Mitigation of the impact of Virtual Machine Introspection Pauses on Multi-core Virtual Machines

Léo Cosseron[1][0009−0005−4317−9703], Louis Rilling[2][0000−0003−4520−6646], and Martin Quinson[1][0000−0001−7408−054X]

[1] Univ. Rennes, Inria, CNRS, IRISA
{leo.cosseron,louis.rilling,martin.quinson}@irisa.fr
[2] DGA

Abstract. Virtual machine introspection (VMI) is a class of monitoring techniques used by malware analysis sandboxes to analyze the behavior of malware samples. VMI introduces execution pauses that can be detected within the virtualized environment, revealing the usage of a sandbox. Thus, many sandboxes implement countermeasures to conceal VMI pauses from the guest However, these countermeasures have limitations on multi-core guests.
We propose several metrics to characterize the side effects caused by a VMI monitor in multi-core guests. We introduce a new strategy to conceal VMI pauses in multi-core guests, and assert its performance compared to the regular approaches used by sandboxes. Our results show that this new strategy improves the stealthiness of VMI pauses on a multi-core system for some metrics, but it still has shortcomings that should be addressed in future work.

Keywords: Virtualization · Virtual machine introspection · Hypervisor introspection

1 Introduction

Malware analysis sandboxes can leverage CPU support for machine virtualization to stay stealth and securely isolated from malware samples [12]. However, whereas sandboxes use Virtual Machine Introspection (VMI) [8] to analyze malware behavior, *Hypervisor introspection* [16, 23, 25] allows malware to leverage side-channels from the virtual machine (VM) perspective (e.g. cache misses, execution delays) to detect whether it is being inspected.

Typical examples of execution delays that can thus be observed from inside the VM are the execution pauses caused by VMI procedures. Such pauses last long enough to be attributed to VMI activity rather than to usual execution on top of a hypervisor [25].

To prevent evasive malware from observing VMI pauses, some sandboxes leverage CPU support for clock virtualization to adjust the virtual clock of the analysed VM and pretend that the clock is stopped during the whole VMI

pause [11]. It is even possible to synchronize the clock of a fake network environment with the clock of the analysis VM to hide VMI pauses while showing realistic network performance metrics [6].

Such clock manipulation techniques have to assume that either time-based evasion techniques do not involve network communications or that the sandbox runs in a closed environment in which the whole network is controlled [11, 19]. These assumptions are different but necessary trade-offs that are adapted to different analysis constraints [26].

The stealthiness of clock manipulation is significantly more limited in the case of multi-core VMs [6, 13]. A first solution to hide such VMI pauses consists in independent modifications of the virtual core-local clocks to conceal the pauses. This unfortunately induces virtual core clocks desynchronizations that can be detected by malware. Alternatively, the VMI pauses can be concealed by pausing some virtual cores while the virtual clocks remain unchanged, but malware can then detect that paused virtual cores execute fewer instructions than the other ones.

In this paper, we present three contributions to hide VMI pauses in multi-core analysis VMs:

- We introduce time-related observable phenomena and example algorithms that allow malware to detect VMI in multi-core VMs.
- We propose a strategy for an analysis sandbox to reduce the ability of malware to detect VMI in a multi-core VM. Our implementation on Xen is open-source [2].
- We evaluate the proposed strategy against the observable phenomena introduced and with the VMI plugins of the DRAKVUF analysis sandbox [12]. This evaluation shows the limitations of stealth analysis with the hardware support in modern CPUs and allows us to suggest research directions to improve this support.

This paper is organized as follows. In Section 2 we present the threat model and technical background about VMI. In Section 3 we detail the issue of hiding VMI pauses in multi-core VMs and introduce the observable phenomena allowing malware to detect VMI. In Section 4 we present a strategy to reduce the observability of VMI on multi-core VMs. An evaluation of this strategy in the DRAKVUF sandbox is presented in Section 5. Related work is presented in Section 6 and Section 7 concludes the paper.

2 Context

Using VMI triggers execution pauses that can be detected by an attacker thanks to *hypervisor introspection*. Section 2.1 presents the threat model while Section 2.2 details the operation of VMI pauses in `libVMI`, including in multi-core VMs settings.

2.1 Threat model

In this work, we focus solely on detecting the use of VMI-related tools by the Virtual Machine Monitor (VMM or hypervisor). Our threat model assumes that the attacker has root and kernel rights in the VM, and has access to high-resolution timers. However, the attacker has neither access to the host nor to the VMM, which are assumed to be isolated from the VMs.

We consider the following techniques to be out of scope: the detection of virtualization, access to co-resident VMs, and access to the host or the VMM.

Moreover, the attacker's main objective is to remain undetected during the attack. The attacker will thus prefer improved stealthiness over increased introspection detection accuracy. This may restrict the hypervisor introspection techniques available to the attacker despite their root and kernel privileges in the VMs.

2.2 VMI pauses

We now present the general behavior of a VMI program or *monitor*, including how it induces execution pauses. Our focus is on `libVMI` [1, 18], which is one of the most popular open-source VMI library. A VMI monitor uses `libVMI` to access information about the execution of a monitored VM. `libVMI` interacts with the VMM to retrieve the relevant information.

`libVMI` enables two types of VMI requests that can eventually trigger execution pauses. *VMI commands* are issued immediately to the target VM or virtual CPU (vCPU) by the VMM. Examples include operations such as pausing a VM or accessing its memory. *VMI events* are callbacks that are triggered by specific actions of the VMs. When such an event occurs, the VMM is configured to trigger a transition from the guest to the VMM (VM Exit on Intel CPUs). If required, the vCPU is paused while handling the event. VMI events can be used to monitor a memory area or the usage of instructions such as `MOV to CR3`. A VMI event callback can affect future VMI events, because it is possible to configure new or existing VMI events from a callback.

In addition, `libVMI` is compatible with multi-core VMs. VMI events only occur on individual vCPUs but can occur concurrently on multiple vCPUs. In `libVMI`, events are processed through a VM-specific ring structure in FIFO order. Additionally, some VMI commands can affect multiple (usually all) vCPUs, such as `vmi_pause_vm`.

3 Usage of VMI on multi-core VMs

Similarly to single-core VMs, VMI pauses can also be detected in multi-core VMs, although new metrics should be used. Section 3.1 discusses in-VM observable phenomena resulting from VMI usage on a multi-core VM. To our knowledge, previous works having studied strategies to conceal the impact of VMI pauses do not address multi-core VMs [6,13]. These limitations are discussed in Section 3.2 and experimental insight is provided in Section 3.3.

