Towards Testing the Software Aging Behavior of Hypervisor Hypercall Interfaces

Lukas Beierlieb
University of Würzburg
Würzburg, Germany
lukas.beierlieb@uni-wuerzburg.de

Lukas Iffländer
University of Würzburg
Würzburg, Germany
lukas.ifflaender@uni-wuerzburg.de

Aleksandar Milenkoski
ERNW GmbH
Heidelberg, Germany
amilenkoski@ernw.de

Charles F. Gonçalves
University of Coimbra
Coimbra, Portugal
charles@dei.uc.pt

Nuno Antunes
University of Coimbra
Coimbra, Portugal
nmsa@dei.uc.pt

Samuel Kounev
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

***Abstract*—With the continuing rise of cloud technology hypervisors play a vital role in the performance and reliability of current services. As long-running applications, they are susceptible to software aging. 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 either deals with the aging properties of hypervisors in general without considering the hypercalls or 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 extensive test campaigns as well as the modeling of hypercall interfaces.**

***Index Terms*—software aging, software rejuvenation, robustness testing, hypercalls, hypervisor security, Hyper-V**

I. Introduction

Researchers discovered the phenomena of software aging in the early '90s as a result of significant software reliability research efforts conducted by large telecoms [1]. Software rejuvenation approaches were developed to address software aging [2]. The impact of software aging on systems is receiving increased attention due to severe failures attributed to software aging [3].

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). In a virtualized environment, governed by a hypervisor, VMs share resources. 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.

Hypervisors often run for extended intervals. Thus, system behavior issues related to software aging, such as performance, scalability, reliability, and robustness degradation, may impact hypervisor operation [4]–[6]. The testing of software aging-related aspects of the behavior of hypervisors, and therefore of virtualized environments, is of immense relevance due to the widespread use of hypervisors. Cloud and workstation infrastructures commonly comprise hypervisors. For example, the Windows 10 operating system supports the Virtual Secure Mode feature [7], which configures Windows 10 to operate as a VM running on top of the Hyper-V hypervisor. Windows 10 co-distributes this hypervisor.

The behavior of the hypervisor's hypercall interface constitutes a significant part of the virtualized environment's overall behavior. Therefore, testing the software aging-related aspects of this behavior is essential. This paper focusses on such testing. The contributions of this paper are:

- Discussions on central open challenges when it comes to testing software aging-related aspects of virtualized environment behavior;
- A framework that facilitates such testing and enables the addressing of these challenges. The framework enables the testing of software aging-related aspects of the behavior of hypervisor hypercall interfaces, with a current focus on robustness as a behavior aspect affected by software aging. It allows the observation and measurement of the behavior of hypercall interfaces by generating and executing tailored test campaigns. The framework is designed as generically as possible. We provide an example of hypervisor-specific details for Hyper-V;
- An outlook on future work towards addressing the open challenges discussed in this work.

The contributions of this paper directly relate to the field of software aging, since they focus on testing software aging-related behavior aspects (i.e., performance, scalability, reliability, and robustness) of a commonly deployed, long-running software (i.e., hypervisors).

The remainder of this paper is structured as follows: Section II discusses key open challenges when it comes to testing software aging-related aspects of the behavior of virtualized environments. Next, Section III introduces the relevant technical background; Section IV presents the framework that we

propose; Section V discusses related work, and Section VI concludes the paper.

II. Challenges

In this section, we discuss key open challenges when it comes to testing software aging-related aspects of the behavior of virtualized environment including aspects such as performance, scalability, reliability, and robustness. The challenges are presented in the form of research questions. The objective of the framework presented in Section IV is to enable the addressing of the challenges.

RQ1: *How should the behavior of a virtualized environment be characterized?*

It is necessary to identify existing or develop new metrics relevant for characterizing such behavior, to evaluate the impact of hypercall execution on the behavior of a virtualized environment. These metrics can be used to characterize a baseline behavior that can be used to detect deviations. With our framework, we plan to identify metrics and workloads suitable for characterizing the behavior of a virtualized environment and that are relevant from a software aging perspective. This characterization enables the construction of tailored models for testing and characterizing relevant behavior aspects of virtualized environments. We also plan to investigate the development of new metrics.

