Simulating the Network Environment of Sandboxes to Hide Virtual Machine Introspection Pauses

Léo Cosseron, Louis Rilling, Matthieu Simonin, Martin Quinson

Simulating the Network Environment of Sandboxes to Hide Virtual Machine Introspection Pauses

Léo Cosseron
Univ. Rennes, Inria, CNRS, IRISA
Rennes, France
leo.cosseron@irisa.fr

Louis Rilling
DGA
Rennes, France
louis.rilling@irisa.fr

Matthieu Simonin
Inria
Rennes, France
matthieu.simonin@inria.fr

Martin Quinson
Univ. Rennes, Inria, CNRS, IRISA
Rennes, France
martin.quinson@irisa.fr

ABSTRACT

Virtual Machine Introspection (VMI) is used by sandbox-based dynamic malware detection and analysis frameworks to observe malware samples while staying isolated and stealthy. Sandbox detection and evasion techniques based on hypervisor introspection are becoming less of an issue since running server and workstation environments on hypervisors is becoming standard and high-end sandboxes manipulate virtual clocks to mask VM execution pauses caused by VMI. However, the fake network environment around a sandbox VM offers opportunities similar to hypervisor introspection for malware to evade. Malware can evaluate the discrepancy between observed performances and a real, presumed network environment of infected targets. VMI pauses also cause visible network performance glitches. To solve this issue we propose to extend virtual clock manipulation to synchronize hardware-accelerated virtual machines with a discrete-event network simulator. The experimental evaluation shows that our proposal can counter attempts to infer VMI activity from network timing observations.

CCS CONCEPTS

• **Networks → Network simulations**; • **Security and privacy → Virtualization and security**.

KEYWORDS

evasive malware, virtualization, network simulation, introspection

ACM Reference Format:
Léo Cosseron, Louis Rilling, Matthieu Simonin, and Martin Quinson. 2024. Simulating the Network Environment of Sandboxes to Hide Virtual Machine Introspection Pauses. In *The 17th European Workshop on Systems Security (EuroSec '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3642974.3652280

1 INTRODUCTION

Sandbox-based dynamic malware detection and analysis frameworks can run malware samples in a Virtual Machine (VM) and rely on Virtual Machine Introspection (VMI) [11] to run a monitor in the hypervisor, staying isolated and stealthy [17]. VMI-based monitors thus inspect the VM guest memory at arbitrary times and intercept events in the guest execution, like in-VM context switches and syscalls [23].

VMI is especially useful to dynamically analyze highly obfuscated advanced malware pieces, that are typically crafted by state-sponsored cyberattack groups. Like the infamous Stuxnet worm [16], such malware can target specific sensitive information systems which often are not directly connected to the Internet or even lie beyond air gaps. To stay stealthy such malware activates its malicious payload only after having checked that it is executing in the target information system, based on information previously collected by intelligence.

To effectively analyze such malware, sandboxes thus need to mimic target information systems. The task of creating such controlled and fake environments is similar to building honeynets [25] and cyber ranges [31], for which automation [5, 30] should make it tractable to configure a sandbox to mimic specific sensitive information systems.

The emulated fake environment around a sandbox is naturally closed if the mimicked information system is beyond an air gap. It is also preferable to keep such environments closed to prevent uncontrolled harm to other entities on the Internet and to prevent malware operators from knowing about the analysis [32].

However, *Hypervisor introspection* [20, 26, 28] leverages side-channels from the infected guest perspective (e.g. cache misses, execution delays) to detect whether it is being inspected. While running on top of a hypervisor is becoming standard for servers and workstations, execution pauses caused by VMI can still be easily detected. For this reason analysis sandboxes manipulate time sources to mask VMI-caused pauses [14].

In addition to VMI, the controlled and fake closed network environment emulated around the sandbox virtual machine can also provide side-channels analysis opportunities. Network-based timing measurements (e.g. observed round-trip-time between two endpoints on the network) can indeed reveal VMI activity, as envisioned by [28], but also show discrepancies with a presumed network topology of infected targets (e.g. a geographically-far public DNS server with a sub-millisecond round-trip time, homogeneous round-trip times with heterogeneous network links). As sandboxes keep on

EuroSec '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0542-7/24/04…$15.00
https://doi.org/10.1145/3642974.3652280

improving to stay stealthy from local-to-the-VM timing measurements, we believe that network-based measurements are practical enough to become an issue in a near future.