3.1 Detecting local VMI pauses in a multi-core VM

Many hypervisor introspection strategies depend on an accurate clock source for detecting inconsistencies induced by VMI pauses. On Intel x86 CPUs, the most accurate clock source is the TSC (Time Stamp Counter), which is a 64-bit counter register that continuously increments at a constant, high frequency. In VMs, the guest TSC is an affine function of the host TSC. It is computed from the host TSC and two parameters (`TSC-offset` and `TSC-multiplier`). The used formula according to Intel [10] is ($\gg$ is the bitwise right shift operator):

$$TSC_{guest} = (\texttt{TSC-multiplier} \times TSC_{host}) \gg 48 + \texttt{TSC-offset}$$

These two parameters are set in the Virtual Machine Control Structure (VMCS). On standard VMMs such as Xen and KVM, each vCPU has a dedicated VMCS, enabling independent values of these parameters for each vCPU.

As for single-core VMs, VMI execution pauses can be detected from the vCPU that was paused by observing gaps in the clock flow of the paused vCPU. The frequency and duration of these gaps are key metrics to infer the usage of a VMI tool [25]. In multi-core VMs, VMI pauses may occur on all or only a subset of vCPUs, making a VMI pattern more difficult to identify using the approaches from single-core VMs.

From in-VM observable phenomena, we propose two new time-related metrics to detect the presence of VMI on multi-core VMs. Indeed, VMI usage has two low-level time-related effects. First, a vCPU in a VMI pause cannot produce in-VM side effects based on clocks, like writing timestamps in a shared memory. Second, a paused vCPU cannot produce side effects based on work done, like writing results of computations in a shared memory.

For clock-based side effects, we propose a *timing metric* based on the key property of the guest TSC that it is synchronized across all the vCPUs. A simple algorithm measures the TSC simultaneously on each vCPU, and stores the results in a shared array. By repeating this process many times and comparing the difference between the highest and lowest TSC measurements from each iteration, it is possible to infer the usage of VMI. On a system without VMI, these differences will usually be very small, with occasional outliers due to a VM Exit or a NMI (Non-Maskable Interrupt). A VM Exit happens when the VMM traps the guest to perform a privileged operation, while a NMI is issued by the hardware and cannot be ignored. To the opposite, the number of outliers likely increases significantly when VMI is used. Indeed, VMI pauses are often longer than regular VM Exits, as they they are typically handled in userspace while most VM Exits are handled directly in the VMM.

For work-done-based side effects, we propose a *performance metric* that measures the loss of performance caused by VMI pauses. During a VMI pause, the vCPU thread is not scheduled, but its clock continues to tick. We measure the quantity of work that each vCPU can achieve during a given amount of time, and compare the results. This can be done by incrementing a counter within a dedicated thread for each vCPU. On a system without VMI, we can expect to observe similar results for each vCPU since each thread is scheduled for roughly

the same amount of time. However, if a VMI monitor is active, the paused vCPUs likely achieve less work than others. By observing a large decrease in performance or a high disparity between the results of each vCPU, an attacker can detect the usage of VMI. This approach can be extended to heterogeneous processors by either comparing only cores with the same specifications or determining the relative speed of different cores.

In addition to these two low-level metrics, higher-level phenomena can be observed in the execution environment in the VM. In Linux-based VMs, an example is the clock stability checking logic of the kernel, which regularly checks that the currently used clock source (usually the TSC) is synchronized across cores and runs at a constant speed [3, 4].

3.2 Limitations of existing strategies to hide VMI pauses

Previous works have proposed solutions to hide the impact of VMI pauses, but these solutions are only applicable to either single-core VMs or to VMI pauses affecting the entire VM (e.g. a VM pause) [6, 25]. In the following, we focus on the TSC clock, as it is the primary clock source on Intel x86 processor. The logic is also mostly valid for other clock sources. The main strategy is to roll back the guest TSC, after a VMI pause, to its value before the pause. This strategy can almost completely hide VMI pauses in single-core VMs. This idea was first proposed to tackle hypervisor introspection in [25], and implemented in a few works [6, 11]. In the rest of the paper, we call this strategy the `Naive` approach.

In a multi-core VM, each vCPU has its own TSC counter, `TSC-offset` and `TSC-multiplier` registers. That way, any clock adjustment can be done either on all vCPUs at once, or only on a subset of vCPUs. After a VMI pause on a single vCPU, the `Naive` approach may adjust only the TSC clock of the paused vCPU. But this will introduce clock-based side-effects, as the TSC clocks will no longer be synchronized across all vCPUs, as shown on Figure 1. Adjusting the TSC clocks of all the vCPUs after a VMI pause regardless of if they were paused will instead introduce work-done-based side effects, as the vCPUs that were not paused will have more CPU time than the others. In addition, for unpaused vCPUs, this would create clock rollbacks observable from the TSC clock local to the vCPU. In Section 3.3, the `Naive` approach is extended to multi-core VMs by adjusting the clock of paused vCPUs only and independently from each other.

An appealing approach would be to pause all vCPUs each time a VMI pause is triggered. Then, by performing a global TSC rollback, the clock would stay consistent between the vCPUs and there would be no clock-based or work-done-based side effects observable from within the VM. However, this idea is impracticable because it goes against the design of VMI tools like `libVMI`, and it could interfere with dependencies between VMI events. Furthermore, there could be concurrent VMI pauses on different vCPUs, while this cannot be handled with this approach. It would also introduce a very high performance penalty.

Hypervisor introspection will thus always be possible on a multi-core VM, although it can be made harder by limiting observable work-done and clock related side effects.

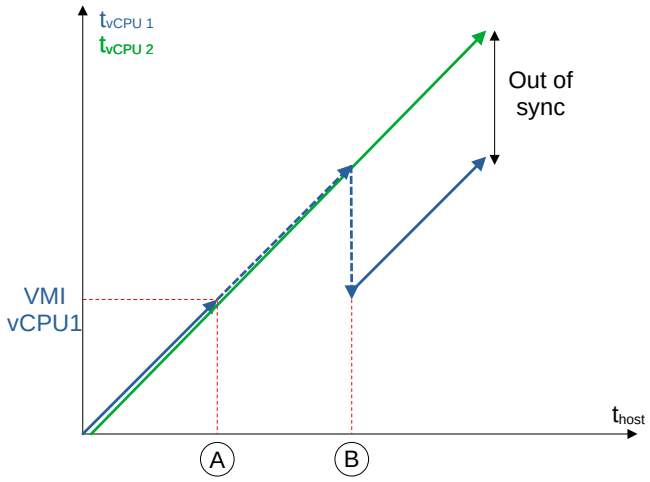

Fig. 1: Illustration of the shortcoming of the `Naive` single-core strategy to hide VMI pauses on a VM with 2 vCPUs. The X axis represents host time, the Y axis virtual time. Both vCPUs clocks are synchronized until (A), where $vCPU_1$ is paused by VMI. At (B), $vCPU_1$ is adjusted to hide the duration of the VMI pause. From this moment on, $vCPU_1$ and $vCPU_2$ clocks are out of sync.

