Transparent Intrusion Detection in Xen Virtual Machines using libVMI: Performance and Limitations

Niels TERESE
INRIA and KrakOS
Grenoble, France
niels.terese@etu.univ-grenoble-alpes.fr

Supervised by: Baptiste LEPERS

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature: TERESE Niels 06/06/2025

Abstract

Virtual Machine Introspection (VMI) offers a powerful mechanism to observe and analyze guest systems from the hypervisor level, enabling advanced security applications without requiring modifications to the guest system.

However, VMI can introduce performance overheads that limit its practical implementation. This study evaluates the performance impact of using `libVMI` for transparent introspection in a Xen-based virtualization environment.

Controlled experiments were conducted on the Grid5000 infrastructure, comparing system performance across various introspection configurations using benchmarks from the `Phoronix Test Suite`.

The results indicate a performance degradation of up to 30% in configurations involving frequent virtual machine pauses and memory flushes. In contrast, optimized introspection setups reduced overhead to less than 1% in some cases.

These findings highlight a critical trade-off between monitoring granularity and performance. This work lays the foundation for improving the efficiency of VMI-based security tools by targeting the primary sources of overhead.

1 Introduction

The security of modern information systems increasingly relies on solutions capable of analyzing system behavior without interfering with normal execution. In this context, Virtual Machine Introspection (VMI) has emerged as a promising approach. It enables the observation of a virtual machine (VM) from the outside, typically from the hypervisor[1] , offering a transparent and robust way to monitor system activity. This capability is particularly valuable for intrusion detection and malware analysis, as it keeps monitoring mechanisms hidden from potential attackers.

Among the available VMI libraries, `libVMI` stands out as one of the most widely used in both academic research and security-related projects. It provides APIs to access the memory and registers of a paused VM through mechanisms offered by hypervisors such as Xen. However, this approach raises a major concern: the performance overhead introduced by frequent VM pauses required for safe and consistent introspection. These interruptions may significantly degrade the responsiveness and usability of the target system, especially in production environments.

In this work, we empirically investigate the performance impact of VMI, using `libVMI` in a Xen-based environment. Our experiments were carried out on the Grid5000 platform, which offers a controlled and reproducible testbed ideal for performance benchmarking. We evaluated the overhead introduced by introspection using tools such as `xentrace` and the `Phoronix Test Suite`, comparing scenarios with and without active introspection under various system loads.

The goal of this study is the following: quantify the performance degradation caused by VMI and identify the key technical factors responsible for this overhead, thus providing insights for future optimization of VMI-based security tools.

2 Xen Project Hypervisor

The Xen Project hypervisor is a lightweight, open-source type-1 hypervisor [2] that allows multiple operating systems to run in parallel on a single physical machine. It is widely used in applications such as server and desktop virtualization, cloud infrastructure, and embedded systems.

2.1 Architecture

The Xen Project architecture is built around a layered virtualization model. Figure 1 illustrates the core components involved.

[1] A hypervisor is a software layer that enables the creation and management of VMs by abstracting and allocating physical hardware resources to them. It mediates access between guest operating systems and the underlying hardware.

[2] A type-1 hypervisor runs directly on the physical hardware, without an underlying operating system. It provides better performance and security compared to type-2 hypervisors (which run atop a host OS).

- **Xen Hypervisor:** The hypervisor is the first software component to run after the system's bootloader. It operates directly on the hardware and is responsible for managing low-level resources such as the CPU, memory, and interrupts.
- **Domain 0 (Dom0):** This is a privileged VM that boots immediately after the hypervisor. Dom0 has direct access to the hardware and contains the drivers and services required to manage the system. It also handles the control and coordination of other VMs (DomUs). Key components within Dom0 include:
  - **System services** - Includes XenStore/XenBus for configuration management, the toolstack interface, and device emulation (via QEMU).
  - **Native device drivers** - Provides hardware support by including physical device drivers.
  - **Virtual device drivers** - Hosts back-end drivers that interface with virtual devices in guest domains.
  - **Toolstack** - Manages VM lifecycle (creation, destruction, configuration) via CLI, GUI, or cloud orchestrators
- **Domain U (DomU)** - These are the unprivileged guest VMs that run user operating systems and applications. DomUs are fully isolated from hardware access and rely on Dom0 for I/O and device management.

