Concurrent and Consistent Virtual Machine Introspection with Hardware Transactional Memory

Yutao Liu†, Yubin Xia†, Haibing Guan§, Binyu Zang†, Haibo Chen†

Shanghai Key Laboratory of Scalable Computing and Systems
†*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*
§ *Department of Computer Science, Shanghai Jiao Tong University*
{ytliu.cc, xiayubin, hbguan, byzang, haibochen}@sjtu.edu.cn

Abstract

Virtual machine introspection, which provides tamper-resistant, high-fidelity "out of the box" monitoring of virtual machines, has many prominent security applications including VM-based intrusion detection, malware analysis and memory forensic analysis. However, prior approaches are either intrusive in stopping the world to avoid race conditions between introspection tools and the guest VM, or providing no guarantee of getting a consistent state of the guest VM. Further, there is currently no effective means for timely examining the VM states in question.

In this paper, we propose a novel approach, called TxIntro, which retrofits hardware transactional memory (HTM) for concurrent, timely and consistent introspection of guest VMs. Specifically, TxIntro leverages the strong atomicity of HTM to actively monitor updates to critical kernel data structures. Then TxIntro can mount introspection to timely detect malicious tampering. To avoid fetching inconsistent kernel states for introspection, TxIntro uses HTM to add related synchronization states into the read set of the monitoring core and thus can easily detect potential inflight concurrent kernel updates. We have implemented and evaluated TxIntro based on Xen VMM on a commodity Intel Haswell machine that provides restricted transactional memory (RTM) support. To demonstrate the effectiveness of TxIntro, we implemented a set of kernel rootkit detectors using TxIntro. Evaluation results show that TxIntro is effective in detecting these rootkits, and is efficient in adding negligible performance overhead.

1. Introduction

Traditional in-VM based virus detection tools are intrinsically limited from isolating themselves from the vulnerable monitored system, and thus can inevitably be infected by the rootkits as well. Virtualization, by adding an additional layer of indirection, provides new opportunities to move rootkit detection tools to the virtual machine monitor (VMM)[1], resulting in strong isolation from the monitored system. Virtual machine introspection (VMI) [1, 2, 3, 4, 5, 6], which retrieves and introspects guest VM's internal states from the VMM, has been widely explored for intrusion detection [4, 1, 2], process monitoring [7], and memory forensics [8].

However, there are several limitations with prior introspection approaches that impede their wide adoption in many scenarios. First, as there will be race conditions on kernel data structures if the VMI tools and the guest VM execute concurrently, some introspection systems are intrusive in stopping the VM for up to tens of seconds [9, 6] to introspect over a VM snapshot, either eagerly [9, 6] or lazily [3]. Further, prior VMI tools provide no effective means to determine when a VMI operation should be performed, forcing users to make a tradeoff between performance overhead and security protection: frequent VMI causes significant service disruption and performance overhead, while infrequent VMI risks of undetected security breaches.

In this paper, we propose a novel approach to VMI, called TxIntro, which leverages hardware transactional memory (HTM) [10] to provide concurrent, timely and consistent VMI. Though HTM's original purpose is simplifying concurrent programming with good performance, we find that the strong atomicity provided by HTM can be used to detect conflicting concurrent accesses to kernel data structures by the kernel and the VMI tools, as well as to provide hints regarding when a VMI operation is needed.

HTM, originally proposed by Herlihy and Moss [10] has recently shifted from research community to commercial products. Example commercial HTM products or proposals include Sun's Rock processor [11], AMD advanced synchronization family [12], POWER [13] and Intel transactional synchronization extensions (TSX) [14]. A concrete embodiment of Intel TSX, the Haswell processor with restricted transaction memory (RTM) support, has been released to the mass market in the middle of 2013. Hence, our approach is readily deployable to commodity platforms.

While ideally it is possible to run the entire operating system code using HTM [15], commodity operating systems have not been integrated with HTM support and commercial HTM is usually a best effort TM with limited read/write set. Hence, TxIntro leverages the strong atomicity of HTM by running only parts of the VMI code using transactional memory.

[1] Here, we refer to the management VM like domain-0 in Xen and Xen itself as the *VMM* (Virtual Machine Monitor) for simplicity, and use the term *hypervisor* to denote the monitor underneath the virtual machines.

TxIntro ensures concurrent and consistent VMI by adding the associated synchronization states into the read set of the CPU core running VMI tools, so that inflight updates to critical kernel data structures will cause transaction aborts in VMI tools, where TxIntro can retry to get a consistent state to check. To provide active monitoring of critical kernel data structures (e.g., syscall/IDT table, kernel hooks), TxIntro dedicates a (logical) CPU core to hold these states to actively monitor kernel updates, which will cause transactional aborts to trigger the corresponding VMI tools.

TxIntro is further built with two techniques to reduce the read and write set of VMI code. To reduce enlarged read and write set due to two-dimensional page walk to access guest VM's memory, TxIntro leverages a technique called *core planting* that dynamically implants a stealthy core to the guest VM to allow direct access of guest VM data. To leverage the asymmetric read/write set of commodity HTM machines, we propose a two-phase VMI approach that uses a pre-run phase to trade read set for write set.

We have implemented TxIntro based on Xen/Linux, which runs on an Intel Haswell machine with RTM support. TxIntro leverages the asymmetric read and write set feature in Haswell, where the read set is 100X larger than the write set (3 MB vs. 26 KB in the tested machine), to hold a relatively large set of kernel data structure for monitoring. Though the emphasis of TxIntro is an architectural mechanism for non-intrusive, timely, concurrent and consistent VMI instead of specific security policies, we have built 9 VMI tools using TxIntro to detect a variant of kernel rootkits. Our evaluation shows that TxIntro is effective in timely detecting such kernel rootkits and intrusions. Performance evaluation shows that TxIntro incurs little disruption to the guest VM and consumes only a small amount of resources.

In summary, this paper makes the following contributions:

- **VMI using HTM.** We introduce TxIntro, a novel approach that provides non-intrusive, concurrent, consistent and timely VMI. We identify the problems associated with prior approaches (section 2), and show how TxIntro successfully solves them (section 3).
- **VMI optimized for HTM.** We describe two novel techniques that significantly reduce the read and write set of VMI code (section 4).
- **An implementation of TxIntro on a Haswell processor.** We show that TxIntro can be practically implemented (section 5.1) and integrated to detect intrusions by applying it to several existing VM monitoring tools (section 5.2).
- **Experimental validation of the benefits of TxIntro.** We evaluate the effectiveness using a number of kernel rootkits (section 6.1) and the non-intrusiveness by using application benchmarks (section 6.2).

2. Motivation and Background

This section first introduces virtual machine introspection (VMI) using LibVMI [16] as an example. LibVMI is an open-source VMI system from Sandia National Laboratories that supports both Xen [17] and KVM [18][2]. We use it because it is relatively mature and should be representative of typical VMI systems. We then describe some potential issues with prior VMI approaches and finally present some background information related to transactional memory using Intel's Restricted Transactional Memory (RTM) as an example.

