Retrofitting LBR Profiling to Enhance Virtual Machine Introspection

Weijie Liu*, Ximeng Liu† *Senior Member, IEEE*,
Zhi Li‡, Bin Liu¶, Rongwei Yu‖, and Lina Wang‖
*Luddy School of Informatics, Computing, and Engineering, Indiana University Bloomington, USA
†College of Mathematics and Computer Science, Fuzhou University, China
‡School of Cyber Science and Engineering, Huazhong University of Science and Technology, China
¶School of Computer and Information Engineering, Hubei University, China
‖School of Cyber Science and Engineering, Wuhan University, China

***Abstract*—Cloud security auditing is a well-established industrial practice for assuring transparency and accountability for a service provider to tenants. However, the multi-tenancy and self-service nature coupled with the sheer size of a cloud implies many unique challenges to cloud forensics. Although Virtual Machine Introspection (VMI) is a powerful tool for attack provenance due to the isolation and high privilege of lying in the hypervisor, the stealthiness of state-of-the-art attacks and the lack of precise information make existing auditing solutions are difficult to fulfill real-time forensics when tracking enormous suspicious behaviors.**

To this end, we propose an instruction-level tracing framework for inspecting the presence of attacks by dynamically tracking shared processor hardware event patterns and analyzing the attack traces. To overcome the challenges of real-time detection and auditing, we advocate Last Branch Record (LBR) profiling, to extract the suspicious execution flows. With the hardware assistance and software-based virtualization introspection, we show that the framework can provide an effective response to threats in different cases, thereby enabling a quick attack provenance with high fidelity. The evaluation shows that our prototype introduces negligible performance penalties.

***Index Terms*—Cloud Forensics, Virtual Machine Introspection, Last Branch Recording.**

I. Introduction

CLOUD service providers always encourage the perception that letting their customers see what is behind their virtual curtain can ease the insecurity to some extent. As a service publisher and platform provider, it is necessary for them to have the ability to detect most kinds of attacks on guest virtual machines in real-time. Virtual machine introspection (VMI), as a cloud forensics method [1], is a compelling technique to enhance system security which can provide strong isolation between untrusted guests and security tools placed out of the guests [2], [3], [4].

The performance of VMI-based Intrusion Detection Systems (IDS) varies widely along several dimensions. Software-only methods suffer the most [5]. Worse yet, due to the lack of attack provenance methods and inaccurate attack tracing technology, an all-around approach to trace a transient attack and to get the digital proof is rare and hard to meet the need for a high precision audit [6]. Compared to the prior hardware-based and software-based defense mechanisms using Hardware Performance Counter (HPC) [7], statistical and numerical data can not represent the high-level semantic information of the guest virtual machine, nor can it locate the instruction level exception, thus leading to a moderate detection rate [8].

Corresponding Author: Ximeng Liu (snbnix@gmail.com)

To track attacks in modern cloud systems, this paper uncovers a more practical and efficient approach by focusing on an Intel CPU feature - Last Branch Recording (LBR), which is a set of loop registers provided by Intel to record the source address and destination address of the latest 32 jump instructions. We propose a tracing system - HYBRID - to locate and analyze real-time state-of-the-art attacks with higher precision and robustness. The main idea of our framework combines software information provided by forensic tools to restore the semantic information of the underlying user program, and the hardware characteristics of modern CPU to identify and distinguish the recorded events. With this specific hardware feature, it is clear some benefits can be delivered such as better performance due to hardware-involved speed up and higher precision due to precise address information retrieved from LBR. An obvious advantage observed is the low-level (hypervisor-managed) memory address translation could be achieved rapidly. Moreover, explicit information about the software abstractions of the operating systems running on them could be easy for bridging the semantic gap, which can further facilitate understanding attack behaviors. To this end, cloud forensic tools can be of greater value to virtual machine introspection with our framework.

Challenges. However, enabling LBR for accurate and efficient VMI is difficult and can not be overlooked. Limitations such as virtualizing LBR are not implementation issues that can be fixed easily but originate from fundamental design challenges that result in conflicts with current virtualization techniques. Secondly, a known issue with measurements involving interrupts is "overflow", wherein LBR stack will be flushed transiently if the processor does not pinpoint exactly which instruction was active at the time of the attack, resulting in a discrepancy between the instruction indicated versus the one being recorded. And in particular, Hypervisor is the main resource manager of the system, where lie system-level services and specific types of security monitoring. However, performing such complex tasks at the Hypervisor level requires a high level of knowledge at the Operating System (OS) and program levels. An important issue of VM introspection is the lack of higher-level knowledge, which is also called the "semantic gap".

Independent introspection technology meets the transparency requirements [9] in the cloud environment and the mainstream trend of the hardware virtualization environment. Part of these solutions is based on software structure knowledge restoration to ensure that the underlying data is reconstructed without losing isolation, such as using the *task_struct* that describes the process structure in the known Linux kernel to restore the process list and using the *init_task* symbol to locate the process list Header to get complete process information at OS level [10]. LibVMI [11] is the representative of this kind of solution. It added a certain semantic restore function for fetch the relevant OS metadata. Although there seems almost no limit to the range of memory information obtained in the introspection technology based on software knowledge, the code used for introspection of the virtual machine is very likely to fail when the VM memory changes. Namely, an attacker can use this failure to invalidate the introspection program.

Contributions. The contributions are outlined as follows.

- New techniques to support LBR on virtualization platforms. We emphasize that the limitations of existing VMI technologies can be overcome by retrofitting the usage of LBR. As we will show, bridging the semantic gap can be accomplished in a virtualized setting.
- Transparent attack provenance approaches. A hardware information transferring method is proposed and designed to fit in multiple cases, where we can provide accurate intrusion auditing using indirect branch validation algorithms.
- A practical cloud forensic framework with solid proof of concept. HYBRID is easy to deploy and can be scheduled on-demand, which allows our solution can scale to tens or hundreds of VMs in cloud settings, making the whole tracing process real-time and efficient. We release the source code of our techniques online [12].

Roadmap. The remainder of the paper is organized as follows. In Section II we explain our background knowledge and related work. Section III-B elaborates fundamental challenges in deploying HYBRID for virtualization environment. We outline our design in Section IV and provide in-depth details about our prototype implementations in Section V. Section VI shows two cases where HYBRID could be put in use - for tracing ROP attacks and side channel attacks. Section VII describes performance evaluation of the prototype. Section IX elaborates related work and finally, Section X provides brief concluding remarks of our framework.

II. Background

A. Virtual Machine Introspection

VMI's purpose is to check the software running status of the virtual machine outside the virtual machine [13]. So, this kind of design was not considered as transparent monitoring, nor was it completely isolated. Since the hypervisor provides strong isolation, it is very difficult to modify or intervene in software running in the hypervisor or the standalone VM [14]. Even if the intruder has full control over the monitored VM, it cannot touch the security tools like the intrusion detection system. On top of that, hypervisor's introspection means that the hypervisor can get all status of the monitored VM: registers, memory, I/O device, etc. This makes it difficult for any VM to escape the monitoring, and the VMI-based malware detection method can fully exploit the high-level privileges for making itself more comprehensive and safe. For example, only minor changes to the hypervisor are required, a VMI forensics tool can be applied to malicious behavior tracing (e.g., for locating cross-VM side-channel operations).

Figure 1 demonstrates the basic working principle of the VMI system. Trusted parts are shown in blue. The VMI-based IDS in the hypervisor collects required hardware, memory and other information to be monitored, and then analyzes the execution of the target virtual machine in a secure/privileged virtual machine. On the right of Figure 1 is the VM running the monitored guest application. The left side is the IDS based on VMI technology, including the following components: (1) the secure VM's interface library (security driver) provides a perspective on the VM system level by explaining the hardware and software status; (2) the security interfaces of the Guest-OS interprets the guest's behavior on both application layer and kernel layer and passes it to the hypervisor; (3) the secure policy compliance module (Introspection API) in the Secure VM executes a specific intrusion detection engine. The hypervisor provides a basis for isolating the IDS from the monitored VM while allowing the IDS to detect the status of the VM.