RQ2: *How does the setup of the virtualized environment under test impact 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 feasible in a straightforward manner with a bare-metal setup. However, the use of another hypervisor to host the virtualized environment under test may impact test results and create representativeness issues. Therefore, it is essential to identify and evaluate the extent of this impact, which includes the identification of the hypercall activities, causing the impact. This identification is a challenging task considering the complexity of hypervisor operation.

RQ3: *What is the performance, reliability, and robustness impact of updates?*

Some operating systems, including hypervisors they host, are updated very frequently. 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 how updates impact the performance, reliability, and robustness of virtualized environments.

RQ4: *Are there effective rejuvenation techniques for virtualized environments?*

Testing virtualized environments for software aging-related issues, such as performance or robustness degradation, enables the identification and systematization of such issues that may occur in practice. This knowledge, in turn, enables the development of rejuvenation techniques for alleviating the issues and the assessment of the efficacy of developed techniques. For example, such a technique is adapting the number of allocated resources to a VM during operation.

III. Technical Background

A. Hypervisors and Hyper-V

In a non-virtualized scenario, physical hardware (i.e., processor, memory, and IO devices) is managed by an operating system, which provides and schedules physical resource accesses 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 can be managed by independent operating systems. This abstraction provides several advantages [8]: Running services in virtual machines rather than directly on hardware allows for higher availability, as in the case of hardware maintenance, VMs are migratable to another hypervisor with little downtime [9]. Dynamically scaling services is possible by starting or stopping VMs running instances of the application. Because VMs are isolated from each other, damage by compromised machines is limited. This isolation allows running any services alongside each other, improving the flexibility of deployment, hardware utilization, and thus operating costs.

One way to classify hypervisors is by whether they have direct control over the hardware or whether they are running on top of an operating system [10]. The former approach is called a Type-1 or bare-metal hypervisor and can utilize its full control for increased performance. Type-2 or hosted hypervisors, on the other hand, can reduce their complexity by relying on the operating system taking care of most of the hardware management.

Virtualization solutions differ by their implementation type. Generally, to virtualize a processor architecture, it is required that all control sensitive instructions affecting the processor state in a way that prevents the hypervisor from functioning correctly, are privileged instructions. These instructions generate a trap event when executed in non-privileged mode [11], allowing the hypervisor to emulate these instructions safely. The x86 architecture did, however, not comply with this requirement, as there were several non-privileged, sensitive instruction [12]. Thus, initial virtualization approaches had to make efforts to prevent their execution at all. Full virtualization allowed VMs to run the same, unmodified operating systems used on physical hardware by patching out critical instructions on the fly using binary translation [13].

In contrast, para-virtualization applied changes to the source code of operating systems themselves. These modifications

also allowed hypervisor and VMs to interact more efficiently, e.g., by using abstract IO interfaces instead of emulating existing, physical devices, leading to reduced overhead and improved performance [14]. Starting in 2005, Intel and AMD added virtualization extensions to their hardware [15], with features including an instruction set that supported trap-and-emulate wholly, an additional privilege mode for the hypervisor and hardware implementations for Second Level Address Translation (SLAT).

Nested virtualization describes the situation when a hypervisor is running inside a guest VM of another hypervisor. This nesting is useful for specific scenarios, e.g., when hypervisors have to be migratable together with their VMs or to assess virtual systems regarding performance or security [16]. Even though privileged instructions of the virtualized hypervisor have to be trapped and emulated and MMU hardware does not provide support for the third memory paging layer. Ben-Yehuda et al. [16] found a nested VM to have less than 15% overhead compared to a regular VM.

Hyper-V is an x86_64 hypervisor developed by Microsoft [17]. It is a Type-1 hypervisor. Thus, it directly controls the hardware. However, to avoid limiting it to specific hardware configurations or bloat the code base with 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 paravirtualized if they support it, but also can use unmodified operating systems, in which case Hyper-V provides emulated devices.