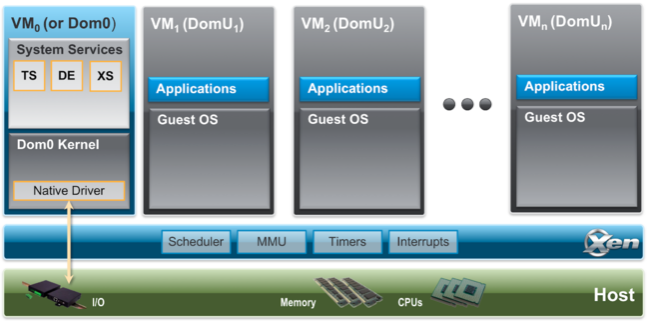

Figure 1: Xen Project architecture overview [Xen,]

3 Analysis of libVMI

`libVMI` is an open-source library that facilitates VMI by enabling external access to a VM's internal state, such as memory and process structures, directly from the hypervisor layer. Unlike traditional in-guest monitoring tools, `libVMI` operates from the outside, typically within the privileged domain (dom0 in Xen), allowing transparent observation of guest systems. It works with Xen, KVM, Qemu, and Raw memory files. Supports both Windows and Linux VMs. Supports 32-bit, PAE, and 64-bit x86 and ARM Cortex-A15 architectures.

The library supports two snapshotting modes for acquiring memory state [Suneja *et al.*, 2015]:

- **Live snap** – captures memory content while the guest VM continues execution. This method minimizes disruption, but may result in inconsistent views of memory due to ongoing changes during capture.
- **Halt snap** – pauses the VM to obtain a consistent and stable snapshot of memory. Although more intrusive, this method ensures coherence and is preferred for forensic analysis where accuracy is critical.

The primary benefit of using `libVMI` is its non-intrusiveness: it allows monitoring without modifying the guest system, reducing the risk of detection or compromise. However, this approach introduces performance challenges, most notably the overhead caused by pausing the VM to access memory safely.

3.1 Memory introspection

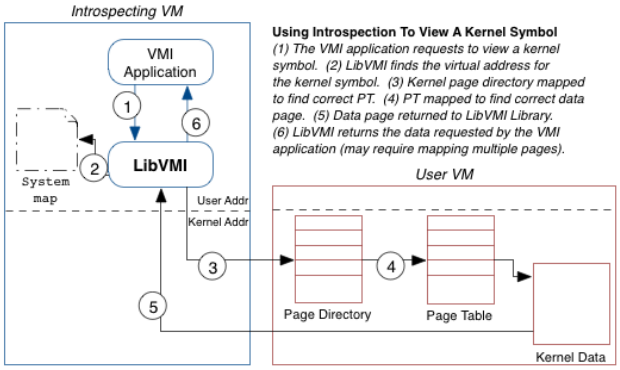

Figure 2: Introspection Detail [libVMI, b]

To perform an effective introspection, `libVMI` requires knowledge of the internal memory layout of the guest operating system. This includes symbol information (available via the /boot/System.map file) and precise memory offsets for key kernel structures. Since the System.map file alone is insufficient for locating runtime memory positions, a helper kernel module, *findoffsets.ko*, must be compiled and executed within the guest. This module extracts the necessary offsets and logs them (typically in /var/log/syslog), enabling accurate mapping of kernel structures during introspection.

