




# Logging Hypercalls to Learn About the Behavior of Hyper-V

Lukas Beierlieb<sup>1</sup>, Nicolas Bellmann<sup>1</sup>, Lukas Iffländer<sup>2</sup> and Samuel Kounev<sup>1</sup>

<sup>1</sup>*Institute of Computer Science, University of Würzburg, Würzburg, Germany*

<sup>2</sup>*Faculty of Informatics, University of Applied Sciences, Dresden, Germany*

*fi*

**Keywords:** Hypervisor, Hyper-V, Hypercall, Logging, Monitoring.

**Abstract:** Hypervisors such as Xen, VMware ESXi, or Microsoft Hyper-V provide virtual machines used in data centers and cloud computing, making them a popular attack target. One potential attack vector is the hypercall interface, which exposes privileged operations as hypercalls. We present a hypercall logger for the Hyper-V hypercall interface that logs the inputs, outputs, and sequence of hypercalls. The logs should improve the testability of the hypercall interface by helping to construct test cases for the hypercall handlers. Related works in hypercall monitoring analyze less detailed hypercall invocation data with intrusion detection systems. Our logger extends the WinDbg debugger by adding additional commands to set software breakpoints on the hypercall handler entry and exit within a debugging session with the Hyper-V hypervisor. The evaluation confirmed that the logs are correct and that the breakpoints slow hypercall execution by 100,000 to 200,000. A case study with the hypercall handler logger helps create test cases for evaluation and demonstrates the logger's suitability.

## 1 INTRODUCTION

Virtual Machines (VMs) are computers whose virtual hardware is provided by a software layer called Hypervisor (HV). The HV runs on physical hardware and uses its resources to create the virtual hardware for the VMs, consisting of Virtual Processors (VPs), memory, and other devices. VMs are unaware of this partitioning and behave like running alone on the hardware, allowing multiple VMs to run on the same physical hardware. The concept behind this abstraction is known as virtualization. The fields of application for HVs and their VMs range from data centers and Cloud Computing (CC) to malware analysis and Virtualization-Based Security (VBS) and beyond (Portnoy, 2023; Hoopes, 2009; Microsoft, 2023b)


For each of the above applications, the HV needs to protect the virtual environments from interfering with each other. This interference may be unintended (heavy workload on one VM affecting another VM), or caused intentionally by an attacker. Attackers gain access to a VM, e.g., by renting one in CC, and then exploit it to attack the HV or other VMs with Denial


of Service, Sensitive Data Disclosure, Privilege Escalation, or Arbitrary Code Execution (Liu et al., 2019).


While VMs have multiple interfaces to the HV, we focus on the hypercall interface, a call-based interface that allows VMs to request services from the HV as hypercalls. Hypercalls are similar to system calls at the Operating System (OS) level. As system calls, hypercalls run at the highest privilege level with direct access to physical hardware, turning them into a powerful attack vector (Milenkoski et al., 2014; VMware, 2007).

We decided to research Microsoft Hyper-V for the following reasons: Hyper-V is widely available because it is part of Windows since Windows 8 and Windows Server since Windows Server 2008. Additionally, Hyper-V supports VBS and makes it usable for VMs via hypercalls, bringing VBS into the scope of our work as an additional attack target. Finally, we chose Hyper-V because, so far, only one critical vulnerability concerning the hypercall interface has been found (Common Vulnerabilities and Exposures (CVE)-2013-3898), which is remarkably few for such complex software (Portnoy, 2023; Microsoft, 2022c).

To facilitate the search for vulnerabilities, we developed a Hypercall Handler Logger (HHL) for Hyper-V, which logs the inputs, outputs, and sequence of hypercall handler calls during VM execution. The HHL logs should improve the testability of the hyper-

<sup>a</sup> <https://orcid.org/0000-0003-2512-9292>

<sup>b</sup> <https://orcid.org/0000-0002-8506-2758>

<sup>c</sup> <https://orcid.org/0000-0001-9742-2063>

call interface by helping to construct test cases for the handlers and the hypercalls.

Security researchers benefit from the HHL in multiple ways. They can use the logged inputs and outputs to trace the handlers' main execution paths and investigate how they can influence execution to get to places they previously identified as critical; focus on handlers having little or no occurrence during VM execution and, therefore, may have been less tested or forgotten to adjust during updates; or try to cause a Denial of Service (DoS) by invoking hypercalls multiple times or in a different order based on the logged inputs and sequence. The reliability testing community evaluates the ability of the system to fulfill its intended functions under given conditions over a specified amount of time (IEEE, 1990). The software aging community analyzes the ongoing degradation of software performance and reliability over time (Cotroneo et al., 2014). Both require inputs that lead to successful hypercall execution. The HHL provides these inputs for hypercalls successfully executed during VM execution.