The main contribution of this paper is as follows: to address both the VMI-induced perturbation of network-based timings and the discrepancy of observable network topologies, we introduce TANSIV, which synchronizes the VMs with an accurate, discrete-event network simulator. Packets emitted by the VMs are received by their destination at the exact date computed by the simulator, with network-based timings unaffected by any potential VMI pause. TANSIV is open-source [7]. We implement the network synchronization along a VMI-hiding technique that makes hardware-accelerated VMs' virtual clocks oblivious to VMI pauses and other VM suspends caused by debugging and analysis tools such as GDB. To the best of our knowledge, this is the first approach to prevent hypervisor introspection techniques that use network-based timing measurements.

As a secondary contribution we discuss issues of VMI hiding in multi-core sandboxes and propose two strategies to make VMI less observable.

The remaining of this paper is organized as follows. Section 2 presents the threat model, how VMI pauses work and how they can be detected using clock sources. Section 3 presents our contributions to hide VMI pauses in networked sandboxes. Section 4 evaluates our approach on different scenarios. Section 5 discusses an extension of our approach on VMs with multiple vCPUs. Section 6 reviews related work. Finally, Section 7 closes this paper and discusses future work.

2 PROBLEM STATEMENT

Most VMI operations require pausing the introspected VM. These pauses can be detected by the VM, which has access to both local and remote time sources [10]. In Section 2.1, we introduce the threat model, then, in Section 2.2 we present how VMI pauses work in practice and why such pauses can be detected by introspected VMs using solely local-to-the-VM clock sources.

2.1 Threat model

First, we assume that attackers do not attempt to detect virtualization. We argue that virtualization is becoming more prevalent and that malware targeting only native systems would miss too many potential victims. Moreover, previous works have shown that it is impossible to build a fully transparent hypervisor [10]. In this work, we focus on hiding the timing perturbations resulting from using VMI, which comes on top of virtualization.

The attacker may have full access to the VM, including root and kernel rights. The attacker can use high resolution timers to ensure the consistency of the time flow, to detect any gaps or rollbacks that could be caused by a potentially introspecting hypervisor. The attacker has no prior knowledge that VMI in particular is going to be used on the VM.

The hypervisor is considered to be securely isolated from the attacker. We disregard timing-based side-channels allowing to steal information from VMs co-located on the same host.

The VM can communicate with other VMs over the network, and these VMs can be controlled by the attacker as well with the same hypotheses. However, we assume the network environment to be closed, with no access to uncontrolled networks or remote servers.

The attacker also knows the topology and the performance characteristics of the network in terms of latency and bandwidth.

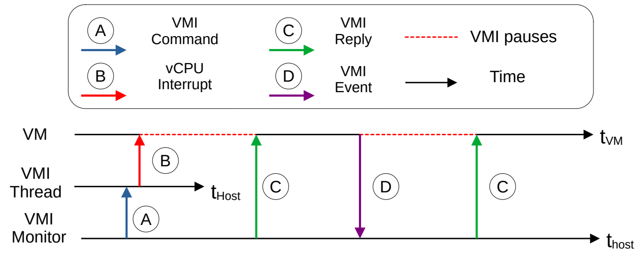

Figure 1: Types of VMI pauses with libVMI. VMI commands are issued by the VMI monitor to the VMI thread. The VMI monitor can add callback functions when a specific condition is met. These VMI events are triggered by the vCPU thread. The VMI pauses count towards the VM clock.

2.2 VMI pauses

In the general case a VMI request is issued by a program called a *VMI monitor*, typically using a dedicated VMI library, to the hypervisor where the target VM is running. Some VMI operations require to pause the target vCPUs, to ensure the VM state stays coherent. This is achieved by triggering a VM-Exit, i.e. a transition from the guest to the hypervisor with Intel hardware virtualization.

Figure 1 illustrates the different types of VMI pauses in libVMI [21], and how they affect the guest clock.

An attacker could infer VMI activity by observing these VMI pauses through timing measurements. Indeed, the guest clock is progressing even when a VM-Exit is handled by the hypervisor. VMI pauses differ from other VM-Exits because they are likely to be longer and/or occur periodically.

Note that this issue exists with any analysis tool that interrupts the VM, such as a debugger attached to the hypervisor.

3 HIDING VMI PAUSES IN NETWORKED SANDBOXES

To prevent introspected VMs from using local and remote clock sources to detect VMI pauses, we manipulate these clock sources from the hypervisor and ensure that timings stay coherent from the VM point of view.

Section 3.1 first presents time manipulation to hide VMI pauses from a guest that uses only local clock sources. Then Section 3.2 presents an extension to counter network-based timings.

3.1 Handling local clock sources

To hide the time gaps caused by introspection pauses, we skew the virtual clock of an introspected VM. This technique makes it difficult to exploit local-to-the-VM timing side-channel measurements to detect or evade VMI.

We first detail clock skewing in the case of single-vCPU VMs in the KVM hypervisor, before presenting an extension to VMs having multiple vCPUs.

