Towards Testing the Performance Influence of Hypervisor Hypercall Interface Behavior

Lukas Beierlieb, Lukas Iffländer, Samuel Kounev
{firstname}.{lastname}@uni-wuerzburg.de
University of Würzburg

Aleksandar Milenkoski
amilenkoski@ernw.de
ERNW GmbH

Abstract

With the continuing rise of cloud technology hypervisors play a vital role in the performance and reliability of current services. Hypervisors offer so-called hypercall interfaces for communication with the hosted virtual machines. These interfaces require thorough robustness to assure performance, security, and reliability. Existing research focusses on finding hypercall-related vulnerabilities. In this work, we discuss open challenges regarding hypercall interfaces. To address these challenges, we propose an extensive framework architecture to perform robustness testing on hypercall interfaces. This framework supports test campaigns and modeling of hypercall interfaces.

1 Introduction

The trend to move applications to the cloud continues without interruption [5]. This ongoing transition from dedicated servers to the cloud results in most services now running in virtual environments.

Virtualization received increasing interest as a way to reduce costs through server consolidation and to enhance the flexibility of physical infrastructures. It allows the creation of virtual instances of physical devices called virtual machines (VMs). Hypervisors implement interfaces that provide call-based connectivity to hosted VMs. One of them is the hypercall interface, which allows a VM to request services from the hypervisor. Hypercalls are software traps from a VM to the hypervisor. They are critical for the operation of VMs. The behavior of the hypervisor's hypercall interface constitutes a significant part of the virtualized environment's overall behavior. Therefore, testing the performance-related aspects of this behavior is essential. This paper focusses on such testing.

The contributions of this paper are **(1)** Discussions on central open challenges when it comes to testing performance aspects of virtualized environment behavior, and **(2)** A framework that facilitates such testing and enables the addressing of these challenges. It allows the observation and measurement of the behavior of hypercall interfaces by generating and executing tailored test campaigns. The framework is as generic as possible. We provide an example of hypervisor-specific details for Hyper-V.

Related work injects hypercalls in a Xen based environment [3], discusses hypercall vulnerabilities [2], and introduces the idea of test campaings [4].

The remainder of this paper is structured as follows: Section 2 discusses key open challenges. Next, Section 3 introduces the relevant technical background; Section 4 presents the proposed framework, and Section 5 concludes the paper.

2 Challenges

In this section, we discuss key open challenges in testing behavioral aspects of virtualized environments. These challenges come in the form of research questions that we will address in our framework.

RQ1: *How to characterize the behavior of a virtualized environment?* It is necessary to identify existing or develop new metrics relevant for characterizing such behavior, to evaluate the impact of hypercall execution.These metrics allow characterizing a baseline behavior to detect deviations. With our framework, we plan to identify metrics and workloads suitable for characterizing the behavior of a virtualized environment. This characterization enables the construction of models tailored for testing and describing relevant behavior aspects of virtualized environments. We also plan to investigate the development of new metrics.

RQ2: *How does the virtualized environment setup impact the test results?* The hypervisor hosting the environment can operate directly on top of the hardware (i.e., a bare-metal setup) or inside a VM hosted by another hypervisor (i.e., a nested virtualization setup), to test the behavior of a given virtualized environment. The nested virtualization setup has the advantage of full control over, and behavior transparency of, the tested virtualized environment. This nesting enables, for example, recovering from crashes caused by tests and storing system and hypervisor states. Such control and transparency are not readily attainable with a bare-metal setup. However, using another hypervisor to host the virtualized environment under test may impact test results and jeopardize representativeness. Therefore, it is essential to identify and evaluate the extent of this impact, including identification of the causing hypercall activities.

RQ3: *How do updates impact on performance, re-*

liability, and robustness? Some operating systems, including the hosted and hosting hypervisors, are frequently updated. For example, some releases of Windows 10 receive monthly updates. Updates often introduce new features and significant reworks of internal mechanisms. Therefore, it is crucial to evaluate the impact on the performance, reliability, and robustness of virtualized environments.

3 Hypervisors and Robustness Testing

Hypervisors: In a non-virtualized scenario, the hardware is managed by an operating system, providing and scheduling resource access to applications running on top. Virtualization describes the concept of introducing an abstraction layer above the hardware. That layer called the hypervisor or Virtual Machine Monitor (VMM) provides a set of virtual resources, which can form multiple virtual machines and be managed by independent operating systems.