The remainder of this paper is structured as follows. Section 2 reviews related work on hypercall logging. Section 3 gives an overview about Hyper-V's hypercall interface before Section 4 describes the HHL design. Section 5 presents logging results and evaluates the runtime overhead. Finally, Section 6 summarizes the paper and gives an outlook on future research opportunities.

## 2 RELATED WORK

This section discusses related work on hypercall logging to illustrate the state-of-the-art. It briefly explains each paper and how they differ from our work, which aims to develop an HHL for the Hyper-V hypercall interface to improve its testability. In the following, we consider the hypercall logging part of hypercall monitoring in Intrusion Detection Systems (IDSs). Monitoring refers to the ongoing supervision of processes to determine if they run as intended. Papers on hypercall monitoring in IDSs often incorporate a hypercall logger to collect data suitable for analysis. (NIST, 2023)

These loggers are comparable to our HHL, with the following differences: First, they have a different purpose; they create logs to detect attacks while we provide them to construct test cases. Second, they are all based on the open-source Xen HV while we work with the closed-source Hyper-V HV.

Mostafavi and Kabiri (2018) suggested an IDS to detect repetitive and irregular-order hypercall attacks

from guest VMs to the Xen HV. In a repetitive hypercall attack, the attacker invokes a specific hypercall repeatedly to waste hardware resources. Irregular-order hypercall attacks, in turn, refer to the calling of successive hypercalls in an unintended order to confuse the HV and cause a temporary stall or crash.

Shi et al. (2016) introduced an IDS under Hypervisor Introspection (HVI) to detect abnormal hypercall sequences with hypercalls that usually do not occur or do not occur in this order, indicating an attack. In addition to the previous paper, they consider the host VM. Their approach relies on nested virtualization, where the IDS resides in the host VM of the outer HV to protect it from attacks.

Wu et al. (2014) proposed an IDS named C<sup>2</sup>Detector to identify covert channels between VMs. Covert channels are hidden communication channels through which attackers can leak information. They arise from the shared physical resources that the HV manages, such as Central Processing Units (CPUs), cache, or memory. Cache-based covert channels encode sensitive data, e.g., in cache access latencies, which are measurable on other VMs.

Le (2009) proposed and implemented two approaches to protect against hypercall attacks on Xen: authenticated hypercalls and Hypercall Access Table (HAT). The first approach protects hypercalls with a Message Authentication Code (MAC). The HAT approach checks only the call site of the hypercall in the HV. It initially scans the guest OS for call sites, stores them per hypercall and VM in the HAT, a database in the HV, and checks incoming hypercalls against it.

## 3 HYPER-V HYPERCALL INTERFACE DETAILS

Hyper-V consists of a Type 1 HV, which runs directly on the physical hardware, and a virtualization stack, which runs as part of Windows or Windows Server in a privileged management VM called the root partition, parent, or host. They implement a microkernel architecture in which the HV performs only the virtualization tasks that require the highest privileges. All others reside in the lower-privileged root partition, reducing critical HV attack surface. The root partition has fewer rights than the HV but more than the other VMs, called child partitions or guests. For example, the root partition can invoke administrative hypercalls to create, delete, and modify child partitions so that the virtualization stack can manage the child partitions. Hyper-V distinguishes between enlightened and unenlightened child partitions, depending on whether the OS inside is aware of virtual-

ization (often called para-virtualization) (Microsoft, 2022c,a).

In the following, we discuss the Hyper-V hypercall interface for enlightened partitions. Part of the information is stated in the Hyper-V Top-Level Functional Specification (TLFS) (Microsoft, 2022c), the rest is reverse-engineered from the Hyper-V binary.

When invoking hypercalls, the caller passes a Hypercall Input Value (HIV) to the HV and receives a Hypercall Result Value (HRV) after the hypercall execution. The exchange of these values happens via General-Purpose Registers (GPRs). For call-specific inputs and outputs, there is choice between a register- and memory-based calling convention. Hypercalls in Hyper-V fall into four groups: simple hypercall, rep hypercall, extended hypercall, and Virtual Trust Level (VTL) call and return (Microsoft, 2022c).