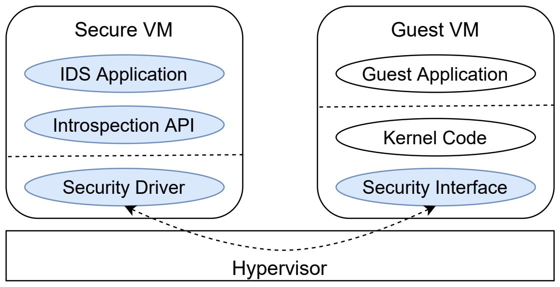

Fig. 1: Traditional VMI-IDS architecture

B. Last Branch Recording

To better monitor the performance and tune programs, CPU manufacturers have integrated a series of hardware support to assist debugging on the CPU. The Last Branch Record (LBR) is a set of circular register stacks that can record the source and destination addresses of the most recent 16/32 branch instructions on the CPU. However, LBR stores these addresses in a round-robin fashion in a set of MSRs. When the LBR buffer is full, LBR will not generate an interrupt, and as a result, the oldest record will be flushed away by the upcoming record [15]. Due to the high density and precision of the branch jump address recorded in the LBR, those entries can be used to record some branch information of the current process. Thus, LBR has a mechanism based on branch instruction type filtering.

Besides, since the branch recorder does not have an overflow interrupt mechanism at this stage, reading the data directly will cause the value is not exactly the value we want. Owing to the error caused by the difference in time, it would result in very bad accuracy in locating attacks. This is also known as the 'skid' problem of hardware counting [16]. To solve this problem, the Performance Monitoring Units (PMU) interrupt mechanism can be introduced. Modern superscalar processors schedule and execute multiple instructions out-of-order at one time. These "in-flight" instructions can retire at any time, depending on memory access, hits in cache, stalls in the pipeline, and many other factors. This can cause performance counter events to be attributed to the wrong instructions, making precise performance analysis difficult or impossible. CPU vendors introduced methods to mitigate some of these drawbacks, e.g., the Instruction Based Sampling (IBS) or Event-Based Sampling (EBS) [17]. Together with these techniques [18], the PMU can be configured to generate an interrupt whenever a counter overflows. When the interrupt triggers, performance monitoring software can record the system state, including the LBR. To collect PMU right, performance monitoring software uses the Perf tool in Linux and the open-source library Libpfm [19], for enabling IBS or EBS. The Perf event supports two modes of collecting HPC status: counting mode and sampling mode. When the HPC reaches a predetermined threshold, an overflow non-maskable interrupt (NMI) is triggered.

III. Problem Statement

In this section, we articulate the threat model and technical challenges in designing HYBRID.

A. Adversary Model

The basic assumption is the CPU hardware is trusted, which is reasonable since it is highly impossible to tamper with the circuit in a cloud platform as a remote user. We also assume that the hypervisor is secure and trusted, as well as the privileged Domain-0 (Dom0). All guest-VM kernels, aka. Domain-U (DomU kernel) should also be trusted, since we cannot eliminate the Direct Kernel Object Manipulation (DKOM) issue. However, our framework can be improved in the future to defend against the DKOM attack, which is discussed in Section VIII.

B. Technical Challenges

Emulating LBR hardware. When monitoring the virtual machine at a low-level view, especially at the attack provenance phase, a tough problem needs to be solved in hypervisor - how to distinguish the hardware events from different virtual machines. Unfortunately, mainstream CPU manufacturers and hypervisor vendors do not implement such LBR virtualization function. As shown in Figure 2, in a virtualization environment, vCPU is an important data structure for solving time division multiplexing of virtual machines. It is assigned to each virtual machine Dom as the current physical CPU of the operating virtual machine Instance. However, the CPU scheduling mechanism makes the CPUs corresponding to vCPUs change at different times. Therefore, the content of hardware information such as LBR in the current CPU is not necessarily generated by the current virtual machine. Unfortunately, current mainstream hypervisors such as Xen/KVM do not provide a emulation/virtualization mechanism for special registers such as LBR. Thus existing hardware-assisted detection methods cannot be directly applied in virtualized environments. How hypervisor uses real hardware information correctly and interprets it as a signature for tracing every user process is exactly what we should address.

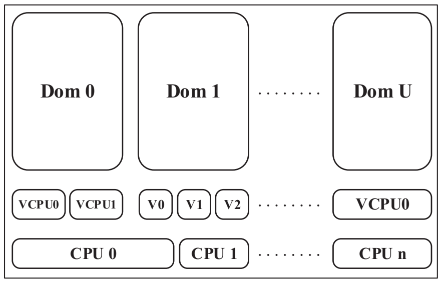

Fig. 2: CPU to vCPU mapping

Overflow issue. According to Chen et al. [20], errors in the distribution of samples are the result of three major factors: (1) synchronization of monitored code with the sampling period, (2) the sampling skid effect, when the address reported by the hardware sample does not necessarily match the address of the instruction causing counter overflow, and (3) the sampling shadow effect when instructions in the shadow of a long latency instruction get low sample counts. If the data sampled by LBR is not accurate, it will be hard to locate the specific instruction address of the attack, thus seriously affecting the correctness of the tracking system. This is particularly important when monitoring whether there is a transient execution attack.

Semantic gap. Since VMI obtains this state information directly from the hypervisor, the semantic knowledge of the guest operating system's abstractions is lost. In addition, all current solutions to the semantic gap problem implicitly assume the guest OS is benign. Although this is a reasonable assumption in many contexts, it can become a stumbling block to the larger goal of reducing the size of the trusted computing base. Automated reconstruction tools may rely on source code analysis or debugging information to extract data structures definitions, as well as leverage sophisticated static analysis and source invariants to reduce false positives during the search phase. However, it's hard to get full expose of guest-VM's source code sometimes. Rather than identifying code invariants from kernel source code, VMI-based dynamic analysis learns data structure based on observing an OS instance [13], yet getting a robust signature is a tough issue.

IV. Design

In this section, we present our design for realizing a hardware-assisted VMI framework, which leverages the LBR with support for multiple purposes. Following we first describe the general idea behind our design and then elaborate on the adversary model, and its components.

A. *Idea*

As we discussed in Section II-A, independent introspection technologies based on hardware architecture knowledge can provide a novel blueprint. These technologies allow VMM to better manage the resources of the system and also try to solve the address translation problems [21], [22]. The transparency of the independent introspection technology based on the knowledge of the hardware architecture is as good as the previous methods. Also, compared with the introspection technology based on software architecture knowledge, the sampling frequency could be much higher, leading to a richer dataset. Moreover, it is difficult for an attacker to change the hardware architecture of the operating system, which makes it more robust. Therefore, we propose a hybrid threat tracing method based on the combination of virtual machine software introspection technology and hardware knowledge.

In particular, we elaborate how we address all challenges highlighted in Section III-B.

C1 - LBR virtualization. We add a thin layer between the hypervisor and the hardware, to implement the function of fetching LBR values in different vCPUs. Also, we leverage the mapping between the vCPUs and Domains maintained by the hypervisor, to hand off fetched LBR values with the correct Domain identifier for later provenance tracing.

C2 - Avoiding overflow issue. Since the LBR includes multiple entries, the sampling naturally occurs in a burst. Carefully crafted LBR control is necessary although this produces more profiles [20]. We use a hardware-assisted approach to serialize the sampled data flow, which forces a command to go through a core with all the caches pre-flushed and waits for all buffered writes to have finished before starting. As a result, LBR-based sampling can derive much more accurate profiles than the traditional sampling-based approach.

C3 - Semantic reconstruction. Due to the nature of the hardware information itself, the amount of information obtained for the introspection program is less versatile and more genuine than approaches based on software knowledge. We translate the addresses into readable thread information with the help of kernel data structures.