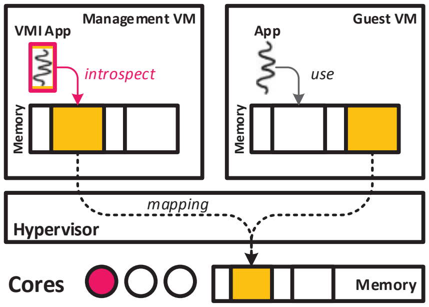


Figure 1: Architecture of LibVMI

```
vmi_pause_vm() ---------------------------------
                                      Copy VM data
start_proc     <— vmi_ksym2v("init_task")
tasks_list     <— vmi_read(start_proc + tasks_offset)
runqueue_tree  <— vmi_read(start_proc + cfs_rq_offset)

/* traverse the all-tasks list */
foreach task in tasks_list:
    allprocs.add(vmi_read(task + pid_offset))

/* traverse the runqueue tree */
foreach run_task in runqueue_tree:
    runprocs.add(vmi_read(run_task + pid_offset))

vmi_resume_vm() --------------------------------
                                      Do VMI check
/ check if there is any hidden running process */
foreach pid in runprocs:
    check_if_pid_in_allprocs(pid, allprocs)

alarm_if_needed() ---------------------- VMI end---
```

Figure 2: Example VMI code to detect hidden processes

2.1. Virtual Machine Introspection

Figure 1 illustrates an architectural overview of typical VMI system such as LibVMI. A VMI tool runs as a user-level process inside the management VM, which uses the library interfaces provided by LibVMI to pull memory from guest VMs for introspection. Figure 2 shows an example VMI tool that checks the process runqueue to detect hidden processes. In the VMM, the tool first iterates the system-wide process list to get the whole set of processes and then fetches the set of all runnable processes by scanning the runqueue of each virtual core. As all processes should be present in the runqueue in order to execute, the tool can detect hidden processes by calculating the difference of the two sets, i.e., processes in the set of runnable processes, but not in the set of all processes.

[2] The support for KVM is relatively less mature than Xen. Hence, we use a Xen-based version in our study.

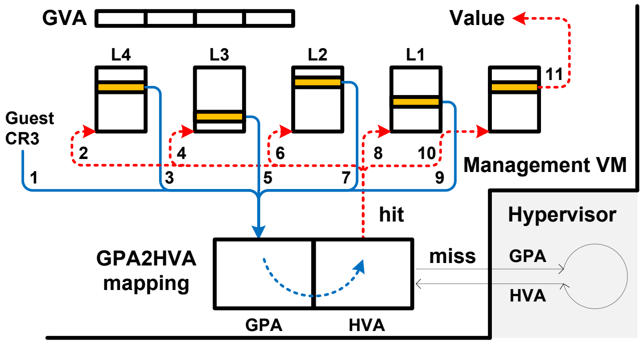


Figure 3: Software page table traversal in *vmi_read_addr*

As the VMI tool resides in a different address space than the monitored VM, it requires a two-dimensional address translation as it only gets a virtual address in the guest virtual machine. The *vmi_read* function gets the virtual address in the VMI tool of the specified guest virtual address. It does so by using the guest virtual address to walk the guest page table to get the corresponding guest physical address, which is then translated into host physical address and finally mapped to the virtual address in the VMI tool. However, as shown in figure 3, such a translation would require five accesses to the guest page table for a 4-level page table and a hash table that translates guest physical address to host virtual address in the VMI tool, as well as some hypercalls[3] to the hypervisor in order to create mappings for guest physical address to virtual address in the VMI tool upon a miss in the hash table.

2.2. Issues with Prior VMI Approaches

Intrusiveness: As shown in Figure 2, the VMI tool needs to stop the guest VM in order to get a consistent state of the guest VM (*vmi_pause_vm()*). However, there are two main issues with such an approach. First, it is essentially intrusive in suspending the VM and the suspension time is usually lengthy. For example, even some simple VMI tools would require stopping the VM from being executed tens of milliseconds, and for frameworks using python like Volatility [19], the stop time is even longer than tens of seconds (e.g., figure 10 in [9]). A lengthy VM suspension time will not only cause disruption to the running services, but also may render some services not function correctly due to timeout of some critical requests (like DMA and heartbeats).

Consistency: Even if the VM is suspended, a VMI tool may still get inconsistent states. This is because some pending updates to the data structures to be checked may still have not completed, causing an inconsistent view by the VMI tool. This may either cause false positives/negatives, or even crash the VMI tool [9, 3]. Hence, some VMI systems need to wait a quiescent state (e.g., all VCPUs are executing in user mode) to stop the VM [9]. However, for today's virtual machines with multiple virtual cores, it is not easy to wait such a quiescent state without enforcing such a state (e.g., pending a CPU in user mode). Further, there may be some time-of-check-to-time-of-use issues, where the suspension time is not the quiescent time due to concurrent execution of multiple cores.

One approach to mitigating the consistency issue is using non-blocking VMI to inspect a guest VM first and then re-inspect again if the first trial detects potential inconsistency. It will then suspend the VM to do checking if the two VMI runs detect a problem [3]. This approach could reduce the frequency of lengthy suspension of VMs. However, it may introduce both false positives and false negatives, as some transient inconsistence will miss a necessary VMI run (thus false negatives) and unsynchronized accesses in two runs may cause false positives. Ultimately, it still needs to suspend the VM to run VMI code.

Timeliness: Further, most prior VMI approaches provide no effective means to infer when a VMI would be necessary. This requires users to make a tradeoff between security and performance: frequent VMI causes significant disruption to running services but infrequent VMI may render some kernel rootkits undetected.

2.3. Transactional Memory

Hardware transactional memory, after nearly two decades of discussion in the research community, has been recently available to the mass market, i.e., Intel's restricted transactional memory (RTM) [14] and hardware lock elision [20][4]. RTM extends the instruction set of x86 with several new instructions called *xbegin*, *xend* and *xabort*, which begins, commits and aborts a transaction respectively. Like all other transactional memory proposals, RTM ensures atomicity and isolation for all running transactions, and implements mechanisms like conflict detection and version management.

For hardware simplicity, RTM implements conflict detection by piggy-backing on the cache coherence protocols, like the early HTM proposal [10]. Specifically, each core maintains a read set and a write set that track the memory addresses read and written in a transaction by extending cache lines with additional read and write bits. Each coherence message will intersect these bits for a cache line in the involved cores to detect conflicts (e.g., read/write and write/write conflicts). A conflict aborts a transaction, whose execution will be directed to the beginning of the transaction, which then switches to a user-provided fallback handler. One interesting feature in RTM and most other HTMs, is that they provide *strong atomicity* such that a transactional execution will be unconditionally aborted by a non-transactional execution if their memory accesses overlap.

Unlike some HTM proposals supporting unbounded transactions [21], RTM is restricted in that it cannot survive from context switches, mode switches and interrupts, which will

[3] Hypercall is similar to syscall but is used to request the hypervisor's services.

[4] We only discuss RTM here as the underlying mechanism is similar, though their interfaces are different.

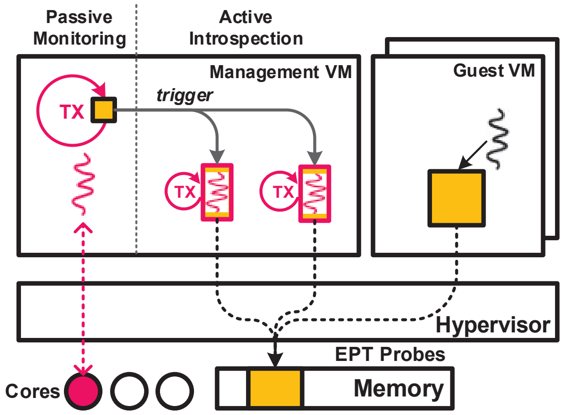


Figure 4: Overview of TxIntro.

abort a running transaction. Further, its read/write set are limited in size and a *capacity abort* will happen if there are any cache misses (e.g., capacity or conflict misses) in the transactionally tracked cache lines. Interestingly, accordingly to our evaluation, though the official Intel Software Development Emulator [22] indicates equal sizes for read and write set as the size of L1 cache, our evaluation shows that its read set is actually around hundred times of the write set (3 MBytes vs. 26 KBytes using a microbenchmark in our tested machine) due to some internal optimizations to track read set. Such asymmetric read/write set size makes it very appealing for VMI, which mostly introspect (i.e., read) guest VM states.