**Basic Workflow.** First, the caller inside the partition specifies the hypercall input, consisting of an HIV, which defines the hypercall to invoke and other hypercall properties, and optionally further call-specific input. The caller triggers a VM exit to the HV. He executes a designated instruction, VMCALL on Virtualization Technology for x86 (VT-x) and VMCALL on AMD-Virtualization (AMD-V). The HV abstracts the different VM exit instructions by providing the correct one to the caller on a memory page in the Guest Physical Address (GPA) space, called the hypercall page. The VP running the hypercall must be in kernel mode at the time of the VM exit since Hyper-V only allows programs in kernel mode to invoke hypercalls (Microsoft, 2022c).

After the VM exit, the HV takes over the execution and starts general pre-processing. It evaluates the reason for the VM exit in the VM Exit handler to call the appropriate handler. In our case, the reason for the exit was the VMCALL, so execution passes to the VMCALL handler, who performs tests (e.g., validity of HIV, calling from kernel mode) and prepares the call for the hypercall handler. Towards the end, the execution splits into extended hypercall, VTL call and return, register-based, and memory-based calling. The register-based and memory-based callings split further into simple and rep hypercall. Finally, the VMCALL handler calls the hypercall handler via the Hypercall Handler Table (HHT)—an HV-internal data structure that stores constant hypercall-specific data. It contains an entry for each hypercall with a pointer to the handler. The handlers implement the hypercall-specific functionality. When finished, each handler returns a value to report its execution status. It is either 0 if the execution was successful, or corresponds to a defined error code (Microsoft, 2022c).

After the hypercall handler, the VMCALL handler

takes over again and starts post-processing. It prepares the potential output of handlers for transfer to the caller and creates the HRV, which provides the caller with feedback about the hypercall execution in the HV. Finally, the VM Exit handler passes control back to the caller.

**Hypercall Input Value.** The HIV is a 64-bit value parameterizing the hypercall properties. Relevant for the logger are the 16-bit call code identifying the call, a bit flag indicating the calling convention (register/memory), one byte specifying variable header size, and rep count a start index for rep hypercalls. The RCX register transfers the value (Microsoft, 2022c).

**Hypercall Result Value.** The HRV is a 64-bit value giving the status of the hypercall execution. Similar to the return value of the hypercall handler, it indicates the overall call execution status. A value of reps completed exists for rep hypercalls. The HV returns the HRV in RAX (Microsoft, 2022c).

**Register- vs. Memory-Based Calling Convention.** Hypercalls with input and output values besides the HIV and HRV allow the caller to choose between a register-based or memory-based calling convention. The register-based concept is faster but only works for calls with a limited input or output size. In basic form, hypercalls with no call-specific output and two or fewer input are supported, with inputs stored in RDX and R8. The XMM Fast Hypercall extension allows using the 128-bit registers XMM0 to XMM5. Thus, the possible input size is 112 bytes. Additionally, it allows the return of output values in the XMM registers that don't carry input values (Microsoft, 2022c).

With the memory-based calling convention, the transfer occurs via two memory pages; one for the input, one for the output. The caller-provided memory pages are 4 KiB in size, sufficient for the input and output of every hypercall. The caller stores their addresses in the registers RDX and R8, respectively (Microsoft, 2022c).

**Simple vs. Rep Hypercalls.** Simple hypercalls behave as described so far; they get called, perform their task, and return. Rep hypercalls, in contrast, facilitate multiple executions of the same operation with a single call, comparable to a series of simple hypercalls. The input next to the HIV consists of a list of input list elements and possibly an input parameter header. The input list elements are the inputs for the individual operation runs, processed in list order. The header contains data that is the same for all list elements. The output comprises an HRV and a list of output list elements if the operation generates one. The transmission of input and output remains the same (Microsoft, 2022c).

The rep count in the HIV reports the number of repetitions to the HV, which equals the number of input list elements. The rep start index, also part of the HIV, indicates the index of the first unprocessed element in the input list because the HV tries to limit the execution time of hypercalls in the HV to 50  $\mu$ s or less before returning control to the calling VP. At this point, some input list elements may still be open, and the rep start index tells the HV which element to continue with when execution resumes. The purpose of the timeout is to prevent the hypercall execution from blocking, e.g., the handling of pending interrupts or the scheduling of other VPs for too long. The HV updates the index, not the caller. The caller doesn't notice the interruption because the HV doesn't advance the Instruction Pointer Register (RIP) of the VP, so the RIP continues to point to the VMCALL instruction. Accordingly, a resume of the VP leads to a new VM exit with an increased rep start index. Hyper-V refers to this concept as hypercall continuation. Reps complete in the HRV reports the number of completed repetitions to the caller (Microsoft, 2022c).

