Blue-Pill Oxpecker: a VMI Platform for Transactional Modification

Seyed Mohammad AghamirMohammadAli, Behnam Momeni, Solmaz Salimi, Mehdi Kharrazi

Abstract—Although multiple techniques have been proposed with the goal of minimizing the semantic gap in virtual machine introspection, most concentrate on passive observation of the internal state, while there are also a number of proposals with which active modification of the VM's internal state is made possible. However there are issues when modifications are applied, such as keeping a consistent kernel state and avoiding a crash. In this paper we propose Oxpecker, a VMI platform for transactional modification. The out-of-VM read access allows an introspector to detect malware in the guest OS (e.g., rootkit) and the transactional write access allows Oxpecker to reliably neutralize the detected threats. To begin a transaction, Oxpecker monitors VM state changes waiting for an idle moment which is free of possible race-conditions in the guest kernel memory. Thereafter, it invokes a VMI client's callback to proceed with reading/writing in its memory. Upon user request or possible exceptions, transaction is rolled back while the transaction ACID properties are maintained at all times. Oxpecker is implemented and evaluated under different real-world workloads. Additionally and as a practical example, a tool is developed, and open sourced, based on Oxpecker with which guest VM processes could be killed.

1 Introduction

Virtualization technologies have been widely deployed over the past decade and have affected the simple client user to large enterprise users. Virtualization has not been without an effect on the security of the machines going virtual. One of the more interesting facts is that the virtualized machine could be observed from the hypervisor level and that brings about a number of interesting capabilities. One of the earliest suggestions was proposed by Garfinkel et. al [1], where they argue that by placing an IDS outside of the VM, they could monitor the VM for malicious activity while the IDS is immune from any tampering from within the VM. Their approach was named VMI (i.e., virtual machine introspection) and has brought about a number of proposals [1], [2], [3], [4], [5], [6] with the aim of monitoring a VM from the hypervisor level.

Virtual machine introspection is inherently passive, as it tries to read a VM state without trying to modify it. Knowing that a threat exists is important, but being able to neutralize it is an even more important capability. For example, consider an anti-rootkit technique which introspects the VM kernel memory without trusting in its OS executable codes. It may monitor running processes by scanning their memory. A rootkit may try to hide by manipulating the kernel data structures or keep running by corrupting the related kernel modules which can be used for its removal. Nevertheless, by an out-of-VM read/write access, the running rootkit and its memory regions can be observed, debugged, and/or removed. This leaves a rootkit empty-handed because any kind of execution or memory access is inherently visible to an out-of-VM introspector.

• *S. M. AghamirMohammadAli, B. Momeni, S. Salimi, and M. Kharrazi are with the S4Lab, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.*
E-mail: {aghamir@ce., behnam.momeni@, s.salimi@, kharrazi@} sharif.edu

Manuscript received RECEIVED-DATE; revised REVISED-DATE.

This manuscript presents a VMI platform for out-of-VM transactional read/write access enabling several functional and security use cases. For example, the internal state of a VM can be actively modified to force a kernel module to execute paths which are not usually traversed, guide a fuzzing algorithm to obtain higher kernel code coverage, or to facilitate reproducing of race conditions, conduct process injection, restore remote access to an encrypted VM, or perform batch updates and reconfiguration patches for a series of VMs to name a few. Some of these applications were discussed previously in [7], [8], [9], however, to the best of our knowledge, there has been no VMI solution which could provide a transactional and out-of-VM read/write access. For example, Zhui Deng et al. [10] provided one of the earlier techniques to realize writable VMI, but it requires a helper process in the VM. Each one of the proposed writable VMI solutions [7], [8], [9], [11], [12], [13] missed at least one transactional property (as discussed in Section 6) and could not support all the noted applications without causing crashes in the guest VM (GVM).

To elaborate on the importance of the transactional write capability, assume an out-of-VM anti-rootkit technique which intends to kill a rootkit process. In addition to knowing the addresses and appropriate read/write times to avoid possible race conditions, a causal relationship between read and written values must be maintained. An out-of-VM process terminator needs to read the linked list of processes, read the previous/next pointer values of the target process, and then write new values in order to point those entries to each other and remove the target process from the linked list. As shown in Fig. 1, the newly written pointer values for task A and C depend on the old read values obtained by following the `next` and `prev` pointers of task B. Therefore, a concurrent addition of task D by a guest process can potentially corrupt the linked list unless their read/write operations are serialized properly.

This writing challenge has two main parts: where to

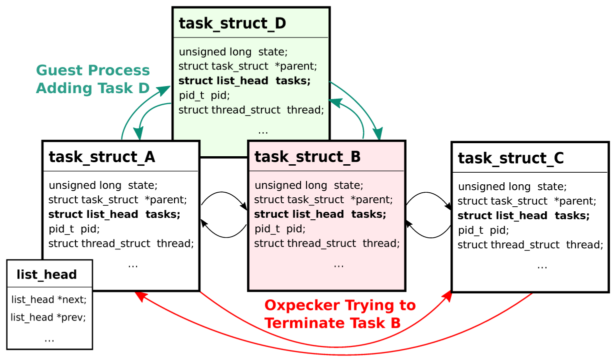


Fig. 1: Linux kernel `process_list` manipulation. GVM and Oxpecker are adding/removing task D/B concurrently.

write and when to write. The first challenge is addressable by an appropriate semantic view of the machine state and is a shared problem with read-only VMI solutions. As for the second challenge, previously proposed solutions exploit the extracted guest OS state to prevent a race condition between the VMM and guest kernel. However, they solve this challenge partially, relinquishing at least one of the transactional write properties. For example, the Exterior [7] uses a dual VM binary code reuse technique to avoid the race condition by injecting instructions in the guest kernel, but it fails to guarantee the consistency property. Our analysis of common OSes reveals a clue for solving this challenge. Normally, each system call tries to finish as soon as possible to minimize the resource contention. This makes it probable for an out-of-VM writer to modify memory without the risk of concurrent modifications. For example, experiments have shown that even under heavy workloads, Linux kernel becomes idle by waiting for less than nine minutes on average.

With the above noted observation, we propose Blue-Pill Oxpecker [1], with which a transactional read/write VMI service for the kernel space is provided. Oxpecker monitors the guest kernel state at the end of each invoked syscall looking for a moment that there is no competing writer process. Then it begins a transaction, allowing a user provided routine to read/write into/from the guest kernel memory in absence of possible race conditions.

Acceptable guest OS performance depends on the hardware acceleration features which allow user-space and kernel-space codes (and syscall invocations) to be performed without slow interventions of the host OS or hypervisor. Whenever an out-of-VM introspection is required, a VM_EXIT event should be handled switching the processor context to the hypervisor, and then back to the GVM once again. This makes it critically important for a practical VMI solution to minimize the number of VM_EXIT events. Monitoring of system calls is paused and resumed periodically by Oxpecker in order to achieve low overhead while also minimizing the time to start for each asked transaction. These optimizations have enabled Oxpecker to impose about 75.84% overhead in the worst case and 1.78% in the best case which is an order of magnitude less than the 2300% overhead of the Exterior [7] tool. We should emphasize that Oxpecker is focused on read/write access to the guest kernel memory, as the importance of out-of-VM solutions is in their ability to deal with infected guest kernel codes. If a threat is confined to the guest userspace, it can be mitigated as discussed in previous works [15], [16] using a guest kernel-level solution too. List of overall contributions follows:

- A simple yet efficient solution is presented in order to decide whether guest kernel memory modifications may cause a race condition,
- An architecture is designed for out-of-VM transactional read/write access to a given guest kernel memory with untrusted executable codes,
- A reference implementation of the proposed architecture is open sourced and its performance and consistency are evaluated empirically,
- Achieving high performance by minimizing the number of VM exit events and periodical VM trapping.

