted memory snapshot.

However, it is important to mention that HYBRID-BRIDGE monitors all the instruction execution (including the data flow), and it never fetches a return address from the untrusted VM (recall stack data is never redirected). Therefore, the only way for attacker to mislead the control flow of our trusted VM is to manipulate the function pointers. However, this can also be detected because we check all the instruction execution: whenever a function pointer value is loaded from untrusted VM and later gets called, we can raise flags (because we can observe this data flow) and stop the function call, though this will lead to a denial of service attack.

Evading Our Introspection. HYBRID-BRIDGE assists VMI developers to reuse inspection tools for introspection purposes. However if system calls and well defined APIs used in inspection tools are not rich enough to do a introspection task then HYBRID-BRIDGE cannot help further. For example, if a Linux rootkit removes a malicious task from *task lined list* then an inspection tool which rely on *task lined list* to enumerate all the running processes would fail to detect the malicious task. Note that both VIRTUOSO and VMST also face this limitation.

More Precise Execution Context Identification for *Bi-Redirectable Instructions*. FAST-BRIDGE depends on the execution context to determine the correct date redirection policy for *bi-redirectable instructions*. While our current approximation design with CSC and PC has not generated any conflict yet, if our SLOW-BRIDGE really detects such a case, we have to resort other means such as instrumenting kernel code to add certain wrapper to further differentiate the context or developing kernel path encoding technique. We leave this as another part of our future work if there does exist such a case.

Reducing VMExit Overhead. In §VII-B, we showed that complicated introspection tools usually cause lots of *VMExits*, which are the main contributor to the FAST-BRIDGE performance overhead. Reducing *VMExits* would be an important immediate task. Part of our future efforts will address this problem. For instance, a possible way to improve the performance of FAST-BRIDGE is not to catch `int3` (no VM Exit) in hypervisor level. Instead, we can introduce an in-guest kernel module and patch the `int3` interrupt handler to switch the page table entries.

Supporting Kernel ASLR. HYBRID-BRIDGE currently works with Linux kernel, which so far has not deployed the kernel space address space layout randomization (ASLR) yet [17]. Addressing kernel ASLR for recent Windows-like system is another avenue of future work.

IX. RELATED WORK

Virtual Machine Introspection (VMI). A common practice to achieve better security is through strong isolation from untrusted environment. Early introspection solutions such as Copilot [45] employs two separate physical machines to provide isolation and uses a PCI card to pull the memory to the monitoring system. Recently, VMI has been increasingly used in many security applications to provide strong isolation. For instance, we have witnessed VMI being used in intrusion detection [22], [43], [44], [15], [19], [20], memory forensics [26], [15], [19], process monitoring [51], and malware analysis [13], [31].

VProbes [4], a general purpose VMI framework, provides basic operation to interact with VMM and guest OS but introspection developer is responsible to traverse the guest OS kernel data structures and find the required data in kernel. VProbes does not provide any automatic mechanism to address the semantic gap.

Min-c [29] is a C interpreter which extract kernel data structure definitions automatically to assist VMI developers. While *Min-c* cuts the development time of introspection tool, it is different from HYBRID-BRIDGE, VIRTUOSO [15], VMST [19], and EXTERIOR [21] (a guest VM writable extension of VMST) in a sense that it does not provide any support to reuse the existing inspection tool and the VMI developer must develop the introspection tool from scratch.

VIRTUOSO, VMST, EXTERIOR, and HYBRID-BRIDGE pursue the same goal, namely, (automatically) bridging the semantic-gap through binary code reuse. HYBRID-BRIDGE outperforms VMST and VIRTUOSO by an order of magnitude in terms of performance overhead. Furthermore, HYBRID-BRIDGE supports a fall back mechanism to address the coverage problem, whereas VIRTUOSO does not have a reliable mechanism to solve this issue. Meanwhile, compared to VMST, HYBRID-BRIDGE has a novel decoupled execution component that runs a lazy taint analysis on a separate VM, which significantly reduce the performance cost.