**VTL Call and Return.** These calls belong to VBS. With a VTL call, it is possible to switch to the next higher VTL, with a VTL return, to the next lower. They undergo individual pre- and post-processing just before and after the handler because not the caller defines the HIV but the code on the hypercall page, there is no output, not even an HRV, and the calling convention is register-based and demands an input of HIV and a 64-bit control input (Microsoft, 2022c).

**Hypercall Handler Table.** The HHT is a table in the HV that stores constant data about hypercalls. It contains an entry for each hypercall indexed according to the call code. The first element of an HHT entry is a function pointer to the hypercall handler. The second element is the call code of the hypercall. The third element is the Hypercall Control Vector (HCV) and contains flags for program control. The following four elements specify the input and output size in bytes, excluding HIV and HRV. Simple hypercalls can have input and output parameters, i.e., for simple hypercalls, fields one and three of the four size fields are relevant. Rep hypercall can have an input parameter header and an input and output list. The input parameter header size is in the field where the input parameter size is in the simple hypercall. The input list size results from field two's input list element size multiplied by the rep count in the HIV. Analogously, the output list size arises from field four. The statistical hypercall group number organizes the hypercalls into groups for which the HV records the number of executions. Padding aligns the HHT entry to a multiple of eight.

## 4 LOGGING APPROACH

This section describes the design of our HHL for Hyper-V, which should log the inputs, outputs, and sequence of hypercall handler calls during VM execution. We designed the HHL with WinDbg, as this seems to be the only option for us to create the logger with reasonable effort. For example, inserting hooks, as a related paper does with the open-source Xen HV, is not as easy with closed-source software. WinDbg is a debugger for Windows OSs, which can also debug the Hyper-V HV (Microsoft, 2018).

The description of the HHL design proceeds step-by-step. First, we deal with hypercall input logging, output logging, then call sequence logging.

**Logging of Hypercall Handler Input.** First, we need to localize the hypercall handlers in the virtual address space of the HV at runtime. Searching for the handler's name is not an option because Microsoft does not publish debug symbols for the HV (ERNW, 2019; Microsoft, 2022b).

Our approach to locating hypercall handlers is the HHT, which stores a function pointer to the handler for each hypercall. The HHT resides at the beginning of the CONST segment. We get the beginning of the segments from their section headers. These are part of the Portable Executable (PE) format, which describes the structure of Windows executables. The executable in PE format starts at the image base. The section headers are behind the DOS header, DOS stub, and NT headers. Starting from the NT headers, we obtain the base address of the section headers by adding the field offset and the size of the optional header. The field offset results from the NT header structure.

The HHT holds one entry for each hypercall, indexed by the call code and with a size of 24 bytes. The function pointer to the hypercall handler is the first element of the entry and has 64 bits or 8 bytes as a memory address on a 64-bit architecture. The memory on the x86-64 architecture is byte accessible, i.e., one address per byte of data. Thus, we find the function pointer per hypercall by reading the first 8 bytes at the address  $\text{CONST base} + 24 \cdot \text{call code}$  (Intel, 2023). The VirtualAddress field in the section header may change when Microsoft releases a new version of the Hyper-V HV as a new build of the HV executables but remains constant between restarts of the HV. However, this doesn't apply to the image base, which changes with every restart due to Address Space Layout Randomization (ASLR). WinDbg provides functions to determine the image base at runtime (Scargall, 2020; Microsoft, 2021b).

Logging can be enabled by placing a Breakpoint (BP) on the begin of the handler. During kernel or

HV debugging, only 32 BPs are available, which is insufficient to log all hypercall handlers simultaneously (Microsoft, 2023a).

Next, we must log the input after a handler call has triggered a BP. Reverse engineering of the HV code showed that the handlers of different hypercall types return different parameters. We extracted them from the code to understand the input. For logging, it is necessary to know the number of parameters, their order, data types, and meanings. We require meaning and order because the handlers receive pointers to input and output areas whose size comes from other parameters, e.g., the size of the input list of a rep hypercall depends on the rep count.