3. Transactional Virtual Machine Introspection

TxIntro leverages hardware transactional memory to provide concurrent and consistent virtual machine introspection that is non-intrusive and transparent to guest VMs. The key idea is to leverage the strong atomicity of HTM to ensure that a VMI tool running in a transactional region can always get a consistent state of guest VM, even if the guest VM is not running in a transaction. This section first describes the basic architecture and algorithm of TxIntro. In the next section, we will further show two optimizations that significantly reduce the read and write set of VMI code so as to accommodate relatively large and complex VMI tools.

3.1. Architecture Overview

Figure 4 shows the overall architecture of TxIntro. The VMI tool mainly runs in the management VM so that it is isolated from the guest VM to be checked. In section 4.1, we further show that parts of its execution can be securely planted into the guest VM stealthily to reduce transactional working set during introspection. TxIntro can monitor multiple VMs simultaneously, essentially embodying a security-as-a-service model for cloud platforms. TxIntro runs in both active and passive modes. In the passive mode, TxIntro monitors updates to critical kernel data structures. Upon detecting any updates to monitored data, TxIntro actively runs the corresponding VMI tools to detect potential kernel rootkits.

3.2. Passive Transactional VMI

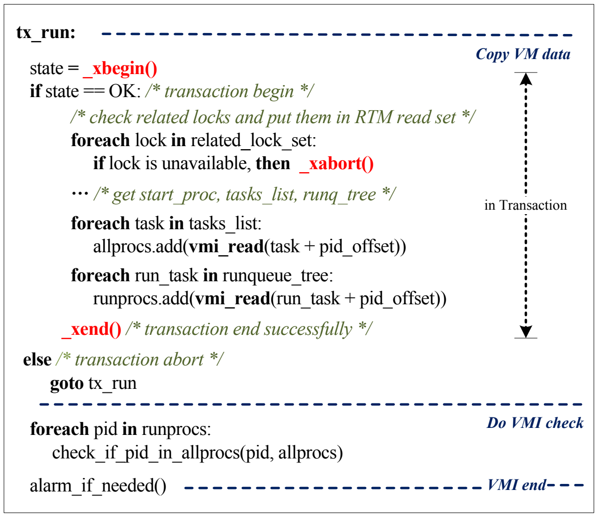
In passive mode, TxIntro dedicates a logical core (e.g., a hardware thread) to monitor critical data structures in guest VMs. To monitor updates to data structures, TxIntro first translates their addresses from guest virtual address to host virtual address in the VMI tool, as discussed in section 2.2, and maintains a copy of these data for rescanning later. TxIntro also places probes in the hypervisor's extended (nested) page table in case the mappings from guest physical addresses to host physical addresses have been changed (which is rare). To start monitoring, TxIntro starts a transaction and adds the addresses of critical kernel data structures to the read set by touching them. TxIntro disables interrupts and context switches in this (logical) core to minimize transactional aborts due to system events.

When there is an update to the monitored data, the transaction abort will be flagged as a conflict abort and TxIntro fetches the virtual address related to this abort through performance monitoring tool (i.e., Intel's Precise Event Based Sampling (PEBS)). In the abort handler, TxIntro determines if the address is due to false sharing (as HTM monitors updates in cache line granularity), or belongs to the monitored addresses. If it is not caused by false sharing, TxIntro then determines which VMI tool should be used according to the address causing the abort. TxIntro then notifies the VMI daemon in another core to start VMI on the corresponding guest VM and returns to the transactional memory loop. The transaction will also be aborted if the address mapping of monitored data has been changed. In this case the hypervisor will send an address change message to TxIntro (i.e., using shared memory), which will abort the transaction. TxIntro will then track the updated addresses using the updated mapping.

Instead of using the monitoring feature provided by TxIntro, it is also possible to use page protection in the hypervisor to intercept updates to critical data structures. However, as the protection granularity can only be done in page level, there could be a number of false sharing that causes frequent traps into the hypervisor, which may cause significant performance slowdown. Another approach would be modifying the guest kernel to aggregate critical data structures together into the same pages to mitigate false sharing [23], which, however, requires significant changes to operating system kernel. In contrast, TxIntro uses HTM to monitor updates in the cache line level, which is with reasonably fine granularity and requires no change to operating system kernel.

3.3. Active Transactional VMI

To support transactional VMI, users may simply replace *vmi_pause_vm* and *vmi_resume_vm* in Figure 2 with *xbegin* and *xend*, as well as adding a simple abort handler that retries the VMI. However, this may still result in inconsistencies as some pending updates are still in-progress. As discussed before, even if the kernel has been paused where there is no con-



```
tx_run:
  state = _xbegin()
  if state == OK: /* transaction begin */
      /* check related locks and put them in RTM read set */
      foreach lock in related_lock_set:
          if lock is unavailable, then  _xabort()
      ··· /* get start_proc, tasks_list, runq_tree */
      foreach task in tasks_list:
          allprocs.add(vmi_read(task + pid_offset))
      foreach run_task in runqueue_tree:
          runprocs.add(vmi_read(run_task + pid_offset))
     _xend() /* transaction end successfully */
 else /* transaction abort */
    goto tx_run

  foreach pid in runprocs:
     check_if_pid_in_allprocs(pid, allprocs)
  alarm_if_needed()
```

Figure 5: Basic version of VMI code to detect hidden processes using HTM.

flicting access with the VMI process, TxIntro can still get inconsistent states, leading to false positives or false negatives.

To address this issue, TxIntro also tracks the synchronization states associated with data touched by a VMI tool. As shown in figure 5, to begin a VMI process, TxIntro starts a memory transaction and first checks the synchronization variables protecting the data structures to ensure that there is no inflight update (e.g., lock is not locked). If so, they are also implicitly added to the read set of this transaction. As a result, any pending or ongoing updates to the data structures will be detected by triggering a transactional abort, where the VMI process can be restarted from the beginning. It should be noted that by checking the synchronization states in the HTM region, TxIntro avoids the issue of time of check to time of use (TOCTTOU), as the checking code is executed atomically with the actual VMI code.

An alternative approach would be to acquire all related locks before the VMI process to ensure a consistent check. However, OS kernels usually have a very complex locking discipline where the locking orders and nesting are very tricky and it is very easy to hang the entire kernel [3, 24]. Further, acquiring the related locks is essentially intrusive to the guest VM and the rootkit could check the lock status to see if some VMI check is in progress.

After all related synchronization variables have been added to the read set, TxIntro starts the VMI process, similarly as the original VMI code.

Handling Transaction Aborts: When there is a transaction abort, TxIntro handles it according to the abort reason. If the abort is caused by system events, TxIntro simply restarts the transaction. If it is caused by a conflict abort, this means that there may be conflicting accesses from the guest VM's code. Hence, the VMI transaction should be restarted as well.

Though the above scheme can work in principle, it may cause permanent aborts due to the limited support of transactional memory in commodity processors for some VMI tools. One potential issue is that it would require some system calls as well as hypercalls in order to create valid memory mappings so that guest virtual/physical addresses can be accessed from the VMI tool's code. To address these problems, instead of directly running VMI code inside the transaction, TxIntro uses a *pre-tx* phase to first run the VMI code, in which TxIntro creates all required mappings, issues necessary system calls and hypercalls.