There are several other key technical challenges in implementing the proposed tracing framework. (1) Transparency, which is, the execution of sensitive operations that identify the protected VM without requiring guests to modify their applications; (2) Minimizing performance impact, that is, we should reduce the overhead of event tracing as much as possible while pinpointing abnormal usage patterns of multiple untrusted virtual machines. Our design overcomes both by integrating minimal components into the hypervisor, as well as using low-level inline assembly to interact with hardware-specific ISA. In practice, by maintaining a lookup table that contains mappings between hypervisor physical address (HPA) to guest-VM physical address (GPA), we shorten procedures that need to do address translation frequently.

B. *LBR enhanced HYBRID*

We propose HYBRID, a novel technique for instruction-level tracing. Figure 3 shows the overview design.

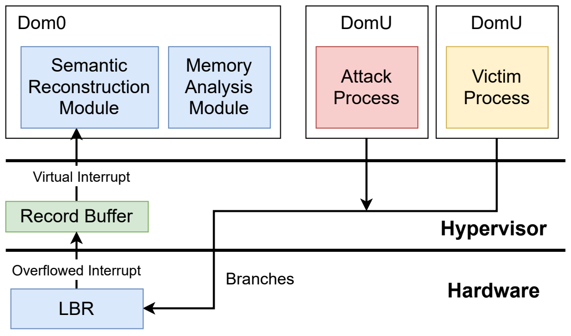

Fig. 3: HYBRID framework overview

The workflow of the HYBRID for the guest virtual machine is described as follows. It can be divided into two parts. The first one is *hardware-assisted event recording*. When LBR is enabled and branches are recorded, these records would be stored temporarily in the hypervisor and transferred to Dom0 via virtual interrupts eventually.

In the second part *attack provenance*, OS metadata will be used for semantic reconstruction and further memory analysis. Particularly, critical hardware events will be recorded for investigation. The tracing module uses software-based introspection to reconstruct the process context, and a potential attacker could be identified and located in a short time with the help of recorded hardware information. One kind of memory manipulating needed to be investigated is memory write operations. Our probe takes a register name, an offset value, and the value which should be written to memory, as parameters, which can be gathered with the branch records in *phase 1*. Besides, the whole *phase 2* can be done offline, allowing HYBRID only to occupy computing resources when necessary, thus reducing its performance overhead to some extend.

In general, HYBRID architecture can keep track of instructions that read/write from memory addresses and provide functions to print the results (e.g., the hex of a chunk of bytes) from memory addresses. The goal of HYBRID is to locate the target of the attack and aid the tracing ability of VMI. Still, any real-time detection techniques can be integrated into HYBRID, to help mitigating attacks in multi-tenant cloud systems. Once malicious behavior is scanned and traced, the virtual machine migration mechanism or other mitigation schemes can be used to move the victim's virtual machine out of the attack scope physically on which it is located, thereby avoiding the second occurrence of the attack.

C. *Phase 1 - Hardware-assisted Event Recording*

As shown in Figure 3, DomU can start the tracing process at the moment of executing the system call and then entering the kernel/hypervisor state by setting a fast system call or hypercall in advance. For example, if a privileged domain initializes the tracing, a customized hypercall can be invoked, to configure and enable LBR and other auxiliary hardware (such as performance counters), making sure that LBR values can be injected with an overflowed interrupt serially (detailed in Section V-B).

In every VM-exit, current branch jump information obtained in the LBR can be sent to the hypervisor by piggybacking the interrupts when PMC sampling mode is turned on. The genuine value LBR information, the virtual machine ID, the time stamp TSC, and the base address of the process page table pointer stored in the CR3 register are recorded in the memory buffer area. Once such information is sampled, the system resumes DomU execution via VM-entry.

D. *Phase 2 - Attack Provenance*

Hardware-level probes gather data as seen by the hardware without any semantic information, and operating-system-level probes gather information by using external knowledge about the operating system. Although there is at least one OS-level approach that frees users from the need to provide such external knowledge themselves, instead of automating the process [23], we prefer our hardware-level approach since it leads to better performance, since at this phase, LBR can still help for semantic reconstruction. Data structure reconstruction generally relies on a learn and search methodology. A learning phase is used to extract information relevant to data structures, generally a data structure signature. A signature can be used to identify and reconstruct data structure instances within kernel memory contents.

Forensic memory analysis (FMA) starts with the view of static memory. Such techniques can be used to address the semantic gap problem in both settings [24]. After Dom0 obtains the recorded data, we leverage such techniques to do memory analysis. It first separates and reorganizes the data according to the virtual machine ID, and uses the VMI technology to get the virtual memory area (VMA) information in the DomU process description structure (task_struct), to accurately locate each section in the running process. Then, the jump instruction address in the LBR record is converted into the offset in the corresponding memory space, and the comparison is performed one by one in the memory analysis module to trace the specific threat process. The whole procedure does not need to modify the DomU kernel or the executable. And DomUs do not need to actively cooperate to pass data via other specific channels, which is completely transparent to guest virtual machines. In the meantime, to be able to audit the events captured in phase 1 of our design, superusers should perform authorization checks through a special API exported by the hypervisor. This is to prevent sensitive system activity information from being used by attackers.

We also implement a fast address space search plugin (detailed in Section V-E) to assist our FMA module. Moreover, since we only use the information stored in CPU registers/hardware (`CR3` values, EPT entries, and LBR values) to reconstruct the data structure.

V. Implementation

In Section IV, we discuss how their approach can be generalized to other hardware/hypervisors. Here, we instantiate the design and implement the prototype on Linux/Intel x86_64 architecture. Specifically, our framework uses the Xen virtualization architecture as the basic VMM and uses the Intel i7-6700K as the example CPU to demonstrate the design. The implementation introduces about 1 kLoCs (C/Assembly) in Xen and about 2 kLocs (C/C++) in LibVMI totally.

A. *LBR Configuration*

LBR could be turned on by the lowest bit of the `IA32_DEBUGCTL` register. Yet if all jumps of the virtual machine are recorded constantly, the amount of information generated is undoubtedly large and it would lead to a high redundancy, which affects the detection performance seriously. As the attacker wants to perform further malicious manipulation and destruction after hijacking the control flow, it is bound to make key system function calls. Therefore, our design only records necessary jump information, while maintaining high detection effectiveness while reducing overhead.

Intel provides a 64-bit `MBR_LBR_SELECT` register to set branch jump type filtering. When the bit is '1', it indicates that the corresponding type of jump instruction is filtered. Since our design focuses on user-level jumps mostly, the flags listed in the Table I are kept at '0', that is, 0x1 is written to the `MSR_LBR_SELECT` register through the WRMSR instruction. Code for enabling this configuration (Figure 4) can be applied to similar architecture with minor modifications.

TABLE I: MSR_LBR_SELECT Settings

Flag	Name	Description
1	CPL_NEQ_0	User-level jump
63:10	-	Reserved bits must be set to 0

Importantly when reading LBR, an inline assembly can be used for setting configurations (shown in Figure 5). We need to first obtain the series and model number of the CPU of the physical host to determine the number how many LBR entries that should be read. This part of implementation is completed on Intel i7 Skylake series processors, which have 32 pairs of LBR entries. In addition, on some types of processors (e.g., Intel Haswell and older versions), even if the LBR flag is cleared in the `IA32_DEBUGCTL` MSR but the TR flag remains set to 1, the CPU will continue to record new branch information to update the LBR stack.

```
void enable_lbr(){
   asm volatile (  "xor %%edx, %%edx\n"
       "xor %%eax, %%eax\n"
       "inc %%eax\n"
       "mov $0x1d9, %%ecx\n"
       "wrmsr" ::   );
       wrmsrl(MSR_LBR_SELECT, 0x1);
}
```

Fig. 4: Enabling LBR.

