2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) | 978-1-6654-2603-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSREW53611.2021.00043

Software Testing Strategies for Detecting Hypercall Handlers' Aging-related Bugs

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

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

Aleksandar Milenkoski
Cybereason
Germany
milenkoski.aleksandar@gmail.com

Alberto Avritzer
eSulab Solutions
Princeton, New Jersey, USA
beto@esulabsolutions.com

Nuno Antunes
Department Of Informatics Engineering
University of Coimbra
Coimbra, Portugal
nmsa@dei.uc.pt

Samuel Kounev
Chair of Software Engineeriung
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 testing to ensure their long-term reliability. Existing research deals with the aging properties of hypervisors in general without considering the hypercalls. In this work, we share our experience that we collected during trying to understand hypercalls and their parameters and use them to construct test cases for hypervisor aging of Microsoft Hyper-V. We present a bug that we detected, which was reported and acknowledged by Microsoft. Further, based on our manual binary code analysis, we propose the idea of automating the analysis process to detect valid parameter ranges and execution conditions of hypercalls without manual effort.**

***Index Terms*—software aging, hypervisor, hypercalls, Hyper-V**

1. Introduction

Today, hypervisors are virtually omnipresent. They are widespread throughout data centers, functioning as the backbone of cloud computing [1] because they allow for server consolidation with huge benefits in efficiency and flexibility. Hypervisors are also prevalent in the modern desktop and workstation infrastructure [2]. This extends to Microsoft shipping their Hyper-V hypervisor directly with many versions of the Windows operating system, and in some cases, even activating it by default [3]. Furthermore, virtualization applies to embedded computing systems, as well. Amongst other examples, the automotive and aerospace domains take advantage of abstracting computing resources [4], [5].

Virtualization 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 providing call-based connectivity to hosted VMs that are aware of being virtual. One of them is the hypercall interface, allowing for VMs to request services from the hypervisor. Hypercalls are software traps from a VM to the hypervisor. Before introducing x86 hardware virtualization in 2006, hypercalls were one solution to run virtualized operating systems. Nowadays, while technically not required, hypercalls are still a common utility to improve efficiency or offer additional features. Microsoft's hypervisor Hyper-V requires an identifying call code and often additional parameters. After processing a hypercall, the hypervisor returns a result value and possibly output values to the caller. Other hypervisors behave very similarly.

The crucial role that hypervisor play in today's infrastructure requires robustness, dependability, insusceptibility to software aging, and high performance. Despite that, there is a gap in the research community concerned with testing these properties, which is necessary for enabling well-informed and unbiased decision-making regarding computing system deployment, configuration, and use.

For now, we chose to focus on the Hyper-V hypervisor for multiple reasons. Hyper-V is relevant due to its widespread use. It is the hypervisor used in the Azure Cloud and shipped with many editions of Windows 10, the Operating System (OS) running on more than 58% of desktop computers as of April 2021 [6], [7]. Also, it is the basis of Microsoft's Virtualization-Based Security (VBS), which is using virtualization to isolate critical data, e.g., credentials or encryption secrets, from the Windows OS, such that even an attacker with administrator privileges cannot access them. Communication between the regular OS and the secure kernel is realized with hypercalls. Further, Hyper-V is proprietary software with limited public documentation. We share our experience testing this hypervisor, demonstrating challenges not present with open-source hypervisors and

978-1-6654-2603-9/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSREW53611.2021.00043

how we overcome them. The contributions of this paper are:

- Discussing strategies to build hypervisor aging testing scenarios consisting of hypercalls
- Sharing our experience of learning how to use Hyper-V hypercalls, especially which values have to be passed for their parameters
- Presenting results of our work, namely a hypervisor crash caused by a software aging-inspired test scenario.
- Proposing the idea of automated code analysis of hypercall handlers to learn about valid parameter ranges and environmental executing conditions more quickly.

The remainder of this paper is structured as follows: Section 2 provides background information about virtualization, Hyper-V, hypercalls, and operating system/hypervisor debugging. Next, Section 3 discusses the strategies for constructing aging tests; Section 4 presents the hypercall-related crashing bug; Section 5 proposes the automated hypercall handler code analysis; Section 6 discusses related work, and Section 7 concludes the paper.