3.3 Evaluation of the `Naive` strategy to hide VMI pauses

We now experimentally evaluate how both metrics introduced in 3.1 can detect in a multi-core setting the `Naive` approach that is classically used to conceal VMI pauses on single-core VMs. To that extent, we use two baselines. `NoAnalysis` constitutes a classic VM setup with no introspection while `Baseline` adds a VMI script to the VM (no attempt is made to conceal the VMI pauses in this case). We opted for DRAKVUF [12], an open-source Xen sandbox based on the `libVMI` library. We used the default DRAKVUF configuration, which traces the execution of the guest operating system. As the guest OS idles and the vCPUs are running on dedicated CPU cores, DRAKVUF mostly traces the execution of the metrics scripts.

The goal of evasive malware is to detect the VMI analysis, so a technique to conceal VMI pauses is effective if the measurement with this technique cannot be distinguished from the `NoAnalysis` measurements. The more a concealing technique differs from `NoAnalysis` on a given metric, the more detectable it is with this metric, and thus the less it is effective.

All experiments in this paper are executed on an Intel i7-1165G7 CPU (4.70 GHz max CPU frequency), in the Xen 4.17 VMM, with a Debian 11 guest. The VM is configured with 8 GB of RAM and 4 vCPUs. Xen was configured to allocate 4 CPU cores exclusively to the VM, to limit potential noise caused by the dom0 OS.

Figure 2 shows the distribution of the results for the clock-based side effects metric, while the values are detailed in Table 1. Symmetrically, Figure 3 shows the distribution of the results for the work-done-based side effects metric, while Table 2 details values.

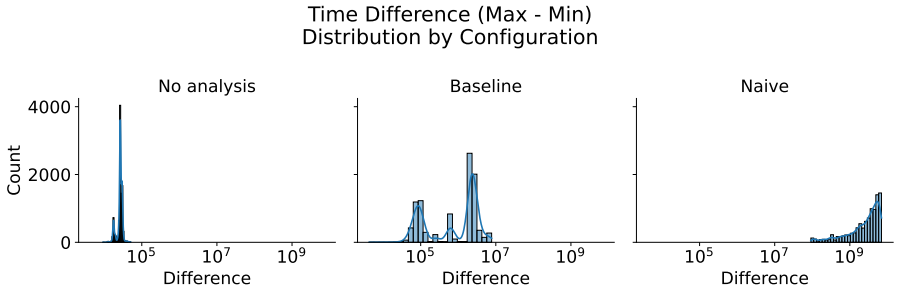

Fig. 2: Histogram of the distribution of the maximum observed time gap between vCPUs for for the `NoAnalysis`, `Baseline` and `Naive` approaches, for the clock-based side effects metric. The difference unit are TSC ticks, with a logarithmic scale. For each iteration, we measure the difference between the highest and lowest TSC timestamps taken by all the vCPUs. We cut values above the 99th percentile to filter outliers (Caused by NMIs for instance). We observe 3 different distributions for each approach. With `NoAnalysis`, the differences between the measurements of TSC are small, between 1×10^4 and 4×10^4. With the `Baseline` VMI script we observe a first peak around the values of the `NoAnalysis`, and a second peak around 2.5×10^6 TSC ticks of difference. This value very likely corresponds to the average duration of a VMI pause. Finally, with the `Naive` approach, we have yet another distribution, which goes up to 7×10^9 TSC ticks of difference, which is about 2.5 seconds on the Intel i7-1165G7 CPU. This is because the vCPUs clocks are out of sync, and drift from each other as time passes.

Table 1: Summary of time differences (Max - Min per measurement) over 10,000 measurements. Unit: TSC ticks. We can see that the values differ by orders of magnitude between the configurations, so this metric can indeed be used to discriminate between the `NoAnalysis` and a tampered multi-core configuration, even if the `Naive` approach is used.

Config	Min	Q1	Median	Q3	Max	Mean	Std
`NoAnalysis`	9.28×10^3	2.59×10^4	2.69×10^4	2.91×10^4	6.06×10^5	2.72×10^4	1.26×10^4
`Baseline`	4.26×10^3	9.95×10^4	2.13×10^6	2.38×10^6	2.10×10^7	1.67×10^6	1.66×10^6
`Naive`	9.17×10^7	1.57×10^9	3.43×10^9	5.24×10^9	7.04×10^9	3.40×10^9	2.09×10^9

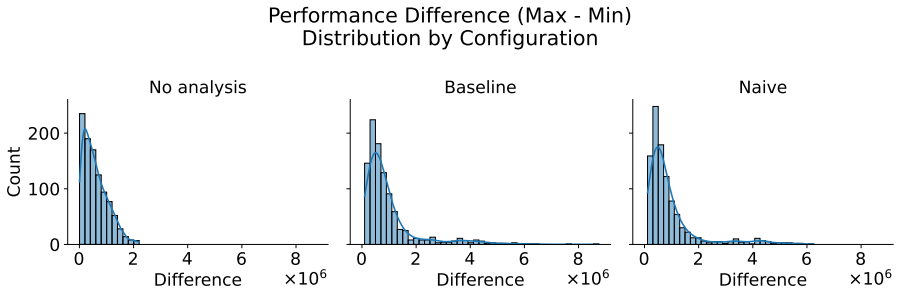

Fig. 3: Histogram of the distribution of the maximum observed performance difference between vCPUs for the `NoAnalysis`, `Baseline` and `Naive` approaches, for the work-done-based side effects metric. For each iteration, we measure the difference between the highest and lowest counter value computed by each vCPU. With `NoAnalysis`, we observe most difference values are below 2×10^6. With the `Baseline` VMI script, the distribution is slightly shifted to the right, and has a longer tail, with many values between 2×10^6 and 5×10^6, indicating more discrepancies in the performance achievable by each vCPU. We observe similar results with the `Naive` approach.

Table 2: Summary of performance differences (Max - Min per measurement) over 1,000 measurements, with 10ms per iteration. Unit: N/A. We can see that `Baseline` and `Naive` results are similar. Compared to the `NoAnalysis`, the standard deviation and mean are noticeably higher, showing more spread out and overall larger performance differences between vCPUs in these configurations.