3.1.1 Case of a single vCPU. In the following, we focus on Intel x86 CPUs and the KVM [15] hypervisor. The main clock source is the TimeStamp Counter (TSC), a register incremented at a constant rate in modern processors. The guest TSC is a linear function of the host TSC, for which the hypervisor can configure the slope and the offset in the vCPU Virtual Machine Control Structure (VMCS), a data structure used by the processor to control each vCPU.

To hide VMI pauses, we measure the pause duration by taking a timestamp in the `VM-Exit` handling code of the hypervisor. Then just before resuming the vCPU, if the `VM-Exit` was related to VMI, we subtract the pause duration from the guest TSC offset. However this adjustment misses the delay between the TSC adjustment and the `vmresume` instruction: the pre-`vmresume` delay.

We thus propose to account for the pre-`vmresume` delay by subtracting a dynamic estimate of this delay from the TSC offset. The estimate is dynamically adjusted by using an approach inspired by KVM's dynamic adjustment algorithm used for the local APIC timer emulation [2].

We extend this approach to QEMU's `GDB` server [4]. The QEMU process notifies KVM to hide a `VM-Exit` corresponding to a `GDB` interrupt by using an extension of the `kvm_run` struct, that is shared between QEMU and KVM with an `ioctl`.

3.1.2 Case of multiple vCPUs. On VMs with multiple vCPUs, adjusting the TSC requires additional care. Each vCPU has its own TSC register. The TSCs are synced during the boot of the system in order to be usable as a global clock. To avoid observable unsynchronized TSC clocks between vCPUs, the guest TSC of each vCPU must be adjusted by the same value and simultaneously. Moreover, to avoid observable rollbacks, this requires all vCPUs to be paused beforehand.

Thus the clock correction approach can be used for VMI requests interrupting all vCPUs at once, such as `libVMI` function `vmi_pause_vm`, or a VM interrupt by `GDB`.

A simple approach could consist in unconditionally pausing all vCPUs on any VMI request or event. However this breaks VMI scripts expecting that at least one vCPU is still running on the system, or scripts waiting for events to trigger on multiple vCPUs before completing a previous event. Thus the straw man approach is to just not hide VMI pauses when only subsets of vCPUs are paused. Correcting the clock for requests or events that only target a subset of vCPUs remains a challenge, which is discussed in Section 5.

3.2 Network interactions

The technique discussed in Section 3.1 does not take into account network-based time sources. The round-trip time of a packet (e.g., a ping) can indeed be impacted if VMI pauses occur on the sender or receiver side during the exchange. Furthermore, our time manipulation technique would desynchronize the clocks of the different VMs on the network, which can be easily noticed. This section shows how, using time manipulation, we interconnect the introspected VMs with a discrete-event network simulator.

We present in the following how the simulator events can be used to synchronize the VMs clocks and how the integration with VMI is performed. In this section we assume that VMI pauses always pause all vCPUs of the introspected VM. Strategies to support VMI pauses of subsets of vCPUs are discussed in Section 5.

3.2.1 Synchronization algorithm. To ensure the consistency of network timings from the introspected VMs perspective, we introduce TANSIV. TANSIV implements the whole network with a discrete-event network simulator and synchronizes the VMs' clocks with the clock of the simulator. Discrete-event simulators run the simulation in an abstract time and make their clock progress forward by jumping to the time of the next occurring event.

TANSIV's synchronization algorithm works by dividing the time into slots of variable duration and bounded by *deadlines*. The VMs run concurrently during each time slot. The packets sent by the VMs are intercepted and appended in a queue with their timestamps. Each VM is paused when it reaches a deadline. Once all VMs are paused, we drain the packet queue in chronological order, advance the simulator clock up to the send timestamp of each packet and inject the packets into the simulation. We then advance the simulator clock to the deadline. If the simulator reports that a packet should be delivered at the deadline, we inject it in the destination VM.

The following two properties allow the accurate timing of packets delivery:

(1) each packet is delivered at the very beginning of a time slot,
(2) no packet sent in a time slot should be theoretically delivered before it is injected into the simulator.

To achieve property 2, the duration of time slots is capped to Λ, the lowest network latency between any two VMs in the network simulation. To achieve property 1, at each deadline δ the next deadline is set to the minimum of $\delta + \Lambda$ and the earliest delivery date of packets currently in the simulator.

To ensure the synchronization of all the clocks, all the vCPUs of a VM are interrupted when it reaches a deadline. Moreover, the time spent during the synchronization with the network simulator must not be taken into account in the guest clock, as detailed below. Figure 2 illustrates this algorithm in a simple scenario with the sending of a single packet, P_1. The sending date $s(P_1)$ is used by the simulator to compute the receiving date $r(P_1)$ at which a deadline is scheduled and the packet delivered to its destination VM.