In what follows, Section 2 reviews related VMI solutions. Section 3 presents the Oxpecker architecture. Practical challenges and details of a reference implementation of Oxpecker are explained in Section 4. Afterwards, Section 5 evaluates the correctness and performance of the proposed architecture. Discussion of presented features and a brief comparison with previous VMI solutions is presented in Section 6. The paper is concluded in Section 7.

2 Related Work

The importance of dominating the guest OS and VMI tools race conditions which concurrently access a GVM's memory, makes semantic gap challenge [17] more intricate for writable VMI. We should note that there have been hardware-assisted approaches for handling of concurrent reading [18] and writing [19] for VMI, leveraging hardware transactional memory (HTM) [20]. These approaches trade flexibility and adaptability in order to obtain performance. Nevertheless, Oxpecker is a software-based VMI platform, hence, hardware-assisted approaches fall outside the scope of this work.

Previous VMI tools fall into two main categories, one is focused on introspection solutions for read-only VMI and another one centers around solutions which try to modify the GVM state. The rest of this section reviews the more relevant VMI solutions in these two categories.

2.1 Read-Only Monitoring

A VMI solution can work from outside of the GVM independently [21], [22], [23], [24], [25], with some collaboration with GVM [26], [27], [28], or with an injected program or agent inside the GVM [29], [30], [31]. Among the proposed solutions, we concentrate on independent out-of-VM solutions as they are intended to work for most of the trusted computation bases without fundamental assumptions.

Some of the earliest VMI solutions were designed for malware analysis [21], [24], [25] where they scan the GVM state for footprints of malicious activities. Some other VMI solutions [32], [33], [34], [35] were designed for digital

1. Oxpecker birds are usually red-billed eating parasites off large mammals. It has been noted [14] that at times, they injure the mammal by pecking the wounds. As in our work, Oxpecker is transparent from the VM's point of view, therefore, we have called it a Blue-Pill Oxpecker.

forensics purposes to monitor the GVM state passively. Both types of these solutions, try to narrow the GVM state to obtain only relevant information for the particular use cases and do not aim to generally solve the semantic gap issue.

A number of other VMI solutions are designed to check the GVM kernel integrity. OSck [36] checks the kernel physical memory mapping, ensures the integrity of sensitive files, OS data, and also detects kernel rootkits. RTKDSM [23] monitors kernel data structures with the goal of resolving the semantic gap and tracks kernel changes by monitoring all write instructions for specific kernel addresses. The ProbeBuilder [37] generates kernel data structure probes by looking for "pointer-offset-pointer" patterns in the memory and finding instructions in the execution flow which can dereference them. This allows probes to be generated in the absence of documented kernel data structures, which was one of the open problems in the RTKDSM.

Other notable techniques are Virtuoso [38] and VMST [22], where they reuse the same binary inside and outside of the GVM and use their data to learn about specific information regarding the memory structure of the GVM.

2.2 Active Modification

Two types of VMI techniques exist which support GVM memory modification. One type requires assistant from within the GVM (e.g., a helper process). Another type is implemented independent of GVM and works from out-of-VM (which are more robust). An early example is the work of Zhui Deng et al. [10] which first selects an already existing guest process and then hijacks it, running an implanted process in the context of that victim process. Functionality of this implanted process is restricted to the victim process in order to decrease the risk of being detected. X-TIER [9] works similarly but injects a kernel module instead of a process. The implanted kernel module which is called an X-module is wrapped by a generic loader to become loadable independent of the exact version of the guest OS. An X-module is then executed furtively by mapping/unmapping its kernel memory pages when it calls into other kernel functions (to stay hidden during execution).

Hypershell [11] design is partially similar to the Process Implanting but allows a userspace process which is running in the host machine to use a helper process within the GVM to interact with the guest kernel. It intercepts all syscalls and when it finds the appropriate guest process, injects the desired syscall arguments to be executed in the GVM.

Expanding on VMST [22], Exterior [7] introduced the first Writable VMI solution which can perform automatic and fine-grained out-of-VM write operations, without injecting an agent or employing the guest OS capabilities. Exterior has a dual-VM architecture in which it redirects the memory state modifications from a secure VM (SVM) to an actual GVM. The introspecting process is executed in the SVM, and all instructions of the SVM are emulated one by one, incurring a high performance overhead. This instruction level monitoring allows Exterior to detect all data operations on various kernel data regions and redirect them to the memory of GVM. The idea of dual-VM architecture is also used in the process out-grafting [39] method. To trace a suspicious process behavior, process out-grafting technique relocates the process from GVM to SVM.

CIVIC [8] employs a cloned VM and injects code in it, hence, needs an extra VM similar to dual-VM solutions, and direct modification or reconfiguration of GVM is not possible. In fact, although CIVIC restricts modifications and side effects to a cloned live VM, the VM states must be kept in synchronization in order to avoid possible inconsistencies.

To the best of our knowledge, previously proposed VMI solutions which support modification in the GVM, fail to provide a transactional write capability which is an essential requirement in a real-world deployment. In the following section, we discuss the proposed architecture for Oxpecker and how it provides a transactional write capability.

3 Oxpecker Architecture

Oxpecker enables an out-of-VM introspector to perform a series of read/write operations on the guest kernel memory within a transaction without requiring any cooperation from a trusted guest OS. The rest of this section presents the related adversarial model, the high level architecture of Oxpecker, how Oxpecker deals with the guest kernel memory, how it forms and maintains a transaction, and details of its components and their corresponding algorithms.

3.1 Adversarial Model and Architectural Assumptions

It is assumed that the guest OS is under the control of the attacker, where it could add/remove guest kernel modules, create/terminate guest processes, and scan guest memory for clues about presence of out-of-VM introspection. The guest OS kernel modules are untrusted and so cannot be called to perform certain operations. However, it is assumed that the guest kernel data structures themselves are known and can be used for consistent out-of-VM introspection. This is a reasonable assumption because even though a malware can add or modify the kernel code, it is unrealistic to assume it would convert the kernel data structures to some alternative format. Such change, although possible, would require the malware to re-implement all other kernel modules which are dependent on those data structures.

Furthermore, it is assumed that a generic method is available to access the GVM kernel memory. That is, a low-level API exists for reading/writing arbitrary bytes from the guest memory, of course, without any consistency guarantees. Such a generic API is provided for Windows and Linux kernels by LibVMI [40] library. The consistent and transactional API of Oxpecker is built on top of this generic read/write API. In fact Oxpecker needs to acquire the knowledge of an appropriate time for applying read/write operations in order to avoid inconsistencies with parallel write accesses to the guest OS kernel.

3.2 Transaction Formation

Fig. 2 demonstrates the Oxpecker high level architecture. The *Coordinator* component (first component shown at top of Fig. 2) acts as a facade layer and allows Oxpecker clients to begin/rollback transactions and read/write to the guest kernel memory. *Coordinator* component needs to know about the appropriate times to begin transactions. It employs the *Action Monitor* component to look for the events which represent potential decrease in the guest kernel activity. Candidate events are then inspected by the *Consistency Checker* component to find out about possible race conditions. If it can ensure about absence of race conditions in the guest kernel, it proceeds by invoking the user callback to perform read/write operations. It also tracks memory changes as a set of old memory values paired with their memory addresses, so it can rollback all changes and ensure atomicity of the transaction if the user canceled the transaction or if an exception occurred while executing the user callback.