We apply the knowledge gained about the parameters to read the handlers' input and write it to a log file with a timestamp and the call code. The remaining section covers our findings on the parameters, divided into the three hypercall types. The hypercall type results from the HCV in the HHT entry of the hypercall.

The HCV of a simple hypercall is 0 or 2, i.e., no flags or only the Variable-Sized Input flag at the 2nd-bit position is active. The first parameter is a pointer to the hypercall input parameters. Reading the hypercall input parameters via the pointer requires their size, which stands in the HHT. The second parameter is a pointer to the hypercall output parameters. It doesn't provide data to the handler but tells it where to store its output. We record the output parameters for output logging to know where the output is after the handler execution. The variable header size is the optional third parameter of the handler for calls with variable-sized input.

Rep hypercalls have an HCV of 1 or 3; the Rep Hypercall flag at the 1st-bit position is mandatory, and the Variable-Sized Input flag at the 2nd-bit position is optional. The first handler parameter is a pointer to the input parameter header and input list. The size of the inputs derives from the HHT, rep count, and, if applicable, variable header size. The rep count is the second handler parameter, encoded in 12 bits of the HIV. The rep start index is the third handler parameter, also stored in the HIV. The last two parameters are output parameters specifying the address to return the output list and reps complete. As with simple hypercall handlers, we keep the output parameters for output logging. The variable header size is an optional parameter.

VTL calls and VTL returns have a set VTL Call and Return flag at the 4th-bit position of the HCV, yielding an HCV value of 4. The first parameter is a pointer to the HV-internal data structure of the calling VP. It is a large and complex structure of unknown size and composition. Therefore, we cannot log and

interpret its contents but instead log the pointer, which acts as a kind of runtime ID of the VP since the addresses of the VP structures do not change at runtime and thus can reveal patterns. The second parameter is a flag that specifies whether the caller is running in a 32-bit or 64-bit mode. The third parameter is the control input.

**Logging of Hypercall Handler Output.** For output logging, we need BPs that trigger once handler execution is complete. We achieve this by placing a BP on the return address of the handler. The return address of a function is the memory address of the first instruction the processor executes after the function returns. The calling function stores it on the stack, and a Return (RET) instruction within the called function loads it into the RIP at the end of the function. So, the return address is a single address that executes after each handler execution and is accessible within the handler; in comparison, the addresses of the handler's RET instructions, which mark the end of the handler, are unknown and require as many BPs as there are RET instructions.

Output logging with BPs on the return addresses of the hypercall handlers involves two challenges. Firstly, the return addresses lie outside the handlers, where all finished hypercalls pass execution to. Calls not supposed to be logged would mistakenly activate the output BPs. For this reason, we add the output BP during input logging and turn it into a one-shot BP, which triggers once and then removes itself. In other words, we insert the BP on each handler call, and the BP triggers and removes itself when the execution reaches the return address. During input logging, we are in the handler and can access the return address on the stack.

Secondly, the hardware can have more than one Logical Processor (LP), and the HV offers more than one VP. The VPs run on the LPs, which operate in parallel on the HV code. LPs running in parallel trigger the output BP set by another LP if they perform a hypercall with the same pre-processing and reach the return address earlier. Moreover, they overwrite each other's output BP if they run hypercalls with the same return address, and we want to log them. We prevent this by making the output BP LP-dependent, i.e., only triggers when the respective LP passes the address. LPs can execute the same hypercall for different VPs simultaneously. Therefore, the HHL needs LP contexts to store the output parameters per hypercall and assign them to the LP during output logging. The hypercall execution of a VP runs on a single LP, as there are no context switches between VPs during hypercall execution in the HV (Microsoft, 2022c). If this were not the case, the output BP would lose its

assignment to the VP after a context switch since the BP remains, but the LP now runs a different VP. In addition, the HHL would need VP contexts.

Each LP can place an output BP. Hence, with output logging, we can no longer set 32 software BPs to track 32 hypercall handlers, but only 32 minus the number of LPs.

We log the handler output on BP triggering in step two. We take the output parameters stored during input logging to read the hypercall output. Furthermore, we read the return value of type HV\_STATUS, which each handler returns to indicate its execution status. The result goes to the log file with a timestamp.

The VTL call and return handlers have no output parameters and, thus, don't require consideration. Handlers of simple hypercalls have one output parameter: the pointer to the hypercall output parameters. Reading them requires their size, given in the HHT as the size of output parameters. Handlers of rep hypercalls have two output parameters: the pointer to the output list and the pointer to reps complete. The output list size results from multiplying the rep count and the size of the output list element in the HHT.