Hyper-V offers various interfaces for VM-Hypervisor and VM-VM communication [18]. These range from the hypervisor taking over control to handle faults accesses to privileged registers and memory addresses over the emulation of privileged instructions, IO ports and memory-mapped IO to the VMBUS, which is a memory-based communication channel for inter-VM communication, and hypercalls. Similarly to how applications can request services from the operating system by issuing system calls, guest operating systems can call into the hypervisor with hypercalls.

B. Robustness Testing

In the words of the IEEE Standard Glossary of Software Engineering Terminology, robustness is *"The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions"* [19]. Thus, robustness testing is concerned with providing unexpected inputs or conditions to the system under test, while trying to detect defects like invalid return values and application states, crashes, freezes, or performance degradations. Robustness testing applies to a multitude of different domains: operating systems [20]–[23], web services [24]–[26], Java server application [27], hypercall interfaces [28], [29], to name but a few. Common steps are the definition of a model of the interface under test, containing information about required parameters, their data types, and the tested range of possible values. When testing operating systems, a filehandle might be tested with values for handles to deleted or altered files [23]. For testing web services, validation of string parameters includes testing with a null value, an empty string, a string with non-printable characters or very long strings [24]. This model can be used to generate test cases automatically. One approach is rating irregular behavior that a system shows in response to the test case execution is to use the CRASH scale [23], [28], an acronym for the severity classes Catastrophic, Restart, Abort, Silent, and Hindering.

IV. Framework

The framework is supposed to provide 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. Due to Hyper-V's widespread application, its support of a variety of hypercalls, and the limited availability of information on its implementation and robustness characteristics, we choose it as a case study for an accurate evaluation of the 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 language features:

- **Hypercalls:** As mentioned before, the calling conventions vary across hypervisors. For them to have the same high-level representation, hypercalls and their parameters are abstractly referenced by name.
- **Order:** To provide means to test if robustness problems occur when performing specified calls in sequence, multiple hypercall can be performed in order.
- **Integer Bounds:** Boundary data type values are a regular testing input. The language supports automatic generation of signed and unsigned maximum integer values, adapted to the size of the used parameter.
- **Repetition:** Single hypercalls can specify a count value and series of calls can be wrapped in a loop with a specified count to ease the execution of a single or multiple recurring hypercalls.
- **Random:** Parameter values can also be tagged to take on a random value. This randomization enables the framework for fuzzing purposes.
- **Timing:** Timing delays can be inserted between a series of hypercalls to allow for varying load profiles. Our framework supports constant, uniformly distributed, and negative-exponentially distributed random values for delay length specification.

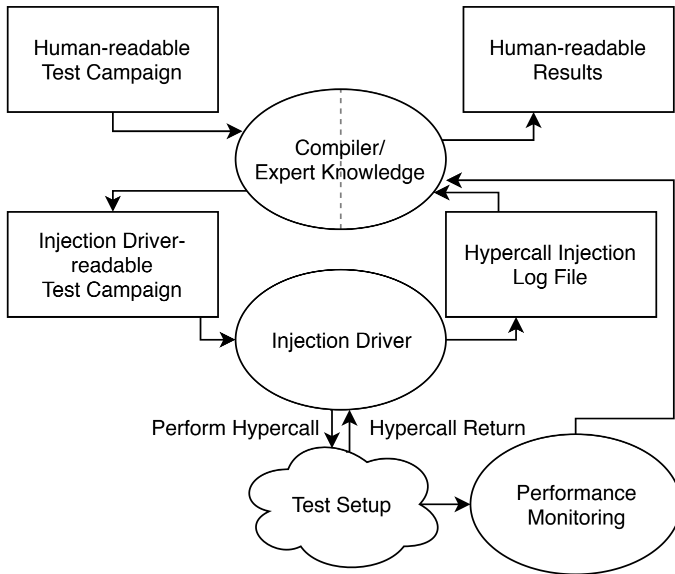

Fig. 1. Framework overview

The module that performs the hypercalls then receives an easy to parse format translated from the high-level test cases. The framework supplies a compiler that parses the JSON file and extracts the hypercalls, delays, and loops. A hypervisor-specific module translates the 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.