The general process and the flow of the components for this introspection are illustrated in Figure 2

3.2 libVMI stack

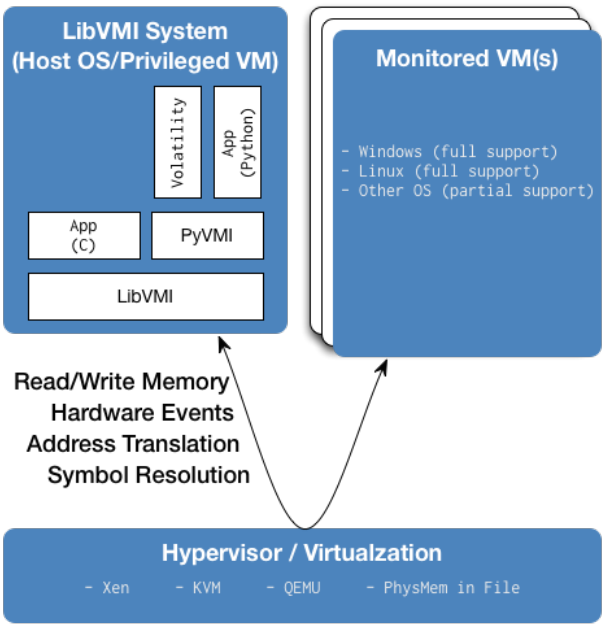

Figure 3: Introspection Detail [libVMI, d]

Figure 3 provides a detailed overview of the software stack underlying the `libVMI` implementation. The framework is composed of the following key components:

- **libVMI API** – the core C library that facilitates memory introspection from the hypervisor level
- **PyVMI** – a Python wrapper for `libVMI`, enabling easier scripting and rapid development
- **Volatility plugin** – an extension that integrates `libVMI` with the Volatility memory forensics framework

3.3 Cache system

To address the inherent performance challenges of VMI, `libVMI` implements an internal caching architecture comprising four distinct caches:

- **Virtual address to physical address cache**
- **Virtual address to process identifier cache**
- **Kernel symbol to virtual address cache**
- **Page cache** - the most performance-critical component

The page cache optimizes memory access by managing mappings of accessed memory pages. Initially implemented as a hash table keyed by physical page addresses, the system encountered performance issues due to rapid growth and slower lookups.

To mitigate this, the developers introduced a least-recently-used (LRU)-style eviction mechanism. This design uses an access-ordered list:

- Recently accessed pages are kept at the front.
- When the list exceeds 512 entries, the older half is evicted.
- Corresponding entries in the hash table are removed simultaneously to prevent unbounded growth.

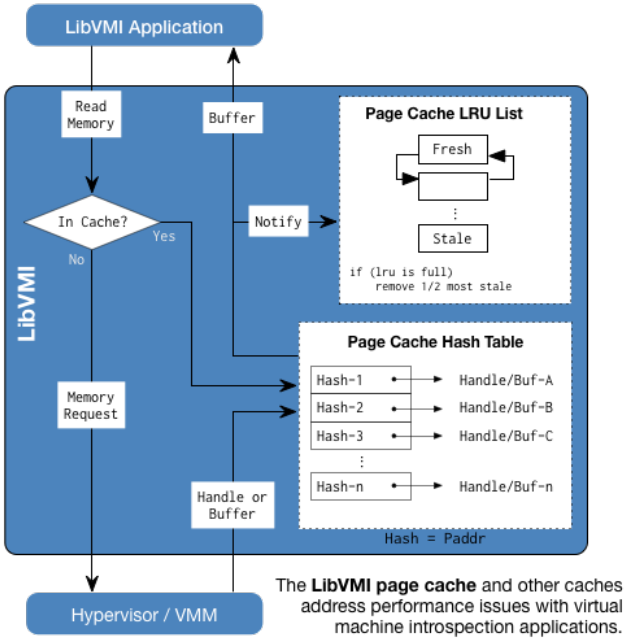

Figure 4: Introspection Detail [libVMI, a]

Figure 4 provides a graphical representation of this page cache algorithm. This approach [Payne, 2012] balances memory usage with access speed and results in significant performance optimization.

4 Used Infrastructure

All benchmarks and experiments were conducted on the Grid5000 Cluster; more specifically, on the Dahu Grenoble's Cluster.

This section details the experimental environment used, including the benchmark tools and configurations used throughout the study.