**Logging of Hypercall Handler Call Sequence** This goal includes recording a timestamp, the call code, the partition ID, and the VP index to track the temporal order of handler calls per VM and VP. The HV guarantees that the partition IDs are unique until an HV restart (Microsoft, 2022c).

The assignment between root VP and LP is constant, i.e., a root VP always runs on the same LP. Guest partitions can have, at most, as many VPs as there are LPs. Unless the system administrator configures a fixed assignment, their VPs can be scheduled on any LP (Microsoft, 2012, 2020).

The location of the partition ID in the Hyper-V HV appears in the handler of hypercall 0x46 HvGetPartitionId, which returns the ID of the partition that issued the hypercall. Accordingly, we could deduce that the partition ID in the HV can be obtained at runtime as [GS Base + 0x103A0] + 0x2380. This query is build-dependent, i.e., only valid until Microsoft changes the HV-internal data structures. This happens frequently, but determining the new value is not elaborate.

Hyper-V models the VP state with a set of registers that it stores internally and loads into the LP on a context switch if there is a physical counterpart. The VP index is in the VP register HvRegisterVpIndex with register name 0x00090003. It has no physical counterpart, hence the HV-defined Model-Specific Register (MSR). For MSR access, the HV code maps the address of the MSR to the name of the VP register and the register name to the HV-internal

VP structure. We found the implementation part that maps 0x00090003 to the HV-internal VP structure by searching for 90003h and 40000002h and tracing the execution path. We verified it by placing a software BP that triggered when we ran Read From Model Specific Register (RDMSR) 0x40000002 in WinDbg while debugging the root partition's kernel (Microsoft, 2022c).

Then, we found a function that receives the VP register name via the GPR ECX. It loads the address of the running VP's HV-internal structure (gs:10398h) into RDI. Then, it evaluates the register name to access the VP structure. The first name tested is 0x00090000 (HvX64RegisterVpRuntime (Microsoft, 2022c)): the function subtracts 0x90000 from the name and branches with Jump if Zero (JZ) if the result is 0. Afterward, the test for 0x00090003 follows: the function reduces the name by another three and checks again with a JZ to 0. Consequently, we end up in the lower right block, where the upper Move (MOV) instruction retrieves the index of the running VP from its VP structure.

Reverse engineering the handler of the HvRegisterVpIndex hypercall, we determined we can query the VP index in the HV at runtime using the HV-internal VP structure at [GS Base + 0x10398] + 0x394. This request is again build-dependent, but determinable with reasonable effort.

The approach enables us to comprehend the relationship between 32 hypercall (limited by BPs). Unfortunately, Hyper-V has 271 hypercalls, 69 are reserved, i.e., they only block call codes for possible future calls. Their function pointers in the HHT point to a default handler implementation, which returns a value indicating an invalid call code.

As an alternative, we considered logging the call sequence on HHT access of the pre-processing when it reads the HHT entry. It only requires one software BP. However, a test has shown that the traffic at this site is so high that frequently occurring hypercalls suppress infrequent ones.

**Implementation.** The whole approach is implemented as a DbgExtEng extension for WinDbg, exposing the four debugger commands: hhl\_log, hhl\_log\_entry, hhl\_log\_exit, and hhl\_convert. However, the user should only enter hhl\_log and hhl\_convert; hhl\_log\_entry and hhl\_log\_exit are for internal purposes. (Microsoft, 2021a)

The code is available on GitHub<sup>1</sup>.

<sup>1</sup><https://github.com/DcartesResearch/HyperV-hypercall-logger>

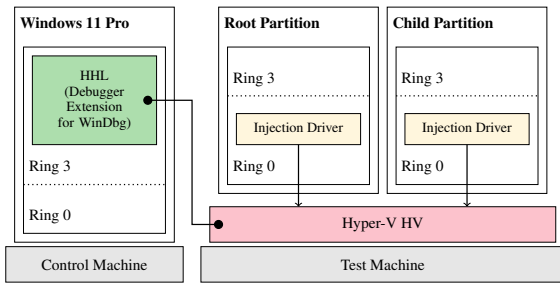


Figure 1: HHL Testbed.