2. Background

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 to take 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 instructions [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 (also called guest partitions) can run para-virtualized 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. Similar to how applications can request services from the operating system by issuing system calls, guest operating systems can call into the hypervisor with hypercalls.

Hypercalls are triggered by special processor instructions. Because Intel and AMD processors have different virtualization extensions to the x86 architecture, the hypercall instructions also vary. Hyper-V prevents the operating

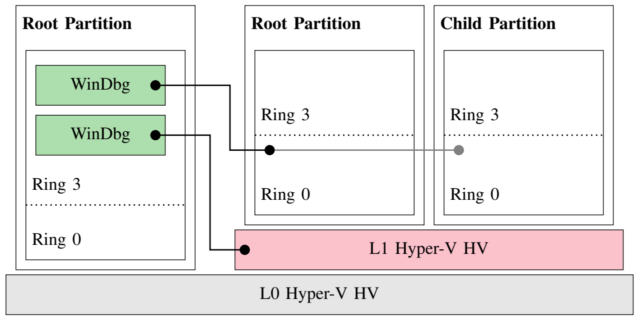

Figure 1. Hyper-V - Debugging Environment [19]

systems in its VMs from dealing with this by overlaying a memory page into the VM, where the correct instructions to perform a hypercall are stored.

A huge advantage when experimenting with hypercalls in Hyper-V is the possibility to debug the hypervisor or Windows instances running as virtual machines. Logically, a machine cannot debug itself. Therefore, a debugger system is required, which can debug the debuggee system. It is possible to use two physical machines for this, but utilizing nested virtualization allows for debugging the nested Hyper-V hypervisor and its partitions. This nested setup is displayed in Figure 1. Two instances of WinDbg, the Windows debugger, run on a virtual machine and wait for the debuggees to connect. Windows and Hyper-V can be configured to connect to a debugger on startup, either via serial or Ethernet connection. Once connected, the debugger can stop the execution of the debuggee, set breakpoints, inspect registers and memory, identical to how usually single applications are debugged.

3. Testing Strategies

One way to utilize hypercalls for testing the software aging behavior of hypervisors is to replicate hypercall workloads that happen at a slow pace during normal operation. If such a workload is susceptible to software aging, artificially replaying it at a considerably higher speed yields chances to detect the aging effects in a reasonable time frame. It is important to note that not all anomalies detected during "accelerated aging" testing are necessarily related to aging. The hypervisor might be in a different state than when a hypercall scenario occurs in regular operation. E.g., replicating hypercalls that send messages or signal events to another virtual machine will likely show differing behavior due to the recipient not anticipating the input. Also, the rapid repetitions of certain hypercalls might be stressful and problematic for the hypervisor, even though at a slower execution pace, no issues may appear. The results in Section 4 highlight a finding that is most probably of this nature.

So, the testing scenarios should be well defined and understood. Specifically, this requires knowledge of the individual hypercalls, their behavior, and parameters. The remainder of this section discusses different techniques to find suitable hypercalls and determine how to perform them properly. First off, monitoring hypercalls is a promising technique to find both, 1) hypercalls that happen during normal operation and are potential aging scenario candidates, and 2) the parameters that have to be passed to them. There are multiple locations where hypercall execution can be intercepted, both in the OS of a VM as well as directly hypervisor. So far, we have tried to use automated kernel debugging to log hypercalls of a VM. While we were able to log some hypercalls, the active measurement caused performance counters to show an increased rate of hypercalls, which also did not match with our capture rate. Considering all the uncertainties, it is unreasonable to treat the results as representative hypercall loads, and further research regarding hypercall monitoring is a part of future research. However, even with perfectly functional monitoring, scenario identification is not a trivial task. As mentioned before, previously working calls might not succeed with a different state of the hypervisor. An example is the creation of a new VM. Among other calls, for every virtual processor, `HvCreateVirtualProcessor` is executed. Here, the partition ID of the new VM has to be specified. Thus, repeatedly creating new virtual machines requires adjusting the partition ID to the current VM every time.