4.1 Dahu Hardware

The Dahu cluster is composed of 32 nodes. Each one of them has the following hardware [Grid5000,] :

- **Model** - Dell PowerEdge C6420
- **CPU** - Intel Xeon Gold 6130 (Skylake-SP), x86_64, 2.10GHz, 2 CPUs/node, 16 cores/CPU
- **RAM** - 192 GiB
- **Network** - 10 Gbps, model: Intel Ethernet Controller X710 for 10GbE SFP+, driver: i40e

4.2 Environment setup

To work with Xen on Grid5000, a preexisting Debian 12 NFS deployment image was used. Xen was then compiled directly within this image, and the GRUB configuration was modified to enable booting into the Xen hypervisor.

This setup enables consistent and repeatable experimentation, and the custom deployment image eliminates the need to recompile everything from scratch, significantly reducing setup time.

It includes the Xen-provided modified Debian environment serving as Dom0, with the full Xen toolstack preinstalled, ensuring reproducibility across deployments and test sessions.

4.3 VM Setup and Specifications

For testing purposes, an Ubuntu VM was created within the Xen environment. The VM is stored in a dedicated home directory on Grenoble's Grid5000 cluster to ensure persistence across sessions.

Ubuntu was installed using the default configuration, with the OpenSSH server enabled to allow remote access.

The VM specifications are the following :

- **Operating system** - Ubuntu 18.04.6 LTS
- **vCPUs** - 8
- **RAM** - 16 GiB
- **Network MAC address** - 00:16:3E:84:00:02
- **Assigned IP** - 10.132.0.2
- **Disk size** - Approximately 30 GiB
- **User** - vm

5 Performance analysis

The performance impact of using `libVMI` was evaluated through a variety of tools and benchmarks. The evaluation considered multiple perspectives: from within the VM, from the Dom0, and from Xen's hypervisor itself.

5.1 Benchmarking tools

The following tools and techniques were used:

- **Phoronix Test Suite** v10.8.4 — for high-level application benchmarks
- **perf** - for low-level profiling of CPU cycles and call stacks, both inside the VM and in Dom0
- **htop** and **xl top** — for real-time CPU and resource monitoring
- **xentrace** and **xenalyze** — for hypervisor-level tracing and event analysis
- **Shell script** – to synchronize `xentrace` and `Phoronix Test Suite` benchmarks. A custom shell script was developed to perform a consistent start of `xentrace` during the execution of a benchmark. This ensured that hypervisor-level tracing occurred precisely during the workload's runtime.

```
#!/bin/bash

# Start Phoronix in background on the VM

sshpass -p "password" ssh vm@10.132.0.2
"timeout 75 phoronix-test-suite batch-run
compress-gzip" &

# Sleep before tracing
sleep 50

# Start xentrace for 10 seconds
sudo timeout 10 xentrace -D
> traces/trace.bin &
XENTRACE_PID=$!

# Wait for xentrace to finish
wait $XENTRACE_PID

echo ">>> xentrace finished"
```

The script launches the benchmark inside the VM using SSH, waits an appropriate delay, then starts tracing using `xentrace`.

- **Python parser for `xenalyze` output** – to summarize and analyze trace data.
  After executing `xentrace`, the resulting binary trace file is processed by `xenalyze`, which produces a human-readable text log. To automate the extraction of meaningful statistics from this log, a Python script named `xen_parser.py` was created. More details in the dedicated Git repository [Terese, 2025b].

5.2 Benchmarking methodology

Multiple benchmarks from the `Phoronix Test Suite` (e.g., smallpt, gzip, c-ray, apache, stream) were used. Each benchmark was systematically executed under multiple configurations to assess the impact of `libVMI` introspection. Specifically, every test was run:

- **Without libVMI** - Serving as the baseline to measure uninstrumented VM performance
- **With libVMI** - introspection enabled, running in Dom0, using multiple implementations

This methodology enabled a fine-grained analysis of how `libVMI` affects performance under both realistic and controlled conditions.