With the injection driver, there is no opportunity for abstraction. It reads the specially crafted campaign file and injects the hypercalls in a hypervisor-specific fashion. The general way of performing a hypercall from a 64bit VM to Hyper-V is to store the hypercall's call code in the RCX register, and pointers to memory pages in the RDX and R8 registers. The former method points to the input page, containing all parameters. The latter approach points to the output page, where the hypervisor can write return values. With everything prepared, control passes to Hyper-V by executing the VMCALL instruction on Intel processors or the VMMCALL instruction on AMD CPUs. These instructions are only executable in a privileged CPU mode. Thus, hypercalls are not injectable from userspace but have to originate from the kernel. We want to test robustness from guest VMs as well as the privileged root partition, which has to run a Windows operating system. Therefore, the selection of implementations is limited to a Windows kernel driver. A userspace application firstly loads the driver into the kernel, then supplies it with the file path of the test campaign file, using a write call to the driver's control device. The driver reads the campaign step by step. If it encounters a time delay action, it sleeps for the specified amount of time. If it encounters a hypercall action, it copies the call code and contents of input and output pages from the file to the corresponding registers and memory locations. Hyper-V provides a way to perform a transition to the hypervisor agnostic of whether it is running on an Intel or AMD platform, by overlaying a page with the correct instruction over the VM's memory. Our driver can execute the provided instruction by retrieving the overlayed memory address (physical guest address), create a virtual memory mapping for it (results in the virtual guest address), casting it to a function pointer and calling it. After the hypervisor finished processing and returned control to the VM, the hypercall result value (placed in RAX), the contents of the output page, as well as the measured execution time are written to the injection log file. This file is another detail specific to the hypervisor and injection module.

A generalized compiler can generate a humanly understandable 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 if unexpected behavior occurs while executing a test campaign. This monitoring is a part of the framework that shows many relations to the software aging topic. As stated in **RQ1**, the first step is to construct a model that describes the normal behavior of the virtualized environment under test. This model should include a set of metrics that capture all relevant aspects of a virtualized environment. There exist lots of benchmarks that are designed to measure the performance of the CPU, the memory bus, disk, and network IO or complete virtualization solutions. However, they usually run over a time frame ranging from multiple seconds to a few hours. It is necessary to measure metrics at very short intervals of probably less than a second to capture short-term performance degradations. Also, it is unclear where to place the performance monitoring agent. Can it run isolated in another guest VM? Does it have to run in injecting VM? In the case of Hyper-V, does it make sense to execute it in the root partition when performing calls a guest partition? Ideally, measurements can run in all of the locations, but there might be interactions between them, causing less predictable behavior. When the metrics are defined and measured for a baseline and a hypercall injection scenario, the model needs to decide which performance deviations are due to various effects like scheduling variations and which are likely to be caused by robustness problems. Another part of the model has to be the detection of crashes and failures. These could potentially happen to only the injecting VM, or it could affect the whole hypervisor. These differences should be detectable using agents sending heartbeat messages to a host outside the hypervisor. When only the signals of the injecting VM stop, its OS probably crashed. When additionally no messages from other VMs are received, probably the whole hypervisor is affected.

For the testing setup, there are two alternatives. Hyper-V is mentioned to illustrate the problem, but the discussion applies to all other hypervisors under tests as well. One approach is to run Hyper-V directly on the hardware, as shown in Fig. 2. This method has the advantage of being a tried and true configuration, with Hyper-V being in complete control of the hardware virtualization extensions. However, it can be