A further approach is the manual construction of testing campaigns with the help of documented information. While the majority of all Hyper-V hypercalls are not documented in detail, at least all their names are public and usually descriptive of their functionality. A review of the 101 hypercalls presents some clear candidates for aging testing. There are several hypercall pairs that deal with the creation and deletion of certain entities, e.g., of partitions and virtual processors (mentioned already), but also event log buffers and buffer groups. Surely it is possible to construct aging scenarios from other hypercalls; however, with these calls, the testing campaign is straightforward: Repeatedly issue the creation and deletion calls. This call selection also provides a high probability of the scenario occurring during regular operation since the calls are likely to be executed at least from time to time. What remains is the determination of the parameters to pass for successful execution. Very helpful is a detailed explanation provided by Microsoft's documentation [18]. For some calls, there exist detailed information, including descriptions of their parameters. Unfortunately, our aging-suitable designated calls are not mentioned. It can be beneficial to take a look at the revisions of Hyper-V documentation for older versions. They can contain information that was dropped for the more recent revisions. So, while there is no mention of the hypervisor's event logging interfaces in the Top-Level Functional Specification (TLFS) from 2016 or 2019, the one from 2012 provides an explanation, including descriptions of the hypercalls that create and destroy event log buffers and buffer groups. Such information is certainly very useful in getting a grasp of the functionality of calls. It has to be taken with a grain of salt, though. It is possible that implementation details changed over the years. We encountered this with the call `HvInitializeEventLogBufferGroup`. The TLFS specifies four bytes to be the parameter `BufferPages`, which specifies how many memory pages each buffer of the group is supposed to have. We found that the recent Hyper-V builds instead require the number of bytes to be passed (has to be a multiple of the page size, 4096 bytes). This is a small yet significant difference, and it is not impossible for more drastic changes to have occurred over time.

Most hypercalls that have no mention anywhere in some documentation, the parameter names and sizes can be looked up in an old, published Hyper-V header file that defines structs for the input and output of calls [20]. Naturally, the details might have changed since its publication. The aforementioned `BufferPages` parameter is here called `BufferSizeInPages`, which does not reflect the current implementation (it should be `BufferSizeInBytes`).

Understanding what parameters describe is certainly helpful, but in the end, it is important to know which values have to be passed. Blindly testing all parameters and checking the returned result value is a theoretically possible approach. However, in practice, there are significant drawbacks. First of all, the parameter space is usually vast. Hypercalls usually have multiple input values, which sometimes are 4-byte values, most of the time 8-byte values. Our performance measurements showed that it is possible to inject more than one million hypercalls per second, but no matter the speed, exploring even a single 8-byte value's possibilities takes too long. We have noticed that many valid values are rather small, but sometimes flags are bit-wise encoded into a parameter, resulting in large integer values if one of the higher bits is set. In any way, again, knowledge about the calls and parameters can aid in shrinking the exploration space into something manageable. Still, some hypercalls might have significant effects on the system, preventing the exploration campaign from continuing and causing incomplete results. Especially privileged calls are likely to cause such issues.

One way to identify the correct parameter values is to intercept hypercalls performed by the operating system and inspect their parameter values. This is a little bit different from the previously described hypercall logging, where we were interested in seeing which different calls are happening. Now, we are interested in a singular call. Using a generic breakpoint on the hypercall page and manually debugging is a tedious task because even during idle, hundreds of hypercalls happen per second, each triggering this breakpoint. Automating the debugging to restart execution until it sees the correct hypercall code is a viable solution, which we have not tested yet. Instead, we set breakpoints into functions of the winhv.sys driver. These functions are essentially wrappers that take parameters like normal C functions, then prepare the registers and memory for the hypercall, and finally call into the hypercall page. A breakpoint set here is only triggered when the OS calls the hypercall using the winhv wrapper function.

To intercept the hypercall, it is, of course, necessary that the call is issued at some point. When researching the parameters of `HvCreateVp`, a significant advantage was that we could trigger calls on demand. Starting a virtual machine using the default Hyper-V user interface causes `HvCreateVp` to be called for each virtual processor of the VM. Thus, we could quickly understand the numbering scheme of partition and processor IDs, and learn that the other two parameters (`Flags` and `ProximityDomainInfo`) are always set to 0, in our case. Unfortunately, this is not the case for all calls. In the case of `HvCreateEventLogBuffer`, we were not able to intercept this hypercall in any condition.