3.4. Scope and Assumptions

Instead of focusing on the specific introspection policies, the focus of TxIntro is on an architectural approach to providing concurrent and consistent VMI, yet is non-intrusive to guest VMs. Hence, TxIntro shares most assumptions with prior VMI systems, including relying on the general knowledge from the OS design principles, correctly interpreting guest VM states and trusting the hypervisor and management VMs. Hence, TxIntro may have both false positives and false negatives like others. For example, a sophisticated adversary controlling the operating system may provide an illusion of displaced kernel text and data, deliberately violate the locking disciplines, or even tamper with the hypervisor and management VM using security vulnerabilities to circumvent the introspection from TxIntro.

As other VMI systems, TxIntro relies on other means like hypervisor based integrity checking and assurance [25] to guarantee the integrity of kernel code and static data. Further, it assumes that a hypervisor can have full access to guest-VM memory, and thus cannot work under some hypervisor-secure systems [26, 27, 28].

4. Reducing Read/Write Set

Unlike some unrestricted transactional memory proposals [21, 29] in the research community, commodity processors like Intel's Haswell and Sun's Rock [11] are usually built with limited read/write set to track memory accesses during transactional execution. Hence, when running some VMI tools that access a large portion of guest VM's state, the transaction may not make forward progress due to capacity aborts where transactional memory accesses exceed the maximum read and write set.

TxIntro is further refined with awareness of the limited working set for a transaction. First, TxIntro optimizes the two dimensional address translation described in section 2.2, which consumes much more memory accesses than it is required due to address translation, to reduce potentially large read set. Second, TxIntro leverages the asymmetric read-/write set feature of Intel's Haswell processor by using a two-phase VMI to further reduce read set and minimize write set.

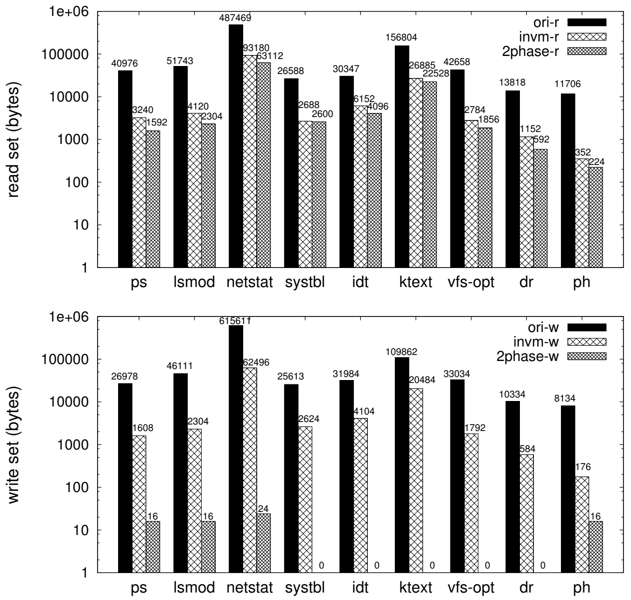


Figure 6: Statistics regarding working set optimization to do VMI for different tools (the unit is byte). Ori-x, invm-x and 2phase-x mean the read/write set for basic, in-VM core planting and two-phase VMI version of TxIntro accordingly.

4.1. In-VM Core Planting

As discussed in section 2.2, a single memory access to guest VM requires two-dimensional walk of both guest VM's page table as well as a hash table. This will significantly increase the working set of the VMI code. Figure 6 shows the working set of 9 VMI tools, many of which significantly exceed the maximum read/write set of a single transaction can hold. To address this issue, we propose an approach called *In-VM Core Planting*, which stealthily plants a virtual core running the VMI code into guest VM's space.

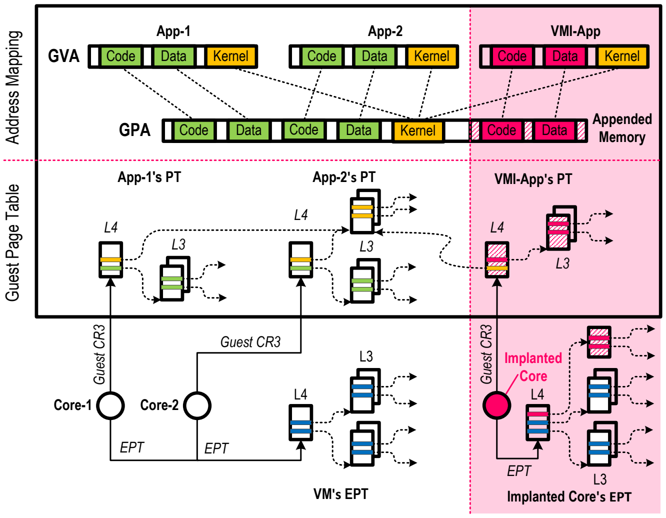


Figure 7: Memory mapping for both the implanted core and guest VM cores.

First, to make the implanted core see the same mapping from GPA to HPA with the guest VM, TxIntro first makes a copy of the extended (or nested) page table of the VM, and marks all the pages as read-only to avoid the VMI code from mistakenly modifying the guest VM's memory, as shown in Figure 7. Then TxIntro further appends a small amount of memory to the implanted core with read/write permission. This part of memory is used only on the implanted core. It is invisible and unaccessible to the guest VM.

Second, TxIntro creates an in-VM page table to run the VMI code. The page table is stored in the appended memory. In the L1 page table directory, the entries of kernel address directly point to the L2 page entries existing in the guest VM memory. Thus the VMI code shares the mapping of kernel address space with the guest VM. The VMI code is running in ring-0, thus it can directly access the kernel memory.

Finally, TxIntro loads the code and data sections of the VMI program to the appended memory, and sets the PC of the implanted core to the start of VMI code. It then issues VMLAUNCH/VMRESUME to run the VMI code. During the execution, TxIntro ensures that no external interrupt will be delivered to the implanted core, all the related pages are pinned to avoid any page fault, and the TLB shoot down interrupt is intercepted to keep guest TLB in sync. At the end of the VMI code is a *cpuid* instruction, which will cause a VMEXIT and trap to the hypervisor.

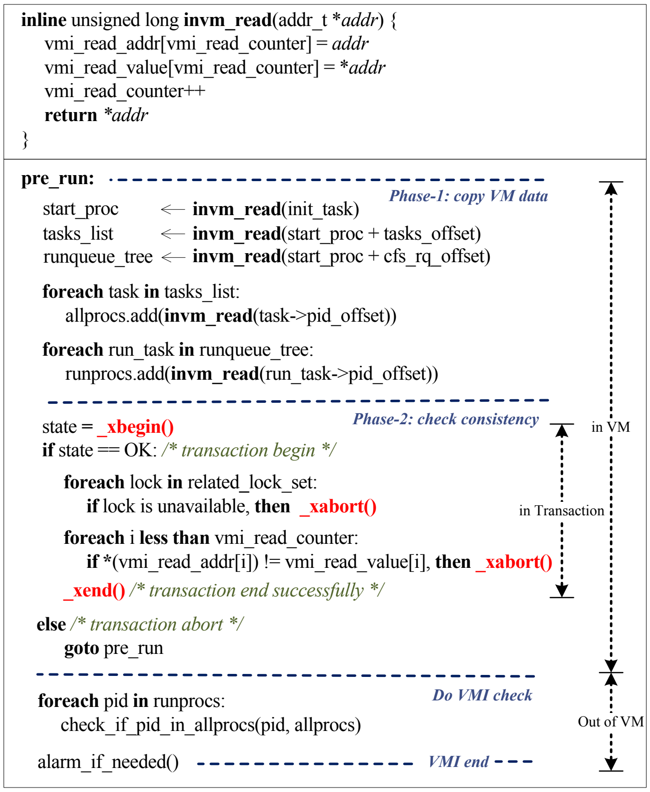
By the carefully manipulated extended page table and in-VM page table, the VMI code can access all the kernel memory without two-dimensional walk of page tables in software. This significantly decreases the working set of typical VMI code. Figure 6 also shows the working set of optimized VMI tools using core planting, which is significantly reduced and can mostly fit into the working set of a typical VM.