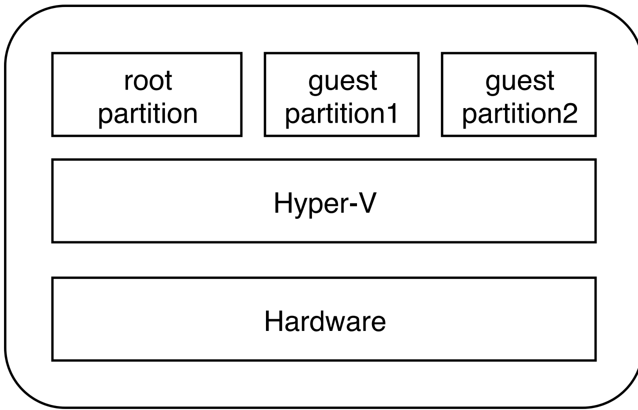

Fig. 2. Bare-metal setup

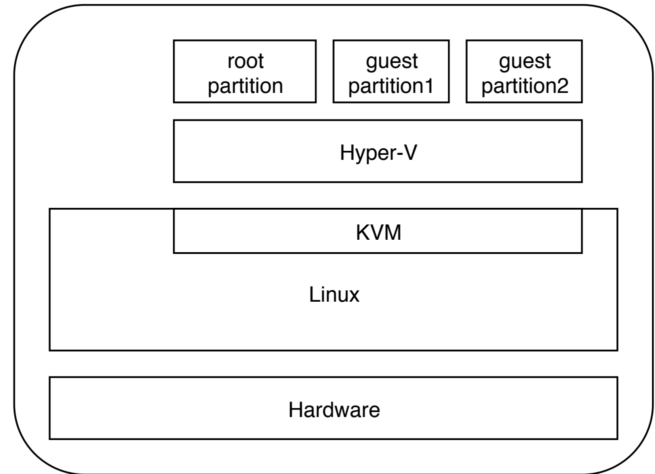

Fig. 3. Setup with nested virtualization

problematic to recover the system from defects. 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. In the best case, the system should be restored after every test campaign, at any rate, to avoid hard to trace interaction effects between campaigns. On the other hand, Hyper-V can itself be run in a virtualized environment, e.g., in a KVM virtual machine, as shown in Fig. 3. With this configuration, having a fully restored system for every test campaign is easy to achieve, e.g., by not writing to the base disk image but an overlay image. At the end of a run, the overlay image becomes disposable, and a new overlay can be created and used for the next test. Nested virtualization is the preferred solution from a test execution perspective. However, **RQ2** states the question whether the results are identical between bare-metal and nested virtualization setups or whether side effects of the VM nesting might produce results that provide a more inconsistent baseline and thus burden deviation detection.

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 versions being distributed alongside Windows OS updates and the Windows OS in the root partition a lot of relevant virtualization code, we want to evaluate how much our model varies between different versions of OS builds.

Finally, after virtualized environments can be tested for software aging-related problems and thus solving the other research questions, a final challenge remains. **RQ4** asks whether it is possible to develop countermeasures in the form of rejuvenation techniques. Such a technique could be adapting the amount of allocated resources to a VM during operation.

V. Related Work

One of the first works that brought software aging and rejuvenation to a greater audience was a report on the Patriot missile defense system. A bug in its software required frequent reboots to keep accuracy [30].

A multitude of works deals with the basics of software aging and rejuvenation [31]–[45]. Related work more specific to this paper covers two categories. The first papers take a look at hypervisors regarding software aging. The second set then tackles robustness testing of hypercall interfaces.

A. Hypervisors and Software Aging

Multiple papers address the combination of virtualized environments and software aging as well as the required rejuvenation cycles. Additionally, some of these works explicitly focus on software aging related to hypervisors.

In [4], Machida et al. present analytic models for virtual machine monitor (VMM) - their term for hypervisor - rejuvenation approaches. They model Cold-VM rejuvenation (shutting down VMs for the process), Warm-VM rejuvenation (suspending VMs for the process), and Migrate-VM rejuvenation (migrating VMs to another host for the process). Furthermore, the paper gives insight into the aging-related trigger intervals for the hypervisor rejuvenation. The authors evaluate these approaches regarding steady-state availability. The findings include that Warm-VM rejuvenation is not always superior to Cold-VM rejuvenation. If the target host has enough capacity, Migrate-VM rejuvenation outperforms the other approaches.

Matos et al. characterize software aging effects in elastic storage mechanisms in [5]. The elastic block storage (EBS) framework Eucalyptus interacts with various components. One of them is the KVM hypervisor while the other is the Eucalyptus Node Controller. The authors find that memory leaks in the node controller due to software aging-related bugs are strongly correlated to a high CPU utilization by the KVM process. Furthermore, they show that the aging effects directly impact the performance of a webserver running on the virtualized infrastructure.