Finally, the method that is very likely to provide a lot of insight, but is also the most tedious, is reverse engineering of the hypervisor executable. Our reverse engineering efforts of the hypercall handling process in Hyper-V showed that at first, general preprocessing happens, e.g., checking that the partition has correct access rights to the memory pages it designated for the input and output values. Then, a hypercall-specific handler routine is called before some general post-processing happens and execution control is given back to the virtual machine. In this case, the call-specific handler is of interest since there the parameter checks are performed. Fortunately, the memory location of a call handler is easy to find. In the CONST memory section of the Hyper-V executable exists a table containing hypercall information, e.g., call code, number of parameters, and most importantly here: address of the handler routine.

We illustrate the structure of these handlers by explaining what is happening in the handler of `HvCreateEventLogBuffer`. At first, some stack memory is initialized for local variables. Then, the registers are prepared for a function call. The function retrieves a pointer to the memory structure with information about the current partition (that issued the call). At a certain offset, a bit is checked - to test if it is the root partition. If not, the function returns with value 6 (`HV_STATUS_ACCESS_DENIED`). In case of success, the first parameter of the call is inspected. It is called `EventLogType`, and according to hvgdk.h, 0, 1, and 2 are valid values [20]. On the contrary, the assembly code clearly shows that only 0 and 1 are allowed; all values above result in the function returning 5 (`HV_STATUS_INVALID_PARAMETER`). Depending on the `EventLogType`, a memory address is calculated and written to a pointer that was passed as a parameter; the rest of the handler indicates that this is a struct storing information about the `EventLogBufferGroup`, of which one exists for each `EventLogType`.

The function returns with either 6, 5, or 0 (HV_STATUS_SUCCESS. The handler checks for the return value to be 0; otherwise, it terminates and forwards the function return value as the result value of the hypercall. This is followed up by many more checks of the parameters; some against static values, where it is straightforward to retrieve the possible ranges, and some against dynamic values in memory. As mentioned before, the function called at the start calculated the address of the corresponding `EventLogBufferGroup`. Now, it is looked up if the index of the new buffer is free or if there already exists a buffer with the same index in the

TABLE 1. PARAMETERS OF HVCREATEVP

Parameter Name	Size [byte]
PartitionId	8
VpIndex	4
Padding	4
ProximityDomainInfo	8
Flags	8

current buffer group. In such cases, it is very beneficial to utilize hypervisor debugging for injecting the hypercall, breaking at its handler, stepping through the instructions, and inspecting which values the memory locations take on during execution. Using the IDA Pro disassembler together with hypervisor debugging allowed us to understand the event log buffer- and buffer group-related hypercalls enough to successfully execute them and construct scenarios with them.

4. Results

In this section, we present our results of creating and running a software aging campaign using the hypercalls `HvCreateVp` and `HvDeleteVp`, which create and delete virtual processors of guest partitions.

Table 1 lists the parameters of `HvCreateVp`, as given by [20]. It is intuitive that `PartitionId` should be an identifier of the VM which the processor should be created for. Another reasonable assumption is that the processors of a partition are numerated, and `VpIndex` specifies which Virtual Processor (VP) to create. `ProximityDomainInfo` is a common term in multiprocessor systems with Non-Uniform Memory Access (NUMA). Setting bits in `flags` probably corresponds to specifying additional options on how to create the VP.

The parameters of the corresponding deletion call are even simpler; only `PartitionId` and `VpIndex` have to be passed.

The calls can only be successfully performed by the privileged root partition. It is unknown and difficult to find out whether the root partition uses these calls to manage its own VPs. Thus, we planned the following testing scenario: Launch a VM (normally, using the official Hyper-V manager), and then continuously create and delete additional processors. There are multiple potential issues. Maybe, there cannot be more cores created than configured. If it is possible, the operating system might behave unexpectedly due to the hot-plug of another processor (the TLFS states this will not occur [18]).