The *Syscall Interceptor* component (shown at bottom left of Fig. 2) is responsible to communicate with the hypervisor in order to setup required traps and receive corresponding notifications upon execution of GVM syscalls. The last component which is shown at bottom right of Fig. 2 is the *Generic Read/Write Introspection* which provides low-level read/write introspection access to the GVM memory without any guarantee about consistency or atomicity of carried operations. This modular approach allows arbitrary read/write introspection schemes to be leveraged into an API with guaranteed transactional properties. For a better presentation of events, a typical operating scenario is exemplified in Fig. 2 through the following ten steps:

1) Client uses *Coordinator* to initiate a transaction,
2) *Coordinator* asks *Action Monitor* to call it back upon activity changes in GVM. *Action Monitor* registers a syscall trap using *Syscall Interceptor* component in the VMM to pause execution of GVM after each syscall,
3) VMM is notified upon every following syscall invocation through `sysret/sysexit/iret` instructions, triggers a `VM_EXIT`, pauses the GVM, and generates an event so the GVM's state can be inspected before execution of that instruction. *Syscall Interceptor* receives `VM_EXIT` events and redirects them to *Action Monitor*.
4) *Coordinator* asks *Consistency Checker* to determine whether all read/write accesses to the GVM kernel memory are guaranteed to be consistent,

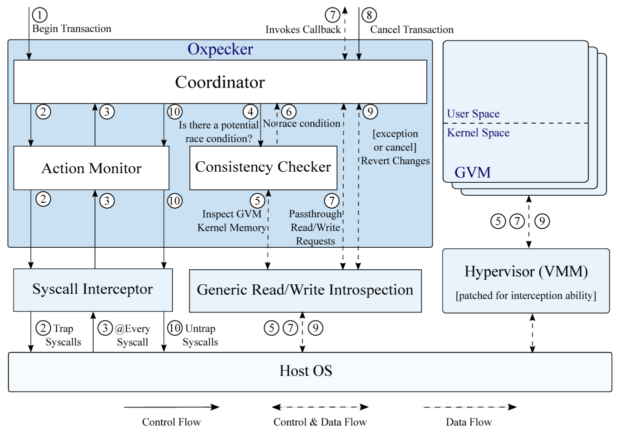


Fig. 2: The Oxpecker architecture. *Coordinator* (top) provides client APIs, manages other components to run user callback, and rollbacks memory changes in exceptional scenarios. *Action Monitor* tracks guest kernel activity at candidate times to begin a transaction. *Consistency Checker* examines existence of potential race conditions. *Syscall Interceptor* registers a trap to be called on every syscall of GVM. *Generic Read/Write Introspection* provides a low-level access to GVM memory without any transactional guarantee.

5) *Consistency Checker* employs the underlying generic read API to introspect the GVM kernel state and detect possible race conditions,
6) *Consistency Checker* informs *Coordinator* that consistent read/write access can be carried now (**step 7**) or the GVM should be resumed to continue its execution while the transaction is kept pending (**step 3**),
7) Knowing that there is no race condition, *Coordinator* runs the client's callback, passing read/write requests through the generic read/write API into the GVM memory space,
8) Client may request the transaction to be canceled before invocation/completion of the registered callback.
9) If an exception occurs or transaction is canceled, *rollback* will be called in order to reapply original values of all modified memory locations,
10) *Coordinator* asks *Action Monitor* to stop notifying about guest state changes which uses *Syscall Interceptor* to allow GVM to run without generating `VM_EXIT` events.

The rest of this section explains the inner design of the aforementioned components.

3.3 Coordinator

The *Coordinator* component sits at the core of Oxpecker architecture to manage the proper usage of different components and maintain prerequisites which should be satisfied before execution of each component in order to ensure the GVM kernel memory consistency. *Coordinator* provides two main APIs for its clients: `beginTransaction` and `cancelTransaction`.

The `beginTransaction` function finds the proper time for reading/writing from/into the GVM kernel memory. This API receives the specific read/write operation details in form of a callback and executes it upon finding the proper time. Fig. 3 shows how this functionality is supposed to work (in the `beginTransaction` function) corresponding to the first seven steps of Fig. 2. The `cancelTransaction` function (shown as the second function in Fig. 3) is provided for resetting the VMM configuration to its normal state. As shown in lines 7-8 of Fig. 3, the VM syscall traps and inspection of the kernel state for determination of a race-free moment are pursued at most for 20 subsequent events. Thereafter, VM is resumed to operate normally for δ seconds and then the monitoring routine starts over. The reason is that when a proper race-free moment is not observed in the short run, kernel is crowded with syscalls and so resuming its operation at normal speed (without unnecessary VM exit events) can increase the probability of finishing competing tasks before the next round of examinations. Moreover, because Nitro just locks the SYSCALL traps, high workload creates a queue from user space process to enter into the kernel space. Therefore, the number of kernel processes increases over time. Performed evaluations confirm this hypothesis, both improving the guest VM's overhead and decreasing the Oxpecker's delay time before application of the callback.

3.4 Action Monitor and Syscall Interceptor

Each guest running process at each given time may execute either in the userspace or the kernel space. There

Fig. 3 Coordinator algorithm to begin/cancel a transaction.

```
function BEGINTRANSACTION($Callback$)
    $done \leftarrow False$
    while $done \neq True$ do
        $listener \leftarrow$ SYSCALLINTERCEPTOR.SETTRAPS(True)
        $count \leftarrow 0$
        for $event \in listener$ do
            if $count \geq 20$ then
                break
            if $event = SYSRET$ then
                $condition \leftarrow$ CONSISTENCYCHECKER()
                if $condition$ then
                    $done \leftarrow True$
                    break
            $count \leftarrow count + 1$
        if $done = True$ then
            $results \leftarrow$ CALLBACK() [atomic]
        SYSCALLINTERCEPTOR.CONTINUE()
        if $done = False$ then
            SLEEP($\delta$)
function CANCELTRANSACTION( )
    $listener \leftarrow$ SYSCALLINTERCEPTOR.SETTRAPS(False)
    for $event \in listener$ do
        SYSCALLINTERCEPTOR.CONTINUE()
```

Fig. 4 Consistency Checker algorithm

```
function CONSISTENCYCHECKER($GVM$)
    $processes \leftarrow$ RETRIEVEPROCESSLIST($GVM$)
    for $process \in processes$ do
        if ISINKERNELSPACE($process.eip$) then
            if $process.status =$ running then
                return False
    return True
```

are three common methods in x86/x86-64 compatible architectures for realization of syscalls: 1) interrupt-based approach which uses a software interrupt such as "`int 0x80`" instruction, 2) the `SYSCALL` group of instructions using `syscall/sysret`, and 3) the `SYSENTER` group of instructions using `sysenter/sysexit`.

By pausing the GVM execution at either entry or exit points of syscalls, it is possible to inspect all guest processes which can potentially cause a race condition when attempting to write into the GVM kernel memory space. The syscall exit point is used as a trigger for performing this inspection.

At the exit point of every executed syscall (i.e. upon execution of `iret`, `sysret`, or `sysexit` instructions), it is expected that all acquired locks over GVM kernel memory objects have been already released. Thus, pausing the system call right at its exit point ensures that no race condition may happen between introspector and that specific process. However, read/write operations may be yet inconsistent with other kernel processes which are running in parallel. Therefore, the *Consistency Checker* is tasked with detecting possible race-conditions with other kernel processes through inspection of GVM kernel memory as explained in Section 3.5. It should be noted that *Action Monitor* needs to freeze GVM temporarily for *Consistency Checker*, hence if there is no race-condition present, it can prolong the freeze in order to complete the transaction at hand.

In summary, the *Action Monitor* component has two objectives: 1) configuring the VMM with a trap listener (using *Syscall Interceptor*), so it causes a `VM_EXIT` to be generated upon each system call of the GVM, and 2) pausing all virtual CPUs of the GVM to give *Coordinator* a chance to inspect its memory space safely.

3.5 Consistency Checker and Generic Introspection

Existent libraries such as LibVMI [40] can be used in place of the *Generic Read/Write Introspection* component. Nevertheless, knowing the proper timing is essential for consistent GVM modifications. It is not trivial to maintain transaction properties [41] without any cooperation of guest OS, but we had two observations making it possible to guarantee consistency of modifications independent of the guest OS implementation details. We should note that the only dependence on OS is limited to its process data structures for enumeration of processes which are active in the kernel space. This dependence can also be generalized using a volatility template [42] to learn about the GVM processes.