One way to classify hypervisors is their level of control over the underlying hardware. The first approach called a Type-1 hypervisor runs directly on top of the hardware and can utilize its full control for increased performance. Contrary to that, Type-2 or hosted hypervisors reduce the complexity by relying on underlying operating systems for the hardware management.

Virtualization solutions differ by their implementation type. Full virtualization allows VMs to run the same, unmodified operating systems used on physical hardware, while para-virtualization requires changes to the source code of operating systems. These modifications also allow hypervisor and VMs to interact more efficiently, e.g., by using abstract IO interfaces instead of emulating existing physical devices to reduce overhead and improve performance.

Nested virtualization describes the situation when a hypervisor is running inside a VM of another hypervisor. Privileged instructions of the virtualized hypervisor have to be trapped and emulated. Memory management hardware does not provide support for the third memory paging layer.

Hyper-V is a Type-1 x86_64 hypervisor developed by Microsoft. To avoid limiting it to specific hardware configurations and implementing countless device drivers, Hyper-V uses a microkernel-based architecture. A specialized VM called the root partition always runs an instance of Windows on top of Hyper-V to provide management features and device drivers. Guest VMs can run para-virtualized if they support it, but can also use unmodified operating systems, in which case Hyper-V provides emulated devices. Similar to how applications can request services from the operating system by issuing system calls, guest operating systems can call to Hyper-V with hypercalls.

Robustness Testing: IEEE defines robustness as *"The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions."* [1]. Thus, robustness testing is concerned with providing unexpected inputs or conditions to the system under test, while trying to detect defects. Often a model of the interface under test is built, containing information about required parameters, their data types, and the tested range of possible values. This model can then be used to generate test cases automatically.

4 Framework

The framework provides a means to test the hypercall robustness of any desired hypervisor. As every hypervisor has its unique hypercall calling convention, it is not possible to generalize every aspect. However, the framework is designed to implement all generalizable steps and provide reusable interfaces for consistent implementation of hypervisor-specific details. We develop a concrete implementation for Hyper-V alongside the generic framework.

The architecture comprises two domains: the test generation and hypercall injection workflow and the execution monitoring during the hypercall injection. Fig. 1 provides a visual overview. Test campaign files describe when hypercall injection occurs and its parameterization. JSON files define the test campaigns allowing humans to arrange tests manually as well as to generate tests automatically from hypercall interface models. The format currently supports the following features: **(1) Hypercalls:** Calling conventions vary across hypervisors. Therefore, we reference hypercalls and their parameters by their name. **(2) Order:** The framework provides a means to test if robustness problems occur when performing specified calls in a specified sequence. **(3) Integer Bounds:** Boundary data type values are a regular testing input. The language supports automatic generation of integer boundary values with regard to the size. **(4) Repetition:** Single hypercalls and series of calls can loop with a specified number of repetitions. **(5) Random:** Parameter values can also be tagged to take on random values. Our framework supports constant, uniformly distributed, and negative-exponentially distributed random values. **(6) Timing:** Delays are injectable between a series of hypercalls.

The framework supplies a compiler parsing the JSON file and extracting hypercalls, delays, and loops. A hypervisor-specific module translates hypercall and parameter names into call codes, parameter sizes, and offsets, calculates integer bounds values according to data type sizes, and exports the campaign in the injection module format.

The injection driver is specific to the used hypervisor. It reads the specially crafted campaign file and injects hypercalls in a hypervisor-specific fashion. Special privileged instructions trigger hypercalls. Thus, hypercalls are not injectable from userspace but have to originate from the kernel, which limits possible implementations to a Windows kernel driver. The driver reads and performs the actions of the campaign step

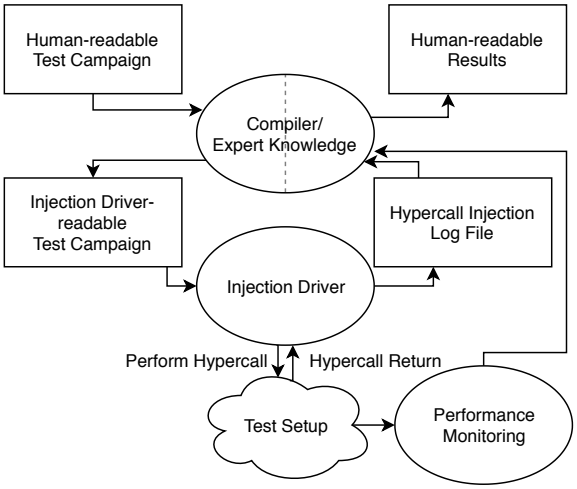

Figure 1: Framework overview

by step. An injection log file collects return and output values, as well as measured execution times. A generalized compiler can generate a human-readable report of the test campaign execution results, using the original campaign file, the injection log file, a hypervisor-specific helper module, and performance measurement data.