B. *Fetching LBR values during VM exits*

When setting the LBR in a virtualized environment, a related field in the Virtual Machine Control Structure (VMCS)

```
void read_lbr(void* info){
   tos = intel_pmu_lbr_tos();
   for(i = 0; i < NUM_MSR; ++i){
       idx = (tos - i) & (NUM_MSR - 1);
       msr_from = MSR_LASTBRANCH_0_FROM_LIP + idx
    ;
     msr_to = MSR_LASTBRANCH_0_TO_LIP + idx;
   asm volatile(   "mov %4, %%ecx\n"
       "mov %%eax, %0\n"
       "mov %%edx, %1\n"
       "mov %5, %%ecx\n"
       "mov %%eax, %2\n"
       "mov %%edx, %3"
       : "=g"(ax1f), "=g"(dx1f),
           "=g"(ax1t), "=g"(dx1t)
       : "g" (msr_from),
           "g" (msr_to)
       : "%eax", "%ecx", "%edx");
   lbrentries[i].from = ax1f;
       lbrentries[i].to = ax1t;
 }
}
```

Fig. 5: Reading LBR during VM exits.

```
cpu = current->arch.hvm_vmx.active_cpu;
on_selected_cpus(cpumask_of(cpu), read_lbr,
    current, 1);
/* enable the LBR */
_intel_lbr_enable();
__vmwrite(GUEST_IA32_DEBUGCTL, 1);
```

Fig. 6: Setting LBR during VM exits

corresponding to the virtual machine needs to be set at the same time. And for the sake of no overflow issue, each time a VM exit occurs, the hypervisor sets it in the VM exit handler, specifically expressed in C (shown in Figure 4).

In the typical code scenario of modern CPU processors, the branch records updated per second may be as large as one million, and the effective capacity of the LBR stack is only 32. The correct way for the processor to stop LBR recording immediately is to use the Performance Monitoring Interrupt (PMI). This feature is enabled by setting bit 11 of the `IA32_DEBUGCTL` register first. We use the CPU performance counter to set the PMU sampling mode to issue a PMI when the branch count indicator exceeds the threshold, which serializes the flow of fetched values, therefore solving the skid issue mentioned in Section III-B.

Once the data in the PMU/PMC reaches the threshold, the PMU/PMC sampling mode is triggered. When the PMC processing interrupt freezes (detailed in Section V-B) the LBR, the `IA32_DEBUGCTL` bit 6 - the TR flag - is also cleared. In our solution, an interrupt is issued and caught by our modified hypervisor. The record module located in the hypervisor is notified immediately to record the current LBR values. Meanwhile, VMCS will keep track of the status of registers including `CR3` and `RIP` of current process in DomU [25]. Our design thus uses different `CR3` values to distinguish protected processes. Specifically, when PMU monitors all processes executed in the guest virtual machine, there are two situations to consider: no context switch occurs during the information collection interval, and a context switch occurs during the interval. When there is no context switch during the non-maskable interrupt (NMI) interval, overflowing NMI causes the VM to exit, and the processor will switch to root mode; at this time, the system will check whether the VM exit is caused by NMI overflow by comparing the VM exit reason field; then it proceeds to the next detection/tracing operation. When there is a context switch during the NMI interval, multiple processes in the guest-VM will accumulate the value of PMU, causing a false positive. To avoid this problem, we store the signature (such as the `pgd_t` field) of each user-level process of the guest virtual machine within the interval. When VM-entry happens, `CR3` value in VMCS can be saved and a correct `RIP` can still be found [1].

C. *Record Transferring*

Although sometimes *Phase 1* is enough for attack detection (examples shown in Section VI), we need a record transferring mechanism, to convert collected information to a readable format, though. To prevent the value of LBR and other registers from being covered and polluted, the hypervisor stores the auxiliary information that is related to the virtual machine into a memory buffer upon the first arrival of LBR profiling.

We take advantage of the architecture of existing Xen hypervisors to implement the record distribution and delivery module. Our implementation does not use a separate daemon to trigger the recording process. Instead, it adds lightweight code to avoid disturbing the state of the system and records hardware values such as performance counters as accurately as possible. To further reduce interference effects, the analysis and tracing module can run on a specific CPU core.

Here, we introduce *Xentrace*, to realize the information distribution and transmission channel by retrofitting it. Our design reuses its event recording mechanism and message passing mechanism and designs a mechanism that can quickly pass the guest VM's VMI detection requirements along with the parameters required by the VMI program to Dom0. This mechanism has universality, does not add too much source code, and has minimal impact on the performance of the system. It maintains the system's security and portability to the greatest extent. We establish a special event format and event parser for recording LBR data to store and parse its data. In addition to the DomU identification (Dom_ID) and the timestamp, the hypervisor also needs to provide an EPTP index to inform the VMI program of the EPT page table base address. The format of each event record in the record buffer is shown in Figure 7.

TSC	Event Mask	Dom_ID	LBR_INFO

Fig. 7: Event record format

The specific operation steps of this module are as follows. Firstly, Xentrace is started up in Dom0. Then, the hypervisor detects whether there is a new virtual interrupt injection, and if it exists, parses the incoming parameters according to the above event format. After that, the module calls the

[1]In Linux kernel, the last 32-bit value of the pgd field is equivalent to the last 32-bit value in `CR3`.

virtual machine introspection program in Dom. Further, it passes the parameters into the VMI program to Xentrace's core function `TRACE_ND()`, and writes data into the buffer provided by hypervisor through this function. Finally, virtual interrupt `VIRQ_TBUF` will be injected to Dom0.

D. Semantic Reconstruction

After the user-level branch addresses are retrieved from the previous step (see Section V-C), they will be further handed off to the last attack provenance modules (the semantic reconstruction module and the memory analysis module) at Dom0.

It should be noted that the object detected in our tracing system is likely to be a function in a shared library, and the 32 LBR indirect branch addresses extracted by the parser in the privileged domain Dom0 are linear. To get the offset information of each shared library, the corresponding base address needs to be subtracted before it can be used for accurate tracking. In addition, when address space randomization (ASLR) is enabled in DomU, the loading base address of the same shared link library in different applications is random. Here we describe how to use VMI technology to obtain the memory descriptor of the process to which the instruction belongs, and collect the load base address of each shared library from the memory mapping information.

The operating system has a corresponding data structure to store the memory mapping information of the process. As shown in Fig. 8, our design takes the Linux system as an example and uses the method provided by LibVMI [26] to obtain the initial address and offset of the data structure (`init_task`). We linearly scan the linked list `task_struct` to find the process corresponding to CR3, until the reconstruction of each data structure is done. Then we look for `mm_struct` in a certain order to restore the process ID (Pid) where the LBR information was captured. Our prototype uses a lookup table to reverse the obtained CR3 information, and the field `pgd_t` can be obtained further. More specifically, the lookup table is set up for rapid translation from HPA to GPA. Those important fields are parameters that need to be passed to the memory management unit (MMU) for address translation. The integrity of the whole semantic reconstruction can be ensured since we get them from the genuine hardware LBR. After scanning all the processes of all VMs, the correspondence between the process Pid and its `pgd_t` is established and it will be stored in a lookup table which is updated in real-time. Later, when the process corresponding to `CR3` needs to be queried, the process number can be obtained by simply looking up the table.

E. Memory Analysis

Real-time memory analysis on the Linux platform is not easy to perform. It requires precise knowledge of the structural layout information in memory, usually obtained by debugging symbols generated during compilation. Here, we use address information (GPA) obtained from the last semantic reconstruction stage to facilitate memory analysis. Particularly, this GPA could be translated to a guest-VM virtual address (GVA) using our Rekall-based plugin. Rekall is an advanced

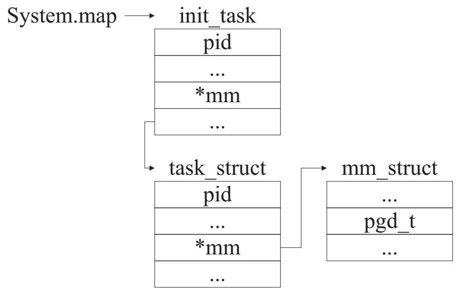

Fig. 8: Critical kernel varibles

auditing and incident response framework [27]. As a memory analysis and forensics framework, it can now be integrated on VMI tools/platforms (such as LibVMI). With the Rekall-based address space plugin and the generated semantics information, we can locate important kernel data structures more accurately, instead of relying on heuristics or feature-based vague threat locations like other tools.