Config	Min	Q1	Median	Q3	Max	Mean	Std
`NoAnalysis`	3.89×10^3	2.25×10^5	4.79×10^5	8.69×10^5	2.21×10^6	5.90×10^5	4.52×10^5
`Baseline`	9.73×10^4	3.83×10^5	6.40×10^5	1.04×10^6	8.75×10^6	9.72×10^5	1.05×10^6
`Naive`	1.11×10^5	3.87×10^5	5.97×10^5	9.96×10^5	6.25×10^6	9.25×10^5	9.89×10^5

4 Strategy to hide VMI pauses on multi-core VMs

This section discusses a strategy to synchronize the different TSC clocks while hiding VMI pauses, with the goal of performing better than the `Naive` approach, that is minimizing the opportunities for an attacker to observe out of sync clocks. Section 4.1 introduces our design principle. Section 4.2 details our proposed `Acceleration` strategy, which accelerates the clocks of late vCPUs. Section 4.3 presents implementation details on the Xen hypervisor.

4.1 Design approach

As discussed in Section 3.2, designing a strategy to mitigate time-based hypervisor introspection implies a trade-off. Rollbacks of the VM clock are ruled out, because they would likely introduce execution bugs in guest applications and kernel services that often assume the clock to be monotonic. Instead, we propose a strategy that authorizes temporarily desynchronization between vCPUs during VMI pauses and then progressively resynchronizes the clocks.

The strategy is designed by assuming 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. Thus, we disregard edge cases such as very long VMI pauses on only a subset of vCPUs, which are irrelevant for any real-world VMI script.

Another important design choice is that when a vCPU is stopped by a VMI pause, we let the other vCPUs continue their execution. Doing otherwise could result in deadlocks when the VMI script expects a VMI pause to occur on multiple vCPUs.

4.2 Adjusting the execution speed of vCPUs

Algorithm 1 Setup of the `TSC-multiplier` after a VMI pause. `current` is the current vCPU. `get_guest_tsc_at` provides the TSC date of a vCPU at a specific host TSC date, while taking into account the state of the vCPU.

Initialize Δ
if VMI pause is over **then**
  `pause_duration` $\leftarrow$ `current.tsc` $-$ `current.vmexit_tsc`
  `current.tsc_offset` $\leftarrow$ `current.tsc_offset` $-$ `pause_duration`
  `current.synchronized` $\leftarrow$ $false$
  `now_current` $\leftarrow$ `current.tsc`
  `now` $\leftarrow$ `rdtsc()`
  `target_tsc` $\leftarrow 0$
  for all $v \in$ vCPUs $\neq$ `current` **do**
    `target_tsc` $\leftarrow \max($`target_tsc`, `get_guest_tsc_at`$(v,$ `now` $+ \Delta))$
  end for
  `tsc_multiplier` $\leftarrow$ `default_multiplier`
  if `target_tsc` $>$ `now_current` **then**
    `tsc_multiplier` $\leftarrow$ (`target_tsc` $-$ `now_current`) $\times$ (`default_multiplier`$/\Delta$);
    if `tsc_multiplier` exceeds 16 bits **then**
      `tsc_multiplier` $\leftarrow$ `max_multiplier`
    end if
  end if
  `current.tsc_multiplier` $\leftarrow$ `tsc_multiplier`
  Update `current.tsc_offset` so that `current.tsc` $=$ `now_current`
  Setup a timer to expire at `target_tsc`
end if

Algorithm 2 Timer expiration handling.

```
current.tsc_multiplier ← default_multiplier
current.synchronized ← true
tsc_offset ← −1
for all v ∈ vCPUs ≠ current do
    if v.sychronized then
        tsc_offset ← v.tsc_offset
    end if
end for
if tsc_offset = −1 then
    Update current.tsc_offset so that current.tsc is the timer expiration date
else
    current.tsc_offset ← tsc_offset
end if
```

Algorithm 3 VMI Pause handling. This algorithm is executed between the VM Exit that triggered the VMI pause and the actual handling of the VMI pause.

```
current.vmexit_tsc ← current.tsc                ▷ Immediately after the VM Exit
...
current.synchronized ← false                    ▷ Handling of the VMI pause
if current.tsc_multiplier ≠ default_multiplier then
    current.tsc_multiplier ← default_multiplier
    now_current ← current.tsc
    if now_current > current.vmexit_tsc then
        delta_tsc = now_current − current.vmexit_tsc
        current.tsc_offset ← current.tsc_offset − delta_tsc
    else
        delta_tsc = current.vmexit_tsc − now_current
        current.tsc_offset ← current.tsc_offset + delta_tsc
    end if
end if
```

We propose a new `Acceleration` strategy to synchronize the TSC clocks of each vCPU while hiding VMI pauses on each vCPU by taking advantage of the `TSC-multiplier` VMCS field. By default, the guest TSC clocks run at the rate of the host TSC clock. However, by using the `TSC-multiplier`, it is possible to accelerate or slow down the guest TSC clock.

With the `Acceleration` strategy, the synchronization of each TSC clock is triggered unconditionally when a vCPU resumes from a VMI pause. As depicted in Figure 4, the main idea is to accelerate the vCPU clock by increasing the `TSC-multiplier` field, until it catches up with the most advanced TSC clock.

The `Acceleration` strategy is implemented within the VMM, between the code handling the completion of the VMI pause and the resuming of the vCPU. It begins by locally hiding the vCPU pause, by subtracting the duration of the

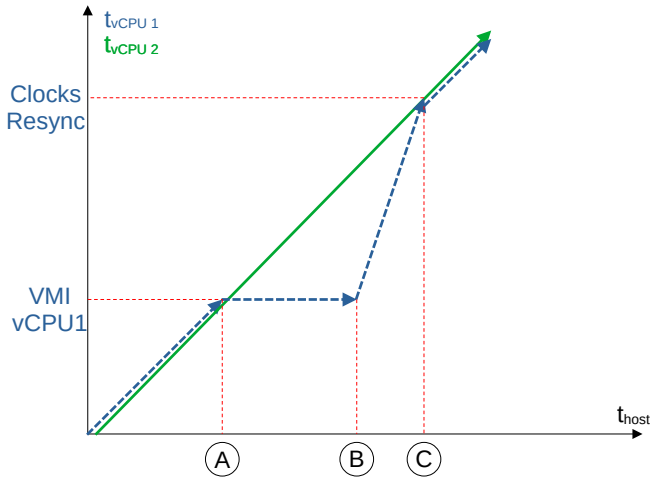

Fig. 4: Illustration of the `Acceleration` strategy to hide VMI pauses with 2 vCPUs. The X axis represents host time, the Y axis virtual time. Both vCPUs clocks are synchronized until (A), where $vCPU_1$ is paused by VMI. When it resumes at (B), its `TSC-multiplier` is set so that its clock runs faster until (C), when it catches up with $vCPU_2$ and the clocks are synchronized.

pause to the vCPU `TSC-offset`, similarly to the `Naive` approach presented in Section 3.2. The vCPU is then flagged as not synchronized.