The second domain of the framework is concerned with performance monitoring to detect unexpected behavior while executing a test campaign. As stated in **RQ1**, the first step is to construct a model that describes the normal behavior. It is necessary to measure at very short intervals to capture short-term performance degradation. Also, it requires investigation whether to place the performance monitoring agent in the injecting VM or isolated in another guest VM. After establishing a baseline, the model needs to decide which performance deviations are caused by robustness problems. Another part of the model is the detection of crashes and failures. These could potentially happen to only the injecting VM or the whole hypervisor.

There are two alternatives for the testing setup. One approach is to run Hyper-V directly on the hardware, as shown in Fig. 2 on the left. This method has the advantage of being a tried and true configuration. However, in case of crashes, a physical reset has to be triggered to reboot the system. Moreover, if the system gets altered permanently, restoring the initial conditions is an expensive operation.

On the other hand, Hyper-V can itself run in a virtualized environment, e.g., in a KVM virtual machine, as shown in Fig. 2 on the right. With this configuration, having a fully restored system for every test campaign is easily achievable. Nested virtualization is the preferred solution from a test execution perspective. However, **RQ2** questions whether the results are identical between bare-metal.

Naturally, every hardware configuration yields different performance results. This limitation requires to determine the baseline metric values for every single setup. However, as pointed out in **RQ3**, baseline performance is also affected by software changes. Especially with Hyper-V distributed alongside Windows OS, we want to evaluate how much our model varies between different versions of OS builds.

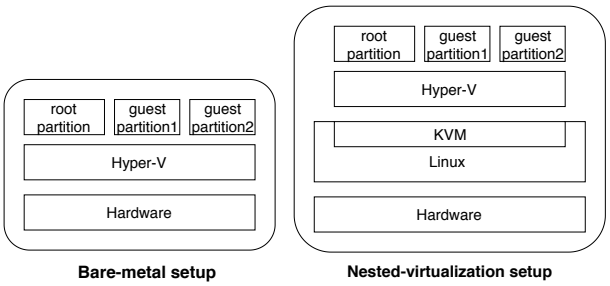

Figure 2: Bare-metal and nested-virtualization setup visualized

5 Conclusion and Future Work

In this work, we described the open challenges regarding the performance influence of hypercall interface behavior in three research questions. Next, we overview the technical background regarding hypervisors and Hyper-V as well as robustness testing. We then proposed a framework architecture for hypercall interface robustness testing. This architecture supports modeling hypercall interfaces, generating test campaigns to assert said models and validating them against new software versions. Furthermore, the framework can execute defined tests while monitoring the system's performance. Finally, we support evaluating the results based on deviations from the baseline characteristics.

In future work, we plan to complete the implementation of the proposed architecture for the Hyper-V hypervisor. Next, we will execute extensive test campaigns on Hyper-V's hypercall interfaces using expert knowledge available inside SPEC.

Acknowledgements

This work was funded by the German Research Foundation (DFG) under grant No. (KO 3445/16-1).

References

[1] IEEE. "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (Dec. 1990).

[2] A. Milenkoski et al. "Experience Report: An Analysis of Hypercall Handler Vulnerabilities". In: *International Symposium on Software Reliability Engineering*. IEEE. 2014.

[3] A. Milenkoski et al. "Evaluation of Intrusion Detection Systems in Virtualized Environments Using Attack Injection". In: *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2015.

[4] C. F. Gonçalves, N. Antunes, and M. Vieira. "Evaluating the Applicability of Robustness Testing in Virtualized Environments". In: *Latin-American Symposium on Dependable Computing (LADC)*. IEEE. 2018.

[5] C. Research. *Public Cloud Services Market Share, Trend And Forecast To 2027*. 2019.