Hybrid-Virtualization. While recently there are a number of systems which combine both hardware virtualization and software virtualization (e.g. TBP [27], Aftersight [32], and V2E [55]), they have different goals and different techniques. In particular, TBP detects malicious code injection attack by using taint tracking to prevent execution of network data. The protected OS is running on Xen and uses page fault to switch execution to QEMU for taint-tracking when tainted data is being processed by the CPU. Aimed at heavyweight analysis on production workload, Aftersight decouples analysis from execution by recording all VM inputs on a VMware Workstation and replaying them on QEMU. Designed for malware analysis, V2E uses hardware virtualization to record the malware execution trace at page level, and uses page fault to transfer control to software virtualization; whereas HYBRID-BRIDGE uses `int3` patch to cause *VMExit* and control the transitions between *redirectable* and *non-redirecatable* instructions at instruction level as well as control the transitions to software virtualization.

Training Memoization. Memoization [39] is an optimization technique that remembers the results corresponding to some set of specific inputs, thus avoiding the recalculation when encountering these inputs again. This has been used in many applications such as deterministic multithreading (via schedule memoization [11]) and taint optimization (e.g., FLEXITAINT [54] and DDFT [30]).

While HYBRID-BRIDGE and FLEXITAINT [54] may seem similar at very high level regarding taint memoization but they operate in different world and face different challenges. FLEXITAINT is an instruction level CPU cache (very similar to Translation Lookaside Buffer) to enhance taint operation with low overhead in CPU, whereas HYBRID-BRIDGE is based on the idea of decoupling taint analysis from the main execution engine (i.e., FAST-BRIDGE) without any taint analysis inside it. For DDFT [30], the substantial difference is that its taint memoization works at user level program much like a compiler optimization to speed up the taint analysis, whereas HYBRID-BRIDGE works at hypervisor level with no intention to speed up the taint analysis itself. Also, our memoization not only does remember the tainted data, but also remember other types of meta-data such as the offset for each return address for bi-redirection instructions.

Binary Code Reuse. Recently, the concept of binary code reuse has gained a lot of attention, and been exploited to address a wide variety of interesting security problems such as malware analysis [8], [34], [56], attack construction [37], and VMI [15], [19], [21]. BCR [8] and Inspector Gadget [34] extract certain malware feature in a self-contained manner and reuse it to analyze the malware. Most recently, TOP [56] demonstrates that we can dynamically decompile malware code, unpack and transplant malware functions.

Dynamic Binary Code Patching. Dynamic binary patching tools such as DDT [35] have been around for more than 50 years. In the past decade, general dynamic binary instrumentation tools such as DynInst [7], DynamoRIO [1], PIN [38], and Valgrind [40] have been used for a wide variety of tasks include performance profiling [52], tracing [42], sandboxing [36], debugging [50] and code optimization [53]. In HYBRID-BRIDGE, we apply dynamic binary code patching technique that is often used by debuggers to set up break point on the monitored program, to trap the guest-OS execution to hypervisor and enforce data redirection policies.

X. CONCLUSION

We have presented HYBRID-BRIDGE, a fast virtual machine introspection system that allows the reuse of the existing binary code to automatically bridge the semantic gap. HYBRID-BRIDGE combines the strengths of both training based scheme from VIRTUOSO, which is fast but incomplete, and online kernel data redirection based scheme from VMST, which is slow but complete. By using a novel fall back mechanism with *decoupled execution* and *training memoization* at hypervisor layer, HYBRID-BRIDGE decouples the expensive execution of taint analysis engine from hardware-based virtualization such as KVM and moves it to software-based virtualization such as QEMU. By doing so, HYBRID-BRIDGE significantly improves the performance of existing solutions with one order of magnitude as demonstrated in our experimental results.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their comments. 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] Dynamorio dynamic instrumentation tool platform. http://dynamorio.org.

[2] Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide. http://www.intel.com/Assets/PDF/manual/253669.pdf.

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

[4] Vprobe toolkit. https://github.com/vmware/vprobe-toolkit.

[5] F. Baiardi and D. Sgandurra. Building trustworthy intrusion detection through vm introspection. In *Proceedings of the 3rd International Symposium on Information Assurance and Security (IAS'07)*, pages 209–214, 2007.

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

[7] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

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

[9] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 555–565, Chicago, IL. October 2009.

[10] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS'01)*, pages 133–138, Elmau/Oberbayern, Germany. 2001.

[11] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*. October 2010.