The first observation is that a modern OS kernel is supposed to be deadlock free. For this reason, it avoids leaving the kernel space while holding a lock because it is not clear when it might return to the kernel space, and another kernel thread or process may require the lock in the meantime. For example, consider the `delete_module` syscall. It takes a lock for the `module_list` in order to remove a target module. Now suppose that kernel stops running `delete_module` syscall while the lock is kept. This makes all other processes which try to access the `module_list` to hang indefinitely until execution of `delete_module` is resumed and the lock is released. In other words, a functional OS kernel needs to release taken locks when a process leaves the kernel space and when it returns to the kernel space, it may reacquire the needed locks.

The second observation is that even in a heavily loaded system in which different threads are invoking syscalls one after another, OS tries to minimize the execution time of each syscall in order to maximize the performance. Measuring the amount of time which is spent in user and kernel spaces, we found that even in a highly loaded system there are times that no process is in the kernel space. Using this observation, Oxpecker can monitor the execution of GVM in order to find an appropriate time which is free of any possible race conditions. Finding a time instant in which only the `idle` process is running in the kernel space is likely as we will show later on. So the trade-off between the guest kernel memory modification delay and the kernel consistency assurance after applying the asked modifications can be resolved reasonably in favor of the guaranteed consistency without a considerable performance loss.

At the exit point of each syscall, all virtual CPUs of the guest VM are paused so the GVM state can be kept unchanged while existence of race-conditions is being examined. Afterwards, the *Coordinator* component is notified which in turn calls the *Consistency Checker (CC)* component. The CC component checks for possible race conditions for read/write operations in GVM by inspecting its kernel memory state. Specifically, CC iterates over all running processes in the GVM kernel space by reading its kernel memory using the *Generic Introspection* API. It checks whether there are any kernel processes which may modify

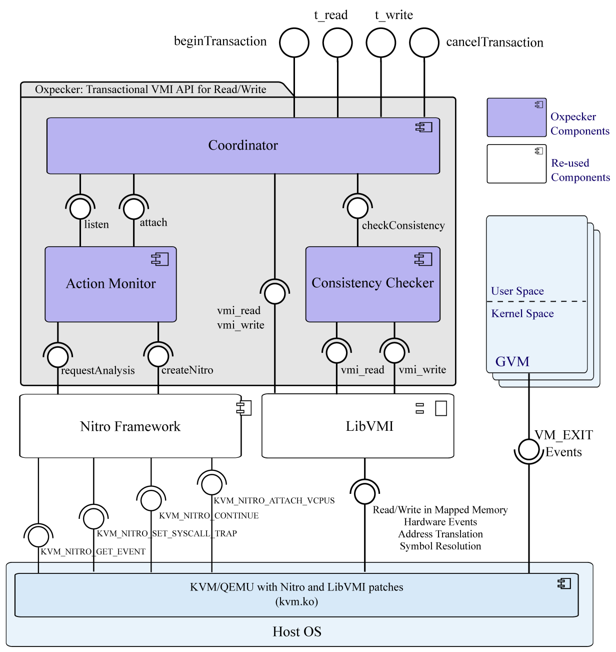


Fig. 5: The Oxpecker component diagram. The *beginTransaction* and *cancelTransaction* APIs configure/deconfigure VMM through *Coordinator* and *Action Monitor* components which in turn use *Nitro Framework* for invoking user callback in a proper time. *Consistency Checker* employs *LibVMI* APIs for introspecting the paused guest VM. All exceptions are handled automatically for maintaining transaction properties.

the memory in parallel or just the `idle` process is running in the kernel space.

As shown in Fig. 4, both of kernel threads and user processes which have switched to the kernel space can be enumerated by reading the GVM kernel memory. The good news is that the number of competing processes in the kernel space can decrease only at the end of syscall or interrupt handling procedures. Using this knowledge and limiting inspection moments to end of syscalls (as detected by *Action Monitor* component) makes it possible to use hardware virtualization while waiting for `VM_EXIT` events and improve performance from emulation speed to an acceptable slow down (about 75.84% max overhead on tested scenarios).

The next section describes practical challenges and implementation details of the Oxpecker components.

4 Implementation

Oxpecker is implemented and evaluated in a customized KVM/QEMU environment using Windows and Linux guest OSes. Fig. 5 shows the Oxpecker component diagram which is designed in three main layers. Our implementation of Oxpecker is available at [43]. The rest of this section elaborates on implementation details of the noted components and their associated practical challenges.

4.1 Hypervisor, Libvmi, and Nitro Framework

Kernel-based Virtual Machine (KVM) [44] in Linux employs a two level architecture. The VMM independent functionalities are provided by `kvm.ko` module while hardware specific details are placed in modules such as `kvm-intel.ko` and `kvm-amd.ko`. Userspace programs can communicate with the KVM modules to create/manage VMs through `ioctl` requests to the `/dev/kvm` device node since Linux version 2.6.20 and QEMU version 1.3.

The QEMU process can map memory pages which are allocated to the GVM memory to its own virtual memory address space in the host machine and so can read/write its contents. But there is a "semantic gap" between the memory pages which are seen in QEMU, depicting the physical memory of the GVM, and the address spaces of different guest processes and/or guest kernel. Other patches to QEMU such as LibVMI [40] deal with this semantic gap problem allowing a host process to access the memory of guest processes. Oxpecker uses the LibVMI [45] introspection API (refer to I in [46]) and builds required consistency guarantees on top of that API LibVMI facilitates access to GVM memory, without providing any guarantee about consistent read/write operations. Specially, the provided write API of LibVMI can lead to GVM crashes. LibVMI supports different hypervisors such as *Xen*, *KVM*, and *QEMU*.

In addition to the generic read/write access to GVM, *Coordinator* needs to be notified about syscall operations so it can inspect the guest kernel state at change instants without degrading the execution performance. The Nitro framework (refer to II in [46]) is responsible to intercept syscalls and notify *Action Monitor* component before returning from each syscall or interrupt handling routine. The original Nitro code could cause about 30% overhead on GVM. However, the overhead in Oxpecker is much less as reported in Section 5, because syscalls are only monitored from the user request moment until the beginning of a transaction.

4.2 Action Monitor and Coordinator Component

Oxpecker uses KVM/QEMU with Nitro [47] patch through the *Action Monitor* component as shown in Fig. 5, to enforce a *VM_EXIT* after all syscalls while using the hardware virtualization performance benefits. However, the original *Nitro Framework* only pauses the virtual CPU which had executed a syscall. Thus, a modified Nitro patch is developed [43] in Oxpecker which pauses all virtual CPUs while emulating the `iret`, `sysret`, and `sysexit` instructions in order to avoid race conditions while checking the number of running processes in the kernel space. It sets/removes syscall traps and notifies the *Coordinator* component right before allowing the guest OS to return from every syscall.

The *Coordinator* component, as the name suggests, coordinates tasks between the different Oxpecker components. As shown in Table 1, six APIs are provided for Oxpecker clients. The main API is `beginTransaction` which takes a callback function and invokes it after finding the appropriate time for consistent read/write access to the GVM.

4.3 Consistency Checker Component

The *Consistency Checker* component is supposed to count the number of running processes in the kernel space. Since recognition of running processes needs knowledge of the GVM kernel data structures, this implementation uses Linux kernel version 4.9 and Windows 7 SP1 for demonstration purposes. Having the data structures which are used in other OS versions is enough to port this component. This approach can be generalized using a volatility template [42] or a framework like rekall [48] (as is used in our reference implementation) to learn about the GVM processes.

4.3.1 Linux Guest OS

The main data structure providing this information in Linux is `task_struct` structure which describes a process or task in the kernel. For each process, the `state` field in the corresponding `task_struct` indicates its running status which can be *running*, *sleeping*, or *stopped*. Moreover, to find that a process is going to be scheduled in kernel or userspace, the `sp` field of the structure pointed to by `thread` field of the `task_struct` is used. LibVMI is configured by reading the *System.map* from the guest kernel. This file contains the "*symbol table*" which maps symbol names and their addresses in memory. Also the `init_task` is a kernel symbol which points to the head of a list of all existing `task_struct` data structures. This list begins with a hard-coded `task_struct`, namely *Swapper* process, which has a zero *pid* representing the *idle* process in Linux which causes no inconsistency as it does not access critical sections.