Machida et al. investigate bug reports of five major open-source projects regarding software-aging in [46]. One of these

projects in the Xen hypervisor. They find that Xen has a surprisingly high number of unresolved issues. Users should be alerted to the immaturity of this software.

Pietrantuono and Russo perform a literature review on software aging in virtualized environments in [6]. Therefore, the paper summarizes the past effort conducted by the community in the cloud domain. The authors investigate model-based, measurement-based, and hybrid analysis approaches. Additionally, they present different rejuvenation techniques extracted from the reviewed material. These include Cold-VM rejuvenation, Warm-VM rejuvenation, Migrate-VM rejuvenation, VI Micro-reboot, VI Resource Management, VM Failover, and VM Restart.

Barada and Swain give a survey on software aging and rejuvenation studies in virtualized environments in [47]. From the collected information, the authors propose an algorithm to choose the correct rejuvenation technique according to the observed aging effect.

While the papers mentioned above target hypervisors or their application environments, these papers do not explore the software-aging related issues of the hypercall interfaces.

B. Hypercall Security and Robustness Testing

In [48], Milenkoski et al. present HInjector. HInjector allows interjecting hypercall attacks in a Xen-based environment while operating a paravirtualized guest VM. Therefore, XEN's code-base is adapted. The injected attacks correspond to known Xen vulnerabilities. HInjector is designed to provide an IDS with training data to protect against hypercall injections.

The same authors present an experience report on hypercall handler vulnerabilities in [49]. They motivate that publicly available information on vulnerabilities of hypercall handlers and the attacks on them is limited. This fact hinders advances in monitoring and securing said interfaces. The authors then characterize the attack surface by analyzing known vulnerabilities. They systematize and discuss the bugs causing these vulnerabilities. Next, they demonstrate attacks to trigger the vulnerabilities and propose an action plan for improving hypercall interface security.

Gonçalves et al. present an assessment of the applicability of robustness testing to the Xen hypercall interface in [28]. They devise a testing campaign through the mutation of valid hypercall invocations with invalid values. Attacks originate from a compromised machine. The results revealed frequent crashes of said machines. In some cases, the hypervisor did not detect these occurrences. The authors conclude that new failure mode scales are necessary for Xen as well as new failure detection mechanisms.

The papers above introduce the testing of hypercall interfaces. [28] explores the idea of using campaigns to define testing scenarios. While these approaches specifically target the Xen hypervisor, our framework is generalized in many parts. Also, these works do not cover the topic of software aging-related bugs as well as generating failures from seemingly valid hypercall invocations.

VI. Conclusion and Future Work

In this work, we described the open challenges regarding software aging in hypercall interfaces. From these challenges, we derived four research questions. Next, we gave an overview of 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 and make the framework publicly available. Next, we will execute extensive test campaigns on Hyper-V's hypercall interfaces using expert knowledge available inside SPEC. These campaigns will probe different versions of Hyper-V to assert the need for rejuvenation steps.

Acknowledgements

This work was funded by the German Research Foundation (DFG) under grant No. (KO 3445/16-1) and was written in Cooperation with SPEC RG Security.

References

[1] A. Avritzer and E. J. Weyuker, "Estimating the software reliability of smoothly degrading systems," in *5th International Symposium on Software Reliability Engineering, ISSRE 1994, Monterey, CA, USA, November 6-9, 1994*, 1994, pp. 168–177. [Online]. Available: https://doi.org/10.1109/ISSRE.1994.341370

[2] Y. Huang, C. Kintala, N. Kolettis, and N. Dudley Fulton, "Software rejuvenation: Analysis, module and applications," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, 07 1995, pp. 381–390.

[3] M. Grottke, R. Matias Jr, and K. Trivedi, "The fundamentals of software aging," in *Software Reliability Engineering Workshops, 2008. ISSRE Wksp 2008. IEEE International Conference on*, 12 2008, pp. 1 – 6.

[4] F. Machida, D. S. Kim, and K. S. Trivedi, "Modeling and analysis of software rejuvenation in a server virtualized system," *2010 IEEE Second International Workshop on Software Aging and Rejuvenation*, pp. 1–6, Nov 2010.