Particularly, our plugin can take over from the address translation shown in Fig. 8. After semantic reconstruction (Section V-D), the branch addresses will be translated to virtual addresses for further forensic analysis with the help of certain tools. For example, an ROP attack can be traced if it uses some shared libraries, with the LBR equipped Rekall plugin. More specific cases will be illustrated in Section VI.

Moreover, combining Rekall and LibVMI, our framework can get all kinds of memory information in a guest-VM. During the analysis at this stage, the system suspends all virtual domains and uses the VMI program to read the current page frames. The current processes of all virtual machines will be traversed under Dom0 to locate the target process, and then the static analysis of Rekall will be performed. The tracing module extracts the Rekall image of the malicious virtual machine, through the interaction provided by the Rekall interface, find LBR address information, and locate the calling function name. For instance, to trace a memory bus-based side-channel attack, the process can be considered malicious and an alert will be issued if the analysis matches frequent atomic functions such as `atomic_set()`, `atomic_fetch_add()`.

VI. Use Cases

In this section, we demonstrate how HYBRID can be used for detecting code reuse exploits and side-channel attacks. Virtual machines installed with Linux kernel 4.15 are deployed to host and simulate these use cases.

A. Tracing Code Reuse Attacks

The user of our method (usually the cloud providers or cloud administrators) would be happy to see a out-of-the-VM forensic tool can help them keep track of malicious behaviors inside a VM. Thus, the first use case that can benefit from our framework is the Return-oriented programming (ROP) attack, where attackers can control the return address in a thread's call stack or save the register of the program jump target to form a control flow hijacking via buffer overflow, UAF (use after free), or other memory corruption vulnerabilities. In the early exploits, the attacker directly redirects the hijacked control flow to shellcode which has been carefully arranged (injected) in the stack or heap for subsequent execution. To this end, indirect branch validation would be an efficient way to thwart code reuse attacks [28].

If the attacker makes the jump target of the control flow pointing to a function entry in memory and places the call parameters of the function on the stack, then the function can be called arbitrarily. If the called function is security-sensitive, such as `system()` or `execv()`, it can cause an effective attack without injecting code, It is often not enough to call a single function. By leveraging the `RET`-like instruction at the end of the function, the attacker can make the target address of the control flow jump always come from the data arranged on the stack, to realize the continuous call of scattered instruction fragments in memory, and form a large-scale code fragment reuse attack with specific functions.

In this case, detection should be triggered when intercepting the system call. More specifically, when the guest-VM makes a system call, it stores the code pointer into the *Sysenter_EIP* register, then our modified hypervisor intercepts the system call for the virtual machine, and simulates the system call handling routine. When DomU switches to the kernel mode, the control registers access field is configured. After that, the control will be transferred to the hypervisor for further processing (see Section V-B). Among the instructions executed by a program, the target address of the indirect transfer instruction is dynamically determined by the contents of the register or memory at runtime, which makes it possible for attackers to tamper with and hijack the control flow of the program. Therefore, we implement strict ROP detection by protecting the Control Flow Integrity (CFI) of shared libraries. In the offline preparation stage (in Dom0), static memory analysis is performed on application programs and their dynamically linked libraries. Table II shows the number of indirect branches in certified libraries. Indirect branch information (valid function entry, valid function exit, etc.) is stored in a pre-defined data structure, for offline analysis.

TABLE II: # of indirect branches in common shared libraries

Library	Size/KB	ret	Ind. call	Ind. jmp
libc.so.6	1 824	5 128	994	742
libpcre.so.3.13	448	278	66	98
libdl-2.23.so	16	34	18	25
librt-2.23.so	32	106	5	57
libz.so.1.2.8	104	180	52	25
ld-2.23.so	160	295	97	33
libssl.so.1.0.0	420	1 032	232	431
libcrypto.so.1	2 308	5 513	697	350

We design and implement a CFI checking plugin in Dom0 to validate calls/jumps in a program using common shared libraries. Algorithm 1 shows the validation performed by our customized phase 2. To better describe the algorithm, we give the definitions below.

Definition VI.1 (Lib_List)**.** The list of shared libraries linked against the program to be scanned.

Definition VI.2 (Entry_Set)**.** The set of every entry point in shared libraries in Lib_List. Each element in $Entry\_Set$ is a Bitmap, whose every bit is related to an offset in a shared library. A value of '1' means allowing control flow to jump into the current shared library at the offset within the segment.

Definition VI.3 (Exit_Set)**.** The set of every exit point in shared libraries in Lib_List. The element is a Bitmap struct as well, while each bit in the Bitmap is also related to a shared library. Value '1' means allowing control flow to jump out of the current shared library at this offset.

Definition VI.4 (LibValidEdges⟨Source, Target⟩)**.** A Hashmap that stores every edge in shared libraries. The source address of a valid branch is taken as the key, and the destination address is taken as the value.

Algorithm 1: Indirect Branch Validation Algorithm

```
Input  : LBR records $branch\_pairs[\,]$;
         $Lib\_List$;
         $Entry\_Set$;
         $Exit\_Set$;
         $LibValidEdges$.
Output: validation result, which is either Pass or Violate.
1  for branch in branch_pairs[ ] do
2      $source\_lib \leftarrow$ getIndex($Lib\_List$, $branch.from$)
3      $sink\_lib \leftarrow$ getIndex($Lib\_List$, $branch.to$)
4      $source \leftarrow branch.from$ - $Lib\_List[source\_lib].start$
5      $target \leftarrow branch.to$ - $Lib\_List[sink\_lib].start$
6      if source_lib != NULL && source_lib == sink_lib then
7          if ⟨Source, Target⟩ not in LibValidEdges[Lib_List[source_lib.name]) then
8              return Violate
9          end if
10     else if from_lib != NULL then
11         if source != Exit_Set[Lib_List[source_lib].name] then
12             return Violate
13         end if
14     else if to_lib != NULL then
15         if target != Entry_Set[Lib_List[source_lib].name] then
16             return Violate
17         end if
18 end for
19 return Pass
```

The input *Lib_List* and valid entry/exit addresses are calculated via Rekall based on the abovementioned VMA information. Specifically, after Dom0 obtains the recorded data, our plugin first separates and reorganizes the data according to the virtual machine ID, and obtains the DomU process description structure (task) by VMI technology. With the virtual memory areas (VMA) information in a struct, we can accurately locate the base address of the dynamically linked shared library. The jump instruction address of the LBR branch pairs is then converted into the offset in the corresponding link library and is compared one by one in the legal branch jump database collected offline, to ensure the integrity of the control flow. To accelerate the computation, we use bitmap for the search operations.

B. Tracing Side Channels

Cross-VM side-channel threats would usually bypass the traditional inspection mechanisms and these attacks are becoming a serious issue to public clouds [29], especially after several severe CPU vulnerabilities are exposed by researchers in early 2018 [30], [31]. Cloud providers are responsible for securing the guest-VM's running environment. Once a side-channel attack happens in the user-land of a guest VM, the hypervisor should be able to catch it on behalf of the cloud provider.

Nevertheless, building an anomaly-based method only with performance counters [7] is difficult for detecting side-channel attacks, since HPC sampling is neither trustworthy [32] nor efficient [8]. In our framework, we not only monitor the number of total events (such as cache miss) in counting mode but also set a certain value as the threshold of NMI overflow in sampling mode (see Section V-B). Hardware events (HPC values) are profiled along with the LBR at the instruction level, therefore attack gadgets can be recorded deterministically even when dealing with side channels.