Note that the above approach does not compromise strong isolation and stealthiness of traditional VMI approaches. This is because the planted core is completely invisible from the guest VM and the VMI code is only with read-only access to the guest VM data. Hence, neither the VMI code nor the guest VM can tamper with the other peer.

4.2. Two-phase VMI-Copy

Though in-VM core planting significantly reduces the working set of typical VMI tools, the write set of some VMI tools still exceeds the capacity of RTM on typical processors, as shown in Figure 6. To address this issue, TxIntro leverages the asymmetric read/write set of commodity RTM and is built with a *two-phase VMI-copy* approach to further reduce read set and minimize write set.

Specifically, instead of directly running VMI-copy code within a transaction, TxIntro runs in two phases. The first phase runs the main VMI-copy code on the implanted core to directly execute in the address space of the guest VM, and without stopping other execution in the guest VM. During the first phase, other than recording the VMI related data (like pid,



```
inline unsigned long invm_read(addr_t *addr) {
    vmi_read_addr[vmi_read_counter] = addr
    vmi_read_value[vmi_read_counter] = *addr
    vmi_read_counter++
    return *addr
}

pre_run:
    start_proc     <- invm_read(init_task)
    tasks_list     <- invm_read(start_proc + tasks_offset)
    runqueue_tree  <- invm_read(start_proc + cfs_rq_offset)

    foreach task in tasks_list:
        allprocs.add(invm_read(task->pid_offset))

    foreach run_task in runqueue_tree:
        runprocs.add(invm_read(run_task->pid_offset))

    state = _xbegin()
    if state == OK: /* transaction begin */
        foreach lock in related_lock_set:
            if lock is unavailable, then _xabort()
        foreach i less than vmi_read_counter:
            if *(vmi_read_addr[i]) != vmi_read_value[i], then _xabort()
        _xend() /* transaction end successfully */
    else /* transaction abort */
        goto pre_run

    foreach pid in runprocs:
        check_if_pid_in_allprocs(pid, allprocs)

    alarm_if_needed()
```

Figure 8: Example code of two-phase VMI-copy.

process names), TxIntro also collects all of the guest memory access trace: a table of guest virtual addresses *vmi_read_addr* and their corresponding dereferenced values *vmi_read_value*. As the guest VM is still executing, the values might be out-of-date or inconsistent.

In the second phase, TxIntro starts a transaction using *_xbegin* provided by RTM. In the very beginning of the transaction, TxIntro checks the related synchronization variables (e.g., locks) protecting the data structures related to the VMI code. If any of the variables indicate that the corresponding data structures are being modified, the transaction aborts and the entire VMI-copy process reruns. Note that if the checking passes, all of the touched synchronization variables have been put in the read set of RTM. After that, TxIntro begins the consistency checking by simply verifying if the currently dereferenced value of every *vmi_read_addr* entry equals to its previous dereferenced value stored in *vmi_read_value*.

As shown in Figure 8: (1); if there is no change in the collected memory during the VMI-copy process, which is most likely, then the consistency checking passes and the transaction ends normally, which means the VMI data we collected in phase-1 is consistent and can be used for further analysis; (2); if there's any write on the collected memory during the VMI-copy process, then there must be some dereferenced values not equaling to the collected ones, which means data collected in phase-1 may not be consistent, and will cause a retry of VMI-copy; (3); if no collected memory changes during phase-1, but there are some memory modifications during phase-2, since the synchronization variables are touched at the beginning of phase-2, it will cause retry of VMI-copy due to transaction conflict abort. However, since the last two cases rarely happen, and the second phase runs very fast, the impact of false positives is small. Even if they occur, it simply requires restarting transactions from the beginning, without causing actual false alarms on VMI.

5. Implementation and Applications

5.1. Implementation Status

We have implemented a working prototype of TxIntro based on Xen and it supports Linux as the guest VM. TxIntro mainly modifies the management VM (i.e., domain-0), as well as a few changes to the Xen hypervisor to support in-VM core planting. TxIntro currently runs on Intel Haswell processor. TxIntro currently uses the extended page table [30] to isolate the planted core from other cores of the guest VM. But there is no fundamental limitation to support shadow page table. TxIntro makes no modification to guest VMs. In total, TxIntro adds or changes around 1,500 and 840 SLOCs to the domain-0 and Xen hypervisor accordingly.

To monitor updates to critical kernel data structures, one critical issue is precisely knowing which address is modified. One intuitive approach would be using the information provided by HTM aborts. For example, the *RTM_RETIRED.ABORTED* event in Intel's Precise Event-based Sampling (PEBS) describes transaction abort information. However, though the field of *Data Linear Address* originally appears in Haswell's specification, Intel later removed it from the ABORTED event. We use another workaround by leveraging other PEBS events: HITM and MISS [5]. Before a transaction, we first read all of the critical data to ensure they are loaded into the cache. Then we enable monitoring any HITM and MISS events. After that, the transaction starts and all of the critical data is read again. Since the data is in cache, the memory read in transaction will likely not trigger any HITM or MISS events. When there is a transaction abort, we first read all of the data structure again, and stop the monitoring. Now we should get some PEBS records, since there must be some other cores changing one or more addresses of the critical data. By checking the records, we usually can get the linear addresses of data that has been modified, and invoke the corresponding VMI tools. In occasional cases where there is no such record[6], we simply issues all VMI tools, which is still very efficient as shown in section 6.2. Retriv-

[5] MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM, and MEM_LOAD_UOPS_RETIRED.L1_MISS. Suppose core-1 is accessing some memory location that has been changed by core-2. If the data has not yet been written back to memory, core-1 will get an HITM event. Otherwise it will get a MISS event.

[6] The current RTM contains some suspicious behavior not well documented. We did encounter such a case several times during evaluation and didn't have a good reason for this.

ing exact conflicting address here is so complicated, that we strongly recommend restoring the *Data Linear Address* field to ABORTED event.

5.2. Applications of TxIntro

Rootkit	Descriptions
adore-ng	hide process, module, self, log; local root escalation
average-coder	hide process, module, self, log, tcp; local root escalation
dbg-reg	give root right to process; hide self
wnps	hide process, module, self, port, file; key logger; root shell
int3backdoor	give root right to process
ipid	passive covert channel for sending message
kbdv3	backdoor that allow root access
kbeast	hide process, self, port, file; key logger; root escalation
override	hide process, port
ktextmod	give root right to process to kill
dkomhide	hide process, module, port using DKOM technique

Table 1: Rootkits descriptions

We have used TxIntro to write 9 VMI tools to detect a set of kernel rootkits, including control-flow based hooking and direct kernel object manipulation, as shown in Table 1. Table 2 lists the major attacking means involved in these rootkits.

For control-flow based hooking, as most of the memory pages (e.g., IDT, kernel text section, etc.) are set to read-only, kernel mode rootkits usually change pages to read/write by unsetting WP (write protection) bit in CR0 to alter them. As a result, most of the above data structures do not have synchronization variables to protect them, except the packet handler list (*ptype_all*), whose consistency is ensured by *ptype_lock* and RCU. Thus *packet handler check* VMI tool should touch the *ptype_lock* and the corresponding RCU write lock, as mentioned in section 3.