[5] R. Matos, J. Araujo, V. Alves, and P. Maciel, "Characterization of Software Aging Effects in Elastic Storage Mechanisms for Private Clouds," *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, pp. 293–298, Nov 2012.

[6] R. Pietrantuono and S. Russo, "Software aging and rejuvenation in the cloud: a literature review," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 257–263.

[7] "Work Package 6: Virtual Secure Mode," 2019, [Online; accessed 1. Aug. 2019]. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Cyber-Sicherheit/SiSyPHus/Workpackage6_Virtual_Secure_Mode.pdf?__blob=publicationFile&v=2

[8] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *2010 Second International Conference on Computer and Network Technology*. IEEE, 2010, pp. 222–226.

[9] L. Iffländer, F. Wamser, C. Metter, P. Tran-Gia, and S. Kounev, "Performance Assessment of Cloud Migrations from Network and Application Point of View," in *Proceedings of 9th EAI International Conference on Mobile Networks and Management (MONAMI 2018)*, December 2017.

[10] Z. Gu and Q. Zhao, “A state-of-the-art survey on real-time issues in embedded systems virtualization,” *Journal of software Engineering and Applications*, vol. 5, no. 04, p. 277, 2012.

[11] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[12] U. A. Force, “Analysis of the intel pentiums ability to support a secure virtual machine monitor,” in *Proceedings of the... USENIX Security Symposium. USENIX Association*, 2000, p. 129.

[13] D. Marshall, “Understanding full virtualization, paravirtualization, and hardware assist,” *VMWare White Paper*, p. 17, 2007.

[14] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, “Full and para-virtualization with xen: a performance comparison,” *Journal of Emerging Trends in Computing and Information Sciences*, vol. 4, no. 9, pp. 719–727, 2013.

[15] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, “Intel virtualization technology,” *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[16] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har’El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, “The turtles project: Design and implementation of nested virtualization.” in *Osdi*, vol. 10, 2010, pp. 423–436.

[17] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, “Benchmarking the performance of microsoft hyper-v server, vmware esxi and xen hypervisors,” *Journal of Emerging Trends in Computing and Information Sciences*, vol. 4, no. 12, pp. 922–933, 2013.

[18] N. Joly and J. Bialek, “A Dive in to Hyper-V Architecture and Vulnerabilities,” Aug 2018, [Online; accessed 6. Aug. 2019]. [Online]. Available: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2018_08_BlackHatUSA/A%20Dive%20in%20to%20Hyper-V%20Architecture%20and%20Vulnerabilities.pdf

[19] IEEE, “IEEE standard glossary of software engineering terminology,” *IEEE Std 610.12-1990*, Dec. 1990.

[20] C. P. Shelton, P. Koopman, and K. DeVale, “Robustness testing of the microsoft win32 api,” in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. IEEE, 2000, pp. 261–270.

[21] A. K. Ghosh, M. Schmid, and V. Shah, “Testing the robustness of windows nt software,” in *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No. 98TB100257)*. IEEE, 1998, pp. 231–235.

[22] P. Koopman and J. DeVale, “Comparing the robustness of posix operating systems,” in *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*. IEEE, 1999, pp. 30–37.

[23] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, “Comparing operating systems using robustness benchmarks,” in *Proceedings of SRDS’97: 16th IEEE Symposium on Reliable Distributed Systems*. IEEE, 1997, pp. 72–79.

[24] M. Vieira, N. Laranjeiro, and H. Madeira, “Assessing robustness of web-services infrastructures,” in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’07)*. IEEE, 2007, pp. 131–136.

[25] ——, “Benchmarking the robustness of web services,” in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*. IEEE, 2007, pp. 322–329.

[26] E. Martin, S. Basu, and T. Xie, “Websob: A tool for robustness testing of web services,” in *Companion to the proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 65–66.

[27] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott, “Robustness testing of java server applications,” *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 292–311, 2005.

[28] C. F. Gonçalves, N. Antunes, and M. Vieira, “Evaluating the applicability of robustness testing in virtualized environments,” in *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 2018, pp. 161–166.

[29] D. Carvalho, N. Antunes, M. Vieira, A. Milenkoski, and S. Kounev, “Challenges of assessing the hypercall interface robustness (fast abstract),” in *The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*, 2015.