In brief, *Consistency Checker* starts by finding the address of `init_task` and `task_struct` list in memory by looking at *System.map*. It then iterates over the list to enumerate running status of all processes. For each `task_struct`, it inspects its `state` value and existence of a kernel stack frame pointer to find all processes which are running in the kernel space and can cause a potential race condition with writes in the kernel memory. This method has a drawback, as it depends on `task_struct` of all processes. An adversary can remove a process (e.g. a rootkit) from the tasks list while keeping it in the *runqueues*. In this situation, that process stays in the kernel space indefinitely. Such processes can be determined by looking at *runqueues* or memory contents with help of the rekall [48] and Volatility [42] framework.

4.3.2 Windows Guest OS

Each process in Windows is represented with an instance of *EPROCESS* data structure containing a *KPROCESS* for kernel control data. Also, running processes may have one or more threads which are tracked by instances of the *ETHREAD* data structure (which similarly keeps its kernel control data in the *KTHREAD*). *EPROCESS* maintains its threads list as a linked list in its *ThreadListEntry* member.

PsActiveProcessHead can be used to retrieve the list of *EPROCESS* instances. To find running processes, this linked list should be traversed, checking the corresponding threads, and for each *KTHREAD* instance, its *state* field should be examined. Finding that a process is running in kernel or user space is trickier. *KernelStack* (a member of the *KTHREAD* data structure) is used to maintain the execution context of each thread by pointing to its stack frame. Value of *eip* (indicating the current execution address) can be extracted by looking at two words behind the *KernelStack*. Similar to the Linux guest OS scenario, this process might be affected by hidden processes. However, we can get the full list of processes by looking into the memory [42], [48].

TABLE 1: Oxpecker APIs. X (in API names) can be 8, 16, 32, or 64 bits.

API	Description
beginTransaction	Asynchronous function to start a transaction.
cancelTransaction	Cancel/rollback an ongoing T-VMI transaction.
t_read_X	To read X bits from a virtual address.
t_write_X	To write X bits at a virtual address.
t_read_X_pa	To read X bits from a physical address.
t_write_X_pa	To write X bits at a physical address.

4.4 Killing Process Using Oxpecker

Oxpecker APIs allow a VMI client to implement arbitrary programs without worrying about possible memory-level inconsistencies. However, the program must handle the behavioral (design-level) inconsistencies by itself. For example, Oxpecker can ensure that there is no race condition while an out-of-VM client tries to kill a guest process (i.e., there are no kept locks on the guest memory), however, the client still needs to check that the target process is not uninterruptible. To exemplify a similar situation with regards to the read API of the LibVMI, we can point out to a scenario that user tries to read a virtual address which is not mapped (for the asked process). In this scenario, LibVMI raises VMI_FAILURE error. Now, assume that user had completed a series of successful read/write operations before triggering that VMI_FAILURE error or killing an uninterruptible process. The rollback mechanism automatically reverts the effect of previous write operations, covering the erred callback routine modifications.

As a practical example, Oxpecker provides a tool (refer to III in [46]) to kill a process in Linux distributions by re-implementing the necessary read/write instructions in *do_exit* function of Linux kernel using the Oxpecker APIs. Details of process killing example and its evaluation results are available in the Oxpecker repository [43].

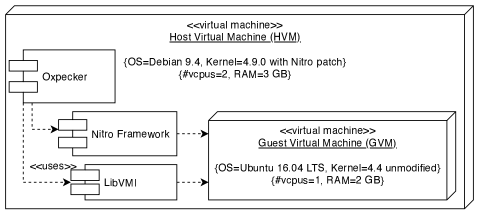
Next section employs Oxpecker in more challenging scenarios to evaluate its functionality and performance.

5 Evaluation

This section evaluates a reference implementation of Oxpecker through two types of experiments. First, the performance effect of Oxpecker is observed under different workloads in order to measure the performance of both the host and guest machines. Second, the write consistency is monitored during Oxpecker concurrent modifications.

Since Oxpecker is not dependent on the guest userspace workload, the experiment scenarios are designed to create different kernel workloads. This is accomplished through the common network-driven (HTTP and FTP) applications in addition to the TPC-H [49] standard benchmark. Fig. 6 shows the Linux GVM setup used in these experiments. Similar deployment was used for Windows GVM.

Two GVM OSes run Ubuntu 16.04 LTS with Linux kernel 4.4 (unmodified LTS version) and Windows 7 SP1 over a one core CPU having 2 GB RAM. Oxpecker does not require modifications in the introspected GVM. The host virtual machine (HVM) runs Debian 9.4 with Linux kernel 4.9.0 over two Intel core i7 4710HQ 2.50GHz CPUs with 3 GB



<<virtual machine>>
Host Virtual Machine (HVM)
{OS=Debian 9.4, Kernel=4.9.0 with Nitro patch}
{#vcpus=2, RAM=3 GB}
Oxpecker
Nitro Framework
<<uses>>
LibVMI
<<virtual machine>>
Guest Virtual Machine (GVM)
{OS=Ubuntu 16.04 LTS, Kernel=4.4 unmodified}
{#vcpus=1, RAM=2 GB}

Fig. 6: Deployment diagram. HVM hosts Oxpecker, Nitro, and LibVMI to inspect/modify GVM kernel memory.

RAM in both scenarios. The HVM kernel is patched with Nitro, so the GVM syscalls can be intercepted efficiently.

In what follows, Section 5.1 presents the first type of experiments conducted to measure the performance of host and guest machines under different workloads in presence of Oxpecker concurrent transactions. The second category of experiments for inspection of possible inconsistencies during concurrent memory accesses is presented in Section 5.2.

5.1 Performance Experiments

Performance of VMI and specially its overhead on GVM is an important metric for evaluation of a VMI solution. An experiment is designed to measure the performance of Oxpecker from two aspects. One aspect is the performance that a client of read/write VMI operations may expect when accessing the GVM. Another aspect is the effect of Oxpecker on the performance of the GVM.

To evaluate the Oxpecker performance in real conditions, different workloads are created on the GVM while the VMI requests are being processed. The Locust [50] testing platform is used to generate a variety of client traffics running in HVM, while a server is running in GVM. Number of client requests affects the guest kernel workload (i.e., server). Locust simulates concurrent users, noted by c, sending requests to the server interleaved by random wait intervals (between 100 to 1000 milliseconds) generated by the HttpUser implementation. For database scenario, GVM runs queries according to TPC-H in an infinite loop, executing 22 queries [51] in each round on a PostgreSQL instance.

Alongside with the client/server and database queries which make the background load on VM, an Oxpecker client sends a series of uniform requests to GVM, waiting for μ seconds between consecutive requests. Oxpecker executes asked transactions after finding the proper time to avoid race conditions while accessing the GVM memory. Therefore, there will be an inevitable delay between beginning of each VMI transaction until its actual execution. This time is used as a criteria to measure the Oxpecker performance.