## 5 EVALUATION

Figure 1 shows the testbed for the evaluation of the HHL. It consists of a control machine and a test machine. The HHL is part of a WinDbg instance that runs on the control machine and maintains a connection to the HV on the test machine. From there, the HHL logs the hypercall handler calls in the HV triggered by hypercalls from the root/child partitions. In the case study, these hypercalls originate from the VM execution. For the performance measurements, a kernel injection driver issues them from Ring 0 of the root or child partition. The connection between the control machine and the test machine for HV debugging exists via a network.

We turned off Secure Boot in the BIOS of the test machine to enable network debugging for the Hyper-V HV. VBS demands Secure Boot. It is possible to bypass this requirement in the Windows Registry. However, we don't know how this affects VBS and decided not to activate it in the root partition. We can use VBS within the guest partition by selecting Secure Boot in the guest VM settings, but only in the case study, as we need to turn on TESTSIGNING to run the injection driver, which is only possible without Secure Boot. TESTSIGNING is permanently active on the root partition.

**Case Study.** Which hypercalls occur during the boot and desktop idle of the root and a guest partition? This is an exciting question and was one of the initial motivations to create the HHL. The logging results<sup>2</sup> list the 74 hypercalls occurring in the four scenarios, marked by an “x” in the columns of the scenarios where they occur. DIdle is short for desktop idle. The hypercall names originate from the HvGdk.h header file (Ionescu, 2020), as it is more up-to-date than the TLFS (Microsoft, 2022c). The case study ran in the testbed described in Section 5 under the following conditions. The boot process be-

gins when the computer starts and ends when the login screen appears. The recording of hypercalls during desktop idle lasts 2 minutes. The BP hit limit is 1 to record each handler call only once. It should ensure that the test machine doesn't hang up and that the HHL captures all hypercalls, not just the most frequent ones. This was necessary because calls like `HvCallUpdatePerformanceStateCountersForLp` were called much more frequently than the rest. Having detailed information about the exact rates of different hypercalls during idle would be very interesting; unfortunately, the rates are considerably too high for the BP-based HHL to keep up. This is a limitation regarding the utilized technology. On the other hand, for the targeted security-, stress-, and dependability testing, the appearing input and output values are usually worth more than the “natural” rate of occurrence of the calls.

**Performance Measurements.** We examine the influence of the logging overhead by issuing hypercalls with and without HHL. The test hypercall is `0x46 HvGetPartitionId`. We refrain from analyzing different hypercalls, callers, and input and output sizes since the overhead caused by the BP events is so dominant that all other factors play a subordinate role.

Each test run issues 1,000 hypercalls with a delay of 0.1 s. The delay is necessary for the WinDbg BPs to work. Without delay or with a delay that is too short, WinDbg will eventually stop triggering BPs, and the HHL will miss the handler entry or exit. In addition, a shorter or longer delay increases the likelihood that the BPs will get stuck during repeated execution. In such a case, a BP constantly triggers and prevents execution from continuing, even in single steps. The 0.1 s is an empirical value for hypercall `0x46`, with which we could execute the test runs without the errors mentioned. The test run without HHL also has a 0.1 s delay to ensure identical conditions.

Figure 2 contains the box plots for the test runs with HHL in logging modes 0, 3, 4, and 7. Logging mode 0 is the bare minimum. It logs the LP ID, the call code, and the handler entry timestamp with a BP on the handler entry. Logging mode 3 involves handler input and call sequence logging, mode 4 handler output logging, and mode 7 handler input, output, and call sequences logging. We skipped logging modes 1, 2, 5, and 6 as the hypercall execution times of 0 and 3 or 4 and 7 hardly differ, so examining their intermediate states 1 and 2 or 5 and 6 provides little information.

The median execution time without logging was  $1.2\mu\text{s}$ , the highest outlier was  $4.7\mu\text{s}$  - around 100.000x faster than modes with one breakpoint and 200.000x faster than modes with two.

<sup>2</sup>[https://github.com/DcartesResearch/HyperV-hypercall-logger/blob/main/logging\\_results/logging\\_results.pdf](https://github.com/DcartesResearch/HyperV-hypercall-logger/blob/main/logging_results/logging_results.pdf)

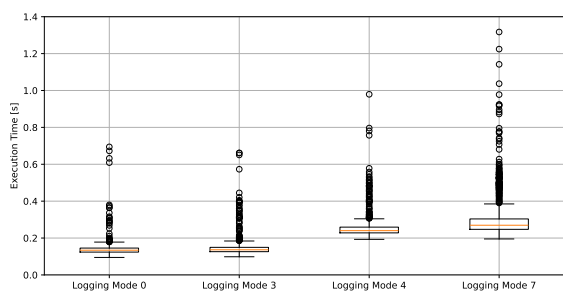


Figure 2: Execution Time of Hypercall 0x46 with HHL.