3.2.2 Integration with VMI. The network synchronization algorithm integrates well with VMI pauses hiding as long as VMI pauses always pause all vCPUs of an introspected VM (see Section 5 for VMI pauses affecting only subsets of vCPUs). Indeed, the network synchronization algorithm just adds another type of VM pause.

To schedule a deadline, we use the VMX Preemption Timer, an Intel-specific timer that takes the form of a register, in which we can load a duration in TSC ticks. When the corresponding vCPU is running, this register counts down at a rate proportional to the host TSC, and triggers a `VM-Exit` when it reaches `0`.

Intel mentions no delay or precision issues with the VMX Preemption Timer [13], although it is known to be broken on some processors [1].

Similarly to VMI pauses, deadlines pauses must be hidden by adjusting the VMs' virtual clocks. The process and implementation

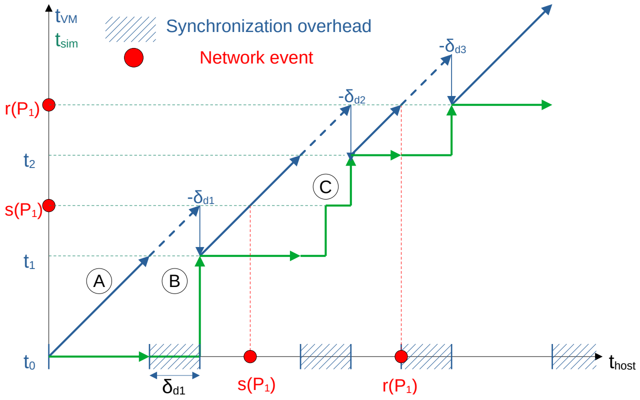

Figure 2: Comparison of the clocks' progressions. The X axis denotes the host time while the Y axis denotes the VM clock (for the blue curve) and the simulator clock (for the green curve). The goal is to keep VM and simulator clocks synchronized. During a time slot such as Ⓐ, the virtual and host clock progress at the same speed, while the simulator clock does not progress. The deadline occurs when Ⓑ starts. Since the virtual clock is not frozen during *B* while we prepare the next time slot, we hide the deadline handling duration δ_{d1} by subtracting δ_{d1} from the virtual clock before resuming the VM. As shown on Ⓒ, the simulator clock progresses by steps, first to the date of a network event, and then to the date of the deadline being handled.

are almost the same as for the hiding of VMI pauses introduced in Section 3.1. The only difference is that we roll back the guest clocks to the deadline date instead of the approximate VM-Exit timestamp to ensure that all VMs are synchronized at each deadline.

A special case to consider is when a VMX Preemption Timer VM-Exit is triggered late because another VM-Exit is being processed at the time of the scheduled deadline.

In this case, just before the end of the VM-Exit processing, the VMX Preemption Timer is loaded with the value 0, which causes an immediate VM-Exit, before executing a single instruction. Then, after the synchronization and the clock correction, we put the vCPU thread to sleep until the end date of the VM-Exit that finished late, before resuming execution. This guarantees that the vCPU cannot see some events that usually cause VM-Exits (notably I/O events) last much less than reality.

TANSIV does not rely on any mechanism specific to KVM, so it should be possible to retrofit it in any hypervisor, such as Xen [3]. On the hardware side, there are no timers equivalent to the VMX Preemption Timer on AMD64 processors with SVM, or on ARMv8, although it is possible to use other timers instead, but at the cost of increased complexity and degraded accuracy.

4 EVALUATION

In this section we evaluate the effectiveness of our synchronization algorithm for hiding VMI pauses from network-based timing measurements. This experiment illustrates how an evasive malware might attempt to detect the presence of VMI by measuring network timings.

We also evaluated the effectiveness of our approach to hide VMI pauses from local timing measurements. We used an approach similar to [28], by measuring the duration of VM pauses when VMI is active. Our results show that large time gaps introduced by VMI are hidden by our virtual clock manipulation, but we do not present them here due to space limitations.

The experiment was performed on a machine with an Intel Core i7-1165G7 CPU and 32GB of RAM. This CPU has a TSC frequency of 2.8032GHz. Each VM is configured to use 8GB of RAM, 1 vCPU that is pinned to a physical core, and a Debian 11 guest OS. We disabled DVFS to fairly compare the execution of different VMI scripts.

We used the Linux/KVM 5.15 hypervisor, libVMI with the kvmi-v12 driver as the VMI library, and SimGrid 3.32 [6] as the network simulator.