As already mentioned in Section 3, using the debugger and breakpoints while starting a virtual machine allowed for the interception and inspection of real `HvCreateVp` calls. We learned that in our configuration, `ProximityDomainInfo` and `Flags`, were always set to zero. Regarding the partition's identifier, we noticed that the first VM started after a reboot of the hypervisor receives an ID of 2, the next an ID of 3, and so on. Similarly, if a partition with n virtual processors is started, there are n `HvCreateVp` calls with `VpIndex` ranging from 0 to $n-1$.

The next step was the verification that more processors could actually be created. We rebooted the whole system, launched a VM with four VPs. Thus, the new partition received the ID 2, and cores 0 to 3 are created.

Listing 1. Virtual Processor Aging Campaign

```
proc main() {
    hcall(["name" -> "HvCreateVp",
           "PartitionId" -> 2,"VpIndex" -> 4]);
}
```

Listing 1 shows the campaign used for verification. It is written in a custom hypercall description language, but even without knowledge, it should be understandable which calls should be executed. All non-specified parameters are set to 0 by default. Executing the verification campaign in the root partition and logging the result value confirms that VP 4 can be successfully created. Also, the OS running in the virtual machine is not affected. Executing the campaign again lets the call fail with a result value of 14 (`HV_STATUS_INVALID_VP_INDEX`). Now executing the corresponding deletion campaign succeeds on the first try, and also fails with code 14 upon the second invocation. Further testing reveals that only one additional core can be added. All following creation calls fail with code 11 (`HV_STATUS_INSUFFICIENT_MEMORY`). The index is however choosable, e.g., it is possible to create VP 7, skipping cores 4 to 6.

With the ability to repeatedly create and delete a single additional virtual processor, we created the campaign displayed in Listing 2.

Listing 2. Virtual Processor Aging Campaign

```
PID = 2;
VPI = 4;

proc main() {
  for (i : range(0, 10000)) {
    hcall(["name" -> "HvCreateVp",
           "PartitionId" -> PID,"VpIndex" -> VPI]);
    hcall(["name" -> "HvDeleteVp",
           "PartitionId" -> PID,"VpIndex" -> VPI]);
  }
}
```

This campaign is then executed repeatedly, such that there is a constant load of VP-related hypercalls.

So, we observed that there is just enough memory to allocate one further virtual processor, which is an indicator for static memory management, which is by design less susceptible to problems arising when aging. Despite that, we executed the campaign repeatedly and actually discovered a problem rather quickly. After some time, the whole system crashed, displaying a bluescreen that indicated an error has occurred in the hypervisor.

We reproduced the problems numerous times. In one case, the crash happened in the 49. campaign execution, meaning there have between 490,000 and 500,000 creations and deletion performed. However, another time, the problem did only occur after 20,747 successfully finished iterations, after more than 200 billion creations and deletions.

Further testing included increasing the campaign to now perform 100,000 repetitions. This resulted in the bluescreen appearing very quickly, typically within the first execution of the campaign. It is reasonable to assume that the higher, more stressful hypercall load caused by less frequent campaign restarting overhead increases the likelihood of crashing. This evidence suggests that we are not dealing with an aging-related problem but rather discovered a flaw in the robustness of Hyper-V.

This issue has also been reported to Microsoft, who could reproduce and confirm the issue, stating the behavior will be corrected.

5. Automated Analysis

Reverse engineering hypercall handlers to understand their functionality and learn their valid parameter ranges is a very general and powerful approach. There is no risk of depending on the availability or correctness of information as there is with documentation since all of the information is necessarily contained in the binary code that implements the call handlers.

Unfortunately, it is not trivial to extract and understand the contained information. Starting off, it is unclear where anything is located in the memory of the hypervisor executable. Manually reverse-engineering the execution path that a hypercall takes through the hypervisor showed us which instructions are part of general pre-and post-processing and where to find the call-specific handler routines.

With this, it is feasible to determine parameters for a given hypercall. Using the call code, the memory address of the handler routine is looked up. Then, going through the handler instruction by instruction can build an understanding of the parameter checks occurring. Depending on the researcher's assembly experience, this process is usually tedious and takes considerable time. Also, a human reverse engineer is prone to misinterpret instructions and draw false conclusions.