For direct kernel object manipulation (DKOM), we then need to leverage some invariants that may not be broken in order to make the kernel work correctly. The most common hooking targets include process list, module list and TCP port hash table, to hide processes, modules and TCP port accordingly. To defend against them, TxIntro's checking tools need to compare two different views of these states to detect hidden ones. To infer when a checking is necessary, TxIntro adds some critical data structures that are not frequently updated to the read set of TxIntro's monitor. For example, TxIntro monitors the module related states like *kset_list_lock*, *module_mutex* and module lists to infer when a module has been inserted, and thus a checking would be necessary. Similarly, TxIntro monitors the hash table (i.e., *listening_hash* and *lhash_lock*) when a TCP port is created or removed. However, it is usually not worthwhile to monitor process creation and deletion, which happens frequently.

As shown in Table 2, a single rootkit usually uses multiple attacking means. Hence, the VMI tool library in TxIntro are written according to the common approaches used by rootkit and user can easily combine these tools together, by either simply running each tool in turn, or putting the check code into a single transaction by using nested transactions.

5.3. Programming Efforts

Writing a VMI tool using TxIntro is mostly similar to other ones like LibVMI and thus share similar assumptions with prior VMI systems, e.g., requires states to be checked reachable through some global data structures. One difference lies in that it needs to know the related synchronization variables, which are usually easy to infer through looking at the definition of data structures. This has already been an essential process of writing a VMI tool. Alternatively, one can use some dynamic algorithms like lockset [31] to automatically derive synchronization variables associated with a data structure. Another difference is that TxIntro needs to write a simple abort handler, which can usually be automated. Table 2 also lists the SLOCs of the 9 VMI library tools, which is of small code size. Actually, one graduate student in our group can write a typical VMI tool within one hour.

6. Evaluation

6.1. Security Evaluation

To evaluate the effectiveness of TxIntro, we have used 9 VMI tools to detect the 11 rootkits shown in Table 1. These rootkits are implemented as Loadable kernel module (LKM), which runs in privileged mode to directly manipulate kernel memory, CPU register, etc. Though in theory they can change anything in the kernel, our survey of existing rootkits indicates that many of them choose to hook kernel function pointers and tend to lurk in the shadow.

In summary, the VMI tools of TxIntro successfully detect all 11 rootkits shown in table 2. Many of them are very sophisticated. For example, *dbg-reg* is a rootkit that provides transparent syscall hooking, but without modifying IDT or syscall table. It makes use of the debug register by setting breakpoints on both syscall handler and sysenter in DR0 (debug register 0) and DR1 registers. Hence, subsequent syscall will cause breakpoint to fire, and the patched *do_debug* handler places global read watch on *sys_call_table[__NR_syscall]* in DR2. When this syscall is called, *do_debug* will place function pointer to hooked syscall in the task's *EIP* register, which finally implements transparent syscall hook. Although this rootkit utilizes the debug registers that TxIntro cannot touch using transactional memory, it actually has to patch the *do_debug* handler using either IDT or kernel text hooking. Hence, TxIntro can detect it by monitoring the IDT handler code and the first (Debugger) entry.

To detect hidden module, the current policy is comparing the module lists derived from procfs (module list) and sysfs (kset object list). Unfortunately, using this policy, TxIntro failed to detect hidden module attack in *wnps*, which uses a sophisticated means to tamper with the modules lists in both procfs and sysfs. However, this is not an essential limitation with TxIntro and we plan to use more sophisticated policies to detect such a case in future. Fortunately, using the *VFS ops*

Rootkit	Control flow based hook						DKOM based hook			Detected?
	Syscall table	IDT	Kernel text	VFS ops	DebugReg	Packet handler	Process hide	Module hide	Socket hide	
adore-ng				⊕				⊕		✓
average-coder				⊕						✓
dbg-reg					⊕			⊕		✓
wnps		⊕		⊕				⊕		✓
int3backdoor		⊕								✓
ipid						⊕				✓
kdbv3	⊕									✓
kbeast	⊕			⊕				⊕		✓
override	⊕									✓
ktextmod			⊕					⊕		✓
dkomhide							⊕	⊕	⊕	✓
lines of code	170	135	150	185	140	125	225	175	275	

Table 2: Techniques used by rootkits and the corresponding VMI tools

hook VMI tool, TxIntro can successfully detect this rootkit.

In all cases, TxIntro's monitoring module will timely detect updates to the monitored data structures like IDT, syscall table and module list, and then issue related VMI tools to detect the rootkits.

False positive/Negatives: As the combat between rootkits and rootkit detectors will continue, VMI tools written in TxIntro may have false negatives in not detecting some new kernel rootkits. A poorly written VMI tool may generate false positives as well. This can be mitigated by using new rootkit detection policies. Here, TxIntro mainly focuses on providing a mechanism to accommodate different detection policies. Note that, due to some suspicious transactional aborts in current Haswell processor [7], the monitoring part of TxIntro may falsely cause transactional aborts and cause some VMI tools to be executed. This will not lead to false positives, as TxIntro still relies on the results of VMI tools. Fortunately, such events happen rarely.

6.2. Performance Evaluation

To evaluate the efficiency of TxIntro, we measure the slowdown in performance and QoS experienced by the guest VM when VMI tools are executing. We also illustrate the statistics regarding the VMI tools, including the execution time and abort rate of each VMI tool.

The platform we use is an Intel machine with a Haswell machine with 4 cores (8 hardware threads using hyperthreading), running at 3.4 GHz and 32 GB memory. Xen is with version 4.2. We use Linux 3.2 as the OS for the management VM and Linux 2.6.24 as the guest production VM. Each guest VM is configured with 2 virtual cores and 2 GB memory. We use SPECINT 2006 [32] and PARSEC [33] to evaluate the performance slowdown and Darwin Streaming Server to evaluate the service disruption.

Performance Slowdown: Figure 9 and Figure 10 illustrate the performance overhead of running SPECINT 2006 and PARSEC while running the 9 VMI tools every 5 seconds. The results show that even in a worst case TxIntro incurs negnigible performance slowdown. This is expected as TxIntro runs isolated from the production VM. Note that there is a small performance varition of running PARSEC in our VM, even after running it for 5 times. However, the overall performance is similar.

Service disruption: To measure the service disruption of TxIntro on guest VMs, we run all 9 VMI tools every 5 seconds. This represents a worst-case stress test where a system administrator constantly issues introspection operations. Figure 11 illustrates the average delay of all 40 clients. TxIntro has similar average delays with a VM without performing VMI, where the differences are caused by run-to-run variations. As the server delay remains negative, this indicates that the clients have sufficient buffers and thus TxIntro causes no impact to clients. We also collect the maximum delay perceived by clients and TxIntro is also with similar values with a VM not being introspected (omitted here for brievity).

VMI execution time: We also measure the execution time of each VMI tool, as shown in Table 3. As the basic scheme of TxIntro may not work due to excessive working set, we only provide the execution time of the optimized versions. As shown in the table, TxIntro is only with a few tens of microseconds for many VMI tools. The worst case is netstat, which spends around 3,674 microseconds due to touching a large number of states. This is orders of magnitude smaller than prior VMI tools like Virtuoso [9] and VMST [34] that take several to tens of seconds to run a simple VMI tool. This short execution time, together with the significantly reduced working set of typical VMI tools (Figure 6), make TxIntro suitable to run large and complex VMI tools.

ps	lsmod	netstat	systbl	idt	ktext	vfs-ops	dr	ph
181	71	3,674	9	14	79	32	4	3

Table 3: Execution time (us) for different VMI tools

Conflict abort rate: We also collect the transaction abort rate when the guest VM is building the Linux kernel in parallel (with many forks). Specifically, we run our test 8 times for each of the 9 tools, thanks to the fast execution of our tools, no conflict abort happens in 8 of the 9 tools' execution, except the *ps* tool. For the *ps* tool, 6 of 8 runs abort and then

[7] For example, we even encountered conflict aborts when running a single threaded application.