During the execution of secret programs such as RSA encryption and decryption (especially, the Montgomery modular multiplication algorithm), the point of attack exists in a large number of branch jumps in the control flow. We detect side channels by inspecting the hardware events that happen during LBR profiling. The first step in detecting hidden side-channel behavior is to identify the events behind hardware resource contention. For example in the case of a memory bus covert channel, the event to be monitored is a memory bus lock operation. Another example is the side channel based on integer operation [33]. The event to be monitored is the number of times a divider is received by a division instruction from one process and waiting for an instruction from another process. Further, if we want to detect the controlled side channel [34], the event that you want to detect is the number of how many page faults occur during a certain period.

Tracing cache-based attacks. As for the cache-based side channel, events to be monitored must be the cache oscillation [7]. The so-called oscillation mode refers to transmit '1' or '0', a Trojan or spy program creates a sufficient number of cache miss events between each other, which allows inferring transmitted bits according to the average access time (i.e. by observing whether the cache hits). This will lead to a conflicting cache access pattern between the Trojan and the victim. Oscillation can be thought of as a periodic characteristic in an event sequence, which is different from a specific period of outbreak of high-frequency memory access. Such events can be detected by measuring event sequence autocorrelation [35]. The alarm threshold can be set by monitoring LLC/L1-loads, LLC/L1-stores, LLC/L1-load-misses, or LLC/L1-store-misses in CPU events.

Tracing memory bus-based attacks. As for covert channels based on memory bus conflicts, we can look for frequent lock events in virtual machines when tracing these threats. Our implementation uses association analysis detection algorithms to detect the presence of cyclic burst mode event sequences. This step consists of two parts: firstly checking if the event sequence has a significant competition pattern (oscillating bursts), and secondly determining whether the burst pattern exhibits behaviors in time series.

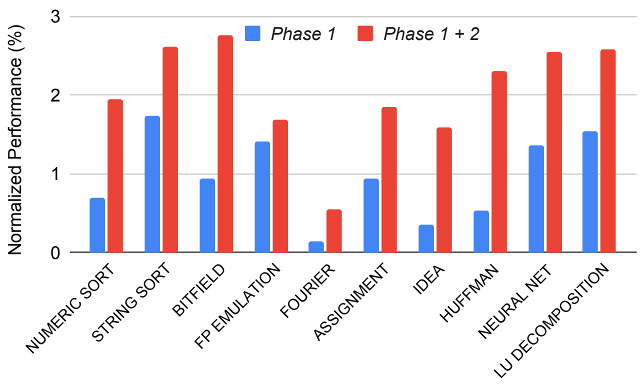

Fig. 9: Performance on nBENCH

If traced operations do not belong to normal memory access, they are considered as potential side-channel attack behaviors via shared cache or memory. Under-reports would be few in our VMI scheme. LBR information of the detected suspicious program is reported to the tracing (analysis) module located in the privileged domain.

Here we demonstrate our approach can locate the process that issues these suspicious operations, owing to that LBR can give precise information at the address level. Alleviating false positives introduced by traditional anomaly detection technology is orthogonal to our tracing framework and it can be integrated into the memory analysis module.

VII. Evaluation

The testbed host is a workstation equipped with Intel i7-6700K, VT-x and VT-d enabled. Every VM on our workstation is allocated with 2 GB out of total 16 GB memory. Additionally, the setting of virtual machines and the hypervisor are the same as what we mentioned in Section V and Section VI.

A. Performance Evaluation

Performance overhead on micro-benchmarks. We further evaluated our prototype on various micro-benchmarks, such as nBench [36], SPECcpu2006 [37], and PARSEC [38]. Figure 9 and Figure 10 show the results of nBench and SPEC 2006 benchmarks running on a Linux VM with different tracing stages turned on. Overall, the overhead of phase 1 with phase 2 both switched on will not exceed 4.5% under such scenarios. Here phase 2 applies the most common VMI function - process list [39]. If we only deploy phase 1, the performance penalty would not be 2.25% bigger than the baseline. Specifically in nBench, *string sort*, *bitfield*, *neural net*, and *lu decomposition* exhibit larger overhead. The tracing has much less influence on *fourier*. While among SPEC benchmarks, *bzip2*, *mct*, and *h264ref* occupies more computation on I/O so they caused more overhead. Meanwhile, *libquantum* has the lowest overhead since it does not need frequent accesses to memory.

We now turn to the performance evaluation on some PARSEC benchmarks. Figure 11 shows the results that phase 1, as

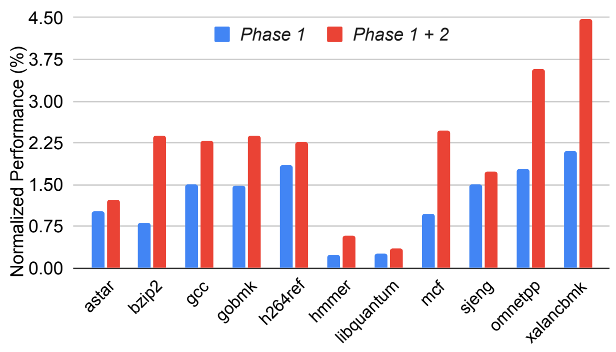

Fig. 10: Performance on SPEC CPU 2006

well as both phase 1 and 2, would introduce to the baseline of the application running. Only phase 1 usually does not impose more than 2.5% overhead. And phase 1 and 2 impose less than 4.6% on any chosen applications. Since *canneal*, *dedup*, and *bodytrack* are most memory intensive, the large amount of memory write operations (in phase 2) potentially lead to more overhead.

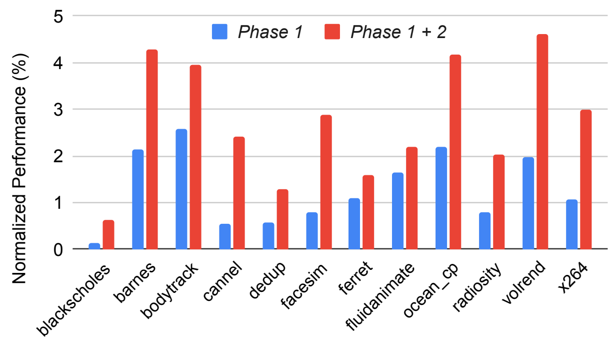

Fig. 11: Performance on PARSEC

Performance overhead on macro-benchmarks. To better understand overhead on application, we measure the execution time of benchmarks in Unix-bench each for 10 times [40].

When only the gadgets chain detection module is turned on, the performance loss of the introduced system is acceptable, with an overhead of 1.51% on the geometric mean (phase 1). As for phase 1 + 2 without memory analysis, the average overhead is around 3%. Among them, the impact on process creation and file read-write indicators is more significant. As shown in Table III, the context switch introduces additional VM exits, which affect the performance of virtual machines to a certain extent. Also, the system is more sensitive to process creation because of traversing the kernel process structure. In addition, our prototype rewrites Xentrace to read/write data into Dom0's memory pages repeatedly, which undoubtedly takes up part of I/O resources.

Forensic time estimation. Here we give the time cost by FMA, while the branch validation algorithm (in Section VI-A) is chosen to be the FMA algorithm. Since it runs on the Dom0, it has little effect on the performance of the guest-VMs. When performing the tests, to make the experiment less affected by various errors, each UnixBench application is executed 10 times. Besides, we set all parameters by making a customized hypercall interface in advance. In the attack provenance phase,

TABLE III: Performance overhead on UnixBench

Benchmark Name	Phase 1	Phase 1 + 2 w/o FMA	FMA
Process creation	1.3%	2.2%	1.33s
Pipe	1.5%	2.6%	1.82s
File reads	1.9%	4.8%	2.54s
File writes	1.8%	5.2%	2.15s
Context switch	1.4%	3.9%	0.27s
Spawn	1.3%	1.6%	0.22s
Bash	1.5%	2.3%	2.76s

when the platform is running N virtual machines, the average time complexity required to traverse the current memory page frame of the current process is $O(N)$. Memory analysis takes about 0.2 to 2.8 seconds. It should be noticed that this solution can be further improved by avoiding pausing the target virtual machine [22] or other fast VMI techniques [41], and the overhead of triggering VM introspection is negligible.