The default value of the `TSC-multiplier` field is 2^{48}, so we have $TSC_{guest} = TSC_{host} + \texttt{TSC-offset}$. As the `TSC-multiplier` is a 64 bits field, we can write down $TSC_{guest} = m \times TSC_{host} + \texttt{TSC-offset}$, where m is a 16 bits integer with a default value of 1. To accelerate the clock, we set $m = \frac{D-T}{\Delta}$, with Δ the duration of the synchronization process, T the TSC clock at VMI pause resume time, and D the targeted TSC clock of the most advanced vCPU at the end of the synchronization.

The computation of D depends on the status of the most advanced vCPU, which can be in one of three states: synchronized, paused, or in the synchronization process. In case m exceeds 16 bits, we set it to $2^{16} - 1$ to avoid an overflow of the `TSC-multiplier` field, although it is unlikely to ever happen with typical VMI pauses duration. The setup of the `TSC-multiplier` is detailed in Algorithm 1.

Then, we set up a timer to interrupt the accelerated vCPU once it reaches the end of the synchronization process. Algorithm 2 presents the handling of the timer expiration. We check if there is another vCPU which has a synchronized clock. If there is one, we reset m to 1 and adjust the `TSC-offset` to match the one from a synchronized vCPU. In the edge case of a `TSC-multiplier` equal to the maximum, this might cause a time jump in the future. Otherwise, if no vCPU is flagged as synchronized, it means that all others vCPUs are paused by VMI, or

in the synchronization process. In this case, we reset m to 1 and adjust the `TSC-offset` so that TSC_{guest} is equal to the end date of the synchronization process. In both cases, once the timer expires, the vCPU is flagged as synchronized.

Finally, we detail what happens if a vCPU is interrupted due to a VMI pause in Algorithm 3. The vCPU is flagged as not synchronized anymore. In case the vCPU was in the synchronization process, we reset m to 1, and adjust its `TSC-offset` so that TSC_{guest} is equal to its value at the beginning of the VMI pause.

4.3 Implementation on Xen

In this section, we present how we implemented the `Acceleration` strategy on the Xen VMM. `libVMI` supports both the KVM and Xen VMM. We opted for Xen because `libVMI`'s support for it is more mature than for KVM. As KVM, Xen is a mature, open source VMM that is commonly used in computer science research [12,18,19,21,23]. Complex VMI programs based on `libVMI` such as the DRAKVUF sandbox only support Xen.

`libVMI` uses Xen's *VM event* system to perform VMI. We identified that all VMI events are processed by the `monitor_traps` function, which takes an argument to flag if the vCPU must be paused or not. That way, we can identify which VM Exits correspond to a VMI pause, and perform clock adjustments accordingly. To do so, we take a TSC timestamp as soon as possible after a VM Exit.

Moreover, the `Acceleration` strategy mandates the use of an accurate timer to interrupt accelerated vCPUs at a precise end-of-synchronization date, when the accelerated TSC catches up with the most advanced vCPU. We opted for the VMX Preemption Timer, which is specific to Intel x86 processors. It consists of a register counter that counts down at a rate proportional to the host TSC only when the corresponding vCPU is being executed in guest mode. When it reaches 0, a VM Exit is triggered. It is suited for our purpose because it has a high accuracy and handles the interruption of vCPUs. Besides, it is not otherwise used by Xen. It is possible to use other timers, such as Xen's software timer, but doing so will likely be detrimental to the accuracy. This could be due to the timer's inherent low accuracy, the timer being based on host time rather than on guest time or having to trigger the VM Exit on the target vCPU from another hardware core because of lack of control on which hardware core executes the timer handler at expiration time.

Nothing in our implementation relies on Xen-specific mechanisms, so the `Acceleration` strategy should be portable to other VMMs supporting VMI, such as KVM.

5 Evaluation

This section evaluates the efficiency of the proposed `Acceleration` strategy. To that extent, we use the metrics introduced in Section 3.1 to assert if the proposed

strategy is able to conceal VMI pauses from an attacker performing hypervisor introspection, and discuss how this strategy performs compared to the other configurations. We use the same experimental setup as in Section 3.3.

We do not evaluate here the ability of the `Acceleration` strategy to hide VMI pauses from time measurements that are local to a single vCPU, because it is not the focus of this paper. Our implementation is a port of the VMI concealment strategy presented in [6] from KVM to Xen, extended to include the `Acceleration` strategy. VMI pause local concealment is enabled in the `Acceleration` configuration.

First, we compare the results of both the performance and timing metrics between the configurations of `NoAnalysis` and analysis with the `Acceleration` strategy. Figure 5 shows that the `Acceleration` strategy performs similarly to the `Baseline` setup for the performance metric. While the `Acceleration` only provides an illusion of continuous time to a paused vCPU, it does not change the actual CPU time allocated to the VM. Consequently, the CPU time spent in a VMI pause is stolen from the vCPU, similarly to the `Baseline` configuration. This leads to more performance differences between each vCPU over a fixed period of time. These differences can be caused by either a subset of paused vCPUs having less available CPU time, or by the vCPU monitoring the duration of the measurement being paused, which gives more time to the other vCPUs.

Next, Figure 6 shows that the `Acceleration` strategy performs differently from both `NoAnalysis` and `Baseline` configurations for the timing metric. According to the design of the `Acceleration` strategy, we must allow out-of-sync vCPUs to run concurrently. Thus, as expected, we can observe timing differences across concurrent vCPUs two order of magnitudes greater than in the `NoAnalysis` configuration, so the usage of VMI could still be detected with a well chosen threshold.

We also observe that the median difference value with the `Acceleration` strategy is 2 times lower than the one from the `Baseline` scenario. This improvement can be explained by the fact that accelerating the TSC clock can partially conceal the duration of the VMI pause before the TSC timestamp is taken by the metric script. By adjusting the duration of Δ (synchronization process duration) in the `TSC-multiplier` computation, it could be possible to conceal this pause duration even more, although reducing Δ too much would make the strategy nearly identical to `Baseline`.