We propose the idea of using automated code analysis for this task. This should be feasible, as the hypercall handlers are very constrained environments. At the start of the handler, the ECX register holds a memory address where the parameters are located. Then, the usual process is that the handler retrieves the parameters and compares them to either static values or values retrieved from other memory locations. This is where the execution branches. On the one branch (success), more checks happen; on the other branch (failure), the handlers usually only set the EAX register to an error status code and return back to the general post-processing. This is how a human determines which parameter values lead to which errors, and following the same algorithm, an automated analysis tool can as well.

Comparisons with dynamic values from memory pose a problem, as the values are only known at runtime, not during static analysis of a binary. Theoretically, the automated analysis tool could also use a live system, a debugger, inject hypercalls, intercept with breakpoints and inspect the memory locations' values at runtime. However, it is crucial to keep in mind that these values may be different when performing calls at a different time or on a different machine.

Apart from the parameter values, other conditions can be found. In the case of `HvCreateEventLogBuffer` we saw that it checked at a specific memory location whether the partition executing the call was the root partition. Investing in manual reverse engineering and labeling such common memory locations can be beneficial for the amount of information the automated analysis can obtain.

The result of the analysis can be a compact report of the conditions found for executing the hypercall, including valid parameter ranges and environmental conditions (e.g., caller is root partition, caller has this or that privilege flag). As a benefit, less time has to be spent on reverse engineering hypercall handlers, and more time can be dedicated to constructing and running software aging testing campaigns.

6. 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 [21].

A multitude of works deals with the basics of software aging and rejuvenation [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36].

Related works more specific to this paper take a look at hypervisors regarding 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 [37], 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 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 [38]. 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 [39]. One of these projects is 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 [40]. 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 [41]. 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.

7. Conclusion

In this work, we shared our experience regarding the construction of hypervisor aging test cases consisting of hypercalls. Due to issues with hypercall monitoring and no possibility to get a representative overview of hypercalls happening during normal execution, we suggested the construction of scenarios using create-delete pairs. Information about the calls and their parameters can be retrieved from official documentation, older versions of the documentation, a published old header file. Blindly testing out hypercall parameters is impractical, but with previously gathered knowledge, the parameter exploration space can be significantly reduced. We described the use of debugging and static code analysis to find valid parameter ranges and environmental execution conditions.

After the testing strategy, we discussed the results we discovered testing with `HvCreateVp` and `HvDeleteVp`, and displayed the scenario that resulted in crashing the hypervisor, which was later confirmed and acknowledged by Microsoft.

Finally, based on our manual reverse engineering experience and the recognition of how limited of an execution environment the hypercall handlers are, we proposed the idea of using automated analysis to retrieve valid parameter ranges and environmental execution conditions automatically. Dynamic analysis using a live system and debugger can help get insight into values retrieve dynamically from memory. In the future, we plan to follow this idea and create an implementation.

Acknowledgements

This work was written in Cooperation with SPEC RG Security.

References

[1] S. Srivastava and S. Singh, "A survey on virtualization and hypervisor-based technology in cloud computing environment," *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, vol. 5, no. 2, 2016.

[2] K. Miller and M. Pegah, "Virtualization: virtually at the desktop," in *Proceedings of the 35th annual ACM SIGUCCS fall conference*. ACM, 2007, pp. 255–260.

[3] "Virtualization-Based Security: Enabled by Default," https://techcommunity.microsoft.com/t5/Virtualization/Virtualization-Based-Security-Enabled-by-Default/ba-p/890167, accessed: 2019-10-27.

[4] D. Reinhardt and G. Morgan, "An embedded hypervisor for safety-relevant automotive e/e-systems," in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. IEEE, 2014, pp. 189–198.

[5] A. Crespo, I. Ripoll, M. Masmano, P. Arberet, and J. Metge, "Xtratum an open source hypervisor for tsp embedded systems in aerospace," *Data Systems In Aerospace DASIA, Istanbul, Turkey*, 2009.

[6] "Desktop Operating System Market Share Worldwide," https://gs.statcounter.com/os-market-share/desktop/worldwide, accessed: 2021-15-05.

[7] "Desktop Windows Version Market Share Worldwide," https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide, accessed: 2021-15-05.

[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. IEEE, 2010, pp. 222–226.