Forensic space overhead. The recorded logs could be a risk to the overhead on the memory and the file system. We also evaluate how much storage our system would spend. Table IV shows the storage cost when our framework scales to a multi-guest situation. The target application we choose to measure is the Apache 2.4 Web server. A client executes a stress test tool - Siege [42] - on another host in an isolated LAN. Siege was configured to send continuous HTTPS requests (with no delay between two consecutive ones) to the webserver. The workload contains 3 classes of files with sizes at 10KB, 100KB, and 1,000KB respectively and the total dataset size is 4.9 GB chosen from the SPECweb2009 benchmark. We measured its throughput in the presence of a different number of running VMs to understand how our implementation would perform.

TABLE IV: Storage overhead

#. of VMs	Workload per request	Throughput	Overhead
1	10 KB	5.83 MB/s	0.21%
1	100 KB	6.19 MB/s	0.12%
1	1000 KB	6.67 MB/s	0.06%
2	10 KB	4.62 MB/s	0.11%
2	100 KB	5.01 MB/s	0.08%
2	1000 KB	5.17 MB/s	0.05%
4	10 KB	3.35 MB/s	0.04%
4	100 KB	3.62 MB/s	0.02%
4	1000 KB	3.74 MB/s	0.00%

Due to the presence of the buffer cache, most disk write requests are consumed at the Xen's frontend disk I/O but wait to be flushed in the backend. This could alleviate the most burden of the space overhead. To be fair, the record buffer size is set as 80 KB, which will only take up a very small part of Xen's heap memory. As we can see in the table, the throughput has been maintained at a relatively high level. Nearly 2.8 gigabytes LBR data was processed in 8 minutes when a 10 KB workload was applied in one VM. Compared to the baseline (pure Xentrace without LBR data transferred), the overhead is always less than 0.3 percent. With the increase of workload, the processing speed of the server will become faster and more LBR information will be generated. However, the overhead is decreasing, which proves that there is no performance bottleneck in our framework.

VIII. Discussion

In this section, we discuss the limitations of our study and potential future research. While HYBRID is able to detect and prevent a range of state-of-the-art attacks, the protection is limited to specific assumptions.

Untrusted guest kernel. If we are examining a machine infected by malware, the malware might even have altered some kernel symbols and thus rendering the guessing method useless. This issue can be solved by comparing LBR values generated from an untrusted kernel to a pure and clean kernel whose kernel version is exactly the untrusted kernel has. We can prepare a suite of security benchmarks, which cover most security-sensitive kernel functions, to test if their execution paths are the same. If the LBR recorded during executing the security benchmark in a privileged clean kernel (such as Dom0) does not match to values recorded in an untrust kernel, then this kernel should be treated as suspicious.

Supporting other intrusion detections. Our design can also work on detecting other cross-VM attacks. Most of the existing side channel detection researches aim at a specific side-channel attack on a specific hardware resource (e.g., cache, memory bus, or TLB). These detection techniques can be perfectly complemented by the tracing framework proposed in our design. Our method also can be used in tracing controlled side channels since the attack behaviors are highly related to memory addresses. We can use EPT-based detection method [43] to identify them and further extract the key information on tampered kernel pages. Also, fuzzers like Agamotto [44], a lightweight virtual machine checkpointing method, can be integrated into HYBRID seamlessly to enable high-throughput kernel driver fuzzing.

IX. Related work

Virtual machine introspection. Current VMI techniques can be classified into two different categories: hardware-level probes and operating-system-level probes [39]. The early implementation of Garrinkel's VMI abandoned the requirement [45] of transparency because the virtualization platform did not have related interfaces, allowing virtual machines to directly connect with the monitoring software. Later VMI such as Antfarm can adapt well to a variety of operating systems, the information it can obtain is extremely limited [46]. Lares [47] can solve the problem that the auditing program cannot be monitored actively in the isolated VM, so does ShadowMonitor [48] that decomposes the whole monitoring system into two compartments and then assigns each compartment with isolated address space. However, these methods are not portable in some cases as discussed in Section I. LibVMI is easy to use and obtains high-level information, but its process of obtaining offset information through artificial gadgets affects its transparency [49], [50]. ShadowContext allows the guest-VM system call code to be reused inside a "shadowed" portion of the context of the out-of-guest inspection program. Besides, ShadowContext is secure enough to defend against a variety of real world attacks. DRAKVUF [51] and its application on Arm have been proposed using *alternate p2m* to switch the memory view, which is effective but cannot guarantee ongoing stealthy operation [52]. All these schemes have advantages and disadvantages. How to balance transparency and isolation is always a problem that needs to be solved. Table V shows the comparison with current VMI techniques.

Unlike other coarse-grained CFI methods, HYBRID does not need to extend the compiler or modify the kernel. It doesn't need to get the source code as well, which keeps the transparency and isolation of the guest-VM.

Security enhancement with hardware-based profiling. Recently, researchers have made extensive use of the aforementioned hardware performance counters to detect security vulnerabilities [55], [56]. The counter can reveal the execution characteristics of the program and further reflect the safe state of the program. In addition, this kind of detection has negligible performance overhead on the program.

We are not the only ones that use hardware features for enhancing system security. Execution time of hardware-assisted software event tracing [57] can be reduced by up to ten times when assisted by hardware, as compared to software tracing with LTTng [58], a high-performance tracer for Linux. Similar techniques can be extended to ARM [59]. Griffin [60] also leverages Intel CPU features which can enforce fine-grained CFI policies with shadow stack as recommended by researchers, and its performance is comparable to software-only instrumentation techniques. Another important aspect of performing a forensic investigation in a cloud environment is preserving privacy while not compromising integrity. Using confidential computing technology such as Intel Software Guard Extensions (SGX) is a good way to protect the privacy of both code and data that need to be investigated [61].

X. Conclusion

In this paper, a cross-VM attack investigation method HYBRID is proposed, which combines a newly-explored hardware feature (Intel LBR) with privileged domain forensics techniques, keeping both advantages of rapid response and rich semantics. The experimental results show that this method can accurately locate the malicious process and has a small performance overhead.

Another brighter side of the HYBRID framework is its flexibility. The design can be instantiated using near-future Alder Lake CPUs with minor modification, especially with the LBR virtualization and its data flow serialization features.

References

[1] B. Manral, G. Somani, K.-K. R. Choo, M. Conti, and M. S. Gaur, "A systematic survey on cloud forensics challenges, solutions, and future directions," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–38, 2019.

[2] K. Ruan, J. Carthy, T. Kechadi, and M. Crosbie, "Cloud forensics: An overview," in *proceedings of the 7th IFIP International Conference on Digital Forensics*, 2011, pp. 16–25.

[3] D. Birk and C. Wegener, "Technical issues of forensic investigations in cloud computing environments," in *2011 Sixth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering*. IEEE, 2011, pp. 1–10.

[4] K. J. Han, B.-Y. Choi, and S. Song, *High performance cloud auditing and applications*. Springer, 2014.

TABLE V: Comparison of VMI approaches. Transparency means whether the framework requires external assistance. Generality means whether an introspection tool can work on various situations.

Approach	Transparency	Generality	Performance
Virtuoso [53]	Limited (need intervention for each guest)	Suitable for most attacks	6s to run *pslist*
Drakvuf [51]	Good (wide scope of runtime information)	Suitable for most attacks	Very heavy
CloudRadar [7]	Excellent	Side channel attack only	$< 5\%$
ShadowContext [54]	Excellent	Syscall-level attack only	75% on average
HYBRID	Good (rely on genuine hardware information)	Suitable for most attacks	5% + FMA time

[5] S. Suneja, C. Isci, E. de Lara, and V. Bala, "Exploring vm introspection: Techniques and trade-offs," *Acm Sigplan Notices*, vol. 50, no. 7, pp. 133–146, 2015.

[6] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 249–266.