Another observation is that the observable drift between TSC clocks is much lower than that from the `Naive` approach. Indeed, while the clock drift grows over time in the `Naive` approach (see Figure 6), the drift is bounded by the duration of a VMI pause with the `Acceleration` strategy. It makes `Acceleration` more stealthy regarding this metric.

Finally, we assess the stealthiness of the strategy against high-level time-based phenomena by checking the behavior of the Linux kernel `clocksource` watchdog [4]. As introduced in Section 3.1, Linux provides a watchdog kernel thread that can assess the TSC stability. This watchdog performs time measurements with the TSC, and compares the results to an alternate clock source,

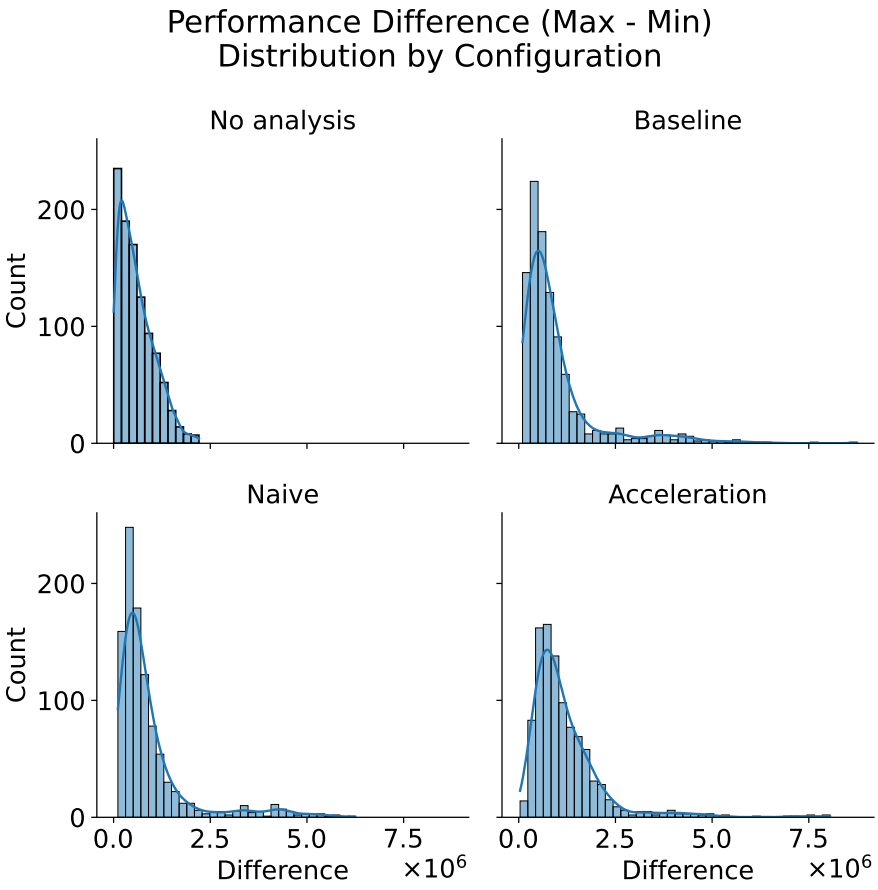

Fig. 5: Histogram of the distribution of the maximum observed performance difference between vCPUs for the `Acceleration`, `NoAnalysis`, `Baseline` and `Naive` configurations. We can see that the `Acceleration` strategy results are close the `Baseline` and `Naive` results, and thus are still noticeably different from the `NoAnalysis`. This can be explained by the fact that the `Acceleration` strategy only accelerates the TSC clock and not the actual CPU speed. The CPU time spent in a VMI pause is not compensated by this strategy.

usually the HPET, considered to be less accurate but more stable. If there is a noticeable drift between both clocksources, the TSC is flagged as unstable and the Linux kernel switches to a more stable clocksource such as the HPET. The threshold to mark the clocksource as unstable is usually rather large, typically in tens of milliseconds. Furthermore, the watchdog also checks if the clocksource remains synchronized across all CPUs [3]. To do so, it performs three clock measurements. A clock measurement is done on a tested vCPU between two mea-

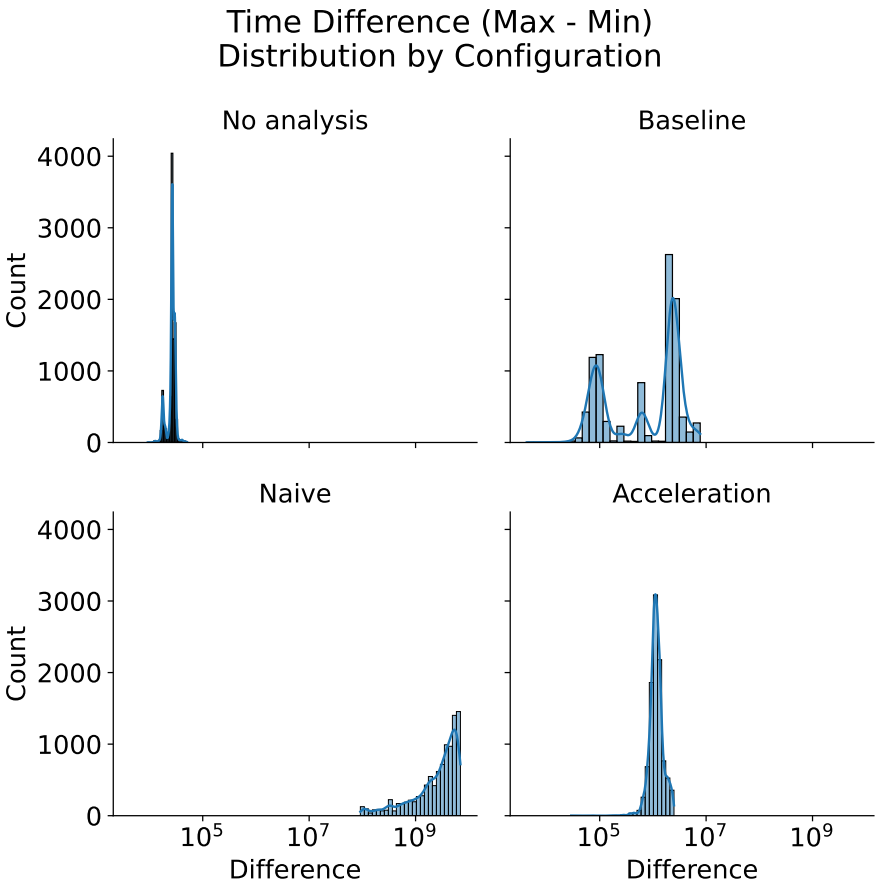

Fig. 6: Histogram of the distribution of the maximum observed time gap between vCPUs for the `Acceleration`, `NoAnalysis`, `Baseline` and `Naive` configurations. The differences unit are TSC ticks. We cut values above the 99th percentile to filter outliers (caused by NMIs for instance). As expected, the `Acceleration` strategy values are far from the `NoAnalysis` configuration, because the strategy allows out of sync vCPUs to run concurrently. Nevertheless, the drift between the TSC clocks of each vCPU is much more limited than the `Naive` approach, with most differences values below 3×10^6 (About 1ms of drift on the Intel i7-1165G7 CPU).

surements on the current vCPU. It then asserts that the three measurements are in order, otherwise the vCPUs are flagged as out of sync and the clocksource marked as unstable.