[30] G. report, “Gao - patriot missile software problem,” http://fas.org/spp/starwars/gao/im92026.htm, 1991, accessed: 2014-09-07.

[31] M. Grottke, A. Nikora, and K. Trivedi, “An empirical investigation of fault types in space mission system software,” in *Dependable Systems and Networks (DSN), 2010 Int’l. Conf.*, 2010.

[32] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, “Fault triggers in open-source software: An experience report,” in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, 2013, pp. 178–187.

[33] F. Qin, Z. Zheng, X. Li, Y. Qiao, and K. S. Trivedi, “An empirical investigation of fault triggers in android operating system,” in *22nd IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2017, Christchurch, New Zealand, January 22-25, 2017*, 2017, pp. 135–144. [Online]. Available: https://doi.org/10.1109/PRDC.2017.27

[34] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert, “Proactive management of software aging,” *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 311–332, 2001.

[35] F. Machida, D. S. Kim, and K. S. Trivedi, “Modeling and analysis of software rejuvenation in a server virtualized system with live VM migration,” *Perform. Eval.*, vol. 70, no. 3, pp. 212–230, 2013.

[36] F. Machida, V. F. Nicola, and K. S. Trivedi, “Job completion time on a virtualized server with software rejuvenation,” *JETC*, vol. 10, no. 1, p. 10, 2014.

[37] J. Zhao, Y. Jin, K. S. Trivedi, R. M. Jr., and Y. Wang, “Software rejuvenation scheduling using accelerated life testing,” *JETC*, vol. 10, no. 1, p. 9, 2014.

[38] J. Zhao, Y. Wang, G. Ning, K. S. Trivedi, R. M. Jr., and K. Cai, “A comprehensive approach to optimal software rejuvenation,” *Perform. Eval.*, vol. 70, no. 11, pp. 917–933, 2013.

[39] K. Trivedi, R. Mansharamani, D. Kim, M. Grottke, and M. Nambiar, “Recovery from failures due to mandelbugs in it systems,” in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 224–233.

[40] M. Grottke and K. S. Trivedi, “Fighting bugs: Remove, retry, replicate, and rejuvenate,” *Computer*, vol. 40, no. 2, pp. 107–109, Feb. 2007.

[41] M. Grottke and K. Trivedi, “Software faults, software aging and software rejuvenation,” *J. Reliab. Eng. Ass. JPN*, vol. 27, no. 7, 2005.

[42] K. Vaidyanathan and K. Trivedi, “A comprehensive model for software rejuvenation,” *Dependable and Secure Computing, IEEE Transactions on*, vol. 2, no. 2, pp. 124–137, April 2005.

[43] J. Alonso, M. Rivalino, E. Vicente, A. Maria, and K. Trivedi, “A comparative experimental study of software rejuvenation overhead,” *Perform. Eval.*, vol. 70, no. 3, pp. 231–250, 2013.

[44] T. Dohi, K. Goseva-Popstojanova, and K. S. Trivedi, “Estimating software rejuvenation schedules in high-assurance systems.” *Comput. J.*, vol. 44, no. 6, pp. 473–485, 2001.

[45] K. Vaidyanathan and K. Trivedi, “A measurement-based model for estimation of resource exhaustion in operational software systems,” in *Proc. of the Tenth Int. Symp. on Soft. Rel. Engineering*, Nov. 1999, pp. 84–93.

[46] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, “Aging-Related Bugs in Cloud Computing Software,” *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, pp. 287–292, Nov 2012.

[47] S. Barada and S. K. Swain, “A survey report on software aging and rejuvenation studies in virtualized environment,” *Int J Comput Eng Technol (IJCSET)*, vol. 5, no. 5, pp. 541–546, 2014.

[48] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, S. Kounev, A. Avritzer, and M. Luft, “Evaluation of Intrusion Detection Systems in Virtualized Environments Using Attack Injection,” in *The 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2015)*. Springer, November 2015. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-26362-5_22

[49] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, and S. Kounev, “Experience report: an analysis of hypercall handler vulnerabilities,” in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 100–111.