Our simulated network topology is constituted of 2 VMs, connected with a 1Gbps link with 2ms of latency. Both VMs are running on the same host. When using TANSIV's synchronization algorithm, the VMs are connected using the SimGrid network simulator. Without TANSIV, they are connected with libvirt NAT forwarding, using NetEm [12] to perform the network emulation of the link. Note that NetEm is configured from outside the VMs. We used the exact same latency parameter for NetEm and SimGrid latency. Thus, the RTT in the NetEm setup is slightly higher, as we do not take into account the delay of the host network stack, which adds a few dozen of microseconds.

We executed the "ping" command 1000 times from one guest to the other. By comparing the Round-Trip Time (RTT) of packets to the expected result (Slightly over 4ms in average with our topology and usual packet processing delays), we can infer potential VMI activity.

For both setups, we perform the experiment three times: (i) without VMI; (ii) with the "vmi-process-list" VMI script, without time manipulation; (iii) with the "vmi-process-list" VMI script, with time manipulation.

The VMI script is based on the "vmi-process-list" example script of libVMI. This script triggers a VMI pause, and lists the processes currently running in the monitored VM. We have chosen to perform a process listing every millisecond to ensure that the VMI script has an impact on all the pings. This heavy load represents a worst-case scenario where VMI has a major impact on network timings. In the second and third experiment, the VM issuing the pings is introspected during the whole duration of the experiment, while the other VM is not introspected at all.

Results are shown on Figure 3. We opted for a boxen plot representation to highlight the outliers. The main level contains the median and 50% of the values. Then each successive level contains half of the remaining points, until we reach the outliers. With time manipulation, in the NetEm experiment, the median RTT is about 0.1ms, much lower than the emulated link RTT latency. NetEm works by adding a FIFO queue in between the network protocols and the network card. The packets are held there and are forwarded to the network card only according to NetEm current configuration. With time manipulation enabled, the packets stay in the FIFO queue during 2ms of host time. Then the packets travel on the network,

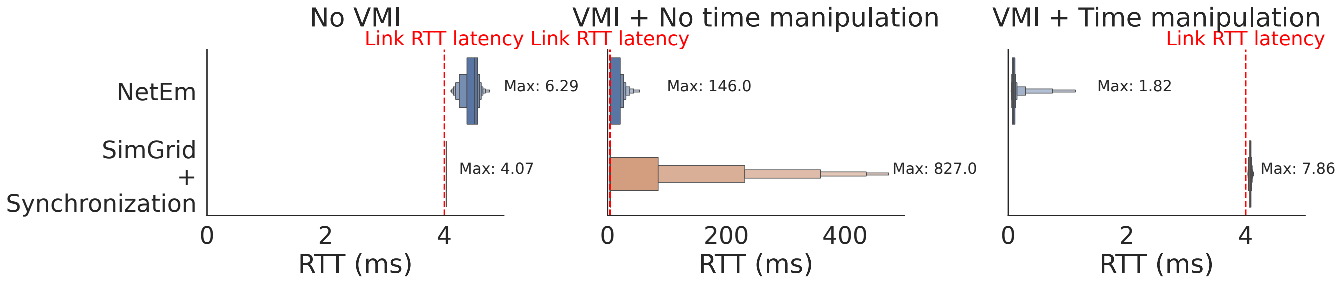

Figure 3: Boxen plot representation of ping RTT on three scenarios. On the left, without VMI, both distributions are close to the theoretical link RTT latency. In the middle, with VMI but without any time manipulation, the RTT of the pings varies greatly, due to VMI pauses delaying the processing of network packets. On the right, with time manipulation, the VMI pause impact is corrected by the combination of VMI pause hiding and network synchronization, whereas it is not sufficient on the `NetEm` approach as the VMs are not in sync anymore.

the ping replies stay in the second `NetEm` queue and come back. While the RTT lasts slightly above 4ms of host time, hidden VMI pauses make it last much less in introspected VM time.

When using time manipulation and TANSIV with the SimGrid network simulator, the simulator's and both VMs' clocks stay synchronized, resulting in consistent network timings, with in particular no value below the link RTT latency, as shown on the plot. While the maximum observed value is high at 7.86ms, it is an outlier as the 99th percentile is 4.13ms.

5 STRATEGIES FOR MULTI-VCPUS SANDBOXES

In this section, we discuss two strategies to re-synchronize the different TSC clocks, with the goal of performing better than the straw man approach, that is minimizing the opportunities for an attacker to observe unsynchronized clocks.

The two strategies share the principle of always locally hiding the VMI pause from the point of view of the vCPU resuming from the pause and differ in how a vCPU resuming from a pause catches up with the TSC clock of other vCPUs. A VMI pause is locally hidden using the algorithm presented for the single-core case in Section 3.1.

The two strategies also assume that VMI pauses concerning only subsets of vCPUs are likely to be short and last a few milliseconds at most before either ending the pause or having all vCPUs paused by VMI. Once all vCPUs are paused by VMI we can reuse the approach for VMI pauses targeting all the vCPUs.