We configured the guest OS to activate the watchdog, and summarized the results in Table 3 after a short run of DRAKVUF (except for `NoAnalysis`). A

Table 3: Summary of Linux TSC `clocksource` Watchdog behavior under different setups.

Setup	**Watchdog Behavior**
`NoAnalysis`	Not triggered
`Baseline`	Not triggered
`Naive`	Triggered (Clock skew too large compared to HPET)
`Acceleration`	Triggered (CPU clocks out of sync)

first observation is that the watchdog is not triggered in the `Baseline` scenario, because the average duration of VMI pauses is lower than the clock skew threshold of the watchdog. We can see that the watchdog is triggered both for the `Naive` and `Acceleration` approach, although for different reasons. Note that it cannot be ruled out that the watchdog could be triggered by each configuration for the reported reason only. Thus, both approaches introduce side-effects to the virtual TSC clocks that can be detected from the Linux kernel.

Table 4: Summary of VMI detection techniques on a multi-core VM. ✓: Unaffected; ✗: Affected; ~: Depends on the detection script (see footnotes).

Detection technique	`NoAnalysis`	`Baseline`	`Naive`	`Acceleration`
Local to a single vCPU time gaps	✓	✗	✓	✓
Performance difference between vCPUs	✓	✗	✗	✗
Clock synchronization between vCPUs	✓	~[3]	✗	~[4]
TSC clock kernel watchdog	✓	✓	✗	✗

As summarized in Table 4, these results show shortcomings of the `Acceleration` vCPU synchronization strategy, although this strategy offers improvements compared to the `Naive` and `Baseline` approaches in terms of stealthiness.

6 Related work

6.1 Hypervisor introspection

It is a well known problem that VMI pauses can be exploited to perform hypervisor introspection [6, 16, 17, 23, 25], and many countermeasures have been studied [6, 7, 11, 13, 24]. However, the proposed countermeasures are either limited to single-core VMs [6], or disregard new timing attacks introduced by the countermeasure's side effects when used on multi-core VMs [11].

[3] The clocks are synchronized, but concurrent measurements can be affected by VMI pauses. The observable delay is bounded by the duration of a VMI pause.

[4] The clocks are out of sync, but the maximum difference is bounded by the strategy.

Countermeasures that introduce alternative virtual clocks have the same limitations. For example, StopWatch [13] proposes replacing the clock with a virtual clock that counts the number of instructions executed in the VM. The authors explain that their design is incompatible with multi-core VMs because the scheduling of instructions across all cores is non-deterministic.

Approaches that skew the clock by adding randomized delay or degrading its resolution [15, 24] can be extended to multi-core VMs if the clock adjustments are well-balanced across all cores. However, these methods introduce undesirable side effects and the degraded timer resolution may still be sufficient to detect VMI pauses.

In [23], the authors mention that a resilient strategy for performing hypervisor introspection in a multi-threaded environment is to use parallel loops that count the number of iterations. This reuses an approach first proposed to detect the usage of virtualization on a multi-core system [22]. We used a similar method for the work-done-based metric. In [14], the authors highlight multiple side channels to detect the presence of a sandbox in multi-core environments, mostly related to the scheduling of virtual cores. In this paper, we mitigated most of these attacks by properly dedicating a CPU core to each vCPU.

Many sandboxes are still configured as single-core VMs [5]. Limiting sandboxes to single-core configurations has been discussed [14], but in addition to the large performance penalty, this makes it easy for evasive malware to detect such sandboxes because most systems, even virtualized, are multi-core [14, 17].

6.2 Time dilation

The strategy we proposed in this work to synchronize the vCPUs clocks is similar to *time dilation* [9] techniques. Time dilation involves scaling the rate at which time and system resources run by a constant factor. In TimeSync [21], this constant factor is updated regularly to synchronize a VM with a network simulator. In this work, we also use a dynamic factor to modify the virtual clock speed. However, TimeSync only slows down virtual time, so that the network simulator running its nominal speed can keep up with the execution of the VMs. To produce the opposite effect, distem [20], a network emulation tool, can throttle some CPU cores to create the illusion to the others that they can achieve better performance than in reality. In this paper, we accelerate the clock speed to produce an effect similar to distem's, but without introducing CPU throttling.

7 Conclusion and future work

In this paper, we introduced two new metrics to characterize the usage of VMI on a multi-core VM. We have shown that both work-done-based and clock-based side effects occur after a VMI pause, and can be observable by an attacker within the VM. Furthermore, the metrics classically used to detect the usage of VMI on single-core VM are also applicable in multi-core settings.

We proposed a new VMI pause concealing strategy named `Acceleration` that takes advantage of Intel's virtualization of the TSC clock source. When evaluated in multi-core VMs, this strategy outperforms the ones used in current sandboxes for some metrics, but still has numerous shortcomings, especially regarding work-done based side effects, as they are not taken into account in the strategy design.

As future work, we plan to conduct a more thorough evaluation of the impact of each DRAKVUF plugin on each studied metric. This will require adjusting the workload in the guest VM, in order to significantly stimulate most DRAKVUF plugins.

A better strategy than `Acceleration` would have to take into account the CPU time in addition to the clock time, in order to better conceal VMI pauses across all cores. This could be achieved by balancing the runtime of each vCPU of a VM over a certain period of time. Another idea similar to the `Acceleration` strategy would be to control the CPU frequency of each vCPU core, to balance the average CPU frequency despite VMI pauses. However, without specific hardware support, this would likely require patching the guest OS, which would degrade the stealthiness of the approach.

Hardware extensions to better control the vCPUs clocks and speed from the VMM would help implementing flexible strategies to reduce observable time-based side-effects of VMI in multi-core VMs. Similarly to the VMX preemption timer, such extensions would likely be the most helpful if their effect is enabled only while the CPU is in guest mode. For instance, having guest TSC clocks that stop running on a VM-Exit would make VMI hiding simpler (no need to measure the VMI pause durations) and thus more accurate. Similarly, such extensions would allow the VMM to slow down a vCPU without intervention of the guest OS and keeping the guest TSC running at a speed looking constant from the guest kernel point of view on the vCPU.