[9] L. Iffländer, C. Metter, F. Wamser, P. Tran-Gia, and S. Kounev, "Performance Assessment of Cloud Migrations from Network and Application Point of View," in *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Melbourne, Australia: Springer International Publishing, Dec. 2018, pp. 262–276.

[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. 05, no. 04, pp. 277–290, 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, Jul. 1974.

[12] U. S. A. Force, "Analysis of the intel pentium's 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. M. 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, May 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] "Hyper-V Top Level Function Specifications," Aug. 2019, [Online; accessed 5. May. 2021]. [Online]. Available: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/tlfs

[19] Microsoft Security Response Center, "First Steps in Hyper-V Research," 2018, https://msrc-blog.microsoft.com/2018/12/10/first-steps-in-hyper-v-research/, accessed on 01.11.2020.

[20] "hvgdk.h on GitHub," Aug. 2020, [Online; accessed 5. May. 2021]. [Online]. Available: https://github.com/ionescu007/hdk/blob/master/hvgdk.h

[21] G. A. O. report, "Gao - Patriot Missile Software Problem," http://fas.org/spp/starwars/gao/im92026.htm, 1991, accessed: 2014-09-07.

[22] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An Empirical Investigation of Fault Types in Space Mission System Software," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, Jun. 2010.

[23] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault Triggers in Open-source Software: An Experience Report," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Nov. 2013, pp. 178–187.

[24] F. Qin, Z. Zheng, X. Li, Y. Qiao, and K. S. Trivedi, "An Empirical Investigation of Fault Triggers in Android Operating System," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, Jan. 2017, pp. 135–144. [Online]. Available: https://doi.org/10.1109/PRDC.2017.27

[25] 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, Mar. 2001.

[26] F. Machida, D. S. Kim, and K. S. Trivedi, "Modeling and Analysis of Software Rejuvenation in a Server Virtualized System with Live VM Migration," *Performance Evaluation*, vol. 70, no. 3, pp. 212–230, Mar. 2013.

[27] F. Machida, V. F. Nicola, and K. S. Trivedi, "Job Completion Time on a Virtualized Server with Software Rejuvenation," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 10, no. 1, pp. 1–26, Jan. 2014.

[28] J. Zhao, Y. Jin, K. S. Trivedi, R. M. Jr., and Y. Wang, "Software Rejuvenation Scheduling Using Accelerated Life Testing," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 10, no. 1, pp. 1–23, Jan. 2014.

[29] J. Zhao, Y. Wang, G. Ning, K. S. Trivedi, R. Matias, and K.-Y. Cai, "A Comprehensive Approach to Optimal Software Rejuvenation," *Performance Evaluation*, vol. 70, no. 11, pp. 917–933, Nov. 2013.

[30] K. S. Trivedi, R. Mansharamani, D. S. Kim, M. Grottke, and M. Nambiar, "Recovery from Failures Due to Mandelbugs in IT Systems," in *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, vol. 0. Los Alamitos, CA, USA: IEEE, Dec. 2011, pp. 224–233.

[31] M. Grottke and K. S. Trivedi, "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, Feb. 2007.

[32] M. Grottke and K. Trivedi, "Software Faults, Software Aging and Software Rejuvenation," *J. Reliab. Eng. Ass. JPN*, vol. 27, no. 7, 2005.

[33] K. Vaidyanathan and K. S. Trivedi, "A Comprehensive Model for Software Rejuvenation," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 124–137, Feb. 2005.

[34] J. Alonso, R. Matias, E. Vicente, A. Maria, and K. S. Trivedi, "A Comparative Experimental Study of Software Rejuvenation Overhead," *Performance Evaluation*, vol. 70, no. 3, pp. 231–250, Mar. 2013.

[35] T. Dohi, "Estimating Software Rejuvenation Schedules in High-assurance Systems," *The Computer Journal*, vol. 44, no. 6, pp. 473–485, Jun. 2001.

[36] 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.

[37] 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.

[38] 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.

[39] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, "Aging-related Bugs in Cloud Computing Software," pp. 287–292, Nov. 2012.

[40] 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. IEEE, Oct. 2018, pp. 257–263.

[41] S. U. B. H. A. S. H. R. E. E. Barada and S. A. N. T. O. S. H. K. U. M. A. R. 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.