[7] R. B. L. Tianwei Zhang, Yinqian Zhang, "Cloudradar: A real-time side-channel attack detection system in clouds," in *19th Intern. Symp. Research in Attacks, Intrusions, and Defenses*, 2014.

[8] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 20–38.

[9] Y. Hebbal, S. Laniepce, and J.-M. Menaud, "Virtual machine introspection: Techniques and applications," in *2015 10th International Conference on Availability, Reliability and Security*. IEEE, 2015, pp. 676–685.

[10] E. Bauman, G. Ayoade, and Z. Lin, "A survey on hypervisor-based monitoring: approaches, applications, and evolutions," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–33, 2015.

[11] B. D. Payne, "Simplifying virtual machine introspection using libvmi," *Sandia report*, pp. 43–44, 2012.

[12] "Hardware-assisted (lbr-based) attack tracing on cloud platforms," https://github.com/StanPlatinum/HYBRID.

[13] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 605–620.

[14] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *ISOC Network and Distributed System Security Symp.*, vol. 3, 2003, pp. 191–206.

[15] P. Yuan, Q. Zeng, and X. Ding, "Hardware-assisted fine-grained code-reuse attack detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 66–85.

[16] C. Pierce, M. Spisak, and K. Fitch, "Capturing 0day exploits with perfectly placed hardware traps," BLACKHAT, 2014.

[17] V. M. Weaver, "Advanced hardware profiling and sampling (pebs, ibs, etc.): creating a new papi sampling interface," Tech. rep. UMAINEVMW-TR-PEBS-IBS-SAMPLING-2016-08. http://web. eece. maine …, Tech. Rep., 2016.

[18] A. Nowak, A. Yasin, A. Mendelson, and W. Zwaenepoel, "Establishing a base of trust with performance counters for enterprise workloads," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015, pp. 541–548.

[19] "PERFMON2," https://perfmon2.sourceforge.net/.

[20] D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng, "Taming hardware event samples for precise and versatile feedback directed optimizations," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 376–389, 2011.

[21] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent {ROP} exploit mitigation using indirect branch tracing," in *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 447–462.

[22] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen, "Concurrent and consistent virtual machine introspection with hardware transactional memory," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 416–427.

[23] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 586–600.

[24] B. Dolan-Gavitt, B. Payne, and W. Lee, "Leveraging forensic tools for virtual machine introspection," Georgia Institute of Technology, Tech. Rep., 2011.

[25] "Intel 64 and IA-32 Architectures Software Developer's Manual," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[26] "libVMI API," http://libvmi.com/api.

[27] M. Cohen, "Rekall memory forensics framework," *DFIR Prague*, 2014.

[28] X. Wang and J. Backer, "Sigdrop: Signature-based rop detection using hardware performance counters," *arXiv preprint arXiv:1609.02667*, 2016.

[29] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus, "Bypassing memory safety mechanisms through speculative control flow hijacks," *arXiv preprint arXiv:2003.05503*, 2020.

[30] M. Lipp, M. Schwarz, D. Gruss, and et al., "Meltdown," https://meltdownattack.com/meltdown.pdf, Report, 2017.

[31] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[32] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 141–150.

[33] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 473–482.

[34] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *36th IEEE Symp. Security and Privacy*, 2015, pp. 640–656.

[35] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[36] "Linux/Unix nbench," 2019, https://www.math.utah.edu/~mayer/linux/bmark.html.

[37] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[38] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Intern. Conf. Parallel Architectures and Compilation Techniques*, 2008.

[39] F. Westphal, S. Axelsson, C. Neuhaus, and A. Polze, "Vmi-pl: A monitoring language for virtual platforms using virtual machine introspection," *Digital Investigation*, vol. 11, pp. S85–S94, 2014.

[40] "byte-unixbench," https://github.com/kdlucas/byte-unixbench, 2019.

[41] T. K. Lengyel, "Stealthy monitoring with xen altp2m," 2016.

[42] "Siege - http load testing and benchmarking utility," https://www.joedog.org/siege-home/.

[43] C. Qiang, W. Liu, L. Wang, and R. Yu, "Controlled channel attack detection based on hardware virtualization," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2018, pp. 406–420.

[44] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2541–2557.

[45] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 477–487.

[46] S. T. Jones, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau *et al.*, "Antfarm: Tracking processes in a virtual machine environment." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 1–14.

[47] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 233–247.

[48] B. Shi, L. Cui, B. Li, X. Liu, Z. Hao, and H. Shen, "Shadowmonitor: An effective in-vm monitoring framework with hardware-enforced isolation," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 670–690.

[49] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, “Using hardware features for increased debugging transparency,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 55–69.

[50] K. Leach, C. Spensky, W. Weimer, and F. Zhang, “Towards transparent introspection,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 248–259.

[51] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, “Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 386–395.

[52] S. Proskurin, T. Lengyel, M. Momeu, C. Eckert, and A. Zarras, “Hiding in the shadows: Empowering arm for stealthy virtual machine introspection,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 407–417.

[53] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, “Virtuoso: Narrowing the semantic gap in virtual machine introspection,” in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 297–312.

[54] R. Wu, P. Chen, P. Liu, and B. Mao, “System call redirection: A practical approach to meeting real-world virtual machine introspection needs,” in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 574–585.

[55] Y. Xia, Y. Liu, H. Chen, and B. Zang, “Cfimon: Detecting violation of control flow integrity using performance counters,” in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12.

[56] A. Tang, S. Sethumadhavan, and S. J. Stolfo, “Unsupervised anomaly-based malware detection using hardware features,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 109–129.

[57] A. Vergé, N. Ezzati-Jivan, and M. R. Dagenais, “Hardware-assisted software event tracing,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 10, p. e4069, 2017.

[58] “LTTng is an open source tracing framework for Linux.” https://lttng.org/.

[59] Y. Du, Z. Ning, J. Xu, Z. Wang, Y.-H. Lin, F. Zhang, X. Xing, and B. Mao, “Hart: Hardware-assisted kernel module tracing on arm,” in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 316–337.

[60] X. Ge, W. Cui, and T. Jaeger, “Griffin: Guarding control flows using intel processor trace,” in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1. ACM, 2017, pp. 585–598.

[61] K. Leach, F. Zhang, and W. Weimer, “Scotch: Combining software guard extensions and system management mode to monitor cloud resource usage,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 403–424.

Weijie Liu received the B.S. degree and Ph.D. degree from Wuhan University in 2012 and 2018, respectively. He is doing a Research Associate at Indiana University Bloomington. His research areas include virtualization security, side channel defenses, trusted execution technologies, and data mining.

Ximeng Liu (S’ 13-M’ 16-SM’ 21) received the B.Sc. degree and Ph.D. degree from Xidian University in 2010 and 2015, respectively. He is currently a “Minjiang Scholars” Distinguished Professor in the College of Mathematics and Computer Science, Fuzhou University. Before that, he worked at the School of Information System, Singapore Management University, Singapore. He received “Qishan Scholars” in Fuzhou University and ACM SIGSAC China Rising Star Award (2018). His research interests include cloud security, applied cryptography and big data security.

Zhi Li is currently working toward the Ph.D. degree in computer science and technology at Huazhong University of Science and Technology (HUST), Wuhan, China. His research areas include security in system architecture and virtualization.

Bin Liu received the Ph.D. degree from Huazhong University of Science and Technology in 2006. He is a Professor in the School of Computer and Information Engineering, Hubei University. His research interests include image processing, machine learning, and multimedia security.

Rongwei Yu received his Ph.D. from Wuhan University in 2009. He is currently an Associate Professor at the School of Cyber Science and Engineering, Wuhan University. His research interests include system security, blockchain security, and mobile network security.

Lina Wang received the Ph.D. degrees from Northeastern University, Shenyang, China, in 1989 and 2001, respectively. She is currently a Professor with the School of Cyber Science and Engineering, Wuhan University, where she is also the Director of the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education. Her research interests include multimedia security, cloud security, and network security.