Fig. 7a and Fig. 7b summarize the statistics of the VMI client delay for both workloads (i.e. FTP and HTTP client-server requests) in Linux and Windows environments respectively. Each experiment is repeated 1000 times for each c value. The time interval between consecutive VMI write requests is configured as five seconds. Two CPUs are used in the evaluation environment (using just one thread per CPU) for HVM and one CPU for GVM. Therefore, 1000 concurrent users exhaust GVM simulating a high load production environment. The Standard Error of Mean (SEM) is calculated according to the $SEM = (Std.Dev)/\sqrt{N}$ formula, normalizing the standard deviation with root of N (the size of sampling data which is 1000 in this case). SEM can be multiplied by 1.96 to obtain an estimated 95% confidence interval. As shown in Fig. 7, SEM values increase when the number of concurrent users (c) increases. Moreover, increasing the c and generating higher workloads in both of the FTP and HTTP test scenarios leads to an increase in the variance and mean of observed delays because it is more probable for guest kernel to be busy processing FTP/HTTP requests. There is an exception in the HTTP application (for both OSes) in which the mean delay decreases when the number of concurrent users is increased from one user to 10 users. This can be explained in terms of the active syscalls because in the $c = 1$ scenario of HTTP application, the system was idle in most of the times. Hence, the rate of SYSEXIT events was lower and Oxpecker had to wait longer for SYSEXIT traps. Consequently, it caused higher delay time during the test with one user in the HTTP scenario. In the FTP scenario, mean delay is higher than HTTP and TPC-H scenarios. This is due to a higher workload created by FTP requests. It performs more operations such as authentication and file operations which increase kernel workload by invoking more syscalls for file/network access.

Finally, Fig. 7 shows the feasibility of Oxpecker in high load situations even when the GVM is exhausted by clients. In the worse case scenario (1000 concurrent user and FTP scenario), the average delay time is less than 9 minutes. That is, one can execute Oxpecker transactions in less than 9 minutes in high load environments.

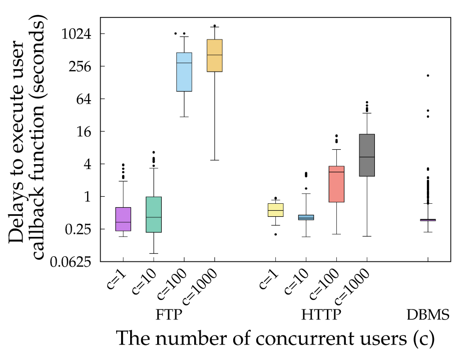
So far, the performance that Oxpecker clients can expect for reading from/writing into GVM is evaluated under different workloads. Another important performance aspect is the overhead which these operations add to the GVM processes. The output of Locust.io [50] while running FTP client for 5 minutes is used to measure this overhead. Fig. 7c depicts the calculated requests per seconds for each c value, indicating the number of concurrent users. Furthermore, the gap time between VMI requests is set to 5 seconds (μ) and the back-off time (δ) is set to 0.5 seconds.

The overhead is calculated according to Eqn. 1 in which $\mathcal{R}(c, \mu, \delta)$ represents the number of requests per second for FTP application in GVM when Oxpecker setups with μ and δ. Also, $\mathcal{R}(c, \infty, \infty)$ indicates the overall time when the Oxpecker is not present at all.

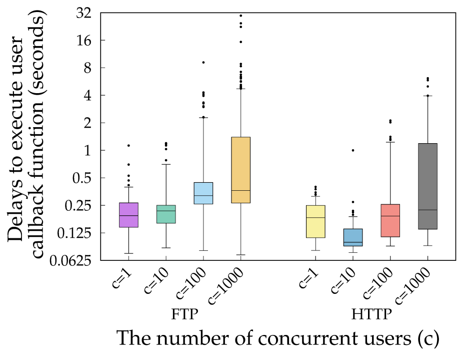
$$Overhead(c, \mu, \delta) = \frac{\mathcal{R}(c, \infty, \infty) - \mathcal{R}(c, \mu, \delta)}{\mathcal{R}(c, \infty, \infty)} \tag{1}$$

As the number of concurrent users and requests rate increase, the Oxpecker's delay and the average response time increase too filling the system capacity. Thereafter, the rate of FTP requests remains unchanged or slightly decreases (see Fig. 7c around $c = 50$). In fact, for $c > 50$, the CPU is overwhelmed, hence, increasing concurrent clients cannot increase the system load and requests per seconds furthermore because extra requests have to wait for their turns to be processed by the FTP server.

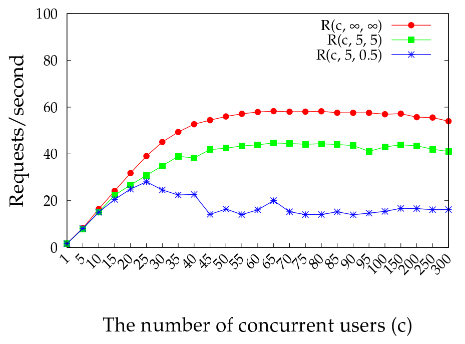
Fig. 7c shows that the overhead of Oxpecker is about 28.62% (or 75.84%) at most and is as low as 2.71% (or 1.78%) at best when δ is set to 5 (or 0.5) seconds. In the



(a) Linux



(b) Windows



(c) Locust delays

Fig. 7: Two initial figures show Oxpecker's delay to begin a transaction in Linux and Windows. Last figure shows the GVM overhead showing the average response time of Locust FTP requests. The c is the number of concurrent users sending requests and $\mu = 5s$ indicates the uniform gap between transaction requests.

worst case scenario, where the background workload is higher on GVM, Oxpecker makes 75.84% overhead on GVM processes which supports the relative usability of Oxpecker in production environment.

5.2 Write Consistency

This section inspects write accesses of Oxpecker to kernel regions while GVM kernel modules can potentially access and modify them concurrently. For this purpose, a kernel module (refer to IV in [46]) is developed which waits for a second using the `msleep_interruptible` and repeats execution of a critical section. Within the critical section, it holds a mutex (i.e. `my_lock`) and busy waits for one second. Concurrently, a series of VMI transactions are initiated after loading this kernel module in the GVM in order to modify the same data item (i.e. `dining_spoon`) in the kernel space. This shared variable is inspected in the critical section and if it is modified while the `my_lock` mutex is being held, it is counted as a crash by increasing the value of `crash` variable. Oxpecker client competes with this kernel module to update this shared variable with this difference that it generates and writes a random number in each cycle. If Oxpecker misses any race condition while trying to update the same data item, the kernel module will be able to detect the concurrent modifications.

This experiment is repeated under two conditions. Once using the LibVMI API which provides no transactional guarantees and once using the Oxpecker API. As expected, the LibVMI scenario generates more crashes as it keeps updating the shared variable at arbitrary moments. But Oxpecker makes the modification while the kernel module is sleeping out of critical section and so the `crash` variable stays at zero.

6 Discussion

Section 2 reviewed VMI solutions including those with memory modification capability. This section compares writable VMI solutions based on the criteria shown in the Table 2. The first four rows indicate the main properties of an ACID [41] transaction. Transactions contain a series of read/write operations. If only one execution thread tried to access data structures, it could proceed flawlessly with this assumption that the transaction operations were not faulty (e.g., dereferencing a null pointer). However, in a real-world system, multiple guest processes invoke syscalls and try to access the same kernel memory pages that a VMI client is interested in, from out-of-VM. As depicted in the first four rows of Table 2, at least one of transaction properties is missed by each solution, except the Oxpecker.

Next group of criteria return to the underlying capabilities that they need to function properly. *"Guest Kernel"* row indicates the trust level and integrity of the guest kernel which is assumed in the proposed solution. Most

TABLE 2: Comparison of writable VMI solutions. (Un)Supported features are marked as (✗)✓. The `1vcpu` shows that only one virtual CPU is supported. `TC` solutions need trusted guest kernel code while `KDS` needs non-tampered kernel data structures. `GUSP` and `GKM` mean that a guest userspace process and kernel module interfaces are provided respectively while `HUSP` means that VMI client can use the interface of a host userspace process (with mapped syscalls). `TRW-API` means that VMI client should use read/write APIs in a transaction context to access the GVM. The 1x (2x) reports that the same (double) amount of GVM memory is approximately required. Finally, `#CS`, `#Int`, `#SC`, and `#Inst` indicate the number of context switches, interrupts, syscalls, and executed instructions.