5.1 Pause vCPUs having higher clock values

This first strategy consists in letting vCPUs having just ended a VMI pause catch up by pausing the vCPUs having higher clock values. To this end, the global re-synchronization of the TSC clocks is delayed until no vCPU is in a VMI pause.

To globally re-synchronize the TSC clocks, we pause all vCPUs but the ones having the lowest values of TSC clock and do so iteratively until all vCPUs catch up with the vCPUs having the highest value of TSC clock. To synchronize the clocks at catch up times, the catching up vCPUs are paused again to set the same TSC offset on all of them and the more advanced vCPUs they have just caught up with.

If a VMI pause is triggered during the re-synchronization process, the re-synchronization is stopped and we resume the paused vCPUs.

This approach is compatible with our synchronization algorithm of Section 3.2. However, there is a risk of deadlock if a VMI pause cannot be completed until a VMI event is triggered on a vCPU paused by a deadline. To avoid this, we can perform the synchronization with only the subset of vCPUs that reached the deadline. This can be repeated for the following deadlines if necessary. When a VMI-paused vCPU resumes and sends a packet, we postpone its timestamp to the beginning of the current time slot (if it was taken before), to ensure the consistency of our network synchronization algorithm.

This approach can be detected because, during VMI pauses, some vCPUs might run concurrently with unsynchronized clocks and, during global resychronizations, catching up vCPUs might observe side effects of vCPUs having already run with higher clock values. The integration with our network synchronization algorithm can result in some packets being delivered too late due to timestamp adjustments. Finally, it may not be suitable for aggressive VMI scripts where just one vCPU is concerned by a VMI pause most of the time.

5.2 Accelerate TSC clocks of late vCPUs

The second strategy takes advantage of the `TSC multiplier` VMCS field. By default, the guest TSC clocks run at the same speed as the host TSC clock. However, by using the `TSC multiplier` we can accelerate or slow down the guest TSC.

In this strategy, the global re-synchronization of TSC clocks is triggered unconditionally when a vCPU resumes from a VMI pause. The idea is to accelerate the vCPU clock by modifying the `TSC multiplier`, until it catches up with the most advanced TSC clock.

We can use the same strategy as the first approach to avoid deadlocks when integrating this approach with the network simulator synchronization algorithm.

This more aggressive strategy may work better with VMI scripts for which the pausing strategy might not be adapted, such as VMI scripts with long pauses or pausing the same few vCPUs. However, the change in clock speed can be detected and might impact some applications.

6 RELATED WORK

In [27] the authors propose to reduce the resolution of the TSC timer on Xen VMs to mitigate timing side-channels, while ensuring the system still behaves normally. They can degrade the resolution of the TSC up to 2μs. [19] presents adjustments to the TSC and other time sources to mitigate timing side-channels, by adding randomized delays. However, in both approaches the timer resolution is too high to prevent the detection of VMI pauses. Disabling the TSC entirely is not an option either, because this breaks multiple applications [19].

StopWatch [18] tackles the issue of timing side-channels attacks by co-resident VMs. The authors replace the real-time clock with a virtual clock that only counts the number of instructions executed in the guest. Although this can prevent VMI pauses from being detected by the VM, it provides unrealistic timings, because VM-Exit events unrelated to VMI (e.g. for I/O) take time inherent to virtualization, and should count towards this virtual clock. The authors of Ether [9], a malware analysis system, claim to adjust time queries from the guest by subtracting the overhead of the malware analysis framework. However, it has been shown in [22] that Ether actually configured the hypervisor to trigger a VM-Exit on every `rdtsc` instruction, and return a counter value incremented by 1 on each call. In the same work, the authors show how this can be exploited to detect the presence of Ether. Finally, the HyperDbg debugger [14] includes two approaches to hide its timing overhead. The first one modifies the guest's TSC MSR, similarly to our approach. The second one emulates the result of `rdtsc` instructions, which requires a VM-Exit, but it can be limited on a specific process rather than impacting the whole system.

[26, 28] focus on how a VM can detect a hypervisor performing introspection, and offer countermeasures. [26] authors propose to use time sources that are hard to tamper with, such as the HPET timer. Although in this paper we only address the TSC, other virtualized local clocks could be addressed similarly. [28] briefly mentions a virtual clock approach that is similar to ours, but does not address networking.