[12] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *Proceedings of USENIX Security Symposium*. August, 2012.

[13] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)*, pages 51–62, Alexandria, Virginia. October 2008.

[14] B. Dolan-Gavitt. Virtuoso: Whole-system binary code extraction for introspection. https://code.google.com/p/virtuoso/.

[15] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (SP'11)*, pages 297–312, Oakland, CA. May 2011.

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

[17] J. Edge. Randomizing the kernel, 2013. http://lwn.net/Articles/546686/.

[18] B. Fabrice. Qemu, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC'05)*, Anaheim, CA. June 2005.

[19] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP'12)*, pages 586–600, San Fransisco, CA. May 2012.

[20] Y. Fu and Z. Lin. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. *ACM Trans. Inf. Syst. Secur.*, 16(2):7:1–7:29, Sept. 2013.

[21] Y. Fu and Z. Lin. 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. March 2013.

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

[23] R. P. Goldberg. *Architectural Principles of virtual machines*. PhD thesis. PhD thesis, Harvard University. 1972.

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

[25] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011)*, pages 147–156, Madrid, Spain. October 4-7, 2011.

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

[27] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'06)*, pages 29–41. 2006.

[28] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. 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)*, pages 279–290, Newport Beach, California. March 2011.

[29] H. Inoue, F. Adelstein, M. Donovan, and S. Brueckner. Automatically bridging the semantic gap using a c interpreter. In *Proceedings of the 2011 Annual Symposium on Information Assurance (ASIA'11)*, Albany, NY. June 2011.

[30] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'12)*, San Diego, CA. February 2012.

[31] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, pages 128–138, Alexandria, Virginia. October 2007.

[32] P. M. Chen, J. Chow, and T. Garfinkel. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC'08)*, pages 1–14, 2008.

[33] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[34] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of 2010 IEEE Security and Privacy (SP'10)*, Oakland, CA. May 2010.

[35] A. Kotok. Dec debugging tape (ddt). *Massachusetts Institute of Technology (MIT)*, 1964.

[36] W. Li, L.-c. Lam, and T.-c. Chiueh. Accurate application-specific sandboxing for win32/intel binaries. In *Proceedings of the 3rd International Symposium on Information Assurance and Security (IAS'07)*, pages 375–382, Manchester, UK. 2007.

[37] Z. Lin, X. Zhang, and D. Xu. Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS 2010)*, Chicago, IL. June 2010.

[38] 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 *ACM SIGPLAN Notices*, volume 40, pages 190–200, 2005.

[39] D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218(5136):19–22, Apr. 1968.

[40] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*, pages 89–100, San Diego, CA. 2007.

[41] J. Newsome and D. Song. 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. February 2005.

[42] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk. Controlling program execution through binary instrumentation. *SIGARCH Comput. Archit. News*, 33(5):45–50, Dec. 2005.

[43] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*. December 2007.

[44] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP'08)*, pages 233–247, Oakland, CA. May 2008.

[45] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - A coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium (Security'04)*, pages 179–194, San Diego, CA. August 2004.

[46] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium (Security'06)*, Vancouver, B.C., Canada. August 2006.

[47] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security (IWSEC'11)*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. November 2011.

[48] N. A. Quynh. Operating system fingerprinting for virtual machines, 2010. In DEFCON 18.

[49] M. Rajagopalan, S. Perianayagam, H. He, G. Andrews, and S. Debray. Biray rewriting of an operating system kernel. In *Proc. Workshop on Binary Instrumentation and Applications*, 2006.

[50] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATC'05)*, pages 2–2, Anaheim, CA. 2005.

[51] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS'11)*, pages 363–374, Chicago, Illinois. October 2011.

[52] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd symposium on Operating systems design and implementation (OSDI'99)*, pages 117–130, 1999.

[53] A. Tamches and B. P. Miller. Dynamic kernel i-cache optimization. In *Proceedings of the 3rd Workshop on Binary Translation*, 2001.

[54] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture (HPCA'08)*, Salt Lake City, UT. 2008.

[55] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin. V2e: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE'12)*, pages 227–238, London, UK, 2012.

[56] J. Zeng, Y. Fu, K. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation-resilient binary code reuse through trace-oriented programming. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, Berlin, Germany. November 2013.