	Process Implanting	X-TIER	Hypershell	CIVIC	Exterior	Oxpecker
Atomicity	✗	✗	✗	✓	✗	✓
Consistency	✓	✗	✓	✓	✗	✓
Isolation	1vcpu	1vcpu	✗	✓	✓	✓
Durability	✓	✓	✓	✗	✓	✓
Guest Kernel	TC	TC	TC	TC	KDS	KDS
Fully Out-of-VM	✗	✗	✗	✗	✓	✓
Flexibility	GUSP	GKM	HUSP	GUSP	HUSP	TRW-API
Memory Usage	1x	1x	1x	2x	2x	1x
#VM_EXIT	#CS	#Int	#SC	0	#Inst	#SC
Usability	◓	◔	◔	◓	◔	●
Year	2011	2013	2014	2017	2013	2018

of solutions need a trusted guest kernel code (i.e. they can be detected by rootkits) except Exterior [7] and Oxpecker which just need known kernel data structures and do not rely on the guest modules themselves. "*Fully Out-of-VM*" row marks solutions which do not need any in-VM agent and instead use the VMM capabilities to pause/resume the VM and read/write from/into its memory. The benefit of these solutions is in their independence of untrusted guest kernel with the drawback of facing the semantic gap problem. Furthermore, the granularity of the introspection API provided for each solution (7th row), such as injecting a kernel module or changing a byte in the memory, indicates which use cases could be implemented over it.

Aside from functionality, proposed solutions can be compared based on their resource usages. Some VMI solutions [7], [8] require cloning GVM for sake of consistency, hence, doubling the host machine memory as shown by 2x in the "*Memory Requirement*" row. The "#*VM_EXIT*" row reports the extra *VM_EXIT* events which are added due to the VMI operations. Each VM exit event acts like a process context switch, but is much costlier due to the GVM state switching. In fact, as the number of *VM_EXIT* events is reduced, GVM will be able to run longer on hardware virtualization uninterruptedly and incur a smaller performance overhead. The "*Usability*" row combines the provided functionality and aforementioned memory and performance factors in a qualitative manner. As more features are provided, usability is increased and as more memory is used and further overhead is imposed, the usability is decreased.

In the rest of this section, details of the previously proposed methods are examined. One of the earliest solutions was Process Implanting [10], in which the number of *VM_EXIT* events is limited to the number of process context switches and consistency is maintained since all modifications pass through the guest syscalls. Thus, it depends on the correct operation of guest syscalls. For example, a rootkit can detect an implanted process by inspecting its memory addresses right before scheduling it or during the execution of its invoked syscalls. In order to resolve this issue and provide isolation from other guest processes, Process Embedding depends on the existence of a single virtual CPU so all context switches can be intercepted sequentially and the implanted process can be removed from the guest OS memory before scheduling the next potentially malicious process. Memory modifications for implanting a process itself are reversed at its removal, but its writes to the guest memory remain durable. These modifications are not atomic as they may be interrupted by exceptions.

Similarly, X-TIER [9] requires a single virtual CPU configuration. It tries to provide atomicity by intercepting interrupts and exceptions during the execution of an injected kernel module. However, if it calls an external function, the processing of interrupts will be resumed, and the guest kernel functions can be executed before resuming the execution of the injected module. Additionally, exceptions in the execution of the guest module itself are intercepted and delayed. Memory pages of the injected kernel module are unmapped at its removal, its modifications to other data structures are durable, but atomicity is not guaranteed. More crucially, if injected module tries to modify a shared memory while another kernel thread is taking its lock, memory inconsistency may arise as the injected module cannot wait for the lock to be released. Such an issue limits its usage to read-only introspections in practice. Also, all interrupts lead to *VM_EXIT* events in order to guarantee the injected kernel module isolation. Finally, injected module can be detected if a rootkit modifies one of the called external functions and check for its return address which is unmapped in X-TIER solution.

Hypershell [11] needs a helper process inside the GVM. Therefore, consistency and durability are achieved, but atomicity is not guaranteed since termination of the helper process leaves half of the operations applied and cancels the rest without any rollback mechanism. Finally, the shared helper process limits the concurrency of introspecting processes. For example, guest file descriptors are shared among independent introspecting processes. Generally, the shared kernel context of the helper process challenges the isolation between host-based introspecting processes.

CIVIC [8] periodically clones the GVM and works on a cloned VM. This approach ensures atomicity, consistency, and isolation while missing the durability for the original GVM as all operations are executed over separate cloned VMs. Furthermore, although it uses a copy-on-write approach to keep memory usage low, the memory usage is doubled in the worst case. Cloned VM introspection mechanism itself is based on injecting a process in the VM and so depends on the presence of trusted guest kernel code.

Finally, Exterior [7] uses a dual-VM setup. A secure VM (SVM) to run the introspecting process, monitor its executed instructions, and emulate their effects to update the GVM. This emulation causes the maximum relative overhead. However, the trust level is much lower and knowing about the non-tampered guest kernel data structures is enough to redirect SVM operations to GVM. Data durability and isolation are achieved, but there is no guarantee about atomicity of executed instructions because they are streamed to the GVM memory. It also avoids redirecting memory accesses of specific sequences of instructions (e.g., lock taking) in order to reduce inconsistencies. Because the GVM is paused at a random moment for SVM operations to take effect, if a redirected memory operation modifies a critical section which had been locked by GVM before being paused, the crash of GVM is highly plausible.

For example, Fig. 8 depicts two sample situations in the `delete_module` syscall which can cause inconsistencies. There is a possibility that `module_mutex` is unlocked by VMI in line 3 while it was locked in the paused GVM. Exterior [7] is immune to these types of errors because it ignores redirection of lock taking routines such as the `mutex_unlock` example. However, if GVM is paused in line 8 and Exterior continues by removing a module, it interferes with the guest OS in line 9 because the `mod` local variable is pointing to a dangling area which was used by an old module. This inconsistency leads to a crash in GVM.

On the other hand, Oxpecker guarantees atomicity by maintaining a transaction and rolling back modifications in case of exceptions. Consistency and isolation are guaranteed because GVM is monitored and paused at a proper moment in order to eliminate any potential race condition. Different transaction requests are also serialized similar to the guest processes. However, this serialization has a drawback. If a

```
1   SYSCALL_DEFINE2(delete_module, const char __user*, ↩
       name_user, unsigned int, flags) {
2     /*some initialization... setting `name` in kernel*/
3     // mutex_unlock by VMI
4     if (mutex_lock_interruptible(&module_mutex) != 0)
5       return -EINTR;
6     mod = find_module(name);
7     /*some further checks*/
8     // delete module by VMI
9     ret = try_stop_module(mod, flags, &forced);
10    mutex_unlock(&module_mutex);
11    /*release some resources and return*/
12  }
```

Fig. 8: Parts of the `delete_module` syscall source code from Linux kernel v4.9. If a VMI client unlocks the `module_mutex` in line 3, two processes enter into critical section causing inconsistency. Also if it removes the module in line 8, line 9 might crash due to dangling `mod` variable.

rootkit keeps running in the kernel space without releasing the CPU, no race-free moment may be identified and Oxpecker transaction will be delayed indefinitely. The good news is that such a behavior blocks concurrent processes to use the same CPU and is not used by rootkits which need to be stealthy. Write operations are committed to the memory of GVM and so durability is also achieved. It requires no in-VM agent and operates in presence of untrusted kernel code since it just needs to know about the non-tampered guest data structures. Overhead of the Oxpecker is relatively low as it needs to intercept syscalls and the rest of kernel/user codes can run using hardware virtualization. The memory usage is also limited to the memory which is required by the out-of-VM client itself.

7 Conclusion

The manuscript presented an architecture for out-of-VM transactional access to the GVM kernel memory, namely the blue-pill Oxpecker. Oxpecker has a three layer architecture providing read/write API for VMI clients in the host machine through Coordinator component in the first layer, monitoring GVM state changes and examining its memory for possible race-conditions in the second layer, and reusing generic VMI APIs in the third layer for updating guest kernel memory after finding the proper race-free moment.