Finally many works have studied the pairing of a network simulator with VMs, but none in the context of VMI to the best of our knowledge. SliceTime [29] synchronizes Xen VMs with the ns-3 [24] network simulator to accurately evaluate the performance of distributed applications. They cut the time into slices, and perform the synchronization at each pause. A major limitation of SliceTime is that the VMs in the simulation have to run sequentially, whereas they can run in parallel with our approach. Finally our synchronization algorithm uses time slices of dynamic duration in order to improve accuracy, as proposed in [8], where they pair QEMU VMs with a SystemC emulator. However unlike our approach their synchronization algorithm does not take into consideration the network topology and does not address the case of hardware virtualization.

7 CONCLUSION AND FUTURE WORK

In this paper we showed how, by synchronizing the VMs with a network simulator, we can solve both the VMI-induced perturbation of network-based timings and the discrepancy of observable network topologies. To this end, we introduced TANSIV, in which this synchronization is implemented along a time manipulation technique to make the virtual clocks of hardware-accelerated VMs oblivious to VM suspends caused by VMI and debuggers. To the best of our knowledge, this is the first approach to prevent hypervisor introspection techniques that use network-based timing measurements. We have also discussed and proposed two strategies to partially hide VMI pauses in multi-core sandboxes.

In our evaluation, we have shown that network timings are significantly more realistic for introspected VMs. While we used a simple network topology, we support simulating more complex networks, within the limits of the simulator.

To extend our work to VMs with multiple vCPUs, we plan to evaluate the effectiveness of each strategy introduced in Section 5 on different VMI workloads. We are considering the use of custom VMI scripts to highlight specific cases, as well as existing scripts that are more representative of a classic VMI setup, such as those created for the DRAKVUF sandbox [17].

In addition to pauses caused by VMI, I/O requests trigger VM-Exits that can last for a duration noticeably different from the emulated hardware I/O operation. We consider studying performance models for I/O devices, and adjusting the virtual clock of the VM similarly to the technique presented in this paper for the VMI pauses.

In this paper, we focused on closed network environments, which is not suitable for VMI-based intrusion detection and restricts analysis to malware samples interacting with known network entities. As future work, we can introduce access to remote hosts on the Internet, while ensuring the timings on the local network are coherent. As we have full control over the local network thanks to the simulator, we can adjust the timings of outgoing and incoming packets to conserve consistent timings from the sandbox perspective.

REFERENCES

[1] 2015. *Intel Xeon Processor 5500 Series Specification Update (Errata AAK139)*. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-5500-specification-update.pdf

[2] 2018. *KVM: LAPIC: Tune lapic_timer_advance_ns automatically.* https://patchwork.kernel.org/project/kvm/patch/1538115136-20092-1-git-send-email-wanpengli@tencent.com/

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003* (Bolton Landing, NY, USA, October 19-22, 2003), Michael L. Scott and Larry L. Peterson (Eds.). ACM, 164–177. https://doi.org/10.1145/945445.945462

[4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference* (Anaheim, CA, USA). USENIX, 41–46.

[5] Pierre-Victor Besson, Valérie Viet Triem Tong, Gilles Guette, Guillaume Piolle, and Erwann Abgrall. 2023. URSID: Automatically Refining a Single Attack Scenario into Multiple Cyber Range Architectures. In *FPS 2023 - 16th International Symposium on Foundations & Practice of Security*. Bordeaux, France, 1–16. https://inria.hal.science/hal-04317073

[6] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. 2014. Versatile, scalable, and accurate simulation of distributed applications and platforms. *J. Parallel Distributed Comput.* 74, 10 (2014), 2899–2917. https://doi.org/10.1016/j.jpdc.2014.06.008

[7] TANSIV contributors. 2018. TANSIV. https://gitlab.inria.fr/tansiv/tansiv.

[8] Massimiliano d'Angelo, Alberto Ferrari, Ommund Ogaard, Claudio Pinello, and Alessandro Ulisse. 2012. A Simulator based on QEMU and SystemC for Robustness Testing of a Networked Linux-based Fire Detection and Alarm System. In *Embedded Real Time Software and Systems (ERTS2012)*. Toulouse, France. https://hal.science/hal-02192275

[9] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008* (Alexandria, Virginia, USA), Peng Ning, Paul F. Syverson, and Somesh Jha (Eds.). ACM, 51–62. https://doi.org/10.1145/1455770.1455779

[10] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility Is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of HotOS'07: 11th Workshop on Hot Topics in Operating Systems* (San Diego, California, USA), Galen C. Hunt (Ed.). USENIX Association.

[11] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003* (San Diego, California, USA). The Internet Society.

[12] Stephen Hemminger. 2005. Network emulation with NetEm. In *Linux conf au*.