## 6 CONCLUSION

The paper presented an HHL for the Hyper-V hypercall interface that logs the inputs, outputs, and sequence of hypercall handler calls during VM execution, with the aim of improving the testability of the hypercall interface.

The most evident follow-up work is to use the HHL for its intended purpose: support in creating test cases for the handlers and the hypercalls. In addition, the HHL provides the foundation for logging other properties of the handlers, the hypercall interface, or the Hyper-V HV, e.g., global handler variables, test results during pre- or post-processing, or MSR access. We omitted extended hypercalls and hypercall 0x6. Therefore, a possible future task is to add them.

An alternative to the WinDbg BP approach could be hooking, which we excluded in this work due to its complexity. For example, one could modify the HV code with WinDbg and apply a system call hooking technique, which overwrites the function pointer to the hypercall handler in the HHT with a pointer to a function that performs input logging, calls the original handler, and finally logs the output. It would be considerably faster and more reliable than the WinDbg BP approach.

## REFERENCES

Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. (2014). A Survey of Software Aging and Rejuvenation Studies. *J. Emerg. Technol. Comput. Syst.*, 10(1).

ERNW (2019). Work Package 6: Virtual Secure Mode.

Hoopes, J. (2009). *Virtualization for Security: Including Sandboxing, Disaster Recovery, High Availability, Forensic Analysis, and Honeypotting*. Elsevier Science.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84.

Intel (2023). Intel 64 and IA-32 Architectures Software Developer’s Manual.

Ionescu, A. (2020). hvgtk.h. <https://github.com/ionescu/007/hdk/blob/master/hvgtk.h>.

Le, C. H. H. (2009). Protecting Xen hypercalls: Intrusion Detection/ Prevention in a Virtualization Environment. Master Thesis, University of British Columbia.

Liu, C., Singhal, A., Chandramouli, R., and Wijesekera, D. (2019). Determining the Forensic Data Requirements for Investigating Hypervisor Attacks. In *Advances in Digital Forensics XV*, volume 569 of *IFIP Advances in Information and Communication Technology*, pages 253–272. Springer, Cham.

Microsoft (2012). Hypervisor Top Level Functional Specification v3.0a.

Microsoft (2018). First Steps in Hyper-V Research. <https://msrc-blog.microsoft.com/2018/12/10/first-steps-in-hyper-v-research/>.

Microsoft (2020). Hyper-V Host CPU Resource Management. <https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-minroot-2016>.

Microsoft (2021a). Debugger Engine Introduction. <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/introduction>.

Microsoft (2021b). Modules. <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/modules>.

Microsoft (2022a). Hyper-V Architecture. <https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture>.

Microsoft (2022b). Hyper-V symbols for debugging. <https://learn.microsoft.com/en-us/virtualization/community/team-blog/2018/20180425-hyper-v-symbols-for-debugging>.

Microsoft (2022c). Hypervisor Top Level Functional Specification.

Microsoft (2023a). Methods of Controlling Breakpoints. <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/methods-of-controlling-breakpoints>.

Microsoft (2023b). Virtualization-based Security (VBS). <https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs>.

Milenkoski, A., Payne, B. D., Antunes, N., Vieira, M., and Kounev, S. (2014). Experience Report: An Analysis of Hypercall Handler Vulnerabilities. *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 100–111.

Mostafavi, M. and Kabiri, P. (2018). Detection of repetitive and irregular hypercall attacks from guest virtual machines to Xen hypervisor. *Iran Journal of Computer Science*, 1:89–97.

NIST (2023). Glossary. <https://csrc.nist.gov/glossary>.

Portnoy, M. (2023). *Virtualization Essentials*. Sybex, Indianapolis, 3rd edition.

Scargall, S. (2020). *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, Berkeley, CA.

Shi, J., Yang, Y., and Tang, C. (2016). Hardware assisted hypervisor introspection. *SpringerPlus*, 5.

VMware (2007). Understanding Full Virtualization, Paravirtualization, and Hardware Assist.

Wu, J., Ding, L., Wu, Y., Min-Allah, N., Khan, S. U., and Wang, Y. (2014). C2detector: a covert channel detection framework in cloud computing. *Security and Communication Networks*, 7(3):544–557.