5.3 Main results

Initially, the default `vmi-process-list` example provided by `libVMI` was used, executing it once per second via an external shell loop. However, this approach included several inefficiencies and limitations, which were progressively corrected:

- **Removed initialization overhead** – The original script reinitialized the `libVMI` context during every execution, which introduced a noticeable performance cost.

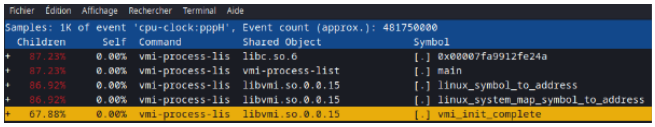

Figure 5: perf record result

This was confirmed using `perf record`, which showed that a large portion of CPU time was spent inside the `vmi_init_complete()` function as shown in Figure 5. To fix this, the loop was integrated directly into the script, and `libVMI` was initialized only once, allowing the same instance to be reused across iterations.

- **Disabled VM pausing** - By default, `libVMI` pauses the VM during introspection to ensure consistency. This behavior was removed by deleting this call `vmi_pause_vm()`, allowing for real-time, non-intrusive introspection
- **Improved robustness** - Without pausing the VM, memory consistency is no longer guaranteed. This caused occasional failures due to invalid or partially-updated pointers. The script was modified to handle such inconsistencies, avoiding crashes or infinite loops (5 iterations max if we cannot find the head pointer for process listing).
- **Enabled continuous introspection** - The one-second sleep in the loop was removed to perform continuous introspection at maximum frequency. This helped assess the worst-case performance impact of sustained monitoring.
- **Implemented page cache flushing** – `libVMI` stores memory pages in a cache, as shown in Figure 4, which can become outdated if the VM is rebooted or shut down. To make sure the process list is always up to date, a call to `vmi_flush_cached_pages()` [libVMI, c] was added at the end of the loop. This forces `libVMI` to reload the memory from the VM, but adds some performance overhead.

This iterative development process allowed both increased performance realism and greater introspection accuracy, while also revealing the trade-offs between monitoring frequency, overhead, and system stability. The modified script can be found on this GIT repository [Terese, 2025a].

5.4 Impact on performance

A variety of benchmarks were executed under several configurations using the `Phoronix Test Suite` v10.8.4, including `smallpt`, `compress-gzip`, `c-ray (4k)`, `build-linux-kernel (defconfig)`, `apache (200)`, and the memory bandwidth tests from `pts/stream` (copy, scale, triad, add types). Each benchmark was run with and without `libVMI` enabled, using different introspection modes.

The most insightful results are summarized below:

1. **Initial test – libVMI with default process list every second**

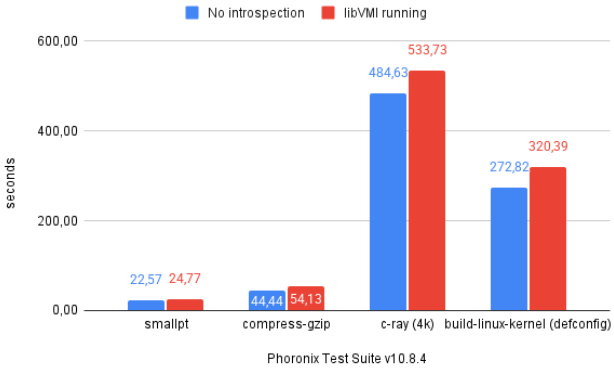

Figure 6: Original libVMI CPU benchmarks results

CPU overhead was substantial in many cases, with up to **21.8%** on `compress-gzip`, **17.4%** on kernel build, and an overall average increase of **14.77%**. Concerning memory, no overhead was found using `pts/stream` benchmark.
This version still included VM pauses and external loop overhead.