[13] Intel. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[14] Mohammad Sina Karvandi, MohammadHosein Gholamrezaei, Saleh Khalaj Monfared, Soroush Meghdadi Zanjani, Behrooz Abbassi, Ali Amini, Reza Mortazavi, Saeid Gorgin, Dara Rahmati, and Michael Schwarz. 2022. HyperDbg: Reinventing Hardware-Assisted Debugging. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 1709–1723. https://doi.org/10.1145/3548606.3560649

[15] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. Ottawa, Ontorio, Canada, 225–230.

[16] David Kushner. 2013. The real story of Stuxnet. *IEEE Spectrum* 50, 3 (2013), 48–53. https://doi.org/10.1109/MSPEC.2013.6471059

[17] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. 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), Charles N. Payne Jr., Adam Hahn, Kevin R. B. Butler, and Micah Sherr (Eds.). ACM, 386–395. https://doi.org/10.1145/2664243.2664252

[18] Peng Li, Debin Gao, and Michael K. Reiter. 2013. Mitigating access-driven timing channels in clouds using StopWatch. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (Budapest, Hungary). IEEE Computer Society, 1–12. https://doi.org/10.1109/DSN.2013.6575299

[19] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th International Symposium on Computer Architecture (ISCA 2012)* (Portland, OR, USA). IEEE Computer Society, 118–129. https://doi.org/10.1109/ISCA.2012.6237011

[20] Preeti Mishra, Emmanuel S. Pilli, Vijay Varadharajan, and Udaya Kiran Tupakula. 2017. Intrusion detection techniques in cloud environment: A survey. *J. Netw. Comput. Appl.* 77 (2017), 18–47. https://doi.org/10.1016/J.JNCA.2016.10.015

[21] Bryan D Payne. 2012. *Simplifying virtual machine introspection using LibVMI.* Technical Report. Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA.

[22] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. 2011. nEther: in-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security, EUROSEC'11, April 10, 2011, Salzburg, Austria*, Engin Kirda and Steven Hand (Eds.). ACM, 3. https://doi.org/10.1145/1972551.1972554

[23] Jonas Pfoh, Christian A. Schneider, and Claudia Eckert. 2011. Nitro: Hardware-Based System Call Tracing for Virtual Machines. In *Advances in Information and Computer Security - 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7038)*, Tetsu Iwata and Masakatsu Nishigaki (Eds.). Springer, 96–112. https://doi.org/10.1007/978-3-642-25141-2_7

[24] George F. Riley and Thomas R. Henderson. 2010. The *ns-3* Network Simulator. In *Modeling and Tools for Network Simulation*, Klaus Wehrle, Mesut Günes, and James Gross (Eds.). Springer, 15–34. https://doi.org/10.1007/978-3-642-12331-3_2

[25] L. Spitzner. 2003. The Honeynet Project: trapping the hackers. *IEEE Security & Privacy* 1, 2 (2003), 15–23. https://doi.org/10.1109/MSECP.2003.1193207

[26] Tomasz Tuzel, Mark P. Bridgman, Joshua Zepf, Tamas K. Lengyel, and Kyle J. Temkin. 2018. Who watches the watcher? Detecting hypervisor introspection from unprivileged guests. *Digit. Investig.* 26 Supplement (2018), S98–S106. https://doi.org/10.1016/j.diin.2018.04.015

[27] Bhanu Chandra Vattikonda, Sambit Das, and Hovav Shacham. 2011. Eliminating fine grained timers in Xen. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011* (Chicago, IL, USA), Christian Cachin and Thomas Ristenpart (Eds.). ACM, 41–46. https://doi.org/10.1145/2046660.2046671

[28] Gary Wang, Zachary John Estrada, Cuong Manh Pham, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2015. Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring. In *9th USENIX Workshop on Offensive Technologies, WOOT '15* (Washington, DC, USA), Aurélien Francillon and Thomas Ptacek (Eds.). USENIX Association.

[29] Elias Weingärtner, Florian Schmidt, Hendrik vom Lehn, Tobias Heer, and Klaus Wehrle. 2011. SliceTime: A Platform for Scalable and Accurate Network Emulation. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011* (Boston, MA, USA), David G. Andersen and Sylvia Ratnasamy (Eds.). USENIX Association.

[30] Muhammad Mudassar Yamin and Basel Katt. 2022. Modeling and executing cyber security exercise scenarios in cyber ranges. *Computers & Security* 116 (2022), 102635. https://doi.org/10.1016/j.cose.2022.102635

[31] Muhammad Mudassar Yamin, Basel Katt, and Vasileios Gkioulos. 2020. Cyber ranges and security testbeds: Scenarios, functions, tools and architecture. *Computers & Security* 88 (2020), 101636. https://doi.org/10.1016/j.cose.2019.101636

[32] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M. Blough, Elissa M. Redmiles, and Mustaque Ahamad. 2021. An Inside Look into the Practice of Malware Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 3053–3069. https://doi.org/10.1145/3460120.3484759