The Oxpecker architecture is implemented based on the Nitro framework and LibVMI in the third layer and evaluated under different workloads. As Oxpecker performance and accuracy are affected by the guest kernel workload, real-world network applications are used to generate a tunable workload. Experiments showed that the max overhead is about 75.84% which is much lower than the 2300% overhead obtained with a somewhat similar Exterior [7] framework which also lacks the transactional writing feature provided by the Oxpecker. The Oxpecker transactional read-/write API can be used to construct higher-level services such as out-of-VM rootkit removal, kernel hot patching, access restoration, and batch configuration updates.

References

[1] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[2] A. More and S. Tapaswi, "Dynamic malware detection and recording using virtual machine introspection," in *Best Practices Meet (BPM), 2013 DSCI*. IEEE, 2013, pp. 1–6.

[3] 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.

[4] B. D. Payne, D. D. A. Martim, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 385–397.

[5] 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.

[6] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 363–374.

[7] Y. Fu and Z. Lin, "Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery," *ACM SIGPLAN Notices*, vol. 48, no. 7, pp. 97–110, 2013.

[8] S. Suneja, R. Koller, C. Isci, E. de Lara, A. B. Hashemi, A. Bhattacharyya, and C. Amza, "Safe inspection of live virtual machines," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*, 2017, pp. 97–111.

[9] S. Vogl, F. Kiliç, C. A. Schneider, and C. Eckert, "X-TIER: kernel module injection," in *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, 2013, pp. 192–205.

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

[11] Y. Fu, J. Zeng, and Z. Lin, "Hypershell: a practical hypervisor layer guest os shell for automated in-vm management," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 85–96.

[12] W. Qiang, G. Xu, W. Dai, D. Zou, and H. Jin, "Cloudvmi: A cloud-oriented writable virtual machine introspection," *IEEE Access*, vol. 5, pp. 21 962–21 976, 2017.

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

[14] J. del Hoyo, A. Elliott, and D. Christie, "Family buphagidae (oxpeckers)," in *Handbook of the Birds of the World*. Lynx Edicions, 2009, pp. 642–653.

[15] J. Y.-C. Cheng, T.-S. Tsai, and C.-S. Yang, "An information retrieval approach for malware classification based on windows api calls," in *Machine Learning and Cybernetics (ICMLC), 2013 International Conference on*, vol. 4. IEEE, 2013, pp. 1678–1683.

[16] T. G. Paul and T. G. Kumar, "A framework for dynamic malware analysis based on behavior artifacts," in *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications*. Springer, 2017, pp. 551–559.

[17] P. M. Chen and B. D. Noble, "When virtual is better than real [operating system relocation to virtual machines]," in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*. IEEE, 2001, pp. 133–138.

[18] 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, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, 2014, pp. 605–620.

[19] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen, "Concurrent and consistent virtual machine introspection with hardware transactional memory," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, 2014, pp. 416–427.

[20] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993.

[21] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Blacksheep: detecting compromised hosts in homogeneous crowds,"

in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 341–352.

[22] Y. Fu and Z. Lin, "Space traveling across VM: automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, 2012, pp. 586–600.

[23] J. Hizver and T. Chiueh, "Real-time deep virtual machine introspection and its applications," in *10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, Salt Lake City, UT, USA, March 01 - 02, 2014*, 2014, pp. 3–14.

[24] 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, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, 2014, pp. 386–395.

[25] N. Rakotondravony and H. P. Reiser, "Visualizing and controlling vmi-based malware analysis in iaas cloud," in *35th IEEE Symposium on Reliable Distributed Systems, SRDS 2016, Budapest, Hungary, September 26-29, 2016*, 2016, pp. 211–212.

[26] M. Carbone, M. Conover, B. Montague, and W. Lee, "Secure and robust monitoring of virtual machines through guest-assisted introspection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2012, pp. 22–41.

[27] D. A. S. de Oliveira and S. F. Wu, "Protecting kernel code and data with a virtualization-aware collaborative operating system," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 451–460.

[28] A. Srivastava and J. Giffin, "Efficient protection of kernel data structures via object partitioning," in *Proceedings of the 28th annual computer security applications conference*. ACM, 2012, pp. 429–438.

[29] M. Carbone, M. Conover, B. Montague, and W. Lee, "Secure and robust monitoring of virtual machines through guest-assisted introspection," in *Research in Attacks, Intrusions, and Defenses - 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings*, 2012, pp. 22–41.

[30] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 545–554.

[31] X. Xiong, D. Tian, P. Liu *et al.*, "Practical protection of kernel integrity for commodity os from untrusted extensions." in *NDSS*, vol. 11, 2011.

[32] Y. Cheng, X. Fu, X. Du, B. Luo, and M. Guizani, "A lightweight live memory forensic approach based on hardware virtualization," *Inf. Sci.*, vol. 379, pp. 23–41, 2017.

[33] J. Dykstra and A. T. Sherman, "Acquiring forensic evidence from infrastructure-as-a-service cloud computing: Exploring and evaluating tools, trust, and techniques," *Digital Investigation*, vol. 9, pp. S90–S98, 2012.

[34] 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.

[35] A. L. Shaw, B. Bordbar, J. Saxon, K. Harrison, and C. I. Dalton, "Forensic virtual machines: dynamic defence in the cloud via introspection," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 303–310.

[36] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 279–290, 2011.

[37] C.-W. Wang, C.-W. Wang, and S. Shieh, "Probebuilder: Uncovering opaque kernel data structures for automatic probe construction," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, pp. 568–581, 2016.

[38] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. T. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, 2011, pp. 297–312.

[39] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, 2011, pp. 363–374.

[40] H. Xiong, Z. Liu, W. Xu, and S. Jiao, "Libvmi: A library for bridging the semantic gap between guest os and vmm," in *Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 549–556.

[41] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

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

[43] (2019, May) Consistent writable vmi api. [Online]. Available: https://github.com/Oxpecker-VMI/oxpecker

[44] 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.

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

[46] (2019, May) Oxpecker technical details. [Online]. Available: https://github.com/Oxpecker-VMI/oxpecker/blob/master/report/technical_

[47] J. Pfoh, C. Schneider, and C. Eckert, "Nitro: Hardware-based system call tracing for virtual machines," in *International Workshop on Security*. Springer, 2011, pp. 96–112.

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

[49] T. P. P. Council, "Tpc benchmark h standard specification version 2.18.0, 2018."

[50] "An open source load testing tool." [Online]. Available: https://locust.io/

[51] G. Rahn, "Tpc-h benchmark kit with some modifications/additions." [Online]. Available: https://github.com/gregrahn/tpch-kit

Seyed Mohammad AghamirMohammadAli received his B.E. in Computer Engineering and B.S. in Pure Mathematics from Sharif University of Technology, Tehran, Iran, in 2010. Later on, he joined the S4Lab at the Department of Computer Engineering, Sharif University of Technology, and conducted research in the area of virtual machine introspection. He obtained his M.S. degree in Information Technology in 2018 from Sharif University of Technology.

Behnam Momeni received his B.E. and M.Sc. degrees in computer engineering and information technology (with first rank) from Sharif University of Technology, Tehran, Iran, in 2010 and 2012, respectively. Thereafter, he conducted research on software deobfuscation in S4Lab, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran receiving his Ph.D. in 2018 (with first rank). His research interests include operating system, software analysis, software security, and vulnerability analysis.

Solmaz Salimi s a Ph.D. candidate in the field of Computer Engineering working at S4lab, Sharif University of Technology. Before that, she obtained her M.Sc. degree in computer engineering from Iran University of Science and Technology, Tehran, Iran, in 2015. Her research interests focus on software security, including software program analysis, vulnerability detection, analysis and exploitation techniques.

Mehdi Kharrazi received his B.E. in E.E. from the City College of New York, New York, in 1999, and M.Sc. and Ph.D. in E.E. from NYU Tandon, Brooklyn, NY, in 2002 and 2006. He is currently an Associate Professor with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. His current research interests include software, system, and network security and is the director of Safety and Security in Software and Systems Laboratory (S4Lab).