2. **Optimized introspection – No pause, internal loop, with correctness handling**

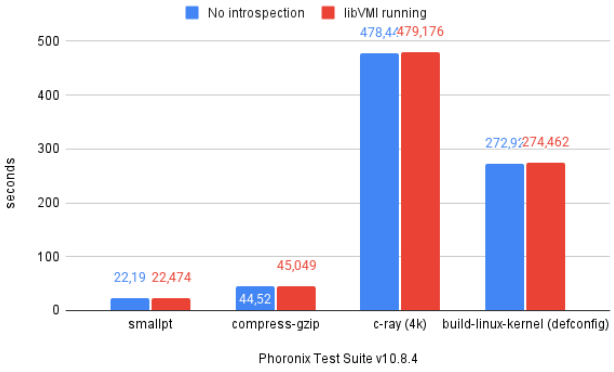

Figure 7: Optimized `libVMI` CPU benchmarks results

After integrating the loop within the `libVMI` script and removing VM pauses, the CPU overhead dropped significantly. The average impact was reduced to just **0.79%**. Memory overhead showed an average overhead of **0.22%** using `pts/stream` benchmark.

3. **Continuous mode – No pause, no flush, loop without delay**
With introspection running continuously, but without cache flushes, moderate overhead reappeared. For example, `compress-gzip` slows the VM by **9.29%**, and `build-linux-kernel` by **5.42%**, with an average impact of **5.29%** across CPU benchmarks. The average memory overhead is now **1.03%**.

4. **Final mode – No pause, continuous, with cache flushing** This mode, aimed at ensuring process list accuracy after VM reboots or shutdowns, introduced the highest overhead.

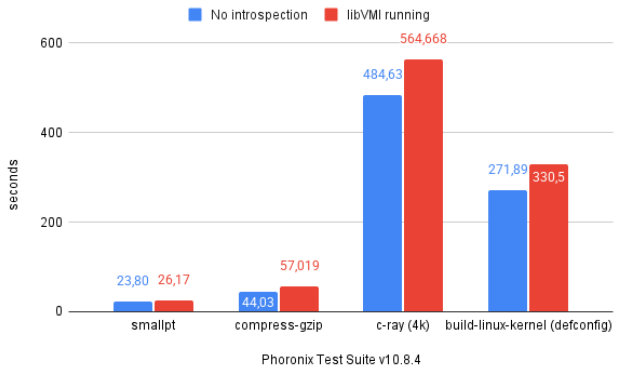

Figure 8: Final `libVMI` CPU benchmarks results

Benchmarks showed significant performance overheads, with `compress-gzip` reaching **29.5%** and `c-ray` **16.5%**. The average CPU overhead reached **19.38%**, while memory bandwidth dropped by over **5.7%**.

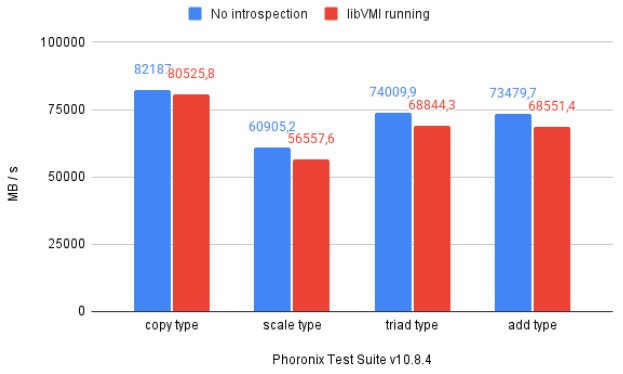

Figure 9: Final `libVMI` memory benchmarks results

5.5 Limitations of traditional profiling tools

To understand the source of this overhead, traditional tools like `perf` were first used inside the VM. However, `perf` failed to reveal meaningful information, especially when the VM was paused during introspection, as the profiling itself was interrupted.

Interestingly, a **2.40%** overhead was measured with `perf`, which can be attributed to the VM's recovery cost after each pause; since `libVMI` was initially configured to pause the VM at each introspection cycle.

5.6 Hypervisor-level tracing

Initial monitoring from Dom0 using `htop` showed no visible changes in CPU usage while running benchmarks (e.g., `smallpt`) inside the VM, regardless of whether `libVMI` was active. However, `htop` does not accurately capture CPU activity at the hypervisor level. To obtain more precise data, `xl top` was used, as it performs a hypercall to fetch real-time vCPU metrics directly from Xen.