References

1. Libvmi: Simplified virtual machine introspection (2010), `https://github.com/libvmi/libvmi`
2. Xen artifacts, TANSIV project (2025), `https://gitlab.inria.fr/tansiv/xen/-/tree/HS3-Xen-Artifact?ref_type=tags`
3. Bootlin: clocksource_verify_percpu identifier (2025), `https://elixir.bootlin.com/linux/v6.15.2/A/ident/clocksource_verify_percpu`
4. Bootlin: clocksource_watchdog identifier (2025), `https://elixir.bootlin.com/linux/v6.15.2/A/ident/clocksource_watchdog`
5. Brengel, M., Backes, M., Rossow, C.: Detecting hardware-assisted virtualization. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9721, pp. 207–227. Springer (2016). `https://doi.org/10.1007/978-3-319-40667-1_11`

6. Cosseron, L., Rilling, L., Simonin, M., Quinson, M.: Simulating the network environment of sandboxes to hide virtual machine introspection pauses. In: Proceedings of the 17th European Workshop on Systems Security, EuroSec 2024, Athens, Greece, 22 April 2024. pp. 1–7. ACM (2024). https://doi.org/10.1145/3642974.3652280
7. Dinaburg, A., Royal, P., Sharif, M.I., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Ning, P., Syverson, P.F., Jha, S. (eds.) Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008. pp. 51–62. ACM (2008). https://doi.org/10.1145/1455770.1455779
8. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2003. The Internet Society (2003)
9. Gupta, D., Vishwanath, K.V., Vahdat, A.: Diecast: Testing distributed systems with an accurate scale model. In: Crowcroft, J., Dahlin, M. (eds.) 5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings. pp. 407–422. USENIX Association (2008)
10. Intel: Intel® 64 and IA-32 architectures software developer's manual volume 3 (3A, 3B, 3C & 3D): System programming guide (12 2021), https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
11. Karvandi, M.S., Gholamrezaei, M., Monfared, S.K., Zanjani, S.M., Abbassi, B., Amini, A., Mortazavi, R., Gorgin, S., Rahmati, D., Schwarz, M.: Hyperdbg: Reinventing hardware-assisted debugging. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022. pp. 1709–1723. ACM (2022). https://doi.org/10.1145/3548606.3560649
12. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A.: Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In: Jr., C.N.P., Hahn, A., Butler, K.R.B., Sherr, M. (eds.) Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014. pp. 386–395. ACM (2014). https://doi.org/10.1145/2664243.2664252
13. Li, P., Gao, D., Reiter, M.K.: Mitigating access-driven timing channels in clouds using stopwatch. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 1–12. IEEE Computer Society (2013). https://doi.org/10.1109/DSN.2013.6575299
14. Lusky, Y., Mendelson, A.: Sandbox detection using hardware side channels. In: 22nd International Symposium on Quality Electronic Design, ISQED 2021, Santa Clara, CA, USA, April 7-9, 2021. pp. 192–197. IEEE (2021). https://doi.org/10.1109/ISQED51717.2021.9424260
15. Martin, R., Demme, J., Sethumadhavan, S.: Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: 39th International Symposium on Computer Architecture (ISCA 2012). pp. 118–129. IEEE Computer Society (2012). https://doi.org/10.1109/ISCA.2012.6237011
16. Mishra, P., Pilli, E.S., Varadharajan, V., Tupakula, U.K.: Intrusion detection techniques in cloud environment: A survey. J. Netw. Comput. Appl. **77**, 18–47 (2017). https://doi.org/10.1016/J.JNCA.2016.10.015
17. Nance, K.L., Hay, B., Bishop, M.: Investigating the implications of virtual machine introspection for digital forensics. In: Proceedings of the The Forth International Conference on Availability, Reliability and Security, ARES 2009, March 16-19, 2009, Fukuoka, Japan. pp. 1024–1029. IEEE Computer Society (2009). https://doi.org/10.1109/ARES.2009.173

18. Payne, B.D.: Simplifying virtual machine introspection using libvmi. Tech. rep., Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA (2012)
19. Proskurin, S., Lengyel, T.K., Momeu, M., Eckert, C., Zarras, A.: Hiding in the shadows: Empowering ARM for stealthy virtual machine introspection. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 407–417. ACM (2018). https://doi.org/10.1145/3274694.3274698
20. Sarzyniec, L., Buchert, T., Jeanvoine, E., Nussbaum, L.: Design and evaluation of a virtual experimental environment for distributed systems. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 172–179 (2013). https://doi.org/10.1109/PDP.2013.32
21. Sultan, F., Poylisher, A., Lee, J., Serban, C., Chiang, C.J., Chadha, R., Whittaker, K., Scilla, C., Ali, S.E.: Timesync: enabling scalable, high-fidelity hybrid network emulation. In: Zomaya, A.Y., Landfeldt, B., Prakash, R. (eds.) The 15th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM '12. pp. 185–194. ACM (2012). https://doi.org/10.1145/2387238.2387271
22. Thompson, C., Huntley, M., Link, C.: Virtualization detection: New strategies and their effectiveness. Univ. of Minn., Minneapolis, MN, USA (2010)
23. Tuzel, T., Bridgman, M.P., Zepf, J., Lengyel, T.K., Temkin, K.J.: Who watches the watcher? detecting hypervisor introspection from unprivileged guests. Digit. Investig. **26 Supplement**, S98–S106 (2018). https://doi.org/10.1016/j.diin.2018.04.015
24. Vattikonda, B.C., Das, S., Shacham, H.: Eliminating fine grained timers in xen. In: Cachin, C., Ristenpart, T. (eds.) Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011. pp. 41–46. ACM (2011). https://doi.org/10.1145/2046660.2046671
25. Wang, G., Estrada, Z.J., Pham, C.M., Kalbarczyk, Z.T., Iyer, R.K.: Hypervisor introspection: A technique for evading passive virtual machine monitoring. In: Francillon, A., Ptacek, T. (eds.) 9th USENIX Workshop on Offensive Technologies, WOOT '15. USENIX Association (2015)
26. Wong, M.Y., Landen, M., Antonakakis, M., Blough, D.M., Redmiles, E.M., Ahamad, M.: An inside look into the practice of malware analysis. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021. pp. 3053–3069. ACM (2021). https://doi.org/10.1145/3460120.3484759