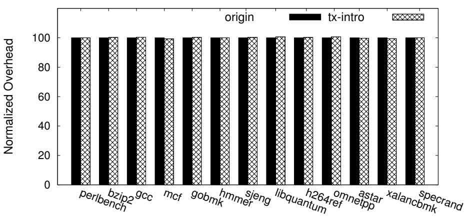


Figure 9: ***SPECINT 2006 perf. overhead.***

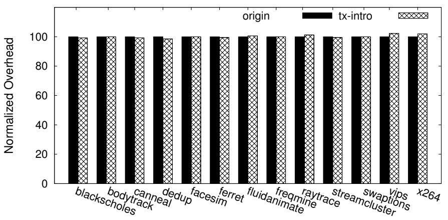


Figure 10: ***PARSEC perf. overhead.***

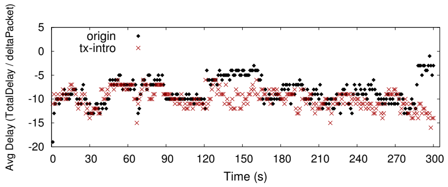


Figure 11: ***Darwin streaming sever delay.***

retry only once before success. Similarly, in the security evaluation, conflict abort seldom happens for the 9 tools unless the rootkit modifies the related data structure like runqueue or process lists.

7. Related Work

Virtual Machine Introspection: Garfinkel and Roseenblum are the first to propose using an "out-of-VM" approach called virtual machine introspection (VMI) to do security checking. Due to the advantage of strong isolation from the VM being monitored, VMI has been intensively studied for secure systems [1, 2, 35, 4, 3, 5, 9, 34, 6]. VMware has also provided a product called vShield that uses a security VM to check the file systems of monitored VMs [36].

Similar to TxIntro, Kruiser [24] also aims at making the monitoring process and guest virtual machine concurrently. However, Kruiser is specifically targeted to heap memory monitoring to detect heap overflow attacks. It uses a ticket-like versioning lock to synchronize monitoring process and guest VM and requires changing kernel memory management code in guest VMs. In contrast, TxIntro is a general framework for VMI and is transparent to guest VMs, thanks to the strong atomicity of hardware transactional memory.

KI-Mon [37] instead designs and implements dedicated hardware to monitor updates to critical data structures in OS kernel to preserve kernel integrity. A special-purpose processor snoops system buses to interpret updates to monitored data structures, which, however, can only monitor updates until they have been flushed from cache to memory[8]. If there is suspicious updates, KI-Mon pulls host memory to do further checking. As the dedicated processor is not synchronized with guest VM's execution, there is no guarantee that the memory snapshot is consistent. By using the strong atomicity of HTM, TxIntro overcomes these issues and runs directly on commodity processors.

Many other systems focus on exploring the applications of VMI [1, 4, 5] and how to easily construct introspection tools [2, 9, 34, 6]. For example, LibVMI [2, 16] provides a library to ease the task of writing VMI tools. Virtuoso [9] and VMST [34] instead leverage binary code reuse to automate the process of writing VMI tools, but at the cost of long VM pausing duration (more than 20s for a simple VMI tool called getpsfile [9] in Virtuoso) and significant performance slowdown (9.3X for user-level VMI tools and up to 500X for kernel-level VMIs), which limit their uses in online monitoring [9]. Exterior [6] further supports changes to a limited set of kernel state, but still requires pausing the monitored VM and excessive performance overhead (23X overhead compared to QEMU).

Currently, there has been very limited consideration of consistent issues. Many systems require pausing the VM being monitored, which, however, cannot guarantee consistent states as the kernel might be paused in a state where an update to data structures is still in progress. Virtuoso [9] chooses to wait the monitored VM into user mode before starting VMI. However, there may be a risk regarding the time of check to time of use (TOCTTOU) between the status checking and the real pausing for VMs with multiple cores.

TxIntro resembles existing VMI systems in many aspects, but does not need to pause VMs and is with a consistent view on VM states and negligible performance overhead.

Using Transactions for Security: Researchers have investigated using transactional memory to monitor data invariants [38, 39] in user code. Harris and Simon Peyton use software transactional memory (STM) in Haskell to actively monitor programmer-specified data structure invariants [38] by hooking STM conflict handlers. Butt et al. [39] further show such functionality can be implemented using hardware transactional memory (HTM) by using a HTM simulator of LogTM-SE [40]. Unlike the monitoring functionality in TxIntro, these systems are intrusive in requiring wrapping the monitored code in a HTM region, while TxIntro leverages the strong atomicity of a HTM (i.e., Intel's RTM) and makes no assumption about the unmodified monitored code. Further, TxIntro is the first to use HTM for virtual machine introspection by applying to kernel code inside a VM and runs on a real hardware with restricted HTM features, while prior systems are for user-level code and runs either on STM or on a full-fledged HTM using a simulator. Hill et al. [38] make a case of leveraging the access summary in LogTM-SE [40] to support watchpoints. TxIntro further shows that it is possible to leverage the strong atomicity of HTM to infer when a VMI might be necessary and describes the challenges and solutions of monitoring key data structures. Further, TxIntro also leverages HTM to provide consistent and concurrent VMI and is built with several techniques to conquer the limited and asymmetric read and write set in commodity HTM.

Chung et al. [41] show how hardware transactional memory can be used to solve metadata races during security

[8] They assume a write-through cache design, which is not the case for commdoity x86 processors

[9] It fetches the name of the executable for a given PID.

checks using binary translation. Birgisson et al. [42] describe a system to ease checking of authorization policies using software transactional memory. Specifically, they group a set of policy checks as well as accesses to authorization policies into transactions, such that it is unlikely to miss some security checks. However, it requires instrumenting operating system code using transactions, which is intrusive and may be impossible for RTM due to frequent permanent aborts by systems events like I/O instructions.

Overall, TxIntro departs from existing work using transactions for security by applying it to VMI, and is the first to leverage the strong atomicity of RTM to ensure consistent and concurrent introspection.

Leveraging Commodity Hardware for Software Security: TxIntro continues the line of research in leveraging commodity hardware features for software security and reliability, including performance counters [43, 44], control speculation [45, 46] and ECC memory [47]. TxIntro adds to the literature by showing that the strong atomicity of commodity hardware transactional memory can be leveraged for concurrent and consistent virtual machine introspection.

8. Discussion and Limitation

Missed monitoring time window: TxIntro might miss some updates to critical data structures when two updates happen simultaneously. This is because, the first update will trigger a transaction abort, making the second update not be detected as the monitoring code is handling the transaction aborts. This is a fundamental limitation with TxIntro using HTM for VMI. Fortunately, these cases are rare in practice as it is unlikely that a rootkit updates two data structures simultaneously[10]. Even so, the missed time window is small and the rootkit may likely be detected during rescan as TxIntro will perform a VMI to related data structure ultimately.

Frequent aborts due to racy execution: A rootkit writer may leverage the mechanism of TxIntro to impede the VMI code from being successfully executed, like frequently touching the synchronization states in the read set of TxIntro. However, as TxIntro is executed stealthily without being known from guest VMs, the rootkit has no knowledge of which data it would access, thus has to frequently touch all of the synchronization states. This may disturb normal execution and provide visible clues to user, which violates the goal of rootkits being stealth. Further, TxIntro can use frequent aborts as a sign to infer the rootkit's existence and report such suspicious events to users to take further actions (e.g., pausing the VM to do VMI).

Huge read/write set not fitting in cache: It would be possible that a complex VMI tool that requires touching a large number of guest states, which makes it hard to fit into the maximum read/write set provided by the processor. For such cases, TxIntro would have to require programmers to split their VMI code into several stages so that each stage can be fit into the read/write set. Fortunately, our optimizations have actually enabled writing of large and complex VMI tools that would be otherwise impossible. Further, we expect forthcoming commodity processors to support larger working set.

Policies vs. mechanisms: It is possible that a rootkit writer may develop rootkits that cannot be easily detected by existing VMI tools, e.g., by breaking the heuristics relied on by VMI tools [48]. This is essentially a typical long-term combat between attackers and defenders. Here, we focus on the mechanisms for VMI, and thus defenders can use TxIntro to provide more sophisticated rootkit detection tools (i.e., policies). For example, it can leverage invariants from hardware states to derive correct view on guest VM [3].

Interference with kernel's future usage of HTM: With the adoption HTM to OS kernel, TxIntro might interfere with kernel's normal usages of HTM, e.g., aborting a normal kernel transaction. We believe this is a common issue of VMI. However, other similar tools not using HTM (e.g., directly inspecting kernel variables) may *always* abort normal kernel transactions with conflicting accesses due to the strong atomicity of HTM, while TxIntro may abort less normal kernel transactions depending on the abort policy. Further, the interference caused by TxIntro is unlikely to be serious as Two-phase VMI-Copy optimization already makes transaction execution time very short, and thus minimizes the chance of conflicting.

9. Conclusion and Future Work

We presented TxIntro, a novel approach that leveraged commodity hardware transactional memory (HTM) support to provide concurrent and consistent virtual machine introspection. Based on the strong atomicity provided by HTM, TxIntro was completely transparent and caused almost no disruption to guest virtual machines. TxIntro was carefully designed with two techniques to reduce the read and write set in commodity best-effort HTM such that a large VMI tool can still run using our framework. TxIntro was shown to be useful by stealthily detecting 11 popular kernel rootkits and incurring negligible performance overhead and service disruption to guest VMs.

We plan to extend our work in several directions. First, TxIntro currently does introspection only and does not directly modify the states of guest VMs. We plan to extend TxIntro so that it can directly kill and remove kernel rootkits from outside the VM. Second, we plan to incorporate prior approaches [9, 34] to automate the process of writing VMI tools. Third, we plan to deploy TxIntro to other HTM platforms to see the performance implications.

10. Acknowledgement

We thank Sylvain Geneves and the anonymous reviewers for their insightful comments. This work is supported by a research grant from Huawei Technologies, Inc., a grant from

[10] Our experiences with more than 10 kernel rootkits confirm this.

Shanghai Science and Technology Development Funds (No. 12QA1401700), a foundation for the Author of National Excellent Doctoral Dissertation of PR China, and China National Natural Science Foundation (No. 61303011).

References

[1] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Proc. NDSS*, 2003.

[2] B. D. Payne, M. de Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Proc. ACSAC*, 2007, pp. 385–397.

[3] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *ACM SIGPLAN Notices*, vol. 46, no. 3, 2011, pp. 279–290.

[4] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proc. S&P*, 2008, pp. 233–247.

[5] Z. Gu, Z. Deng, D. Xu, and X. Jiang, "Process implanting: A new active introspection framework for virtualization," in *Proc. SRDS*, 2011, pp. 147–156.

[6] Y. Fu and Z. Lin, "Exterior: using a dual-vm based external shell for guest-os introspection, configuration, and recovery," in *Proc. VEE*, 2013, pp. 97–110.

[7] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring," in *Proc. CCS*, 2011, pp. 363–374.

[8] B. Hay and K. Nance, "Forensics examination of volatile system data using virtual introspection," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 74–82, 2008.

[9] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. S&P*, 2011, pp. 297–312.

[10] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. ISCA*, 1993.

[11] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance sparc cmt processor," *Micro, IEEE*, vol. 29, no. 2, pp. 6–16, 2009.

[12] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman, "Asf: Amd64 extension for lock-free data structures and transactional memory," in *Proc. MICRO*. IEEE, 2010, pp. 39–50.

[13] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the power architecture," in *Proc. ISCA*. ACM, 2013, pp. 225–236.

[14] Intel Corp., "Intel 64 and ia-32 architectures optimization reference manual," https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2013.

[15] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "Metatm/txlinux: transactional memory for an operating system," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 92–103, 2007.

[16] "vmitools: Virtual machine introspection tools," http://code.google.com/p/vmitools/.

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[18] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[19] A. Walters, "The volatility framework: Volatile memory artifact extraction utility framework," 2007.

[20] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proc. MICRO*, 2001, pp. 294–305.

[21] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proc. HPCA*, 2005, pp. 316–327.

[22] Intel Corp., "Intel software development emulator," http://software.intel.com/en-us/articles/intel-software-development-emulator, 2013.

[23] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proc. CCS*, 2009, pp. 545–554.

[24] D. Tian, Q. Zeng, D. Wu, P. Liu, and C. Hu, "Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring," in *Proc. NDSS*, 2013.

[25] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, 2007, pp. 335–350.

[26] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. SOSP*, 2011, pp. 203–216.

[27] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 437–450, 2012.

[28] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks." in *Proc. HPCA*, 2013, pp. 246–257.

[29] C. Blundell, J. Devietti, E. C. Lewis, and M. M. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, 2007, pp. 24–34.

[30] Intel Corp., "Intel 64 and ia-32 architectures software developer's manuals," http://download.intel.com/products/processor/manual/325384.pdf, 2013.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

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

[33] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proc. PACT*. ACM, 2008, pp. 72–81.

[34] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proc. S&P*, 2012, pp. 586–600.

[35] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proc. CCS*, 2007, pp. 128–138.

[36] VMware Inc., http://www.vmware.com/products/datacenter-virtualization/vsphere/endpoint.html.

[37] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang, "Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object," in *Proc. Usenix Security*, 2013.

[38] M. D. Hill, D. Hower, K. E. Moore, M. M. Swift, H. Volos, and D. A. Wood, "A case for deconstructing hardware transactional memory systems," Department of Computer Sciences, University of Wisconsin–Madison, Tech. Rep., 2007.

[39] S. Butt, V. Ganapathy, A. Baliga, and M. Christodorescu, "Monitoring data structures using hardware transactional memory," in *Runtime Verification*. Springer, 2012, pp. 345–359.

[40] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Logtm-se: Decoupling hardware transactional memory from caches," in *Proc. HPCA*, 2007, pp. 261–272.

[41] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis, "Thread-safe dynamic binary translation using transactional memory," in *Proc. HPCA*, 2008, pp. 279–289.

[42] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode, "Enforcing authorization policies using transactional memory introspection," in *Proc. CCS*, 2008, pp. 223–234.

[43] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *Proc. DSN*, 2012, pp. 1–12.

[44] X. Wang and R. Karri, "Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," in *Proc. DAC*, 2013, pp. 1–7.

[45] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong, "From speculation to security: Practical and efficient information flow tracking using speculative hardware," in *Proc. ISCA*, 2008, pp. 401–412.

[46] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-c. Yew, "Control flow obfuscation with information flow tracking," in *Proc. MICRO*, 2009, pp. 391–400.

[47] F. Qin, S. Lu, and Y. Zhou, "Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs," in *Proc. HPCA*, 2005, pp. 291–302.

[48] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "Dksm: Subverting virtual machine introspection for fun and profit," in *Proc. SRDS*, 2010, pp. 82–91.