This revealed a clear pattern: without `libVMI`, all 8 vCPUs of the VM were used at full capacity (approximately 800% CPU usage), while Dom0 consumed only about 1.8%.

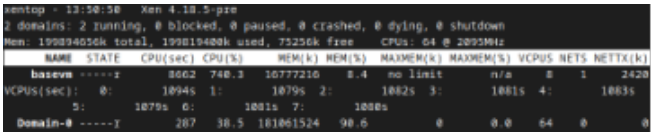

Figure 10: `xl top` while `libVMI` is running

When `libVMI` was enabled, the VM's vCPU usage dropped to around 740%, and Dom0's CPU usage increased significantly to approximately 39%, as shown in Figure 10.

This behavior is consistent with observations reported in previous research [Utomo *et al.*, 2019], which noted similar CPU distribution shifts between Dom0 and the guest VM during introspection.

To gain a deeper understanding of the overhead introduced by `libVMI`, hypervisor-level tracing tools were used : `xentrace` and `xenalyze`. These tools provide insight into low-level VM behavior, such as CPU scheduling, hypercalls, and privileged operations, at the Xen hypervisor layer.

We focused on the `compress-gzip` benchmark, which showed some of the most significant slowdowns (30%) in previous tests. The trace data was parsed using `xenalyze` and the python script [Terese, 2025b] used to extract domain-level statistics.

The parser output revealed indirect effects:

- The VM maintained similar *Running* time across runs, confirming that the CPU scheduling was not affected.
- Hypercall activity in the VM increased by 15%, despite a total overhead of 30%.
- Hypercall activity in Dom0, particularly *iret*, *stack_switch*, *mmu_update*, and *vcpu_op* increased significantly, indicating overhead from repeated memory page operations triggered by `libVMI`.

6 Conclusion

The results of this study suggest that the performance overhead introduced by `libVMI` is not induced by direct CPU starvation or scheduling delays, but rather from more indirect effects, such as:

- Frequent memory mapping and unmapping
- Cache invalidation within the guest VM.
- Coordination overhead within the hypervisor when managing memory between Dom0 and the guest.

These mechanisms help to explain why conventional tools such as `perf` and `xentrace` do not capture the full extent of slowdown. Custom instrumentation (e.g., synchro-

nized tracing and automated xenalyze parsing) was essential to reveal the indirect effects of introspection. From the hypervisor's perspective, the VM appears to be running normally. However, real-world execution time and CPU usage, measured by tools such as `xl top` and `Phoronix Test Suite`, clearly reveal a performance impact.

References

[Grid5000,] Grid5000. Dahu hardware. https://www.grid5000.fr/w/Grenoble:Hardware#dahu.

[libVMI, a] libVMI. Cache system. https://libvmi.com.

[libVMI, b] libVMI. Introspection detail. https://libvmi.com/docs/gcode-intro.html.

[libVMI, c] libVMI. libvmi api details. https://libvmi.com/api.

[libVMI, d] libVMI. Software stack. https://libvmi.com/docs/gcode-intro.html.

[Payne, 2012] Bryan D Payne. Simplifying virtual machine introspection using libvmi. Technical report, Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA (United States), 09 2012.

[Suneja *et al.*, 2015] Sahil Suneja, Canturk Isci, Eyal de Lara, and Vasanth Bala. Exploring vm introspection: Techniques and trade-offs. *SIGPLAN Not.*, 50(7):133–146, March 2015.

[Terese, 2025a] Niels Terese. Git repository containing the modified libvmi example. https://github.com/NielsTRS/libvmi/blob/master/examples/process-list.c, 2025.

[Terese, 2025b] Niels Terese. Xenalyze parser repositoty. https://github.com/NielsTRS/xenalyze-parser, 2025.

[Utomo *et al.*, 2019] Agus Priyo Utomo, Idris Winarno, and Iwan Syarif. Detecting hang on the virtual machine using libvmi. In *2019 International Electronics Symposium (IES)*, pages 618–621, 2019.

[Xen,] Xen. Xen architecture. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview.