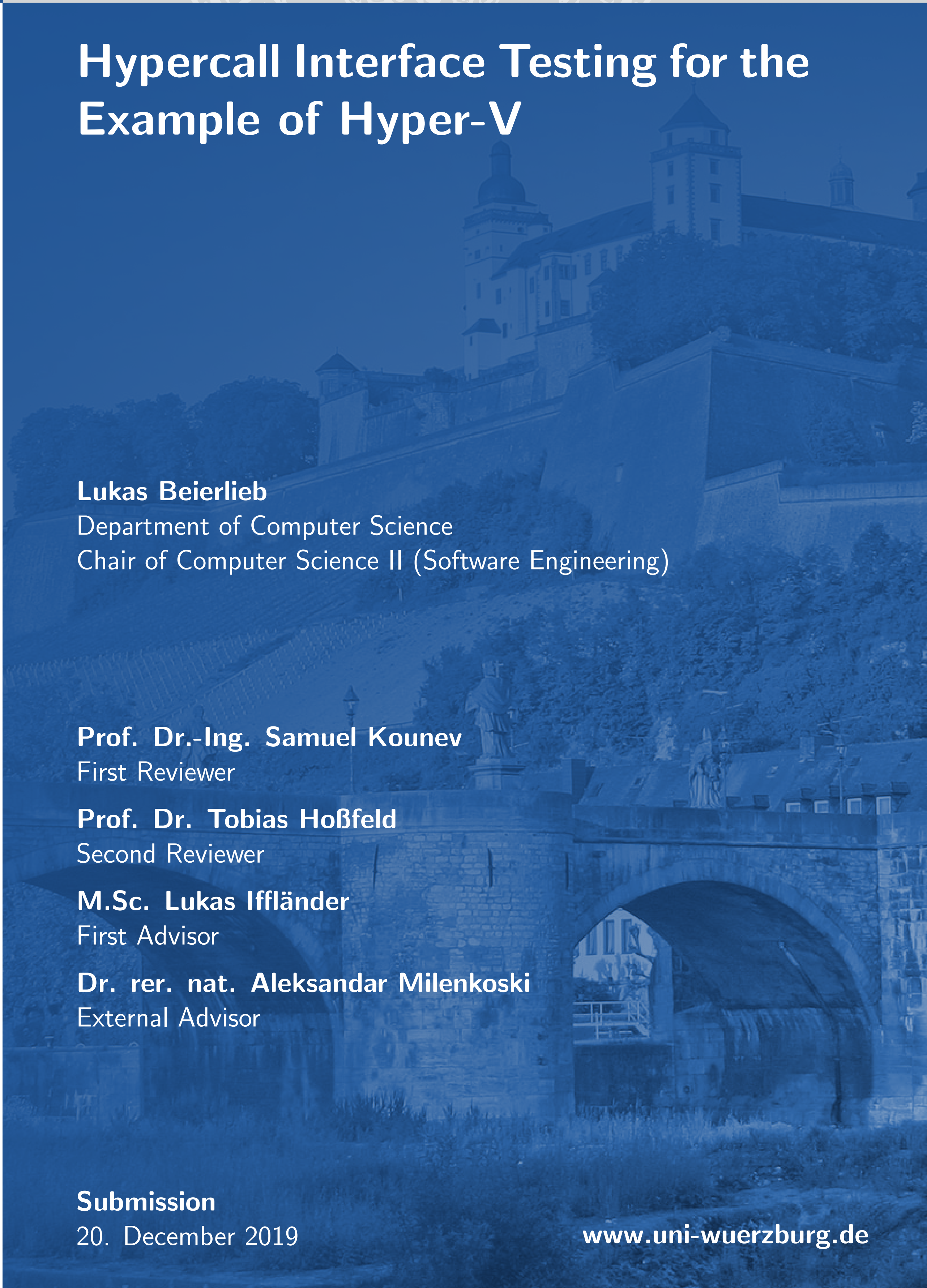
Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG



Hypercall Interface Testing for the Example of Hyper-V

Lukas Beierlieb
Department of Computer Science
Chair of Computer Science II (Software Engineering)

Prof. Dr.-Ing. Samuel Kounev
First Reviewer

Prof. Dr. Tobias Hoßfeld
Second Reviewer

M.Sc. Lukas Iffländer
First Advisor

Dr. rer. nat. Aleksandar Milenkoski
External Advisor

Submission
20. December 2019

www.uni-wuerzburg.de

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, 20. December 2019

. . . .

(Lukas Beierlieb)

Abstract

Virtualization describes the concept of abstracting physical devices by creating virtual instances of them. In the case of virtualized computing resources, the abstraction layer is referred to as the hypervisor. Virtual Machines (VMs) can communicate with their hypervisor through so-called hypercall interfaces. Hypervisors are critical entities in today's infrastructure. Thus, it is crucial to ensure the performance, isolation, reliability, robustness, and security properties of the hypervisor, and specifically of the hypercall interface. Testing the interface cannot prove that no issues exist, but it can help to find problems. Microsoft's Hyper-V hypervisor is especially interesting for testing because it is widely used, proprietary, and enables the Virtualization-Based Security technology.

There are existing efforts to test the robustness properties of hypercall interfaces. However, they specialize in testing the Xen hypervisor. This thesis focuses on building a generic framework that allows performing hypercall testing campaigns on arbitrary hypervisors. Testing campaigns consist of a sequence of hypercalls and delays. Because hypercall calling conventions differ, a hypervisor-specific injection module is required, which usually has to run in a VM's kernel. The injection modules use binary files to retrieve the testing campaign and log execution results. We propose the Hypercall Campaign Description Language (HCCDL), a domain-specific language tailored towards describing hypercall campaigns. We implement a compiler that is able to parse and evaluate HCCDL campaigns. During evaluation of a campaign, every hypercall and delay requested is reported to an implementation of the campaign listener interface. Such implementations are hypervisor- and task-specific. They provide functionalities like the creation of binary campaigns or the generation of human-readable execution reports. We realize the generic parts of the framework, as well as the hypervisor-specific details for Hyper-V.

The evaluation of our Hyper-V hypercall injection module shows that delays defined in a campaign are executed with high precision. The mean error measured is as low as the resolution of the timestamps used to measure, i.e., 0.1 microseconds. On the test machine, the injector was able to invoke more than 2 million hypercalls per second. However, when details about the execution have to be logged, hypercall throughput decreases.

A case study assesses the behavior of Hyper-V's hypercall interface. To learn about which hypercalls occur during normal operation, we developed a hypercall monitoring approach. However, using the monitoring tool results in substantial increases in hypercall activity. This prohibits the acquisition of representative data. Hyper-V's Top-Level Functional Specification documents 24 hypercalls. 6 of them are classified as potentially suitable to be used for load testing, 4 are identified as suitable. These suited calls are invoked at different load levels while measuring their execution times. For some of the hypercalls, the mean execution times increase when they are issued at a slower rate, which might be explainable with caching effects. However, this behavior is not exhibited by all the tested hypercalls. During load testing, a robustness problem has been identified, as well. The hypercall `HvNotifyLongSpinWait`, which is supposed to be only a hint to Hyper-V's scheduler, crashes the whole hypervisor.

Zusammenfassung

Virtualisierung beschreibt das Konzept der Abstraktion physischer Geräte durch die Erstellung virtueller Instanzen. Im Falle von virtualisierten Computerressourcen wird die Abstraktionsschicht als Hypervisor bezeichnet. Virtuelle Maschinen (VMs) können mit ihrem Hypervisor über sogenannte Hypercall-Schnittstellen kommunizieren. Hypervisoren sind ein unverzichtbarer Bestandteil unserer heutigen Infrastruktur. Daher ist es essenziell die Eigenschaften von Hypervisoren und insbesondere derer Hypercall-Schnittstellen hinsichtlich Performance, Isolation, Zuverlässigkeit, Robustheit und Sicherheit zu gewährleisten. Das Testen der Schnittstellen kann nicht beweisen, dass keine Probleme existieren, aber es kann helfen existierende aufzufinden. Hyper-V ist ein von Microsoft entwickelter Hypervisor. Dieser ist für Tests besonders interessant, da er weit verbreitet ist, sein Sourcecode nicht öffentlich zugänglich ist und er die *Virtualization-Based Security*-Technologie ermöglicht.

Es gibt bereits Arbeiten, die die Robustheitseigenschaften von Hypercall-Schnittstellen testen. Diese sind jedoch auf den Xen-Hypervisor spezialisiert. Diese Thesis konzentriert sich auf den Aufbau eines generischen Frameworks, das es ermöglicht, Hypercall-Testkampagnen für beliebige Hypervisoren durchzuführen. Testkampagnen bestehen aus einer Folge von Hypercalls und Wartezeiten. Da sich die Aufrufkonventionen für Hypercalls unterscheiden, ist ein hypervisor-spezifisches Injektionsmodul erforderlich, das in der Regel im Kernel einer VM laufen muss. Die Injektionsmodule verwenden Binärdateien, um Testkampagnen abzurufen und die Ergebnisse der Ausführung zu protokollieren. Wir definieren die Hypercall Campaign Description Language (HCCDL), eine domänenspezifische Sprache, die auf die Beschreibung von Hypercall-Kampagnen spezialisiert ist. Wir implementieren einen Compiler, der in der Lage ist, HCCDL-Kampagnen einzulesen und auszuwerten. Bei der Auswertung einer Kampagne wird jeder angeforderte Hypercall und jede angeforderte Wartezeit an eine Implementierung des *Campaign Listener*-Interfaces weitergereicht. Solche Implementierungen sind hypervisor- und aufgabenspezifisch. Sie bieten Funktionalitäten wie die Erstellung von Binärkampagnen oder - in Verbindung mit einer Ausführungsprotokolldatei - menschenlesbaren Ausführungsberichten. Wir realisieren die generischen Teile des Frameworks, sowie die hypervisor-spezifischen Details für Hyper-V.

Die Auswertung unseres Hypercall-Injektionsmoduls für Hyper-V zeigt, dass die in einer Kampagne definierten Wartezeiten mit hoher Präzision eingehalten werden. Der gemessene mittlere Fehler ist so niedrig wie die Auflösung der zur Messung verwendeten Zeitstempel; das sind 0,1 Mikrosekunden. Auf der Testmaschine konnte der Injektor mehr als 2 Millionen Hypercalls pro Sekunde ausführen. Wenn jedoch Details über die Ausführung protokolliert werden müssen, sinkt der Hypercall-Durchsatz.

In einer Fallstudie wird das Verhalten der Hypercall-Schnittstelle von Hyper-V untersucht. Um zu erfahren, welche Hypercalls im Normalbetrieb auftreten, haben wir einen Hypercall-Monitoring-Ansatz entwickelt. Das Benutzen des Monitoring-Tools führt jedoch zu einem deutlichen Anstieg der Hypercall-Aktivität. Dies verhindert die Erfassung repräsentativer Daten. Die *Top-Level Functional Specification* von Hyper-V dokumentiert 24

Hypercalls. 6 von ihnen sind als potenziell geeignet für Lasttests eingestuft, 4 als geeignet. Die geeigneten Aufrufe werden in verschiedenen Laststufen aufgerufen, während ihre Ausführungszeiten gemessen werden. Für einige der Hypercalls erhöhen sich die durchschnittlichen Ausführungszeiten, wenn sie mit einer langsameren Rate aufgerufen werden, was mit Caching-Effekten erklärbar sein könnte. Dieses Verhalten wird jedoch nicht von allen getesteten Hypercalls gezeigt. Während der Durchführung der Lasttests wurde auch ein Robustheitsproblem festgestellt. Der Hypercall `HvNotifyLongSpinWait`, der nur ein Hinweis an den Scheduler von Hyper-V sein soll, bringt den gesamten Hypervisor zum Absturz.

Contents

1. Introduction **1**
1.1. Motivation 1
1.2. Idea 2
1.3. Benefiting Communities 3
1.4. Research Questions and Goals 4
1.5. Outline 4

2. Background **5**
2.1. x86_64 Architecture 5
2.2. Virtualization 9
2.3. Hyper-V 12

3. Related Work **17**
3.1. Robustness Testing 17
3.2. Performance Isolation 19
3.3. Software Aging 19
3.4. Performance Benchmarking 20

4. Approach **21**
4.1. Hypercall Testing Framework Design 21
4.2. Hypercall Campaign Description Language 24
4.2.1. Requirements 25
4.2.2. Choice of Realization 26
4.2.3. Language Design 27
4.3. Framework Implementation for Hyper-V 31
4.3.1. Hypercall Injection Module 31
4.3.2. Campaign Listener Interface Implementations 36
4.4. Hypercall Monitor 37

5. Implementation **39**
5.1. Campaign Description Language Compiler 39
5.2. Campaign Listener Implementations for Hyper-V 46
5.3. Hyper-V Injection Module 48
5.4. Hypercall Monitor 52

6. Evaluation **55**
6.1. Testbed Setup 55
6.2. Delay and Timestamp Accuracy 56
6.3. Hypercall Injection Performance Evaluation 58

7. Hyper-V Hypercall Performance **65**
7.1. Hypercall Monitoring 65
7.2. Hypercall Survey 68

7.3. Load Testing 72

8. Conclusion **81**
8.1. Summary 81
8.2. Assessment 82
8.3. Future Work 83

List of Figures **85**

List of Tables **87**

List of Listings **89**

List of Abbreviations **91**

Bibliography **95**

Appendix **101**
A. Approach 101
B. Repeated Measurements: HvExtCallQueryCapabilities 102
C. Repeated Measurements: HvFlushVirtualAddressSpace (Guest) 105
D. Repeated Measurements: HvFlushVirtualAddressSpace (Root) 108
E. Repeated Measurements: HvNotifyLongSpinWait (Param = 0) 111
F. Repeated Measurements: HvNotifyLongSpinWait (Param = max) 113

1. Introduction

This chapter introduces and gives an overview of the topic of this thesis, which the following chapters will cover in detail. Section 1.1 sets the context of this work and motivates the importance of the presented topic. The idea to address the motivated problem is explained in Section 1.2. Section 1.3 lists the various research communities that benefit from the contributions provided by this work. To conclude the introduction, Section 1.5 presents the structure of the remainder of this thesis.

1.1. Motivation

Virtualization describes the concept of abstracting physical devices by creating virtual instances of them. In the case of virtualized computing resources, the abstraction layer is referred to as the hypervisor. The hypervisor itself is controlling the physical resources and responsible for providing a virtual computing environment, consisting of virtual processors, memory, and other virtual or emulated devices, to applications or operating systems. This concept was implemented for the first time in the 1960s by IBM, in order to run multiple computing tasks on one mainframe at once.

Today, hypervisors are virtually everywhere. They are widespread throughout data centers, functioning as the backbone of cloud computing [1]. Hypervisors are also prevalent in the modern desktop and workstation infrastructure [2]. This goes as far as 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 is applied to embedded computing systems, as well. Amongst other examples, the automotive and aerospace domains take advantage of abstracting computing resources [4, 5].

Infrastructure as a Service (IaaS) is one of the service models offered by cloud computing. Cloud providers use hypervisors to manage virtual computing environments, often referred to as Virtual Machines (VMs), which can be rented by customers. There is a substantial market for IaaS. In 2017, this service model achieved a global revenue figure of 23.6 billion U.S. Dollars, which increased to 31 billion dollars in 2018 and is predicted to continue rising over the course of the next years [6]. There are several requirements for hypervisors to meet in an IaaS scenario, which also apply to other use cases. When renting VMs to customers, cloud providers guarantee performance in the form of Service Level Agreements (SLAs) and Service Level Objectives (SLOs). As mutually untrusted VMs can run on the same physical machine, the hypervisor has the essential responsibility of providing complete isolation, e.g., regarding performance and security, between virtual instances. High load or deliberate attacks of one virtual machine must not have a measurable impact on co-located virtual environments. The hypervisor is supposed to have as little downtime as possible, requiring high reliability and robustness to invalid and malicious behavior of VMs. The long uptimes also demand an absence of aging effects.

However, as with any complex software system, design or implementation mistakes do exist, which can comprise the required properties. Due to their public indexing and documentation, security vulnerabilities provide a good insight into hypervisor-related issues. Elhage found that the KVM hypervisor allowed unplugging of virtual devices that are not supposed to be removable during runtime, and exploited this fact to execute arbitrary code with hypervisor privilege [7]. A way to perform arbitrary code execution in VMware hypervisors was proposed by Kortchinsky, who took advantage of a virtual video card not verifying memory accesses [8]. In an IaaS scenario, attackers could use both vulnerabilities to leak data of co-located customers or cause a Denial of Service (DOS). Both have devastating consequences for the provider.

1.2. Idea

The previous section established that hypervisors are a critical entity of various modern infrastructures and that their performance, isolation, reliability, robustness, and security have to be ensured. While testing cannot prove that a hypervisor does not have issues, it can help to identify and subsequently fix problems. Thus, this thesis concerns itself with the testing of hypervisors.

However, there are multiple interfaces for VM-hypervisor-interactions suitable for examination. As already mentioned, the hypervisor provides virtual devices, which are managed by the VMs' drivers. Also, every time a privileged instruction is executed in a virtual environment, the hypervisor takes over control and emulates the instruction's actions to the virtual processor it is executed on. Finally, many hypervisors provide a call-based interface, which virtualization-aware VMs can utilize. Analogously to the systemcalls, which allow userspace-kernel-communication, the calls to the hypervisor are called hypercalls.

This thesis selected hypercalls due to their use in Virtualization-Based Security (VBS) [9]. The previous section already stated that many Windows operating systems include the Hyper-V hypervisor. When Hyper-V is active, VBS can be used. Then, the Windows operating system running on top of Hyper-V is split into two isolated execution environments. In one, the regular kernel and operating system run; the other is used for security-critical operations, e.g., storing password hashes or encryption. This separation provides the advantage that even an attacker with the highest privileges in the normal partition cannot access the sensitive contents of the secure partition. Communication between the regular and the secure partition is realized through hypercalls.

Thus, this thesis sets out to build a framework that is as generic as possible and allows the manual invocation of hypercalls to test hypervisors for various properties. As a concrete case study, we choose Hyper-V for multiple reasons. The first reason is the existence of VBS, which utilizes hypercalls, and promises additional security. Also, its inclusion in many Windows versions has lead to a wide distribution of Hyper-V in desktop infrastructure. Finally, it is the hypervisor used in the Azure Cloud, which has been chosen to provide secure cloud infrastructure for the U.S. Defense Department [10].

Of the mentioned requirements, we evaluate the robustness of Hyper-V's hypercall interface through a case study. 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*" [11]. This work focuses on quantifying the impact of stressful environmental conditions. The case study evaluates the influence of different load levels on the hypercall execution times. This load testing means that valid hypercalls are executed repetitively, with a delay between each call. The duration of the delay determines the load level. When performing load testing on, e.g., web servers, response times increase due to congestion. The case study of this work is supposed to test the Hyper-V hypercall interface for similar behaviors.

1.3. Benefiting Communities

Multiple communities benefit from this work's contributions, i.e., a framework for testing the hypercall interfaces of various hypervisors, as well as research about and evaluation results for Hyper-V.

One beneficiary is the robustness testing community, which is often concerned with exposing systems under test with invalid parameters and input values. The goal of robustness testers is to find inputs that result in a crash or other malfunction of the test system. There is overlap to the community of security researchers, which concern themselves with finding inputs that they can use to exploit the system. With crashes being a form of DOS attack, they are relevant both to security researchers and robustness testers. Also, the security-related term "fuzzing" can be described as model-based robustness testing. Both communities benefit from the development of a hypercall testing framework, which enables them to perform robustness or fuzzing hypercall test campaigns.

The framework can also be used to execute hypercall campaigns for testing performance isolation. While a hypervisor might effectively isolate workloads running in different virtual machines, hypercalls present a different challenge because they require processing time of the hypervisor. Using the framework to invoke hypercalls in one VM while benchmarking the performance of a co-located virtual environment can reveal such issues.

Also profiting from this work is the software aging community. Amongst other use cases, the framework enables accelerating the hypervisor's aging process artificially. Hyper-V and other hypervisors offer management hypercalls that can create or delete items, e.g., virtual environments, processors, or communication channels. Repeatedly issuing such creation and deletion calls can simulate long timespans of regular operation and might reveal problems like memory fragmentation. In fact, a vision paper of this work regarding the aspects of software aging has been published [12].

Another vision paper has been published with a focus on performance evaluation and benchmarking [13]. Analyzing the performance characteristics of hypercalls can also interest researchers trying to apply virtualization to embedded systems, as execution times have an essential role in real-time environments. The load testing results of the case study provide an insight into the execution times of different Hyper-V calls. Additionally, the framework can be used to perform further measurements.

Finally, this work is helping people who are interested in understanding the Hyper-V hypercall interface, e.g., because they want to develop software to communicate with it. Hyper-V is proprietary, so no source code is publicly available. There is the Top-Level Functional Specification from Microsoft covering the most important aspects of interacting with the hypervisor [14]. Furthermore, it is possible to reverse-engineer the binary files. There are also reverse-engineering results publicly available [15]. This thesis provides a summary of the information necessary to understand the hypercall interface, as well as measurement results of hypercall execution times, which give a hint about their internal complexity.

1.4. Research Questions and Goals

We state the following research questions in order to develop a generic framework for hypercall testing and perform a case study in the form of load level testing with Hyper-V:

- **RQ1:** *What is the workflow to perform hypercall injection test campaigns?*
- **RQ2:** *Which extent of workflow abstraction is achievable?*
- **RQ3:** *How to inject hypercalls into Hyper-V?*
- **RQ4:** *Which hypercalls occur during normal operation?*
- **RQ5:** *How does Hyper-V react to increasing hypercall loads?*

From these research questions, we derived the following goals:

- **G1:** *Compare the hypercall interfaces of different hypervisors and identify similarities and differences.*
- **G2:** *Implement a generic framework for designing, executing, and evaluating hypercall test campaigns with interfaces for tasks that are hypervisor-specific and can't be generalized.*
- **G3:** *Implement the hypervisor-specific interfaces for Hyper-V, including the hypercall injection module.*
- **G4:** *Validate the correctness and evaluate the performance of the Hyper-V injection module.*
- **G5:** *Identify hypercalls suited for repeated execution and evaluate the impact of different load levels.*

1.5. Outline

The remainder of this thesis is structured as follows. Chapter 2 explains the necessary technical background regarding the x86_64 processor architecture, virtualization generally, and Hyper-V specifically. Existing research efforts in the area of hypervisor testing are presented in Chapter 3. Chapter 4 first identifies the generic workflow of performing hypercall testing campaigns. Based on that, an approach for the framework and its individual components is developed. It also introduces the idea of a hypercall monitor, which can log all hypercalls issued in a Hyper-V virtual machine. The proposed approaches are implemented in Chapter 5. Chapter 6 evaluates how accurately the developed Hyper-V hypercall injector can perform testing campaigns, as well as how much overhead logging information about hypercall executions introduces. The case study is described in Chapter 7. Finally, Chapter 8 concludes this work by summarizing its content and providing an overview of future research possibilities.

2. Background

This chapter establishes a knowledge base that contains the information about virtualization necessary to understand this thesis. First, the basics of the x86_64 processor architecture are explained in Section 2.1. For one, this is required because Hyper-V is a hypervisor for the x86 and x86_64 architectures, and manually performing hypercalls to it involves architecture-specific details. On the other hand, x86_64, with its significant market share in server and desktop infrastructure, has an essential role in the development of virtualization technology, which is explained in Section 2.2; knowledge about a Central Processing Unit (CPU) architecture aids in understanding how it can be virtualized. Finally, Section 2.3 sets the focus on Hyper-V, describing how this hypervisor fits into the concept of virtualization.

2.1. x86_64 Architecture

The following information and many more details about the x86_64 architecture can be found in Intel's Software Developer Manual [16]. The history of the x86_64 processor architecture (also called x64 or amd64 architecture) begins as early as 1978, when Intel released the 8086 microprocessor. Its 16-bit-based architecture was extended to 32-bit with the Intel 80386 in the year 1985, and further extended to 64-bit by AMD in 2003. Throughout its evolution, backward compatibility has been kept almost entirely. Because of this, the x86_64 architecture has five different execution modes, with their primary differentiation being in how they access memory. However, this section is going to focus on the *long mode*, which is the 64-bit execution mode, and thus the mode in which Hyper-V and its virtual machines execute in the majority of their time.

Registers

Registers are small memory locations inside of the CPU, which are very fast to access (typically in only a single CPU cycle). There are different types of registers for different purposes. General-purpose registers do not have a usage policy enforced on by the hardware, the software running on the CPU is free to decide how to make use of them. The general-purpose registers available in x86_64 are listed in Table 2.1. The first eight listed exist in their 16-bit form since the original 8086 architecture. They were called AX, BX, CX, . . . , and were 16 bits wide. When the architecture was extended to 32-bit, the registers EAX, EBX, ECX, . . . were introduced, with "E" meaning "extended". To ensure backward compatibility, the 16-bit register names then referred to the lower 16 bits of the

Register Name	Intended Use
RAX	Accumulator Register
RBX	Base Register
RCX	Counter Register
RDX	Data Register
RSI	Source index Register
RDI	Destination index Register
RBP	Base Pointer Register
RSP	Stack Pointer Register
R8-R15	Additional Registers

Table 2.1.: x86_64 General-Purpose Registers

corresponding extended register. The step from 32-bit to 64-bit had an analogous consequence. The original register names were prepended with an "R" and widened to 64 bits. The lower 32 and 16 bits can still be accessed using the extended and original register names, respectively.

Furthermore, for x86_64, additional general-purpose registers were implemented. They are named numerically, from R8 to R15. It is possible to access the lower 16 and 32 bits by appending "W" (word, describes 16 bits) and "D" (double word, represents 32 bits), respectively. For R8, these would be R8W and R8D. While the numerically-named registers have no intended use stated by their name, the mnemonic of the other registers were chosen after what they were supposed to be used for by software. RAX and RDX are often utilized for providing operands to and storing results from arithmetic operations. RBX is supposed to hold memory addresses that are base pointers to larger memory structures, which are accessed using the base pointer and an offset to the specific element of the structure. This technique is required when accessing structs or arrays in the C language. In loops, RCX is the register intended to store the loop index. When copying memory from one location to another, RSI and RDI are used to keep track of the source and destination addresses, respectively. Using the registers as intended makes it easier to understand pieces of assembler code. However, it is irrelevant to the CPU whether these conventions are adhered to. Yet, it makes sense to use the RBP and RSP registers to manage the stack, because there are specialized stack-related instructions, which implicitly use RBP and RSP. The stack is a memory structure that is used to provide temporary storage, e.g., for local variables of functions or function parameters, and to keep track of which function called which other functions by storing return addresses. x86_64 also has a number of registers that are specialized to store values for floating point operations and Single Instruction, Multiple Data (SIMD) instructions, as well as a number of control registers.

Memory

When executing in its long mode, software running on an x86_64 processor is not able to access memory locations by their physical addresses. Instead, virtual memory is used, which allows the processor to provide multiple linear addresses spaces with a 64-bit addressing. Operating systems utilize this feature to isolate processes. Security is increased by giving every processes its own address space. Additionally, this also simplifies memory management for programs, as they are free to use the whole address space and do not have check if some other process uses some certain memory ranges.

The linear virtual address spaces are divided into pages, with a typical size of 4.096 bits. A data structure called page table is responsible for storing to which physical memory

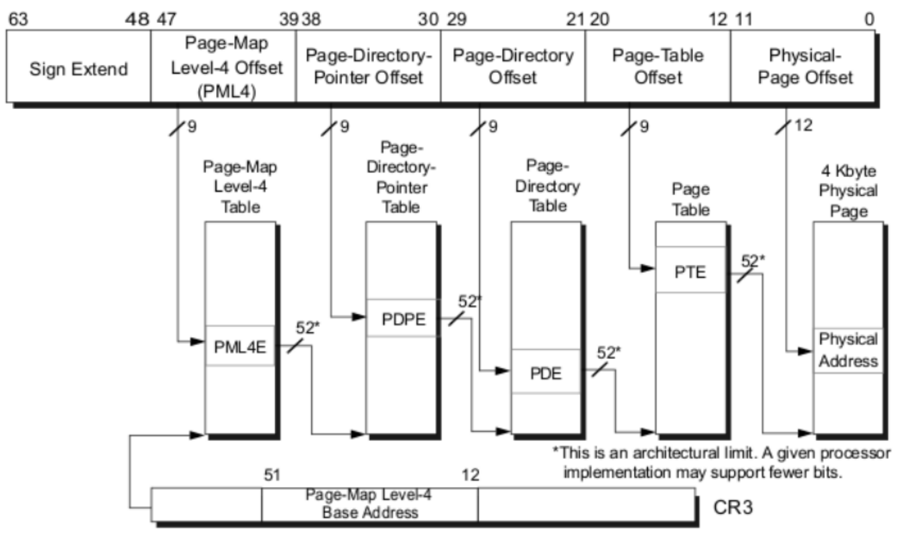


Figure 2.1.: Visualization of the x86_64 Paging [17]

address the page is mapped. Only pages that are used do have a valid entry in the page table. This approach offers a lot of flexibility for operating systems. If all (or even just one) of the virtual address spaces require more memory than there is physical Random Access Memory (RAM) installed, pages can be copied to disk while they are not actively used. When those pages are accessed again, they are copied back to RAM, forcing some other memory out to disk. This strategy is known as swapping. Another advantage is the possibility of mapping pages of different virtual address spaces to the same physical memory. This is especially useful when multiple instances of the same program are running. The code has to be available in every virtual address space; however, it is enough to have it in physical memory once. The same can be applied when a file is mapped to memory, and multiple processes have read-only access to it. It also enables shared memory between the otherwise isolated processes.

However, this greatly improved flexibility and security comes at the price of execution speed and memory usage. For one, the page tables have to be stored in memory, reducing the size of available physical memory that can be used for actual computing. Furthermore, the CPU has to, in principle, access memory multiple times for a single memory access. Both memory consumption and the number of memory access required to resolve a virtual address depend on the page size and the depth of the page table structure. Larger page sizes require fewer entries in the page tables and thus induce less memory overhead. However, pages lose their flexibility with increasing size.

It is possible to use small pages and keep page table sizes low by using multiple layers of translation tables. Every layer adds one memory access of cost to each translation, though. The x86_64 Memory Management Unit (MMU), which implements the address translation in hardware, requires the page table to be set up with a depth of 4. The software is responsible for managing the entries in the page table and inform the hardware about the address of the base table by writing it to the control register CR3. Figure 2.1 visualizes the process of translating a virtual to a physical address for a system that uses 48 bit for physical addressing. Using 48 bits allows addressing 256 tebibytes of physical memory. There is currently little benefit to being able to access more. However, memory translation would be penalized by using the full 64 bits for addressing. The MMU reads the base address of the first-level table, called Page Map Level 4 (PML4), and uses bits 47 to 39 as an offset to find the Page Map Level 4 Entry (PML4E), which is a pointer to the base of the Page Directory Pointer (PDP), where the next bits can be used as an offset to find the Page Directory Pointer Entry (PDPE). The same procedure happens for the Page Directory (PD) and the Page Table (PT), which finally delivers the physical address of the given virtual address. To reduce the cost of repeated memory accesses, the MMU features a cache that stores the last performed translations. This cache is called the Translation Lookaside Buffer (TLB). A operating system has to ensure that every time a process switch is scheduled that the pointer to the process's PML4 is written to the CR3 register.

Device Communication

Communication with peripheral devices happens via multiple channels. For one, there are designated instructions in the x86_64 instruction set that allow reading from and writing to Input/Output (I/O) ports, which in turn are mapped to hardware registers of devices. Instead of using the IN and OUT instructions, together with a separate address space of I/O ports, there is the possibility to have device hardware registers memory-mapped. To the processor, interacting with the hardware works the same as reading and writing to the RAM. It is the task of the MMU to distinguish based on the memory address whether the access has to be performed to memory or to a device register.

Transferring a considerable amount of data, e.g., reading or writing files on a hard drive, pushing texture data to the graphics card, or sending and receiving network packets via the Network Interface Card (NIC), via one of these communication paths is inefficient, because it requires a CPU core to actively transfer the data. Instead, devices are able to access memory independently, which means the processor can perform useful work while the data is transferred. This technique is called Direct Memory Access (DMA).

Also, there are several occasions when devices need to inform the CPU of events that require processing, e.g., when a key on the keyboard is pressed, or the NIC wrote a packet to memory using DMA, which can now be handled by the processor. It is possible to use polling, which describes repeatedly reading hardware registers until an event is signaled there. The polling interval determines how much time the CPU spends checking hardware, which is time that could be used for useful processing, and the delay between an event happening and the processor actually reading about it. However, devices can trigger interrupts, which are managed by a Programmable Interrupt Controller (PIC) or Advanced Programmable Interrupt Controller (APIC). They compare the priority of incoming interrupts and consider which interrupts are disabled by the CPU. Based on that, they determine which interrupt should be handled next. Then, the state of a processing core is saved (register values, instruction pointer), and the instruction pointer is set to the address of the corresponding interrupt handler routine, which performs the action required for this specific interrupt event.

An illustrating example is how the Intel e1000 NIC delivers packets received from the network to the processor. Intel's Software Developer Manual for Gigabit Ethernet Controllers [18] describes this process in detail. The driver of the NIC is responsible for allocating memory that is used to communicate with the NIC. A circular buffer has to be set up. The entries of the buffer, which is essentially a queue, are called descriptors. They hold a pointer to an area of memory reserved for writing received packets, as well as a status byte, which indicates whether a new packet resides at the pointer address. The driver uses hardware registers to inform the NIC about the location of the receive descriptor queue. When the NIC receives a valid packet over the network, it looks up the descriptor at the head of the receive queue and writes the packet contents to the address specified by the descriptor. After the status byte of the descriptor and the head pointer of the queue are advanced, an interrupt is triggered to signal to the CPU that new packets can be retrieved from the descriptor queue.

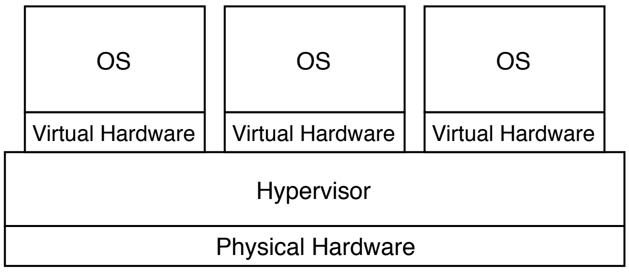


Figure 2.2.: Virtualization

Privilege Levels

The x86_64 architecture defines four privilege levels, called Ring 0 to Ring 3. Ring 0 has the highest privilege and no restrictions. With increasing Ring number, the restrictions regarding access to I/O ports, certain memory pages, and certain instructions increase as well. Typically, modern operating systems only use two privilege levels: Ring 0 for the kernel and Ring 3 for user space code. Applications running in Ring 3 are basically sandboxed to the point where the only possibility to interact with peripherals is to trigger a switch to kernel space and request it to perform the required action, like opening, reading, writing or closing files, sending or receiving network packets, or print something on the screen.

2.2. Virtualization

With the x86_64 architecture known, it is easier to understand how it can be virtualized. It can also be generalized to other processor architecture as the same or similar fundamentals apply.

Popek and Goldberg defined the notion of and requirements for virtualization as early as 1974 [19]. They defined a Virtual Machine (VM) as "an efficient, isolated duplicate of the real machine". They require a Virtual Machine Monitor (VMM), today also known as hypervisor, which is in control of the actual physical resources and provides a functionally identical execution environment to the VM or VMs. This concept is visualized in Figure 2.2. Code that executes correctly on the physical machine should also function in the virtual environment. The virtual execution also should not be significantly slower than the physical performance. Machine emulators, which model the architecture and interpret instructions in software, are not sufficient to meet this requirement. A physical processor should execute as many of the instructions as possible.

To allow a VM to execute instructions while keeping it isolated in its virtual environment, Popek and Goldberg classify the instruction set. Privileged instructions are required to trap when executed in an unprivileged processor state. A trap means the processor is switched to its privileged mode, and a routine is executed that handles the violation. They also define the notion of sensitive instructions, which are sensitive to the processor state. In the case of the x86_64, using arithmetic instructions to modify the content of the general-purpose registers is not seen as changing the processor state However, e.g., changing the page table by writing to the CR3 register or disabling interrupts is.

A CPU architecture is virtualizable if all sensitive instructions are also privileged. If such a sensitive, privileged instruction is executed by a virtual machine, the processor traps and calls the corresponding handler routine set by the hypervisor. The handler can then emulate the actions of the instruction and modify the state of the "virtual processor", without affecting the physical processor state. Non-privileged, sensitive instructions either allow virtual machines to alter the state of the physical CPU or behave dependent on its state. Retrieving state of the physical instead of the virtual processor can lead to the wrong information in the guest program or operating system and result in malfunction. The problem with the x86 and x86_64 architectures is that they are not virtualizable in their original specification, because there are several sensitive, but non-privileged instructions [20]. As explained by VMware in a white paper, two approaches were implemented to work around this limitation [21].

Full Virtualization

The name "Full Virtualization" refers to the fact that VMs are fully virtualized, meaning operating systems that are designed to run on real hardware can be run in the virtual environment without modifications. This is problematic because of the non-privileged sensitive instructions, which exist in operating system code but cannot be trapped by the hypervisor and would thus behave incorrectly. VMware solved this problem by applying binary translation techniques to code of VMs before it is run, rewriting critical instructions.

Paravirtualization

The second approach to avoid the execution of non-privileged sensitive instructions is Paravirtualization. "Para", in the sense of "side by side" or "alongside", implies that virtual machines are not running in a perfectly accurate virtual environment, assuming they are executing on real hardware. Instead, the operating system managing the VM is modified to be aware of its virtual existence. That way, critical instructions can be replaced with calls to the hypervisor. Furthermore, it makes sense to replace even the privileged instructions, which could be trapped, with such calls, because it aids execution speed [22, 23]. The call-based interface that the hypervisor provides to the virtual machines is called the hypercall interface, and the calls are subsequently named hypercalls. This approach was implemented by the open-source hypervisor Xen [24].

Both strategies have their advantages and disadvantages. Full virtualization sacrifices a bit of execution speed, but allows to run any operating system in its virtual environments. Paravirtualization requires modification to guest operating systems, but this additional expense is rewarded with less overhead and better performance.

Hardware-Assisted Virtualization

The problem of having a non-virtualizable CPU architecture was solved with architectural extensions. Intel introduced Virtualization Technology for x86 (VT-x) [25] in 2005, and AMD followed with AMD Virtualization (AMD-V) [26] in 2006. While they are not compatible, they add the same features from a high-level perspective. Additionally to the privilege concept of the four rings, a hypervisor mode is added. When a hypervisor is present, all sensitive instructions executed in non-hypervisor mode generate a trap to the VMM. This allows complete virtualization without using binary translation or guest operating system modifications. Because of the performance benefits of paravirtualization, Intel and AMD added a VMCALL and VMMCALL instruction, respectively, which transfer control to the hypervisor.

Virtual Machine Memory

As already established in Section 2.1, operating systems use virtual memory to provide isolated, linear address spaces to processes, which are mapped into the physical address space by page table structures. However, with machine virtualization, it becomes necessary to virtualize memory twice. The address spaces of processes of guest operating systems (from now on referred to as Guest Virtual Address (GVA)) are mapped into the "VM physical memory" (Guest Physical Address (GPA)). Similarly, the GPAs of all VMs are mapped into the actual physical address space (System Physical Address (SPA)). Because processors supported only a single level of page table memory translation, the translation from guest virtual addresses to guest physical addresses had to be emulated in software. This was addressed by hardware manufacturers with a technology generally referred to as Second Level Address Translation (SLAT). AMD started shipping their implementation, called Rapid Virtualization Indexing (RVI), in 2007, and Intel introduced their Extended Page Tables (EPT) technology in 2008. Both allow the processor to be aware of the GVA to GPA, and the GPA to SPA tables and perform memory translation for both tables in hardware.

Device Virtualization

Contrary to CPU and memory, peripheral devices typically cannot be shared between VMs, because they are designed to be handled by a single driver, not multiple. Subsequently, the hypervisor has to be in complete control of the physical devices, while emulating virtual instances to the VMs. If paravirtualization techniques are used, the virtual device drivers can use hypercalls or other interfaces provided by the hypervisor to manage the state of its virtual devices. Without paravirtualization, the hypervisor has to present emulations of real hardware, for which drivers exist in the guest operating system. Of course, copying the behavior of hardware usually is more expensive than a purpose-designed interface. An advantage is that physical and virtual hardware do not have to be the same. A hypervisor on a system without a NIC can still provide virtual NICs to its virtual machines, and allowing them to communicate over a virtual network. Similarly, a hypervisor connected to a network can choose not to provide NICs to some VMs.

This layer of virtualization does induce too much overhead for some scenarios. Workloads making heavy use of Graphics Processing Unit (GPU) acceleration or 10 Gbit/s networking, with potentially millions of packets per second, are throttled by this virtual device bottleneck. AMD's I/O Memory Management Unit (IOMMU) [27] and Intel's Virtualization Technology for Directed I/O (VT-d) [28, 29] enable giving virtual machines full control over Peripheral Component Interconnect (PCI) devices. This is also known as PCI passthrough. A virtual machine is able to use the device at basically native speed, but other VMs and the hypervisor are prohibited from using the device simultaneously. Some hardware components implement Single-Root I/O-Virtualization (SR-IOV), which allows them to appear as multiple devices, and can thus be passed through to multiple virtual machines. The hardware is designed to be managed by multiple drivers and multiplexes the requests in hardware accordingly. A typical use case is to provide VMs with low-overhead networking using a single SR-IOV-capable NIC [30].

Nested Virtualization

Running a hypervisor inside a virtual environment of another hypervisor, as depicted in Figure 2.3, is called nested virtualization. This is not straightforward and not supported in all scenarios. Microsoft even states that Hyper-V itself is the only nested hypervisor supported to run in a Hyper-V VM. One of the challenges is that memory needs a third level of translation. Also, the top-level hypervisor has to decide whether it has to handle traps itself or has to forward them to the nested hypervisor.

Hypervisor Types

Up until now, the hypervisor has been described to be directly in control of the hardware. Such a hypervisor is referred to as a Type 1 hypervisor or bare-metal hypervisor. VMMs like Xen, VMware ESX, and Hyper-V fall into this category. On the other hand,

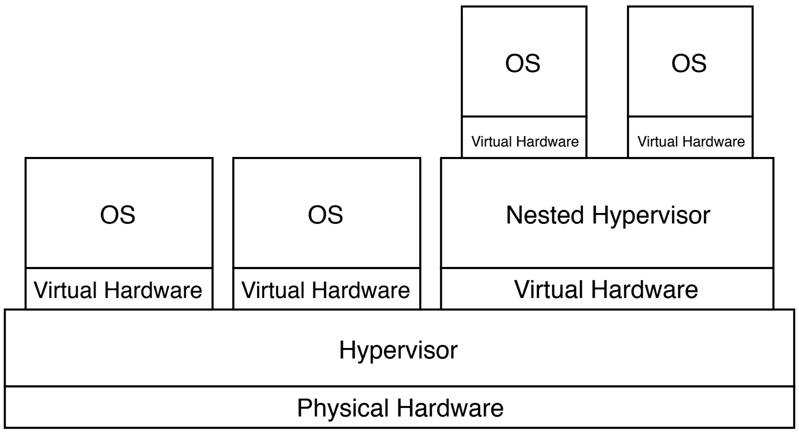


Figure 2.3.: Nested Virtualization

hypervisors like Linux's Kernel-based Virtual Machine (KVM), VMware Workstation, and VirtualBox are Type 2 hypervisors or hosted hypervisors. They are not in sole control of the hardware but run on top of an operating system. They have the advantage of being able to utilize already existing operating system functionalities, including device drivers and CPU scheduling.

There are also different implementation approaches to bare-metal hypervisors. With their ESX hypervisor, VMware chose to build device drivers and all other required functionalities into the hypervisor. This results in a self-contained system without dependencies but also limits possible hardware platforms to those supported by the VMM drivers. Xen implements a microkernel approach. A management VM called dom0 (Domain 0, Domain zero), running, e.g., GNU/Linux, is used to manage the hardware using the operating system's device drivers. Other management tasks are also delegated to dom0.

2.3. Hyper-V

Hyper-V is an x86_64 hypervisor developed by Microsoft, which runs on x86_64 processors that provide hardware virtualization features. The high-level architecture is explained in detail by Microsoft's documentation [32]. Figure 2.4 gives a visual overview of how Hyper-V runs virtual machines.

Similar to Xen, Hyper-V employs a microkernel architecture. One of the virtual machines, which are rather called partitions in Hyper-V context, is the root partition. The Windows operating system running in the root partition works together closely with the hypervisor. It is responsible for managing all hardware devices using their corresponding Windows drivers. It is also the root partition's task to emulate virtual devices to other partitions, as well as other management jobs. The hypervisor is as much as possible used only to delegate tasks into the root partition. Hyper-V's hypercall interface provides many calls that can only be issued by the root partition, and are intended for management tasks,e.g., starting, stopping, and modifying child partitions (guest VMs).

Child partitions are categorized into enlightened and unenlightened partitions, which is the Hyper-V term for paravirtualization. Operating systems aware of being virtualized by Hyper-V can make use of the hypercall interface and the VMBus, which is a shared memory-based communication channel, and used for paravirtualized device communication.

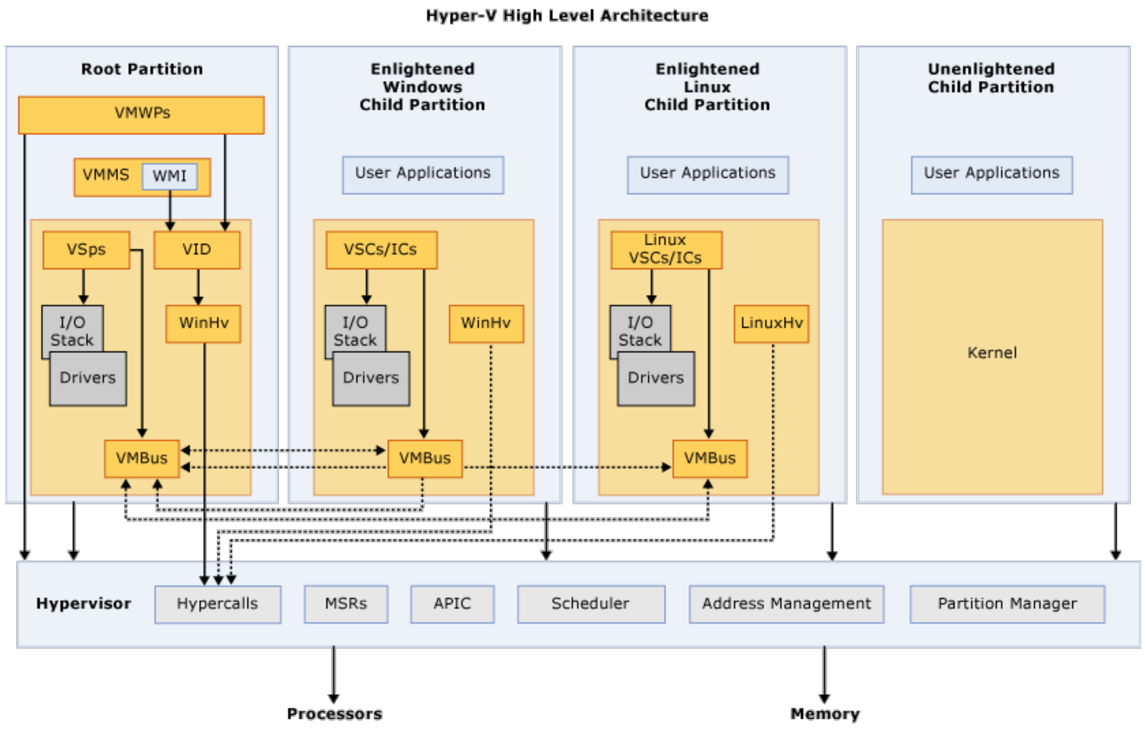


Figure 2.4.: Hyper-V High-Level Architecture [32]

Hypercall Interface

The Hypervisor Top Level Functional Specification [14] describes the hypervisor interfaces for enlightened partitions. This includes how the presence of Hyper-V can be detected, and how information about unsupported features can be queried. A large part of the documentation is concerned with the hypercall interface and a selection of available hypercalls. Because this thesis includes the development of a module that is supposed to perform manually specified hypercalls to Hyper-V, the available calling conventions mentioned in the specification are explained here. There are differences between how 64-bit and 32-bit guests have to issue calls. Here, only the 64-bit variants are presented.

Hyper-V differentiates between simple calls and rep calls (“rep” for “repeat”). Normal calls have a single request for the hypervisor, which is guaranteed to be processed when the hypervisor returns control back to guest code. Rep calls contain one or more independent, identical requests. Hyper-V processes as many of them as possible in $50\mu s$. Then, control is returned to the caller, which has to check if all request were processed. If necessary, the caller can reissue the hypercall with the missing elements.

Both kinds of hypercalls require a hypercall input value, which is 64 bits large. Table 2.2 shows its composition. Based on the call code, the hypervisor knows how to handle the

Bit range	Size [bit]	Name
15:0	16	Call Code
16	1	Fast Flag
26:17	9	Variable Header Size
43:32	12	Rep Count
59:48	12	Rep Start Index

Table 2.2.: Hyper-V Hypercall Input Value

call, because its designated call code identifies every hypercall. The fast flag specifies where the parameters for the call are located. If the flag is 0, the default, “slow”, memory-based calling convention is used. If the flag is 1, the parameters are passed by one of the register-based calling conventions. The calling conventions are explained in detail later. The variable header size field specifies the size of the variable header, if the issued hypercall requires that. The rep count and rep start index are specified for rep calls. The count specifies the total number of requests to be processed. The hypervisor starts processing the elements beginning with the start index. The first time a call is performed, the start index is set to 0. If not all elements were able to be finished, the start index is set to the index of the first unprocessed element, and the hypercall is repeated. All bits not explicitly mentioned in the table are reserved and have to be set to 0.

The hypervisor returns a hypercall result value to inform the caller about details of the call's execution. The result value is 64 bit in size. Its components are shown in Table 2.3. If the call was performed successfully, the result is 0. Otherwise, an error identification

Bit range	Size [bit]	Name
15:0	16	Result Code
43:32	12	Reps Complete

Table 2.3.: Hyper-V Hypercall Result Value

code is placed here. There are many result values defined in the documentation. Amongst others, there are codes to signal an invalid hypercall code, invalid rep counts, invalid parameters, and missing privileges. Reps completed specifies how many rep elements have been processed. This indicates whether a rep call is fully completed or has to be executed again with an increased starting index.

As mentioned in the explanation of the hypercall input value, there are memory- and register-based calling conventions. Table 2.4 lists the expected register setup for the default, memory-based convention. The hypercall input value is supposed to be placed in

Register name	Input/Output	Content
RAX	out	Hypercall Result Value
RCX	in	Hypercall Input Value
RDX	in	Input Page GPA
R8	in	Output Page GPA

Table 2.4.: Hyper-V Default Hypercall Calling Convention

the RCX register. If the requested hypercall expects parameters from the caller, they have to be passed in memory. The caller has to reserve a memory page (4096 bytes), aligned to the page boundaries. Parameters have to be written on that page according to the offset defined by the hypercall documentation. For the hypervisor to find the page, the GPA is stored in the RDX register. If a hypercall does not take parameters, the content of RDX is ignored. Some hypercalls also output data. Again, the caller has to reserve an aligned memory page and store its GPA in register R8. The hypervisor writes the outputs on this page at offsets defined by the documentation. The content of R8 is ignored if a hypercall does not provide outputs. With input value and potentially page addresses placed in the registers, the guest code can transfer control to the hypervisor by using the VMCALL (Intel) or VMMCALL (AMD) instruction, respectively. The hypervisor processes the hypercall, writes output values to the output page, and places the hypercall result value in the RAX register before returning to the caller.

The fast, register-based hypercall calling convention is only usable by calls that take two or less parameters and do not produce output. Table 2.5 displays the register assignment for fast calls. The RCX register is unchanged; it still holds the hypercall input value.

Register name	Input/Output	Content
RAX	out	Hypercall Result Value
RCX	in	Hypercall Input Value
RDX	in	Input Parameter 1
R8	in	Input Parameter 2

Table 2.5.: Hyper-V Fast Hypercall Calling Convention

The up to two input parameters are placed in registers RDX and R8, respectively. If one or both registers are not needed, their contents are ignored. The fast calling convention also requires the call into the hypervisor, which processes the request and stores the result value in RAX.

Hyper-V can support another form of hypercalls, the XMM fast call. On x86_64, there are 16 XMM registers. They are 128 bit wide and were added for SIMD purposes with the Streaming SIMD Extensions (SSE). The XMM fast call register setup can be seen in Table 2.6. Again, RCX contains the hypercall input value. Now, the RDX, R8, and six

Register name	Input/Output	Content
RAX	out	Hypercall Result Value
RCX	in	Hypercall Input Value
RDX	in/out	Input Parameter 1
R8	in/out	Input Parameter 2
XMM0	in/out	Input Parameter 2
XMM1	in/out	Input Parameter 2
...	...	...
XMM5	in/out	Input Parameter 2

Table 2.6.: Hyper-V Fast Hypercall Calling Convention

XMM registers are used to store up to $2 * 8$ bytes + $6 * 16$ bytes = 112 bytes of parameters. Registers that are not needed for parameters can be used by the hypervisor to hold output values. Hypercall execution and result value work as in the other conventions.

3. Related Work

Section 1.3 listed the communities that the contributions of this thesis are relevant to. This chapter takes a closer look at the different community topics, specifically their efforts regarding hypervisors and hypercalls.

Section 3.1 covers robustness testing. Performance isolation is reviewed in Section 3.2. Then, Section 3.3 is devoted to software aging. Finally, Section 3.4 reviews efforts of the performance benchmarking community.

3.1. Robustness Testing

Section 1.2 already used the words of the IEEE Standard Glossary of Software Engineering Terminology to define robustness as "*The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions*" [11]. Thus, one aspect of 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. When reviewing the available literature, this aspect seems to be targeted exclusively. No mention of the "stressful environmental conditions" can be found.

Robustness testing applies to a multitude of different domains: operating systems [33, 34, 35, 36], web services [37, 38, 39], Java server application [40], hypercall interfaces [41, 42], 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 [36]. 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 [37]. This model can be used to generate test cases or campaigns automatically.

One approach of rating irregular behavior that a system shows in response to the test case execution is to use the CRASH scale [36, 41], an acronym for the severity classes Catastrophic, Restart, Abort, Silent, and Hindering.

Ormandy presents an empirical study that tested the robustness of multiple hypervisors [43]. He did not use hypercalls in this work. Instead, the handlers of privileged instructions, as well as emulated I/O devices, were evaluated. The author executed random byte sequences in virtual machines to stress the instruction handling, and random

I/O port accesses to test the emulated devices. A hypervisor failed the robustness test when it crashed. Multiple flawed implementations, which would crash when executing particular instructions or receiving invalid I/O requests, were detected. Among the affected hypervisors were VMware and Xen.

Milenkoski et al. [44] developed a tool, dubbed hInjector, to inject hypercalls into the Xen hypervisor from a Linux guest VM. It was not developed as a robustness testing tool, but as a load generator to evaluate the effectiveness of intrusion detection systems based on virtual machine introspection. However, in a modified form, it has been used for robustness testing in [41] and [42].

Carvalho et al. [42] present the idea of utilizing robustness testing techniques to evaluate the secureness and robustness of the hypercall interface of hypervisors. The paper introduces an approach for assessing a given hypervisor, reports on practical experience with the Xen, and presents encountered challenges.

The evaluation technique begins with studying the hypercall interface, which consists of determining existing calls, together with their parameters and parameter domains. However, as guest operating system kernels are the only instance using the interface, documentation is often scarce. Even for the open-source Xen hypervisor, they had to analyze assembly source files in order to understand all operations.

With the acquired knowledge, tests can be generated. The calls performed in test campaigns are based on a parameter mutation strategy, which defines rules like passing null or empty strings to string parameters or invalid memory locations for pointer parameters. A challenge arising in this step is dealing with more complex, struct-based parameters.

Finally, the prepared tests have to be executed, and the behavioral response has to be monitored to detect and classify occurring failures, e.g., crashes, abnormal terminations, or error codes. Here, more problematic are the silent failures, which might not be directly detectable but affect the results of succeeding tests. Also troublesome are calls that affect the state in such a way that a reboot or even reinstall is necessary to restore the original state.

Gonçalves et al. [41] further assess the applicability of robustness testing to the Xen hypercall interface. As a target environment, a public cloud IaaS infrastructure is chosen in which one of the guest VMs is compromised and has the ability to perform arbitrary hypercalls. On non-compromised VMs, a benchmark is run that measures hypervisor and VM performance with database workloads.

Their approach is similar to the one presented by [42]. To inject hypercalls, an altered version of the hInjector [44] was used. The authors had to gather knowledge about the Xen's hypercall interface to create test cases with specially crafted invalid inputs. Their generated test cases covered 66.6% of the calls available.

One focus of the paper is the rating of irregular system responses. They found that the CRASH scale was not descriptive enough, and defined new categories like `GUEST_CRASH`, `GUEST_KILLED` (by the hypervisor), and `GUEST_HANG`, indicating an unresponsive VM.

These hypercall robustness testing papers inspired the topic of this thesis, hence also the robustness-focused title. However, the differentiation is that we prioritize building a framework that can be used to test any hypervisor, instead of using specialized testing tools, e.g., for Xen, and concentrating on generating test cases. Although, the Hyper-V case study in Chapter 7 has to do exactly that: Learn about the available hypercalls and their parameters. Instead of trying to cause crashes with invalid calls or parameters and classifying the impact, we search for valid calls that can be executed arbitrarily often, and measure the hypervisor's response to varying load levels of those calls. Putting the system under stress using high hypercall load levels can still be considered robustness testing, according to the IEEE definition.

3.2. Performance Isolation

Virtualization and cloud computing enable resource sharing on a large scale. While this has the advantage of better utilizing physical resources, it can lead to interferences between customers. To prevent that, cloud providers employ measures to isolate users and guarantee them a minimum level of performance [45].

Performance isolation is often directly associated with cloud computing and virtualization. However, there are other systems requiring performance isolation, as well.

Verghese et al. [46] present a performance isolation framework that offers fair resource sharing on conventional multi-core processors. Fair scheduling techniques are used to share CPU time and disk accesses between processes.

Multiple works deal with the performance interference aspect in regard to I/O workloads. In [47], Pu et al. performed measurements of various co-location scenarios. Amongst other facts, they found that running multiple isolated network-intensive workloads on the same hardware induces high context switching overhead. By contrast, co-located CPU-intensive I/O workloads suffer from high CPU contention.

Yang et al. [48] proposed vExplorer, a framework for distributed VM I/O performance measurements. The framework is able to execute representative I/O accesses, and thus generate I/O load in the system.

While the impact of hypercalls appears to be covered seldom, one paper measured that hypercall load resulted in at least 40% reduced I/O throughput [49]. The study was conducted with Xen. The `fork` function, which is used to create new processes, issues hypercalls to allocate resources for the new process. Repeated execution of `fork` was used to generate the hypercall load.

The framework developed in this work allows specifying hypercall loads exactly, and can thus be used for more extensive performance isolation testing in the future.

3.3. Software Aging

Software aging refers to the phenomenon that software that is running for extended periods of time tends to experience performance degradations or crashes. Rejuvenation describes techniques to counteract the aging process and mitigate more severe issues. Rebooting is one way to rejuvenate a system completely, but depending on the field of application, there are also less intrusive measures.

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 [50].

A multitude of works deals with the basics of software aging and rejuvenation [51] to [65]. There is also a variety of papers taking a look at hypervisors regarding software aging.

To give an example of the content, in [66], 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.

Machida et al. investigate bug reports of five major open-source projects regarding software-aging in [67]. 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.

Barada and Swain give a survey on software aging and rejuvenation studies in virtualized environments in [68]. 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.

Hypercall campaigns might be able to accelerate aging behavior and thus make it more accessible.

3.4. Performance Benchmarking

A performance benchmark is a test to quantify the performance of a system and represent it as a single metric or a set of metrics. Benchmarks allow comparing the performance of different systems based on their achieved metrics.

With complex systems, e.g., CPUs, there is no way to find an absolute performance ranking, because there are many disciplines. Staying with the CPU example, there are integer arithmetic, floating-point arithmetic, and memory or cache accesses, to name but a few. A processor achieving the highest score in a benchmark testing pure integer calculation performance might be the slowest when it comes to calculating floating-point numbers. Subsequently, it is essential to compare the results of a benchmark whose workload is representative of the required performance. This applies to the benchmarking of virtualized systems and hypervisors, as well. Because of their complexity, it is not trivial to design benchmarks able to characterize their performance.

Only a few works are covering the topic of virtualization or hypervisor benchmarking. However, two papers containing hypercall benchmarking have been found. Le presents two approaches to secure Xen's hypercall interface [69]. He introduces authenticated hypercalls, which require virtual machines to attach a policy and a message authentication code as additional arguments to hypercalls. Xen is then able to verify the passed policy and from which memory location the call was issued. Based on that information, it can decide whether the hypercall should be processed or ignored. In the other security approach, the hypervisor simply stores a table of valid memory call sites, and filters calls according to that.

The author then performed hypercall benchmarking, based on the hypercall execution time metric, which was able to quantify the overhead that the security mechanisms induced.

Dall et al. redesigned the KVM hypervisor to work on the ARM processor architecture [70] efficiently. Again, the authors benchmarked the execution times of hypercalls to demonstrates the superior efficiency of their approach.

In the load testing campaign, we measure hypercall execution times, as well. However, in this case, we are not interested in comparing the results of different machines. The measurement variations of a single machine are crucial. The load testing can be seen as a robustness benchmark, though.

4. Approach

This chapter is supposed to describe the design decisions that have been taken in the development of the hypercall testing framework. First, Section 4.1 identifies a generic hypercall testing workflow and designs a framework architecture accordingly. Then, Section 4.2 develops a specification for a hypercall campaign description language. Concluding the approach of the framework, Section 4.3 discusses the hypervisor-specific parts of the framework in regard to Hyper-V. And concluding the whole chapter, Section 4.4 presents the idea of a hypercall monitoring tool.

4.1. Hypercall Testing Framework Design

To identify the workflow of hypercall testing, it makes sense to start with abstracting the framework to a black box, which internal workings are unknown, and only look at the required interactions with its environment. These interactions are either (1) input from the user, (2) output to the user, or (3) side effects, e.g., an observable increase of CPU utilization.

The framework has one of each category, which is graphically shown in Figure 4.1. It has to be able to accept information from the user that describes which hypercalls are supposed to be performed at which time. On the contrary, for every supplied hypercall testing campaign, the framework is supposed to deliver information about how the execution of the campaign went, e.g., the required execution time or which values were returned by the hypervisor. Both the test campaign and the report should be in a human-readable and -understandable format. Additionally, the framework has to cause the side effects that result from the execution of the specified hypercalls. It is theoretically possible for

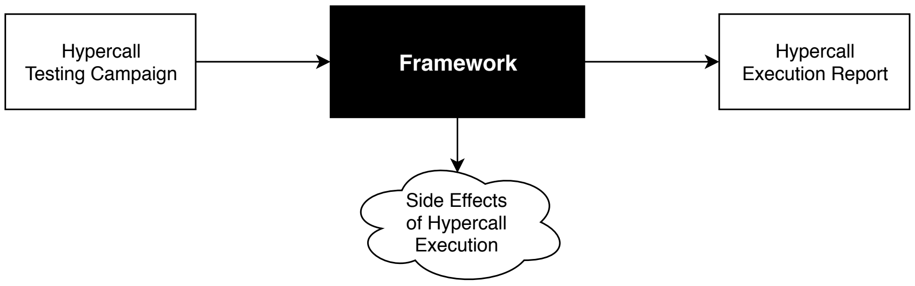

Hypercall Testing Campaign
Framework
Hypercall Execution Report
Side Effects of Hypercall Execution

Figure 4.1.: Inputs, Outputs, and Side Effects of the Hypercall Testing Framework

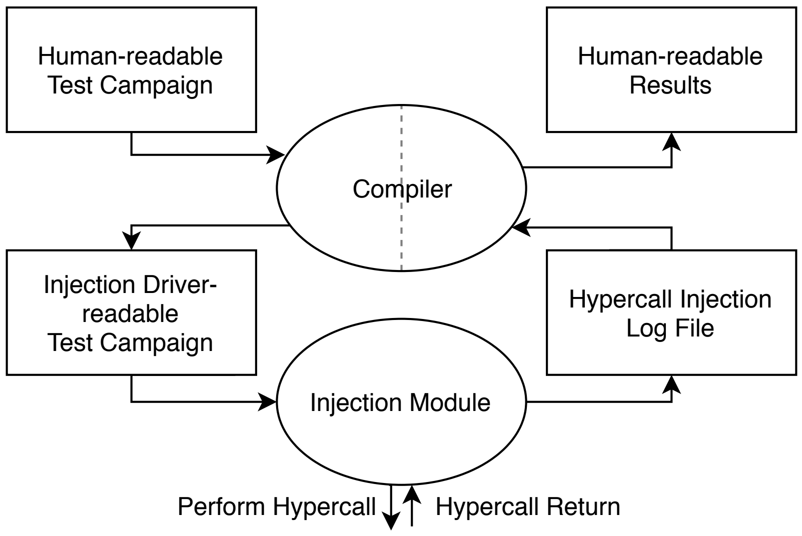

Human-readable Test Campaign
Human-readable Results
Compiler
Injection Driver-readable Test Campaign
Hypercall Injection Log File
Injection Module
Perform Hypercall
Hypercall Return

Figure 4.2.: Architecture of the Hypercall Testing Framework

the framework to deliver a valid execution report without actually executing hypercalls, e.g., by simulating the whole test setup and determining result values, execution times, and other information that way. However, the framework is supposed to produce the side effects that come with the execution of hypercalls. Then, it is possible to capture other metrics of the system the framework running is on and quantify the impact of these side effects, e.g., that the hypervisor requires more CPU time while when hypercalls are issued with high frequency.

The black box behavior described could be implemented as a single module in a monolithic way, but there are several additional requirements that suggest distributing the workflow across multiple distinct components. Taking a look at restrictions of hypercall execution, Intel and AMD implemented the VMCALL and VMMCALL instructions to be non-privileged instructions. While thus hypercalls can theoretically be initiated from user space, Hyper-V, Xen, and KVM do only process calls executed from Ring 0. A custom operating system, or standalone application, running self-contained with full privilege directly in a virtual machine is not a practical solution. First, this would require re-implementing hardware- and hypervisor-specific boot and initialization routines. Furthermore, input of the test campaign and output of the results is difficult. A hard disk driver, network driver, or another form of communication channel would have to be implemented. Also, with this method, there would be no way to retrieve metrics from the operating system, which is managing the VM, because there is no operating system. Finally, this approach is not applicable to inject hypercalls from the privileged VM in microkernel-based hypervisors.

These considerations lead to the conclusion that hypercalls should be performed from the kernel of a standard operating system. Windows, as well as Linux, support loading and executing code into the kernel during runtime, in the form of drivers and loadable kernel modules, respectively. From now on, this component is generically referred to as kernel module or injection module. Both have to be implemented using the C language. For other operating systems, this probably applies as well, or at least a similarly low-level language is used. Managing strings in languages like C is a cumbersome task, which suggests that the injection module should not be responsible for parsing the human-understandable test campaign and generating the final execution report.

Concluding the design considerations of the hypercall execution framework and thus answering the first research question:

RQ1: *What is the workflow to perform hypercall injection test campaigns?*

The identified workflow, which translates to the proposed framework architecture, is shown in Figure 4.2. The user has to design a test campaign, which he can express in a text-based format. The compiler translates this into a binary format, which describes the same campaign but is as easy as possible to process for the injection module. The campaign is executed by the injection driver, which also logs details about the execution in a format that is easiest for it. The information encoded in the binary log could be translated directly into a textual format. Having only information about execution times or result values is probably of little use, though, because the information about which call the results come from is missing. Therefore, the binary log file, combined with the original campaign file, is translated to a complete, informative, human-readable report about the hypercall execution results.

RQ2: *Which extent of workflow abstraction is achievable?*

The second research question of this thesis asks which parts of the proposed framework can be implemented generically, and which are too specific to the hypervisor under test. To answer this question, it is essential to know how much hypercall interfaces differ from each other. This is stated in the first goal:

G1: *Compare the hypercall interfaces of different hypervisors and identify similarities and differences.*

When discussing the injection kernel module, it already has been mentioned that the hypervisors Hyper-V, Xen, and KVM accept only hypercalls executed from Ring 0. The mentioned hypervisors all use registers and, in some cases memory (with pointers to it placed in registers) to store hypercall parameters. This is unsurprising, as they are no other viable options, given the processor architecture. It appears that if the binary campaign format instructs the injection module which value to place into which register or at which memory address, a generic module is possible. But looking at the default calling convention explained in Section 2.3, there are hypervisor-specific details involved that are not describable with a supposed to be simple binary format, e.g., reserving page-aligned memory for the input and output page.

The injection module cannot be generalized for another reason, as well. A Linux LKM (Loadable Kernel Module) in a Linux VM can be used to issue hypercalls to Hyper-V. But if privileged hypercalls should be executed, this is only possible from Hyper-V's root partition, which has to run a Windows operating system and thus requires a Windows driver to perform the hypercalls. In turn, Windows is currently not supported to run in the dom0 of Xen. Subsequently, for each hypervisor, a specific injection module should be implemented. The binary formats for campaign input and log output should also be specific to and optimized for the module.

With the binary formats being hypervisor-specific, as well as the human-readable report, because it depends on the log information provided by the kernel module, the compiler cannot be fully generic. If the human-readable input can follow a generic structure and still describe hypercalls to any hypervisor, parts of the compiler can be designed generically, though. As said before, hypercall calling conventions differ in which values are placed in which registers or memory addresses. Accordingly, it is possible to describe all necessary information of any hypercall with a list of key-value pairs that map storage locations to their values. Apart from describing the hypercalls that have to be executed, the campaign also has to define if and how long the kernel module should wait between calls.

So, the proposed design of the compiler consists of a generic part, which has the responsibility of parsing the human-readable campaign. For every defined hypercall or delay

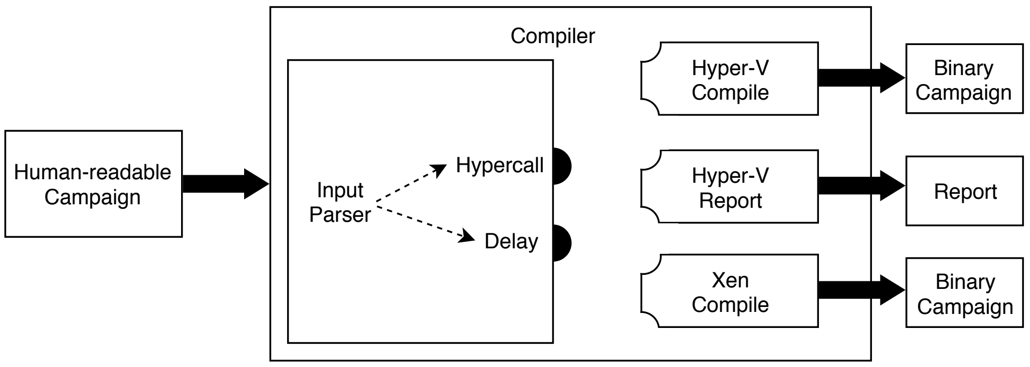


Figure 4.3.: Architecture of the Campaign Compiler

encountered, it transmits the necessary information to a specialized component, dubbed campaign listener, which can then perform a concrete action. The data transmitted for hypercalls is a list of key-value pairs, and for delays, it is an integer value specifying the duration of the delay. Figure 4.3 gives a visual overview of the architecture. The campaign listener to be used is chosen based on the task to be accomplished. When a textual test campaign is supposed to be translated into a binary campaign for an injection driver, a listener implementation that encodes the calls and delays into the injection driver's binary format is chosen to be called from the input parser. To compile a campaign for another injection module, the corresponding encoder has to be chosen. The campaign listener interface can also be used to generate reports. Such a reporting module has to have access to the log output of the injection module. Every time the input parser notifies about a call or delay, the report implementation has all the preliminary information about the action, and can combine that with the data from the binary log to generate an informative, human-readable report entry. The compiler has to define a concrete interface for the campaign listeners, which the input parser can call into, and a comfortable way of using implementations of this interface to process a campaign file.

In conclusion, the generalizable aspects of the framework include:

- The design of a format or language that is suited to describe hypercall campaigns
- A parser that can extract hypercalls and delays with their required information out of a campaign file
- An interface that the parser can call into, which allows notification about hypercalls and delays, including additional information
- Means to add and use different implementations of this campaign listener interface easily

Whereas the following aspects are dependent on the hypervisor that is supposed to be tested:

- A campaign listener implementation generating binary campaign files for the injection module
- The injection module
- A campaign listener implementation that uses information from the campaign file and the binary log to produce a human-readable report

4.2. Hypercall Campaign Description Language

This section is devoted to designing a format for human-readable test campaigns, which should be tailored to be comfortable and easy to use for the user writing the campaigns.

The previous section already established that in order to be usable by humans, campaigns have to be text-based and that describing hypercalls and delays is the goal. Hypercalls need a list of key-value pairs and delays a duration value. However, there are more features to be considered, which are listed and explained in the next subsection.

4.2.1. Requirements

- **Imperative Style**: A campaign should be imperative, meaning it should consist of a sequence of instructions.
- **Hypercalls**: One of the available instructions should be requesting a hypercall. The instruction has to specify a list of key-value pairs that define all details of how to perform the call. The list is passed to the selected campaign listener interface implementation.
- **Delays**: Another available instruction has to be to request a delay before another hypercall. A numeric value denoting the duration has to be passed, which in turn is supplied to the active campaign listener.
- **Variables**: Hypercall and delay requests should not only be able to take immediate values as their parameters, but they should also accept variables. The data types necessary are described in the next entries.
- **Numeric Values**: Numeric values are needed to specify delay durations and probably most values for hypercall parameters. For this reason, it should be possible to specify numbers in decimal, as well as binary and hexadecimal format.
- **Strings**: Campaign listeners might choose to use names instead of values in some cases, e.g., to refer to hypercalls by name instead of specifying their call code. Also, strings are used as the key in key-value pairs.
- **Key-Value Pairs**: To annotate values with a description, key-value pairs pair a string as the key with any other data type as a value.
- **Lists**: Lists are supposed to be a collection of values. There are no restrictions on data types or homogeneity of the contained elements.
- **Operations**: There should be operators to allow for the combination of values. One of the main goals of all features of the campaign definition language is to construct lists of key-value pairs to describe hypercalls. Therefore, there should at least be the possibilities to add elements to lists, to combine lists, and to perform arithmetic operations on numeric values.
- **Loops**: For stress testing, it might be necessary to describe millions of hypercalls in one campaign. The language should feature loops in order to allow the repetition of sequences of instructions. It should be possible to nest loops without restrictions. Loops should also allow iterating over a list of values, processing one element at a time.
- **Built-in Instructions**: The language should provide a type of more complex instruction, possibly parameterized, which resolve to usable values, and provide specialized functionality. To help robustness testing, there should be an instruction that resolves to boundary integer values for a given bit size, e.g., 0, 1, the signed maximum, and the unsigned maximum. For fuzzing purposes, an instruction to retrieve uniformly-distributed random values should exist. Random values following a negative-exponential distribution are commonly used to generate load profiles [71]; thus there should also be an instruction to generate those.

- **User-defined Instructions**: To be able to re-use instruction sequences without copy-pasting of text, it should be possible to it parameterize and call instructions, in a manner similar to functions, methods, and procedures in classical programming languages.

4.2.2. Choice of Realization

With all the required features considered, it has to be decided on a concrete implementation. There are three major options available, each providing different advantages and disadvantages.

The first possibility is to utilize one of the standardized file formats for storing structured data. Commonly used formats used are JSON (JavaScript Object Notation), XML (Extensible Markup Language), and YAML (YAML ain't Markup Language). To show how a campaign could look like using each of these formats, examples are provided. The campaign to be encoded consists of a loop that repeats a hypercall, followed by a delay, for a thousand times. Listing 4.1 contains the JSON variant, Listing 4.2 the campaign in YAML, and Listing 8.1 in the Appendix shows the same using XML.

Listing 4.1: Example: JSON

```
[
 {
  "instruction" : "loop",
  "loopcount" : 1000,
  "loopbody" : [
   {
    "instruction": "hycall",
    "rcx": 3425,
    "rdx": 8
   },
   {
    "instruction": "delay",
    "duration": 50
   }
  ]
 }
]
```

Listing 4.2: Example: YAML

```
---
- instruction: loop
  loopcount: 1000
  loopbody:
  - instruction: hycall
    rcx: 3425
    rdx: 8
  - instruction: delay
    duration: 50
```

An advantage of using one of the existing data formats is the existence of parsers, which can convert the text into structured data accessible in the programming language of choice. However, the parsers only check the syntax of the input. They cannot verify that the semantics actually conforms with the way hypercall campaigns should be defined. There is another significant disadvantage, though. The standardized file format syntax force the user writing a hypercall campaign to include text that does not add information to the campaign but is necessary to comply with the format. Out of the three given examples, XML is the most verbose, with the necessity of providing opening and closing tags. JSON relies on braces and brackets to structure data, and YAML uses indentation. Still, the mandatory uniform structure causes overhead and hampers readability, especially when it comes to advanced features like variables, operations, or definition of user-defined instructions.

The second approach is not to store the campaign as data and parse it, but to write the campaign as code of a programming language. The campaign can use the full functionality of the language, which fulfills most of the considered description features without any implementation expense. The compiler or interpreter of the language also takes care of ensuring correct syntax and semantics. The campaign listener interface can be implemented as a library, providing functions to call when the campaign wants to signal a hypercall or delay request. Some programming languages are more suited than others, especially regarding the complexity of defining data structures like lists. Python, with its property of being loosely typed, and the native syntax for dictionaries (basically a list of key-value pairs), should make for a suitable choice for describing hypercall campaigns with low overhead. An example is shown in Listing 4.3.

Listing 4.3: Example: Python

```
import campaignlistener_interface as ci

def repeated_execution(count, del_duration):
    for c in range(count):
        ci.hypercall(["rcx": 3425, "rdx": 8])
        ci.delay(del_duration)

repeated_execution(1000, 50)
ci.delay(1000)
repeated_execution(1000, 10)
```

With Python as a sample language, it is safe to say that it not only provides all considered features but brings a lot of additional functionality. Campaigns are comfortable to describe, without much textual overhead, and the result is also easy to read and comprehend. Also, there is no inherent disadvantage of choosing this approach.

The final major realization possibility is the design and implementation of a Domain-Specific Language (DSL). It is probably not possible to implement a language that is significantly more concise and comprehensible than, e.g., Python. Also, a DSL requires implementing an interpreter, making sure it functions as intended. And then, despite the effort, there still will be a lot less functionality than in an existing language. Nonetheless, this approach is selected. While there is no real advantage with the currently proposed campaign listener-based compilation, a custom language with limited functionality is better suited to compile into a binary format that does not only list calls and delays but is a program itself. This might be necessary for extensive campaigns, where storing every single call and delay might create an unusably huge binary campaign file, but using loops and variables could keep the file small. The DSL proposed is named Hypercall Campaign Description Language (HCCDL), and its syntax and semantics are defined in the next subsection.

4.2.3. Language Design

The language will be defined in a bottom-up approach, starting with data types, going over how operators and values can be used in expressions, to finally defining procedures and the global campaign structure.

There are four data types in HCCDL. Two of which are elementary, the other two are recursive data types. There are numerical values, which can be expressed in the decimal format by writing a sequence of digits (`12`, `0`, `25746523`, `01`). Leading zeros are allowed, and there is no limit on the size. The binary representation of numerical values starts with `0b`, followed by a sequence of zeros and ones (`0b0010`). Similarly, a hexadecimal number is defined by `0x`, followed by a sequence of hexadecimal digits (`0xff00f8`). The other elementary type is string: A possibly empty sequence of non-quotation mark-characters, enclosed by quotation marks (`"this is a string"`, `"[{}+"`, `""`). Recursive data types contain other elementary or recursive data types. Key-value pairs consist of a string as the key and a value of any other data type. Key-value pairs are created using the `->` operator. On the left side, there has to be a string, and on the right side, a value (`"key1" -> 2`, `"key2" -> "a value"`, `"key3" -> ("nested" -> 7)`). Finally, the last data type is the list. Lists contain zero or more values of any data type and do not have to be homogeneous. Brackets are used to describe lists (`[]`, `[1, 2, 3]`, `["mix", 0b10, "k" -> "v", ["nested", "list"]]`).

Variables can be used to store values. The language is loosely typed, so no data type has to be defined for variables. In fact, a single variable can store, e.g., a string at first, and later get assigned a numeric value. Assignments are performed using the `=` operator (`var1 = 1`, `var2 = [1, 2]`). When a variable name is used without being on the left side of `=`, it evaluates to its stored value.

There are several more operators, which can be used to modify and combine values. A numbers sign can be changed using the unary `-` (`-12`, `-0xf8`). It is possible to use the unary `+`, which has no effect, though. It can be used for explicitness. The binary operators `+`, `-`, `*`, `/`, and `%` can be used with numbers, performing their conventional operation. `+` is also applicable to merge lists (`[1, 2] + [3, 4]`), and to add elements to lists (`[1, 2] + 3`, `0 + [1, 2]`). With two strings as parameters, `+` will evaluate to their concatenation. The key, as well as the value, can be extracted from a key-value pair using `.key` and `.val`, respectively (`("a" -> 1).key`, `("a" -> 1).val`). List elements can be retrieved from a list by denoting the index in brackets after the list (`["one", "two", "there"][1]`). The first element has index 0, so the example would evaluate to `"two"`. Operators differ in their precedence, as defined in Table 4.1.

Precedence	Operators
1	`[]` (list access)
2	`.key`, `.val`
3	`+`, `-` (unary)
4	`*`, `/`, `%`
5	`+`, `-` (binary)
6	`->`
7	`=`
8	`[]` (list creation)

Table 4.1.: HCCDL Operator Precedence

All of the elementary values and operators mentioned up until now are considered expressions. Even a variable assignment is an expression; it evaluates to the value assigned. Expressions can be nested arbitrarily deeply into each other, as long as the types match the operators' constraints. To fulfill the requirement of imperativeness, statements are introduced. An expression followed by a semicolon is a statement (`3;` (useless statement), `x = "k" -> ([1,2,3] + [4, 5] + 6)[4 + (-2)];`). Another statement is the for loop. It takes a list of values, a variable name, and a statement. For every element in the list, the element is assigned to the variable name, and the statement is executed. To not only be able to execute a single expression in a loop, a sequence of statements enclosed by braces is also a statement. An example is given in Listing 4.4.

Listing 4.4: HCCDL For Loop Example

```
list = [1, 2, 3, 4, 5, 6]
doubled = []

for (i : list) {
    double_i = 2 * list[i];
    doubled = doubled + double_i;
}
```

The requirements demand the existence of “user-defined instructions”, similar to functions in the C language. Thus, in HCCDL, the concept of procedures exists. They can take a number of parameters, and consist of a sequence of statements. The parameters basically act like pre-initialized variables. For uniformity reasons, HCCDL follows the procedural programming paradigm and requires that statements can exist only inside of procedures. Invoking other procedures is an expression, as well. In the case of user-defined procedures, a procedure call will evaluate to the evaluation result of the last executed expression in the called procedure. Listing 4.5 shows how procedures are defined and called.

Listing 4.5: HCCDL Procedure Example

```
proc do_something(parameter1, parameter2) {
    ["para1" -> parameter1, "para2" -> parameter2]
}

proc caller() {
    first = do_something(1, 2);
    second = do_something("hello", "world");
}
```

The `do_something` procedure wraps its parameters in a list of key-value pairs. Because this creation of the list is the last expression in this procedure, it is also what a call of this procedure will evaluate to. Thus, at the end of the `caller` procedure, `first` will have the value `[1, 2]`, and `second` will have the value `["hello", "world"]`. The compiler will start the campaign by manually calling the procedure with the name `main`, which has to exist.

As of now, we have defined a language capable of performing calculations and creating data structures, but there has been no mention of how to issue hypercalls and delays. Those are provided as built-in procedures. They are called the same way as user-defined ones but are handled differently by the compiler. Following built-ins are available:

- `delay(duration)`: Requests a delay of length duration. Duration has to be a numeric value. Because procedure calls are expressions, a delay call has to evaluate to some value. This call is only relevant due to its side-effect of requesting a delay; thus, it returns a special value `None`, with which nothing can be done.

- `hcall(params)`: Requests a hypercall. `params` has to be a list of key-value pairs, which describe the hypercall to be performed. Which entries are expected depends on the active campaign listener implementation, and is not part of the generic language. `hcall` also evaluates to `None`.

- `integerBounds(bitcount)`: Evaluates to a list containing the values 0, 1, the signed maximum, as well as the unsigned maximum for integers of the size `bitcount`.

- `randExp(scale)`: Evaluates to a random number, following a negative-exponential distribution with scaling factor `scale`. Beware that such random values are usually floating-point values. This procedure returns random samples rounded down to the nearest integer. `scale` has to be a numerical value.
- `randomUniform(bitcount)`: Evaluates to a random integer lying in the interval [0, $2^{bitcount}-1$]. According to the uniform distribution, all possible outcomes are equally likely. `bitcount` has to be a numerical value.
- `range(lower, upper)`: Evaluates to a list of integers of the format [`lower`, `lower`+1, `lower`+2 ..., `upper`-1]. Thus, the lower bound is inclusive, and the upper bound is exclusive. Both parameters have to be numeric values.
- `rangeStep(lower, step, upper)`: Evaluates to a list of integers of the format [`lower`, `lower` + `step`, `lower` + 2*`step`, ..., `upper`-1], if `lower` + n*`step` = `upper`-1, otherwise the next smaller number in the sequence is the last element of the list. All three parameters have to be numeric values.
- `signedMax(bitcount)`: Evaluates to the signed maximum of an integer with size `bitcount`, using the two's complement representation. This equals $2^{bitcount-1} - 1$. `bitcount` has to be a numerical value.
- `unsignedMax(bitcount)`: Evaluates to the unsigned maximum of an integer with size `bitcount`. This is $2^{bitcount} - 1$. `bitcount` has to be a numerical value.

While procedures can exchange values by passing parameters, this might cause textual overhead for values that are needed in many places. Therefore, global variables are introduced. They can be either uninitialized or initialized with a numeric value. To allow for more complex initializations, the user has to possibility to write an `init` procedure, which, if it exists, is executed by the compiler before the main procedure. Listing 4.6 demonstrates the usage of global variables.

Listing 4.6: HCCDL Global Variable Example

```
uninit1, uninit2;
init_jm = 396;

proc init() {
    uninit1 = [1, 2, 3];
    uninit2 = ["a" -> init_jm, "b" -> uninit1];
}

proc main() {
    x = uninit2;
    delay(init_jm);
}
```

A hypercall campaign can be composed of uninitialized and initialized global variables, an `init` and a `main` procedure, and other user-defined procedures. The `main` procedure is the only mandatory component.

There is an additional remark to specify, which did not fit into the flow of the bottom-up explanation. Identifiers, which are the names of variables, parameters, and procedures, are allowed to consist of lower- and upper-case letters, digits, and underscores. However, the first character cannot be a number.

This concludes the definition of the HCCDL - Hypercall Campaign Description Language. The implementation of a compiler that is able to parse the language, evaluate the campaign, and deliver hypercall and delay requests to a hypervisor-specific listener is described in Section 5.1. The rest of this chapter is dedicated to explaining how Hyper-V-specific components for the framework are designed.

4.3. Framework Implementation for Hyper-V

There are three places in the framework that require a hypervisor-specific implementation. The first is part of the HCCDL compiler. It is one of the campaign listener implementations, which are called by the compiler every time a campaign requests a hypercall or a delay. The purpose of this implementation is to encode the calls and delays into a binary campaign file that is understandable for the kernel-based hypercall injection module. Speaking of, this kernel module is the second hypervisor-dependent component. Finally, the third module is an implementation for the campaign listener interface, as well, but this time, its task is to match hypercalls and delays with result values from a binary log file produced by the injection module. It is supposed to then output a human-readable report about the execution of the hypercall campaign.

While it would make sense to describe the approach taken for the compiler interface implementations directly after the definition of the language, the details of the injection module take precedence. The reason is that the interface implementations both interact with binary files of the kernel module; subsequently, it makes sense to describe them, together with the module, first. This is the content of Section 4.3.1. Then, Section 4.3.2 covers the listener interface implementations for Hyper-V.

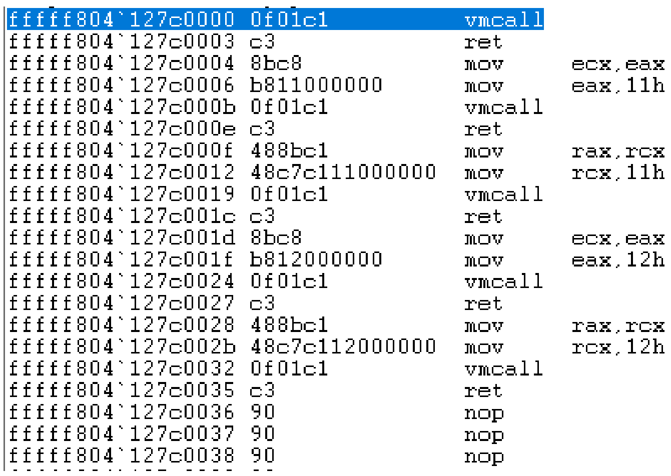
4.3.1. Hypercall Injection Module

The hypercall injection module is supposed to take a binary campaign as an input, perform the campaign, which means to execute hypercalls as described, and wait specified delays before issuing the next. Furthermore, information about the execution should be written to a binary log file.

The choice of how to implement the kernel module is easy, because there is no choice. The problem has already been addressed during the design of the framework in Section 4.1. As mentioned there, it is possible to create a Loadable Kernel Module (LKM) for Linux or any other operating system that can perform hypercalls to Hyper-V. Yet, because many calls are only usable by the root partition, which runs a Windows OS, a Windows driver is the only way to test all of Hyper-V's available hypercalls.

Section 2.3 already went into detail about which calling conventions are supported by Hyper-V: the memory-based, default call; the register-based fast call for hypercalls with two or fewer parameters and no output value; and finally, the XMM register-based extended fast call for hypercalls with up to 112 bytes of parameters. In this thesis, we restrict the injection driver to only support the memory-based calling technique. This does not really prune any functionality regarding executable calls, because all hypercalls can be issued using this convention. However, due to the complexity of handling variably sized inputs, the driver also does not support rep calls and calls with variable header size. With these restrictions in place, the hypercall input value (see Table 2.2) reduces to only the hypercall call code, stored in the bits from 15 down to 0. So, the binary campaign has to provide the call code of each requested hypercall, which the driver has to store in the RCX register.

For the driver to comply with the memory-based calling convention, it also has to reserve two memory pages (4096 bytes in size, aligned to 4096-byte memory boundary). One of them has to be filled with parameters, which have to be supplied by the campaign. The GPA (Guest Physical Address) of this page has to be placed in the RDX register. The



```
fffff804`127c0000 0f01c1           vmcall
fffff804`127c0003 c3               ret
fffff804`127c0004 8bc8             mov     ecx,eax
fffff804`127c0006 b811000000       mov     eax,11h
fffff804`127c000b 0f01c1           vmcall
fffff804`127c000e c3               ret
fffff804`127c000f 488bc1           mov     rax,rcx
fffff804`127c0012 48c7c111000000   mov     rcx,11h
fffff804`127c0019 0f01c1           vmcall
fffff804`127c001c c3               ret
fffff804`127c001d 8bc8             mov     ecx,eax
fffff804`127c001f b812000000       mov     eax,12h
fffff804`127c0024 0f01c1           vmcall
fffff804`127c0027 c3               ret
fffff804`127c0028 488bc1           mov     rax,rcx
fffff804`127c002b 48c7c112000000   mov     rcx,12h
fffff804`127c0032 0f01c1           vmcall
fffff804`127c0035 c3               ret
fffff804`127c0036 90               nop
fffff804`127c0037 90               nop
fffff804`127c0038 90               nop
```

Figure 4.4.: Contents of Hyper-V's Hypercall Overlay Page (Intel)

second page is the output page, which is written to by the hypervisor during hypercalls. So, for the output page, no information is required from the campaign. The driver only has to store the GPA of the output page in register R8 when a call is issued.

Section 2.3 also already explained that, when registers and memory are set up, control is transferred to the hypervisor either by the VMCALL or the VMMCALL instruction, depending on whether the system is running on an Intel or AMD CPU. However, it did not mention that the hypervisor takes care of abstracting the architectural differences away.

Hyper-V has the possibility to deploy overlay memory pages to the address spaces of its partitions. They are called overlay pages because the normal page with the address at that the overlay page is deployed is no longer accessible. Instead, reads and writes happen to the overlay page. Hyper-V uses such an overlay page to standardize the way hypercalls are invoked by placing the correct instruction at the beginning of the page. Thus, instead of directly executing the correct instruction, hypercall invocation code can redirect execution to the address of the overlay page, where the hypervisor placed the correct instruction. The second instruction conveniently returns execution to the caller. The contents of the hypercall page on an Intel machine are displayed in Figure 4.4.

The figure is a screenshot of the Windows Debugger (WinDbg). Each line corresponds to one instruction. The leftmost column shows the instructions' beginning memory addresses, specifically GVAs (Guest Virtual Address). Right of those are the bytes that encode the instruction, displayed in hexadecimal representation. And the rightmost column shows the instructions' assembly representations. As described, at the beginning of the hypercall page is the VMCALL, followed by a RET, which jumps back to the caller. There are also other calling variants following, which move some registers before the VMCALL, but they are not relevant to the injection driver design.

So, at which memory address does Hyper-V deploy this overlay page? In fact, the guest operating system chooses the location and tells the hypervisor by writing it to one of the Model-Specific Registers (MSRs), which are one kind of control registers available in the x86_64 architecture. It is possible to read the MSR afterward and retrieve the specified address. However, this address is a GPA, but execution can only be redirected to GVAs. The driver could either traverse the page table to find the existing translation to this GPA or add an entry to the page table that maps an until now unused GVA to the target GPA. When a GVA is established, the driver can invoke hypercalls by executing a CALL instruction and passing the hypercall page GVA.

After performing a call, the hypervisor has placed the result value in the RAX register.

Depending on the call, it might also have written to the output page. Thus, it is crucial that the driver is able to log these values. Additionally, the driver should be able to take timestamps right before and after performing a call, to measure how long the hypervisor took to process the request. The same can be done for delays to verify that they are carried out accurately.

With the necessary details discussed, it is possible to address the third research question:

RQ3: *How to inject hypercalls into Hyper-V?*

It is answered by providing the course of action of the hypercall injection Windows driver in pseudocode, as given in Listing 4.7.

Listing 4.7: Course of Action of the Hyper-V Hypercall Injection Driver

```
hypercall_page_gpa = retrieve_from_MSR()
hypercall_page_gva = get_gva_for_gpa(hypercall_page_gpa)

input_page_gva = allocate_kernel_memory_page_aligned()
input_page_gpa = lookup_gpa_for_gva(input_page_gva)

output_page_gva = allocate_kernel_memory_page_aligned()
output_page_gpa = lookup_gpa_for_gva(output_page_gva)

while (!end_of_campaign) {
    entry = campaign.next_entry()
    if (entry.is_hypercall()) {
        start_timing()
        RCX = entry.call_code
        RDX = input_page_gpa
        R8 = output_page_gpa
        execution_jump_to(hypercall_page_gpa)
        result = RAX
        stop_timing()
        log_timing()
        log_result(result)
        log_output_page()
    } else if (entry.is_delay()) {
        start_timing()
        wait(duration = entry.delay_duration)
        stop_timing()
        log_timing()
    }
}
clean_up_memory()
```

The individual steps should be comprehensible because all of them have been explained throughout this section; the pseudocode is only supposed to bring everything together and in order.

Binary Campaign Format

For the definition of the binary campaign format, the following requirements have been established:

- **Call Code**: The call code of the hypercall to be executed has to be placed in the RCX register. It is 16 bits in size.
- **Input Page**: The content of the input page describes the parameters of the hypercall. The driver has to copy it to the prepared input page, which fulfills the memory alignment demanded by Hyper-V. The total input page has a size of 4096 bytes. Because no documented hypercall actually uses the whole page and many calls only need a few bytes at the beginning of it, it is beneficial to encode only a portion the page. This makes it necessary to specify how large the encoded portion is. As the maximum size is 4096 bytes, it cannot be specified with 1 byte (highest value 255), but 2 bytes suffice (highest value 65535).
- **Repetition Count**: This has not yet been mentioned. If, e.g., in the case of stress testing, the same call is repeated many times, it is beneficial to reduce campaign entries by repeating a campaign entry multiple times. There is a trade-off between how many repetitions one entry can encode, and how large the memory overhead is when no repetitions can be used due to different call codes or parameter values. Two bytes are chosen to encode the repetition count, which allows one hypercall campaign entry to be executed 65535 times.
- **Entry Type**: As the campaign consists not only of hypercall entries but also delay entries the driver needs a way to identify the type of the entry to know how to interpret the entry's values and whether it should perform a hypercall or wait.
- **Waiting Time**: In case of a waiting entry, the driver has to know for how long it should wait before continuing with the next entry. The call code, repetition count, and input page size occupy 6 bytes of memory. To avoid copying the waiting time and expand it to 8 bytes (necessary to access the value), only four bytes are used to store the delay duration. With a microsecond resolution, this still allows for delays of over 71 minutes.

Thus, an entry in the binary campaign file is 7 bytes large, and is, depending on its type, structured as displayed by Figure 4.5. The driver can check the first byte of an entry

byte	Hypercall Entry	Delay Entry
0	0xca	0x51
1	Call Code	Waiting Time
2		
3	Repition Count	
4		
5	Input Page Size	Unused
6		

Figure 4.5.: Hypercall Campaign Entry Structure

to identify it as a hypercall or as a delay entry. According to the type, the driver can retrieve the encode information and perform the action as defined. However, in the case of a hypercall that requires parameters to be placed on the input page, the next "Input Page Size"-many bytes of the campaign file have to be copied to the hypercall input page. Consequently, the structure of a hypercall campaign file can look like the example provided

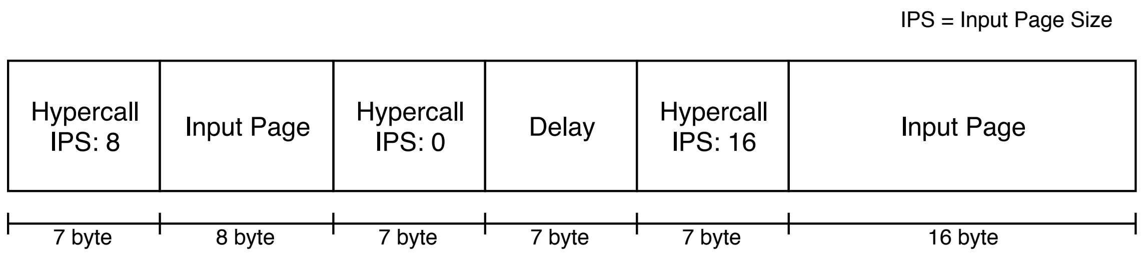


Figure 4.6.: Hypercall Campaign Structure

in Figure 4.6. At the beginning of a campaign file, there always is an entry. Therefore, the injection driver reads 7 bytes. The first byte is 0xca, so the entry describes a hypercall. The input page size, stored in bytes 5 and 6, is 8. The driver reads the next 8 bytes of the campaign file and writes them to the input page. After performing the first call, the driver reads the next 7 bytes. Again, it is a hypercall entry, but it does not require values on the input page. The same applies to the following delay, only the 7-byte entry is needed to be read before moving on to the next entry. Finally, for a hypercall with a specified input page size of 16, the driver reads 16 bytes of the campaign and stores them on the input page.

Binary Log Format

Throughout the course of this section, the following information has been identified as potentially interesting and worth logging:

- **Result Value**: The hypervisor places the result value in RAX before returning execution control to the caller. It can be used to check if hypercalls were processed successfully, or, if not, which error is indicated. Storing RAX requires eight bytes.
- **Output Page**: The output page potentially contains output values written by the hypervisor. Storing the complete page takes up 4096 bytes. Similar to the input page, no documented call actually uses the whole page, often only a few bytes are used, if at all. However, without explicit knowledge about the output parameters of the call performed, the same variably sized storing is not applicable. When running large campaigns for stress testing, logging of the output page is probably not relevant, so the driver stores the whole 4096 bytes. It should, however, be possible to enable and disable logging the page at all.
- **Timing Measurements**: An important feature is the possibility to measure the execution times of hypercalls because they can provide information about the behavior of the hypervisor. As shown in the pseudocode algorithm for the injection driver (see Listing 4.7, execution time is calculated from timestamps taken right before and after hypercall or delay execution. To quantify the time needed to execute overhead code, i.e., loading campaign entry and performing logging, it makes sense to provide the possibility to store the raw timestamps, as well. The Windows kernel system time is the number of 100ns-intervals since January 1, 1601, which requires 8 bytes to store.

There are different kinds of testing campaigns with different goals. Some might be interested in which result value is returned when varying input parameters, but do not care about execution times. Others might want to precisely specify the rate of hypercall invocation, and measure the execution times. Logging the output page in this case will cause overhead, and thus decrease the rate defined by delays in the campaign. So, logging result and output values should be disabled here. Yet another tester might want to only use campaigns to generate load but measure other metrics of the system. Then, no logging at all is the preferred solution.

Thus, the driver should support selecting if the result value should be logged, if the output page should be logged, if timing should be logged, and - if that is the case - whether the execution times suffice or the raw timestamps have to be stored.

This concludes the definition of the behavior of the injection driver, as well as its input binary campaign file format and output binary logging file format. Details about the actual implementation are provided in Section 5.3.

4.3.2. Campaign Listener Interface Implementations

With the generic campaign description language HCCDL defined in Section 4.2 and the concrete binary formats of the Hyper-V injection driver specified in Section 4.3, it is now possible to design the campaign listener interface implementations that connect them together. The first step is to identify, based on the information required by the binary campaign, which key-value pairs have to be delivered by hypercall requests in the HCCDL campaign to describe a hypercall fully. This is relevant to both listeners, the one creating the binary campaign, and the one generating a report. After that, considerations specific to the implementations are discussed.

Hyper-V Hypercalls as a List of Key-Value Pairs

Figures 4.5 and 4.6 point out that a hypercall is entirely defined by specifying the call code, providing an input page, which consists of all the parameters, and stating a repetition count. The user writing HCCDL campaigns should not have to use arbitrary call counts to specify calls, but rather refer to them by name. The same applies to parameters. On a low level, parameters are only a certain value of a certain byte size written at a certain offset in the input page. The user, however, should refer to a parameter by name, and only provide the value. The user should not have to pass a repetition count value into the list of key-value pairs describing the hypercall; this should happen implicitly, when the same call is repeated multiple times, e.g., in a loop. All of those design choices are tailored towards making campaign files easily readable and comprehensible. In the end, the campaign author is not concerned with the exact call code, or which value is placed where on the input page; he is concerned about which call he is executing and what value is assigned to which parameter.

So, in summary, when writing HCCDL campaigns for Hyper-V, the key-value pair list has to contain one key-value pair with the key “name” and, encoded as a string, the name of the requested hypercall. Additionally, for every parameter the specified hypercall has, one key-value pair can be passed, with the name of the parameter as the key and a numeric value as the value.

Campaign Listener for HCCDL to Binary Campaign Compilation

To recap, the module to be described in this paragraph is an implementation of the campaign listener interface, which is called by the HCCDL compiler every time a `hypercall` or `delay` procedure is evaluated. The purpose of this specific implementation is to translate every hypercall and delay request into its equivalent of the injection driver’s binary campaign format. By the time the compiler finished evaluating the high-level campaign, this implementation has to have created a binary representation of the whole campaign, ready to be executed by the driver.

When a campaign listener is called because a hypercall is requested, the compiler provides the list of key-value pairs specified by the campaign author. This should include the name of the requested hypercall, as well as values for the parameters. To create a binary campaign entry from that, the listener needs a knowledge base, which stores for every hypercall what its call code is, together with the information which parameters exist, how many bytes they require, and at which offset they are stored on the input page. Given the hypercall's name, the implementation can look up the call code, and, based on the parameter with the largest offset and its size, determine the required input page size. A buffer, holding the input page portion, can be allocated and initialized with zeros. Now, the implementation can iterate over the given parameter values, and store them at the correct locations in the input page buffer; unmentioned parameters keep their zero-initialization. If the last entry in the existing binary campaign file has the same call code and input page content, its count is simply incremented. If the last entry is a delay, a different hypercall, or the repetition count reached its maximum of 65535, the entry and the input page buffer are appended to the campaign.

Encoding delays is straightforward in comparison. The compiler delivers the delay duration, which is only converted to binary, wrapped in a delay entry (0x51 as the first byte), and appended to the binary campaign.

Campaign Listener for Generating Reports

Up until now, the report generation listener implementation has been treated as a singular instance. However, it actually makes sense to implement different report generators for different purposes. This paragraph is devoted to describing their common behavior; the actually implemented versions are explained later in Chapter 5.

Report generators have two sources of input. For one, their respective listener interface methods are called when a hypercall or delay is encountered while evaluating an HCCDL campaign. Additionally, they have access to a binary log file that was created by an injection driver executing the said campaign.

For every hypercall requested, the generator gets passed the name and the values for the parameters. It can also read log values corresponding to the execution of that call in the log file. As discussed in Section 4.3.1, it is configurable which values are logged. With all this information about the call, a report generator just has to define how to format and output it.

The same goes for delay requests. The compiler provides the delay specified in the campaign, and the log file might provide timestamps or a time value that represents the measured duration. Again, it is the choice of the report generator how to include this information into the report.

4.4. Hypercall Monitor

The fourth research question is not concerned with injecting user-defined hypercalls, but with which hypercalls happen during regular operation of a virtual machine, which might be one step into the direction of characterizing and understanding the behavior of Hyper-V and its hypercall interface. To enable the monitoring of hypercalls, this section designs a tool capable of intercepting and logging hypercalls invoked from a Windows operating system to Hyper-V.

The tool making this possible was already mentioned previously. Figure 4.4 shows the contents of Hyper-V's hypercall page as a screenshot of the Windows Debugger (WinDbg).

WinDbg provides the possibility to perform kernel debugging of Windows operating systems. Like debugging applications, it is possible to set breakpoints, execute code in a step-by-step manner, and inspect memory and registers, but with the kernel code. WinDbg cannot debug the operating system it is running on, the debuggee has to run on another machine (a virtual machine works as well). Communication between debugger and debuggee can be realized through a serial or Ethernet connection.

If there is a way to set a breakpoint that triggers every time a hypercall is invoked, debugger commands can be used to retrieve the hypercall input value and the parameters of the call, either from the registers or the memory, depending on the hypercall calling convention. In Section 4.3.1, the hypercall overlay page has been explained and used to invoke hypercalls from the injector driver. Hyper-V's functional specification gives the impression that every hypercall is invoked using the overlay page. The reason why this can be assumed is that the specification warns about that using any other means than the hypercall page to perform hypercalls might cause an Undefined Operation (UD) exception.

So, apparently, every hypercall will execute the first instruction of the hypercall page. The GPA of the page can be retrieved from the corresponding MSR. It is not possible to place breakpoints on physical addresses. Luckily, the company ERNW reverse engineered parts of Hyper-V and Windows' communication with it and documented the results in [15]. According to them, Windows stores the GVA used to call the hypercall page in the variable `HvlpHypercallCodeVa`. The value of the variable can be retrieved using WinDbg debugger commands. However, it is still not possible to set a traditional breakpoint at the start of the hypercall page. Here, traditional breakpoints are meant synonymous with software breakpoints. They rely on the instruction "INT 3", which triggers a software interrupt stating that control should be transferred to an attached debugger. The instruction has a size of only 1 byte. When the user sets a breakpoint at an instruction, the debugger replaces the first byte of this instruction with "INT 3". When the CPU reaches the address of the original instruction and wants to execute it, instead the software interrupt is triggered, which stops the current execution and transfers control to the debugger. When the user wishes to continue execution, the original instruction is reconstructed, and the CPU's instruction pointer reset. Hyper-V does not allow the guest operating system (and the attached debugger, for that matter) to write to the overlay page; only reading and execution are permitted.

Apart from the described software breakpoints, processors provide hardware breakpoints. Usually, they are used to break on read or write access to memory locations of interest, but they can also detect execution. As the CPU keeps track of the hardware breakpoints, it is not necessary to modify the hypercall page.

It is not practical to retrieve the hypercall invocation data by manually issuing debugger commands, so the whole process has to be automated. The implementation of this debugger script is covered in Section 5.4.

5. Implementation

Chapter 4 identified the workflow of executing hypercall testing campaigns, and, based on that, proposed a hypercall testing framework. A DSL called Hypercall Campaign Description Language (HCCDL) has been designed to allow a comfortable means of defining hypercall testing campaigns for any hypervisor. Section 5.1 describes the implementation of the generic parts of the HCCDL compiler. Then, Section 5.2 covers the realization of the Hyper-V-related campaign listeners. The development of a Windows driver that can actually execute hypercall campaigns is discussed in Section 5.3. At last, Section 5.4 completes this chapter with implementation details about the WinDbg-based hypercall monitor.

5.1. Campaign Description Language Compiler

Section 4.2 already fully described the syntax and semantics of HCCDL. This section explains the implementation of a compiler that can parse this language, evaluate it, and deliver the encountered hypercall and delay requests to a generic interface (campaign listener), which can be implemented by hypervisor-specific modules. To parse HCCDL code, ANTLR (ANother Tool for Language Recognition) is used. All information about ANTLR can be found in the book about it [72], or in the reference manual [73].

ANTLR is a parser generator. That means that it takes a grammar, consisting of lexer rules and parser rules, as input, and generates code that will parse text matching the grammar into an Abstract Syntax Tree (AST). ANTLR supports generating parser code in the languages Javascript, Python, Java, and C#.

Lexer rules are used to transform the input sequence of characters into a sequence of tokens by grouping characters into tokens. A rule consists of a name, which can later be used by the parser rules, and a definition using RegEx (Regular Expression syntax. Lexer rules can be ambigiuous. If multiple rules match, the one producing the longest token is chosen. If multiple rules match with the same length, the first rule (in definition order) is chosen. Listing 5.1 shows the lexer rules used in the HCCDL grammar.

Listing 5.1: HCCDL Lexer Rules

```
STRING: '"' (~'"')* '"';
DEC: [0-9]+;
HEX: '0x' [0-9a-f]+;
BIN: '0b' [0-1]+;
PROC: 'proc';
FOR: 'for';
KEY: 'key';
VAL: 'val';
IDENTIFIER: [_a-zA-Z][_a-zA-Z0-9]*;
WHITESPACE: [ \t\n\r]+ -> skip;
```

The first four rules represent the elementary data types. Strings are a sequence of arbitrary non-quotation mark characters, enclosed by quotation marks. Decimal, hexadecimal, and binary numeric values consist of their prefix, together with a sequence of respective digits. `PROC`, `FOR`, `KEY`, and `VAL` represent keywords in the language. These rules make sure that they are not tokenized as generic identifiers. The last rule prevents spaces, tab, and newline characters from being tokenized, they are ignored. Doing this has the advantage that parser rules can focus on parsing actual tokens because they don't have to include whitespace tokens in between other tokens.

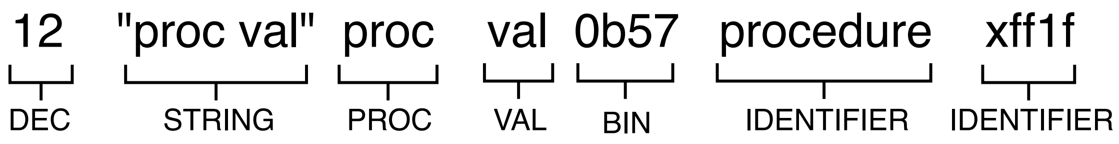


Figure 5.1.: HCCDL Lexer Rules Applied to Sequence of Characters

Figure 5.1 shows the rules tokenizing a sequence of characters. The spaces in between do not get carried over to the sequence of tokens. `proc` and `procedure` demonstrate the resolution of ambiguity. In the case of "procedure", the `PROC` rule matches and would create the token "proc". `IDENTIFIER` matches as well, and would create the token "procedure". Because the second option produces the longer token, the `IDENTIFIER` rule is chosen. However, the character sequence "proc" is also matched by both, completely. Thus, the earlier defined rule, which is `PROC`, is applied.

With the input campaign file transformed, the parser rules can put the tokens together to more complex syntactic constructs. When the language was defined, a bottom-up approach was used to start from the tangible elementary data types and construct a more and more global view of the program from there. Now that the language is already completely known and only the rules have to be created to match it, it makes sense to begin with defining what a program is, and recursively go into more and more detail until everything is defined.

When parsing input, a parser rule has to be given that is supposed to match to whole input. The rule `program`, shown in Listing 5.2, is this rule for the HCCDL grammar. It states that a program consists of a sequence of procedure definitions and global variable declarations. This rule does not enforce the existence of the `main` procedure. Such requirements should not complicate the grammar; they can be checked after the parsing. Next, following the top-down approach, procedure and global variable rules should be stated. Both are also part of Listing 5.2. There are two alternatives for declaring global variables. Multiple can be declared uninitialized, separated by commas.

Listing 5.2: HCCDL Program Structure Parser Rules

```
program:
    (procedure | globalvar_declaration)+;

globalvar_declaration:
    IDENTIFIER (',' IDENTIFIER)* ';'
  | name=IDENTIFIER '=' val=number ';';

procedure:
    PROC IDENTIFIER parameterlistDeclare '{' statement* '}';

parameterlistDeclare:
    '(' (IDENTIFIER (',' IDENTIFIER)*)? ')';
```

Alternatively, it is possible to initialize a global variable with a number. Both variants have to be concluded with a semicolon. The `number` rule will be explicitly shown, as it is defined to be either a decimal, hexadecimal, or binary token.

More interesting is the definition of a procedure. It requires the keyword "proc" at the beginning, then an identifier, which names the procedure, a list of zero or more identifier (parameter names), separated by commas and enclosed in parentheses. The procedure body consists of zero or more statements, enclosed in braces.

Statements are defined analogously to their worded description in Section 4.2. Listing 5.3 displays the rule.

Listing 5.3: HCCDL Statement Parser Rules

```
statement:
    FOR '(' IDENTIFIER ':' expression ')' statement
  | '{' statement* '}'
  | expression ';';
```

A statement is either a for loop, which can iterate over a list of values (here the `expression` has to evaluate to a list, however, this cannot be checked by the syntax), or a block of statements, or an expression.

Listing 5.4: HCCDL Program Structure Parser Rules

```
expression:
    list=expression '[' index=expression ']'
  | expression '.' op=(KEY | VAL)
  | op=('+' | '-') expression
  | expression op=('*' | '/' | '%' ) expression
  | expression op=('+' | '-') expression
  | expression '->' expression
  | IDENTIFIER '=' expression
  | '[' (expression (',' expression)*)? ']'
  | '(' expression ')'
  | IDENTIFIER parameterlistCall
  | IDENTIFIER
  | number
  | STRING;
```

All elementary data values, all operators, and procedure calls are expressions. Thus, the rule is composed of many alternatives, which can be seen in Listing 5.4. Apart from the elementary data types and variables (the `IDENTIFIER` rule), all expressions are recursively defined. The order of alternatives is essential to resolve ambiguities, such as the example given in Figure 5.2. Both parse trees are valid in terms of the rules, but which one of

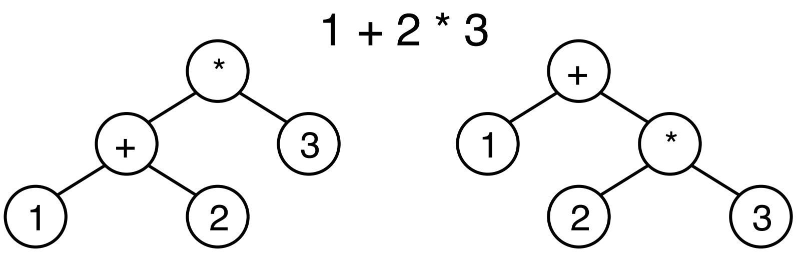


Figure 5.2.: Parser Rule Ambiguity Example

them will the parser build? Because the multiplication alternative is defined before the addition, it is applied first. Thus, the parse tree on the right side is correct.

These are all the grammar rules necessary to parse campaigns written in HCCDL. As mentioned before, the `program` rule has to match the entire campaign. From there, the parser generated by ANTLR builds the parse tree. Lexer rules and other parser rules used in a parser rule become child nodes. Lexer rules are always leaves. To aid in understanding the concept, Figure 5.3 shows the parse tree of the sample HCCDL campaign in Listing 5.5.

Listing 5.5: HCCDL Campaign for Parse Tree Example

```
x = 5;

proc main() {
    x = x + 1;
}
```

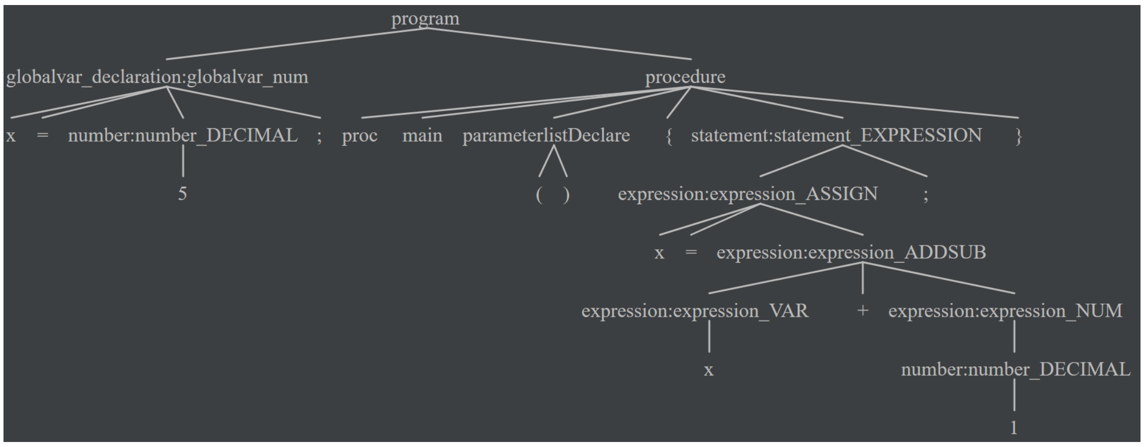

program
globalvar_declaration:globalvar_num
procedure
x
=
number:number_DECIMAL
;
proc
main
parameterlistDeclare
{
statement:statement_EXPRESSION
}
5
(
)
expression:expression_ASSIGN
;
x
=
expression:expression_ADDSUB
expression:expression_VAR
+
expression:expression_NUM
x
number:number_DECIMAL
1

Figure 5.3.: HCCDL Parse Tree Example

For implementing the compiler, which has to process such parse trees, Java has been chosen. Figure 5.4 shows the somewhat unusual folder structure of the project.

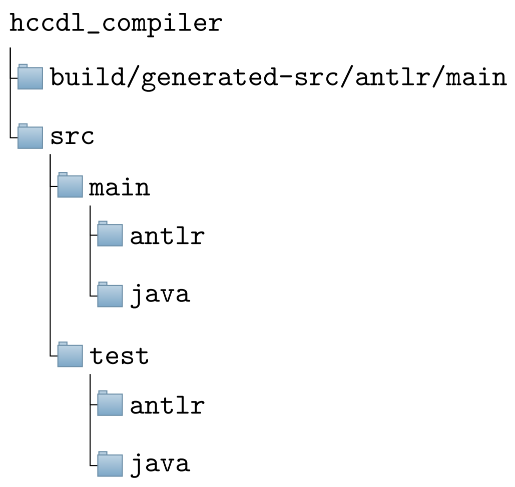


Figure 5.4.: HCCDL Compiler Java Project Structure

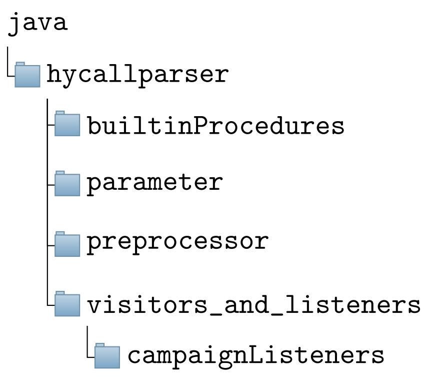


Figure 5.5.: HCCDL Compiler Java Package Overview

The `main` and `test` folders do contain not only a `java` folder but also an `antlr` folder. In `scr/main/antlr` reside the grammar file. When ANTLR is invoked, it generates Java source code for a parser according to the grammar. The Java source files are placed into `build/generated-src/antlr/main`. Subsequently, there are two distinct source folders. A Gradle build file has been written, which is configured to compile and execute the project correctly.

The parser code generated by ANTLR builds a parse tree out of Java classes, which correspond to the parser rules of the grammar. ANTLR provides two software design patterns to traverse through the parse tree: Listener and Visitor. When implementing a listener interface for a grammar, for every rule, an `enter` and an `exit` method exists. The parse tree is traversed in a depth-first manner. Every time a child node is encountered, the listener's corresponding `enter` method is called, and the node object is passed for context. When all children of a node have been traversed, the `exit` method of the listener is called.

Visitors are interfaces with a `visit` method for every parser rule. The `visit` methods have a generic return value, whose type has to be set when implementing a concrete visitor. In the beginning, the `visit` method for the root node is executed. The `visit` methods have to decide which of their child nodes should be visited, and how return values can be aggregated. However, before the visitors and listeners implemented are explained, the other packages (see Java package structure, Figure 5.5) should be explained first.

Package `hycallparser.parameter`

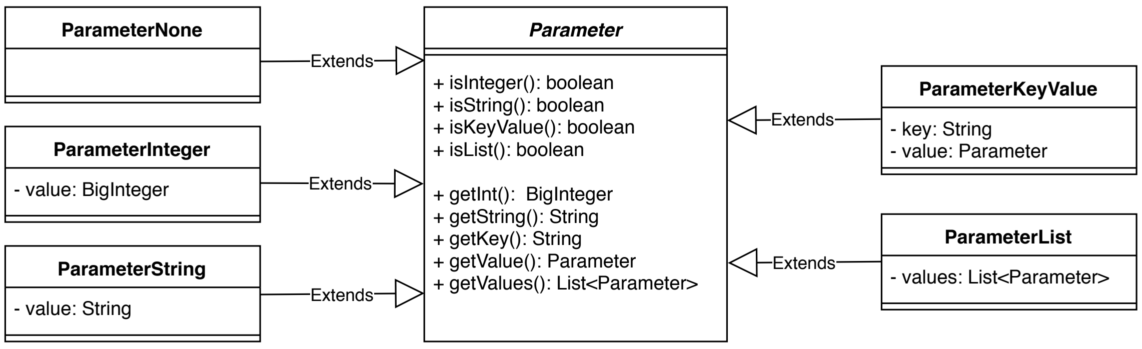


Figure 5.6.: UML Class Diagram of the `parameter` Package

The hycallparser.parameter package provides classes to store values of the different data types available HCCDL. Polymorphism has been used such that key-value pairs and lists can hold values of any data type. By defining all the methods in the Parameter class, all getters can be called on any Parameter. UnsupportedOperationExceptions are thrown on wrong calls. Implementing it this way has the advantage of not needing to cast classes to the specific implementation.

Package hycallparser.preprocessor

There is only one class in the preprocessor package: the CampaignFileLoader. Given a filename of a hypercall campaign file, it loads the content of the file into memory and searches for occurrences of #include statements. For every #include detected, it tries to load the file specified after the #include and replaces the statement with the file content. The newly added content might also contain #includes, so the CampaignFileLoader recursively repeats this process until no new occurrences are found.

Package hycallparser.builtinProcedures

All implemented built-in procedures reside in this folder and are instances of the Procedure interface, which is presented in Table 5.6.

Listing 5.6: Interface hycallparser.builtinProcedures.Procedure

```
public interface Procedure {
    String getName();
    int getParameterCount();

    Parameter perform(List<Parameter> args);
}
```

More built-in procedures can be implemented easily because, at compiler startup, the class ProcedureManager uses the Java Reflections API to create an instance of every Procedure implementation. During the evaluation of a campaign, when a procedure call is issued, but there is no user-defined procedure with the given name, the ProcedureManager queries the names and parameter counts of the Procedure implementations, and if it finds a match, the matcher's perform method is executed. So, adding a new built-in procedure requires only a new implementation of the Procedure interface, stored inside the builtinProcedures package.

Package hycallparser.visitors_and_listeners

All implemented visitors and listeners, which implement interfaces from ANTLR generated source code, are contained in this package, together with several helper classes. There is no single instance that processes the complete parse tree alone. Various listeners and visitors have been implemented for specifics tasks.

The NumberVisitor only implements the methods for visiting the number parser rule. For this visitor, the visit methods return BigIntegers. The idea is that if any object has to process a parse tree node corresponding to the number rule and wants to retrieve the numeric value, it can create a NumberVisitor instance, tell it to visit the node and get the numeric value as a return value. The NumberVisitor encapsulates the functionality of finding the integer value of decimal, hexadecimal, and binary strings.

The same principle applies to the ExpressionEvaluator. If any object has a parse tree node corresponding to the expression rule, it can tell an ExpressionEvaluator object to visit the node to retrieve the evaluation result of the expression. The return value of the `visit` methods is `Parameter`. Depending on the expression to be evaluated, an `ExpressionEvaluator` instance can either directly return the evaluation result (if the expression is just an elementary value or a variable lookup), or, if there are further expressions nested, call itself to evaluate the child expressions' values before performing the actual operation.

There are two helper classes, which keep track of state during the execution of campaigns. The `ProgramState` manages global variables and user-defined procedures. The `ProcedureState` keeps track of local variables for procedures. While there is only one global `ProgramState`, every called procedure has its own `ProcedureState`.

The first action happening to a campaign's parse tree is that a `ProgramStructureListener` traverses the tree, identifies all global variables and user-defined procedures, and creates a `ProgramState` with this information. The `ProgramState` object is passed to a `ProgramExecutor`, which checks whether an `init` procedure exists in the `ProgramState`. If it does, the `ProgramExecutor` creates a new instance of a `ProcedureExecutor` and lets it execute `init`. After `init` finished, the `ProgramExecutor` checks the `ProgramState` to see whether the `main` function exists, and either lets a `ProcedureExecutor` execute it, or throws an error stating that there is no `main` procedure.

Package `hycallparser.visitors_and_listeners.campaignListeners`

This package is responsible for managing the available hypervisor-specific modules, the campaign listeners. Campaign listeners do have to implement the `CampaignListener` interface in order to be informed about hypercalls and delays. Listing 5.7 shows the declaration of the said interface.

Listing 5.7: Interface `visitors_and_listeners.campaignListeners.CampaignListener`

```
public interface CampaignListener {
    void setParameters(String[] args);
    void encodeHypercall(List<ParameterKeyValue> parameters);
    void encodeDelay(long delayMicroseconds);
    void finish();
}
```

Additionally to the mentioned hypercall- and delay-related methods, there is also the `setParameters` method. It gets called before the evaluation of the campaign and transmits the command line arguments that the compiler application received to the listener. This is required, e.g., for report generators, because they have to know the file path to the binary log file. Also, the `finish` method is called at the end of the campaign. It can be used to close open files or perform other cleanup actions. The campaign listener that should be used for compilation can be set by passing the name of the requested listener as a command-line parameter. Similarly to the built-in functions, Java Reflections are used to instantiate the correct `CampaignListener` implementation dynamically. However, the `CampaignListeners` can be located anywhere in the source files. In the `campaignListeners` package, only an implementation of the `CampaignListenerProvider` interface has to exist, which can instantiate the `CampaignListeners`. It also defines the names for them, which have to passed via command-line argument.

This concludes the implementation of the generic parts of the framework, with the interface to the hypervisor-specific details provided via the `CampaignListener`, which means the second goal of this work has been achieved:

G2: *Implement a generic framework for designing, executing and evaluating hypercall test campaigns with interfaces for tasks that are hypervisor-specific and can't be generalized.*

5.2. Campaign Listener Implementations for Hyper-V

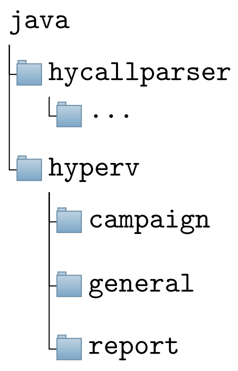


Figure 5.7.: Structure of the `hyperv` Package

The implementation of the Hyper-V-tailored campaign listeners is located pretty much exclusively in the subpackages of the package `hyperv`. In addition to that, an implementation of the `CampaignListenerProvider` interface has been created in the package `hyperv.visitors_and_listeners.campaignListeners`. It is responsible for reporting all Hyper-V campaign listeners to the compiler.

Package `hyperv.general`

No campaign listeners are implemented in the `general` package. Here, only utility classes exist. The class `ByteTransformUtils` offers methods to convert between byte arrays of arbitrary sizes and the numerical value they store. The transformations expect the binary data to be in Little-Endian format, which means the byte array starts with the least-significant byte at index 0. The Little-Endian format is chosen because the injection driver stores data this way (native byte ordering of x86_64 is Little-Endian). The conversion is needed in both directions because the binary campaign generator has to transform integer values to byte sequences of specific sizes, while the report generators read byte sequences from the binary log and need to retrieve their numerical value for human-readable display.

The knowledge about hypercall call codes, offsets, and sizes of the parameters on the input page, as well as the sizes of the output values on the output page, is provided by the `HyperVHypercallDict` class. It is essentially just a wrapper around the hypercall data stored in a JSON file to allow the other classes to retrieve this information by calling Java methods instead of having to handle JSON parsing at multiple locations. This class also implements the method `getRequiredInputPageSize`, which returns how small the encoded input page can be for the binary campaign, based on the hypercall's parameters.

Package `hyperv.campaign`

All the classes related to generating binary campaign files for the injection driver reside in the `campaign` package. For helper classes, there is the `InputPageGenerator`, which allows storing values at offsets, and, based on those, can build a byte array to be placed in the binary campaign. Furthermore, `HyperVHypercall` is a class for representing hypercall entries of the binary campaign. Thus, it stores the call code, the repetition count, and an instance of `InputPageGenerator`. The class provides a method to generate a byte array containing a corresponding binary campaign entry, followed by the variably sized input page.

`HyperVCampaignGenerator` is the implementation of the `CampaignListener` interface that is supposed to produce the campaign files for the injection driver. No output is written until, at the end of the campaign, the interface method `finish` is called. During execution, a list of bytes is used to store the binary entries. The file path where the output should be written to has to be passed as an argument to the interface method `setParameters`.

When `encodeHypercall` is called during a campaign, the `HyperVHypercallDict` is used to retrieve the call code, and offsets and sizes of the parameters. With this information, first, an `InputPageGenerator` is created, and the parameters are written to it, then, a `HyperVHypercall` is instantiated. If there is another instance remembered, with the same call code and identical input page, the count of the old instance is increased. If the hypercalls differ, the remembered one is converted to bytes and appended to the campaign. The new instance is remembered to compare it with the next hypercall. By using this approach, a sequence of identical hypercalls can be encoded in one campaign entry, without modifying the list of bytes that represents the campaign.

Package `hyperv.report`

The campaign listeners supposed the make binary campaign logs human-readable are located in the `report` package. The `ResultFileWalker` is a wrapper around the binary log file and allows to step through it while delivering arbitrarily sized byte arrays. There is also a wrapper class for the flags indicating which information is included in the campaign: the `LogFlagDecoder` reads the bits representing the flags and uses boolean values to make them easily accessible. Because raw timestamps are 8-byte values holding the number of 100 nanosecond-intervals since January 1, 1601, they have huge values. Converting a series of timestamps to a series of relative timestamps with the first time being 0, and all difference between timestamps staying the same, is provided by the `RelativeTimeTracker`.

One of the campaign listeners implementations is the `HyperVConsoleReporter`, which prints the information - inputs as well as logs - about hypercalls and delays to console. A sample of its output can be seen in Listing 5.8.

Listing 5.8: `HyperVConsoleReporter` Output

```
Hypercall:
        Name: HvFlushVirtualAddressSpace
        Exec time: 6.8us
        Result value: 0
Delay:
        Expected: 1000us
        Actual: 1010.6us
Hypercall:
        Name: HvSignalEvent
        Exec time: 5.4us
        Result value: 18
```

The other campaign listener implemented is `HyperVCSVTimingReporter`, which was used for parts of the evaluation in this thesis. Depending on whether execution times or raw timestamps are included in the binary log file, it outputs the event type and the timing information in the CSV (Comma-Separated Values) format. The event type is either "delay" or the name of the hypercall. In the case of timestamps, the `RelativeTimeTracker` is used to make the times more readable. If no file path argument is supplied, output is written to console, otherwise to the specified file.

Listing 5.9 shows a sample of the CSV-formatted output for a log file containing timestamps.

Listing 5.9: `HyperVCSVTimingReporter` Output

```
Event,start,end
hcall_HvFlushVirtualAddressSpace,0.0,9.9
delay,62.5,1396.4
hcall_HvSignalEvent,1508.5,1516.1
delay,1565.3,1615.6
hcall_HvSignalEvent,1678.1,1682.4
Compile time: 83
```

5.3. Hyper-V Injection Module

Microsoft gives an introduction to writing drivers for Windows operating systems in [74]. The relevant details are summarized here. Drivers are mostly associated with device communication. If the operating system wants to interact with a device, it requests an operation of the driver, which in turn knows how to communicate with the hardware device in order to perform the request. But there is also the concept of "software drivers", which do not manage any hardware but still need access to kernel resources. They are also referred to as "kernel services". The hypercall injection driver is exactly that. It needs to be a driver to run code with Ring 0-privilege to be able to invoke hypercalls to Hyper-V.

There are different frameworks available to use for driver development. In 1993, Windows NT was released with a new driver model, which today is called the legacy NT model. It is still usable in modern Windows operating systems; however, it can only be used to implement software drivers. A driver has to have a `DriverEntry` function, which is executed when the driver is loaded and should set up everything the driver needs. This setup includes registering handler functions for the different kinds of I/O Request Packet (IRP) that the driver can processes. IRPs are data structures used to transport information between the operating system and a driver or between drivers. Amongst others, there are the IRPs IRP_MJ_CREATE, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_CLOSE, and IRP_MJ_DEVICE_CONTROL. The operating system translates I/O requests from user space into IRPs and delivers them to the IRP handler function of the driver.

Together with Windows 98, the Windows Driver Model (WDM) framework was released. It introduced the concept of Plug and Play (PnP) and power management to drivers. PnP means the operating system loads the appropriate drivers for each device automatically, on startup as well as when hot-plugging devices. Drivers have to support the case that multiple devices can be present that require the same driver. Power management functionality enables drivers to react to changes of the power state of the system and possibly react by changing the state of the managed devices.

Writing a WDM driver with PnP functionality requires 1500-4000 lines of boilerplate code, according to Ionescu [75]. This was addressed with the Windows Driver Foundation (WDF) framework. It is based on WDM but tries abstracting away the boilerplate implementations as much as possible. For any WDF driver, PnP works out of the box. Also, it eases dealing with IRPs by providing I/O queues. WDF contains the User-Mode Driver Framework (UMDF), which is used to implement drivers that can run in user space, and Kernel-Mode Driver Framework (KMDF) for drivers that have to run in the kernel.

WDF/KMDF was chosen to implement the injection driver for the fact that there is a sample driver made available by Microsoft[1] that is a software driver implemented with KMDF to demonstrate different ways that user space applications and drivers can communicate. At first, the application that comes with the driver loads the driver into the

[1]`https://github.com/Microsoft/Windows-driver-samples/tree/master/general/ioctl/kmdf`

kernel. The `DriverEntry` function of the driver is executed. The function tells the driver execution framework that it is not a PnP driver, and thus will not provide the otherwise mandatory `AddDevice` function. As a non-PnP driver, it has to register a `DriverUnload` function. With PnP drivers, devices accessible by user space are created automatically for all plugged-in hardware devices. To still provide a device that can be used by applications to issue I/O requests to, a control device is created, callback functions for IRPs are set, and a symbolic link is created, which makes the device accessible to applications. When the driver is loaded, the application interacts with it by issuing a `CreateFile` request to the symbolic link of the driver's control device, proceeded by several ioctl, read, and write calls to the file.

This sample code has been adapted to implement the hypercall injection driver, whose behavior was described in pseudocode in Listing 4.7. In the `DriverEntry` function, code has been added to prepare for the execution of hypercalls. Access to the hypercall page is achieved by using the `__readmsr` function to retrieve the content of MSR 0x40000001, whose bits 63:12 contain the physical page number of the hypercall page. The remaining 12 bytes are used for flags. By setting them to 0, the physical address of the hypercall page is determined. With `MmMapIoSpace`, a new virtual address mapping for this physical address can be created. Both the input and output pages are allocated using `MmAllocateContiguousMemory`. Because 4096 bytes (the page size) is requested, this function allocates the memory page-aligned, as demanded by Hyper-V. The hypercall calling convention also demands the physical addresses of the pages. These are determined using `MmGetPhysicalAddress`.

Then, unchanged, the user space app creates file for the control device, but only performs a single write operation to it. In the buffer that is supposed to hold what should be written, it places the file path of the binary campaign file that should be executed, and a set of flags describing which values should be written to the log file. The path and expected values are given to the application via command-line arguments. The `FileEvtIoWrite` function in the driver is the one registered to process write requests. Given the file path in the buffer, this function is supposed to execute the campaign and output a log file. For the log file path, ".out" is appended to the path of the campaign.

Apart from the flags that describe whether execution times, raw timestamps, hypercall result values, or the output page should be logged, there is another flag, which specifies how file access should be handled. In one scenario, the driver sequentially reads campaign entries and input pages as needed. Thus, the driver uses very little memory, and it does not matter how large campaign files or log files are. However, performing file access costs time, which introduces small artificial delays to the campaign. So, there is also a variant, which reserves in-kernel memory buffers for the campaign and the log file, reads the campaign complete into this memory, performs it using only memory accesses, and in the end, writes the log buffer out to file in one operation. This strategy cannot be used when the buffers require more memory than the driver can allocate.

In both cases, a handle to campaign and log files are opened in the beginning using `ZwCreateFile`. The flags bits passed by the application are written to the beginning of the log file. This way, the report generators know which values are included in the log file and can interpret them correctly. So, even the user executing the campaign did specify to log no values, there is a log file, containing only the flags.

There is something that has not been mentioned when explaining the campaign listener that generates the binary campaign: The binary is not solely composed of hypercall entries, input pages, and delay entries. At the beginning, the generator places a header, which states how many bytes it takes to store the campaign, and how many hypercalls and delays it consists of. Each value is encoded with 4 bytes. In the case of the file-based approach, the driver reads the 12 header bytes and discards them. But if the memory-based execution is chosen, two memory buffers are allocated using `ExAllocatePool`, with their size depending on the values of the header. The buffer for the campaign is as big as the specified campaign size. The log buffer's required size is calculated based on the requested log values, the size of the lag values, and the number of hypercalls and delays in the campaign. Listing 5.10 presents the calculation.

Listing 5.10: Calculation of Log Buffer Size (in byte)

```
log_buffer_size = ((flags_received->exectime != 0) * 8 +
                   (flags_received->timestamps != 0) * 16 +
                   (flags_received->result != 0) * 8 +
                   (flags_received->output != 0) * 4096)
                     * info.num_calls +
                  ((flags_received->exectime != 0) * 8 +
                  (flags_received->timestamps != 0) * 16)
                     * info.num_waits;
```

Then, the campaign is read completely and stored in its dedicated buffer.

The file-based approach notices the campaign end when it receives a `STATUS_END_OF_FILE` when trying to read a new campaign entry. However, there is no way to know when the end of the memory buffer is reached - at least until the whole operating system crashes due to access to unmapped virtual memory. Thus, beforehand, the driver calculates the first memory address after the buffer and stores it in `campaign_buffer_end`.

First, the execution strategy for the memory-based technique will be explained. Then, the differences when using the file variant are laid out. The memory approach keeps track of two pointers. At the start, they are pointing to the begin of the campaign and log buffer, respectively. The campaign pointer is cast to a pointer to a `struct campaign_entry` to have access to the individual components and advanced to point to the first memory address after the entry. The struct resembles the campaign entries described in Figure 4.5, and its definition is displayed in Listing 5.11.

Listing 5.11: Injection Driver `struct campaign_entry`

```
#pragma pack(1)
struct campaign_entry {
        UINT8 type;
        union {
                struct {
                        UINT32 waiting_time;
                        UINT16 reserved;
                } waiting_info;
                struct {
                        UINT16 callcode;
                        UINT16 count;
                        UINT16 input_page_size;
                } hypercall_info;
        } info;
};
```

It is necessary to specify the `pragma pack` because otherwise, the compiler would insert padding after the `type` field to align the rest of the struct. This is beneficial for performance, but then the struct would not match up with the campaign entries in memory.

The first byte can be used to differentiate between hypercall and delay. Then, the `info` union provides access to the other 6 bytes, depending on the type. In case of a delay entry, the driver reads the `waiting_info.waiting_time` field, and passes the value to the function `KeStallExecutionProcessor` to sleep for this number of microseconds. Right before and after, the system time is taken using `KeQuerySystemTimePrecise`. If execution times should be logged, the difference of the timestamps is calculated (stored as an 8-byte value), the result written to the log pointer, and the pointer is increased by 8 bytes. If the raw timestamps should be logged, the first is written to the log pointer (also 8 bytes in size), the pointer is advanced, the second timestamp is written, and the pointer advanced again.

When instead a hypercall entry is encountered, more work has to be done. The input page size is retrieved from the entry. This number of bytes is copied from the campaign buffer to the input page (which was reserved in `DriverEntry`). Accordingly, the campaign buffer is increased by this number. With the input page set up, the registers have to be set up according to Table 2.4: RCX holds the call code, RDX the physical address of the input page, and R8 the physical address of the output page. Conveniently, the calling convention for functions passes the first three arguments in these registers. Furthermore, the calling convention demands the return value to be placed in RAX, where the hypervisor places the result value. Thus, it possible to perform a hypercall by casting the virtual address of the hypercall campaign (obtained in `DriverEntry`) to a function pointer of a function with the correct signature and call it with the call code and the physical addresses of the input and output pages. The return value will be the hypercall result value. Listing 5.12 shows the C code necessary to perform this.

Listing 5.12: Injection Driver Hypercall Invocation

```
result =
    ((UINT64(*)(UINT64, UINT64, UINT64))
    hypercall_page_virtual)(call_code,
                            input_page_physical,
                            output_page_physical);
```

Again, timestamps are taken before and after the call and are logged exactly as for delays. If the result value should be logged, it is written to the log pointer, which is subsequently advanced by 8 bytes. Similarly, if the output page has to be stored in the log, the whole 4096 bytes are copied from the output page to the log buffer. Of course, then, the pointer is increased by 4096. The hypercall invocation and subsequent logging are executed as many times as instructed by the repetition count of the campaign entry.

After the processing of a campaign entry, the driver continues by reading the next campaign entry from the buffer. However, before accessing memory, it checks whether the campaign pointer has reached the `campaign_buffer_end`. If it has, the driver writes the log buffer to file and completes the write request, returning control back to the application, which unloads the driver and terminates.

The course of action for the file-based implementation is similar. The only difference is that every time values have been read and written to a buffer, and the pointer has been advanced, now it is read directly from the campaign file using `ZwReadFile`, and written directly to the log file using `ZwWriteFile`. It stops when reading from the campaign yields `STATUS_END_OF_FILE`. Because the log file is already written, the driver only has to close the file handles before finishing.

Section 6.2 will test the accuracy of the delays, while Section 6.3 evaluates the performance of the module, i.e., how fast it can issue hypercalls and how significant the overhead induced by logging is.

This concludes the implementation of the last of the Hyper-V-specific parts of the framework. Therefore, the third goal has been achieved:

G3: *Implement the hypervisor-specific interfaces for Hyper-V including the hypercall injection module.*

5.4. Hypercall Monitor

The implementation of a debugger script that can monitor hypercalls invoking by a Windows operating system, proposed in Section 4.4, is covered by this section.

WinDbg provides a scripting language natively. However, it is a very unintuitive language to use. Listing 5.13 shows an example of it being used to compute factorials, taken from [76].

Listing 5.13: WinDbg Scripting Language: Factorial Computation [76]

```
.block {
 .if (${$arg1} > 1)  {
 $$>a<c:\scripts\FactorialR.wds ${$arg1}-1
  r $t1 = $arg1
  r $t0 = @$t1 * @$t0
 } .else {
  r $t0 = 1
 }
 .printf "Factorial(%p) = %p\n", ${$arg1}, @$t0
}
```

Listing 5.14: Minimal pykd Script to Monitor Hypercalls

```
import pykd
import time

RUNTIME = 60 #seconds

def millis():
    return int(round(time.time() * 1000))

start_time = millis()

pykd.dbgCommand("ba e 1 poi(nt!HvlpHypercallCodeVa)")
pykd.go()

while True:
    cur_time = millis()
    if cur_time > RUNTIME*1000 + start_time:
        break

    rcx = pykd.reg("rcx")
    call_code = rcx & 0xffff
    is_fast_call = (rcx >> 16) & 0x1
    pykd.go()
```

Fortunately, WinDbg also supports plugins. The pykd[2] plugin allows controlling the debugger with Python scripts. The minimal version of a pykd script that enables the monitoring of hypercalls is presented in Listing 5.14.

For the debugger to invoke a script, the debuggee has to be paused. Skipping the timing related code for now, the first thing happening is that a debug command is executed. "ba" is the command that allows setting hardware breakpoints; the acronym stands for "break on access". Three arguments are required. The first is the type of access, which can be either read, write, or execute. The breakpoint should be triggered when the hypercall instruction is executed, thus "e" for "execute" access is passed. The second argument defines access size, and is only relevant for reads and writes. For execution breakpoints, 1 has to be passed, even when the size of the binary representation the instruction at the breakpoint encoding is requires multiple bytes. Finally, the third parameter is the memory address of where the breakpoint should be set. As pointed out by in 4.4, the virtual address of the hypercall page is stored in the symbol `HvlpHypercallCodeVa`, which is part of the `nt` module. The breakpoint should not be set on the address of the symbol, but on the address that is stored at the address of the symbol. This is achieved by using the `poi` command. Now, every time the instruction at the beginning is about to be executed, the processor will halt the operating system, and give control to the debugger.

With the breakpoint set up, it is time to let the debuggee run using the `go` function and wait until it invokes a hypercall, which subsequently triggers the breakpoint. Then, the pykd script will continue with the code in the endless while loop. At the end of the loop is another `go` call, so the script would infinitely get executed on the breakpoint, continue, get executed, continue, and so on. To break this loop, system time is used. In the beginning, the script saves the current time. Every time a breakpoint is hit, the current time is compared to the start time. When the difference gets larger than the specified runtime, instead of letting the debuggee run again, the script terminates and gives control back to the command line of WinDbg.

In the loop, before the debuggee is unpaused, details about the hypercall can be retrieved. Reading the value of the RCX register, and looking only at the bits 15:0 reveals the call code. Meanwhile, the 16th bit specifies if the default or a fast calling convention is used. In the minimal script, this information is not processed further. However, the available timestamp, call code, and fast flag can be easily printed to console or to a file. Furthermore, it is possible to retrieve the passed parameters. Depending on the calling convention, they are either stored in memory, to which the physical address is noted in RDX, or spread across RDX, R8, and some XMM registers. Memory at a physical address can be dumped using the `!dd` WinDbg command. By performing a single step execution using `pykd.step()`, the hypercall can be executed. The result code can be read from RAX, and output values, again, depending on the calling convention, either from memory or registers.

Results from actually using the hypercall monitor to profile a hypercalls are presented in Section 7.1.

[2]`https://githomelab.ru/pykd/pykd`

6. Evaluation

To quantify the performance characteristics that the injection driver implemented in Section 5.3 shows during the execution of hypercall testing campaigns written in HCCDL, this chapter covers its evaluation. First, the testbed that has been used is described in Section 6.1. Then, in Section 6.2, the execution time logging feature of the driver is used to check how accurately delays are performed. Finally, Section 6.3 determines the maximum rate of hypercall invocation and compares that for different execution and logging strategies of the driver.

6.1. Testbed Setup

This section describes the structure of the testbed that was used for executing hypercall campaigns in Sections 6.2, 6.3, and 7.3, as well as for monitoring in Section 7.1.

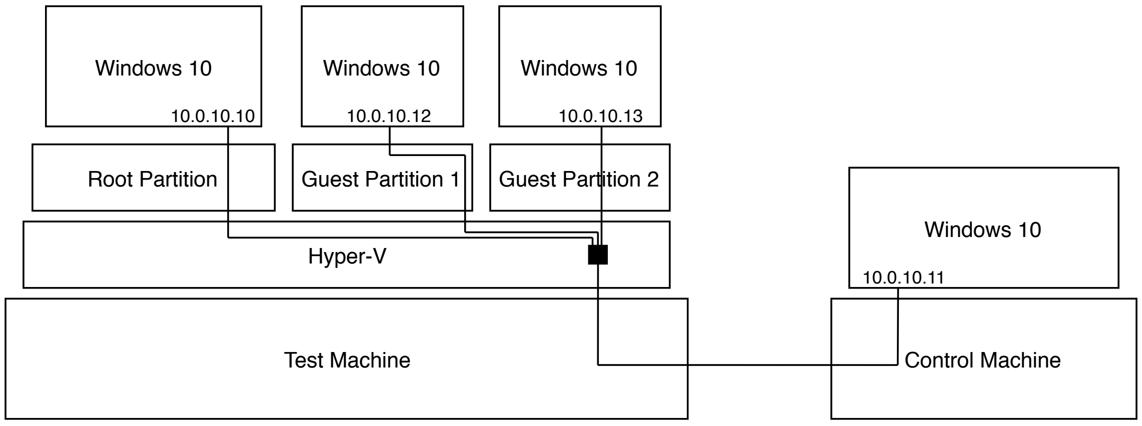


Figure 6.1.: Structure of the Hypercall Testing Testbed

Figure 6.1 shows an overview of the setup. On the test machine, Hyper-V is installed, and the root partition runs Windows 10 Version 1909. Up to two guest partitions are active, both with Windows 10 Version 1909 as operating systems, as well. A virtual network switch connects all active partitions. Note that the switch is most probably managed by the root partition, not the hypervisor, as illustrated in the figure for visual reasons. The virtual switch also connects to the physical Ethernet port of the test machine. Through that link, it connects to the control machine, which is used to run WinDbg to kernel debug the root and guest partitions over the network. Hardware details about the test machine are listed in Table 6.1. Table 6.2 shows the configuration used for the guest partitions. Both use identical settings.

CPU	AMD Ryzen 7 1700
Cores/Logical Processors	8/16
CPU Base Clock	3.64 GHz (OC)
L1 Cache	768 KiB
L2 Cache	4 MiB
L3 Cache	16 MiB
RAM	2 * 16 GiB
RAM Speed	DDR4-2933
Storage	Samsung 960 EVO 500 GB
Sequential Read Speed	Up to 3.2 GB/s
Sequential Write Speed	Up to 1.9 GB/s

Table 6.1.: Test Machine Hardware Specifications

Virtual Processors	4
RAM	4096 MiB
Disk Size	135 GB

Table 6.2.: Guest Partition Configuration Parameters

As the scenarios differ in which software is executed on which machine or partition, the scenarios have to describe their specific use of the testbed themselves.

6.2. Delay and Timestamp Accuracy

When performing load testing, it is critical that the scenario described by a campaign is executed accurately. One aspect playing an important role in this is how accurate the delays created by the function `KeStallExecutionProcessor` are. To quantify this, variations of the campaign that Listing 6.1 presents are tested.

Listing 6.1: HCCDL Campaign to Test Delay Accuracy

```
count = 1000;
del = 1;

proc main() {
        for (_ : range(0, count)) {
                delay(del);
        }
}
```

The campaign shown describes 1000 delays with a waiting time of 1 microsecond. Three additional campaigns exist, with the delay huration being 10, 100, and 1000 microseconds, respectively.

Each campaign is executed 30 times on the root partition of the testbed's test machine, with execution time logging enabled. The binary log files were converted to CSV files using the `HyperVCSVTimingReporter` campaign listener. Using the R language, all the CSV files are loaded, and the difference between the measured execution times and the requested times calculated. The first interesting fact is that the measured execution time is never lower than the request duration. 96819 of the 120000 requests achieve having the measured time match the requested; all other measured a longer duration.

The mean of these deviations is calculated for every run. Thus, for each of the 4 requested delay durations, there are 30 means of 1000 deviations each. Figure 6.2 displays a boxplot

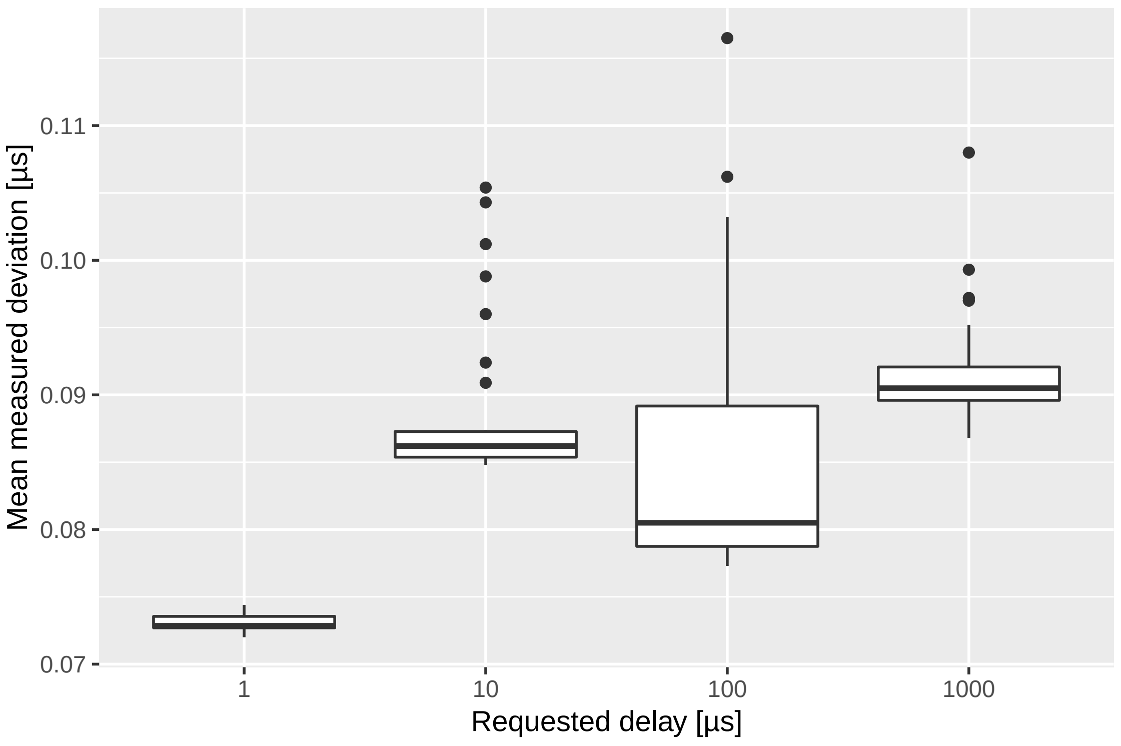


Figure 6.2.: Boxplot of the Measured Delay Deviations

of the data. The boxes show the range from the first to third quartile, which is referred to as the Interquartile Range (IQR). The bar in the middle represents the median. The upper whisker expands to the highest value that is less than the third quartile plus the IQR. Analogously, the lower whisker expands to the lowest value that is larger than the first quartile minus the IQR. All points outside of the whisker ranges are considered outliers and explicitly displayed as points.

For delays of 1 microsecond, the measurements are very consistent, with a minimum mean error of 0.072, and a maximum mean deviation of 0.0744. The timestamps the driver takes are increasing every 100 nanoseconds; hence, the resolution for the execution time measurements is 0.1 microseconds. With the mean of the execution times being even significantly less than the timestamp resolution for all of the delays, the accuracy of the delay function appears to be very high, while the time needed to take timestamps (which naturally might introduce a small error in the measurement), seems very short.

The deviations for the longer delays are mostly below the 0.1 microsecond mark, as well. However, they are measurably higher and have larger variances. We assume that most of the deviations are similar to the values of the 1 microsecond delay, but a small number of outliers increase the mean deviations.

Average values are not very robust to outliers. Performing 1000 delays with a duration of 1 microsecond takes only 1 millisecond, plus the campaign parsing overhead. It is likely that the driver seldom gets interrupted by more important kernel activities, considering the short execution time. Switching to the longer delays, maybe other code interrupts the

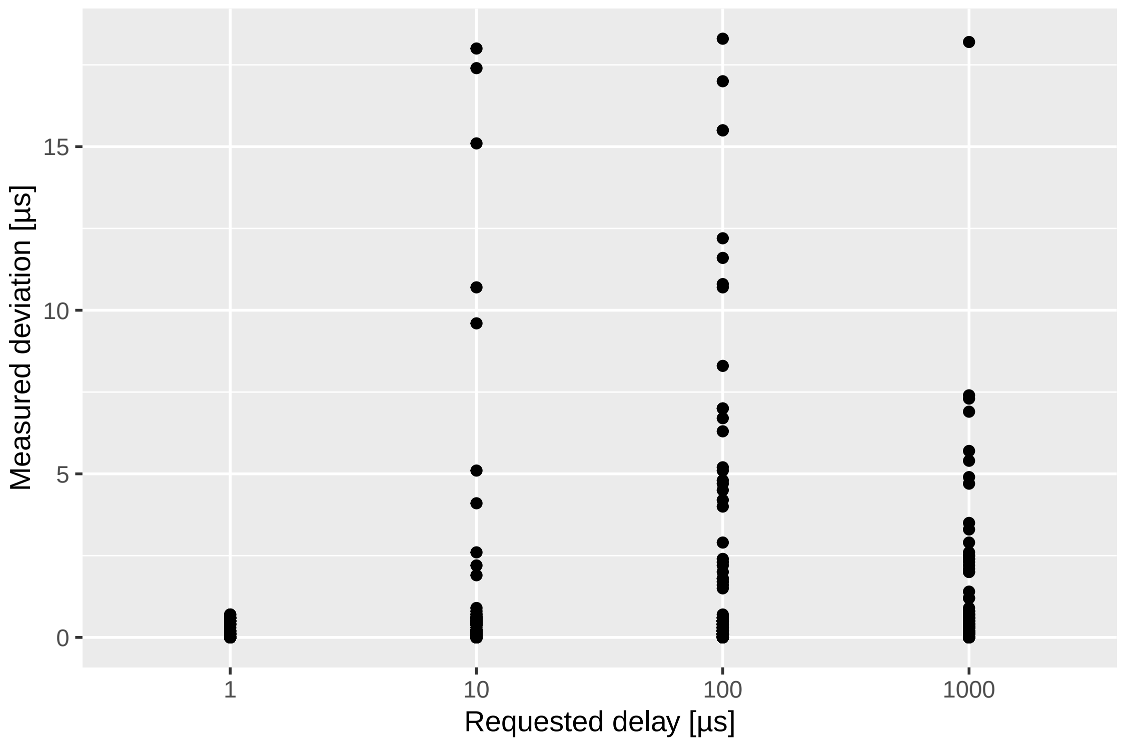


Figure 6.3.: Scatterplot of the Measured Deviations

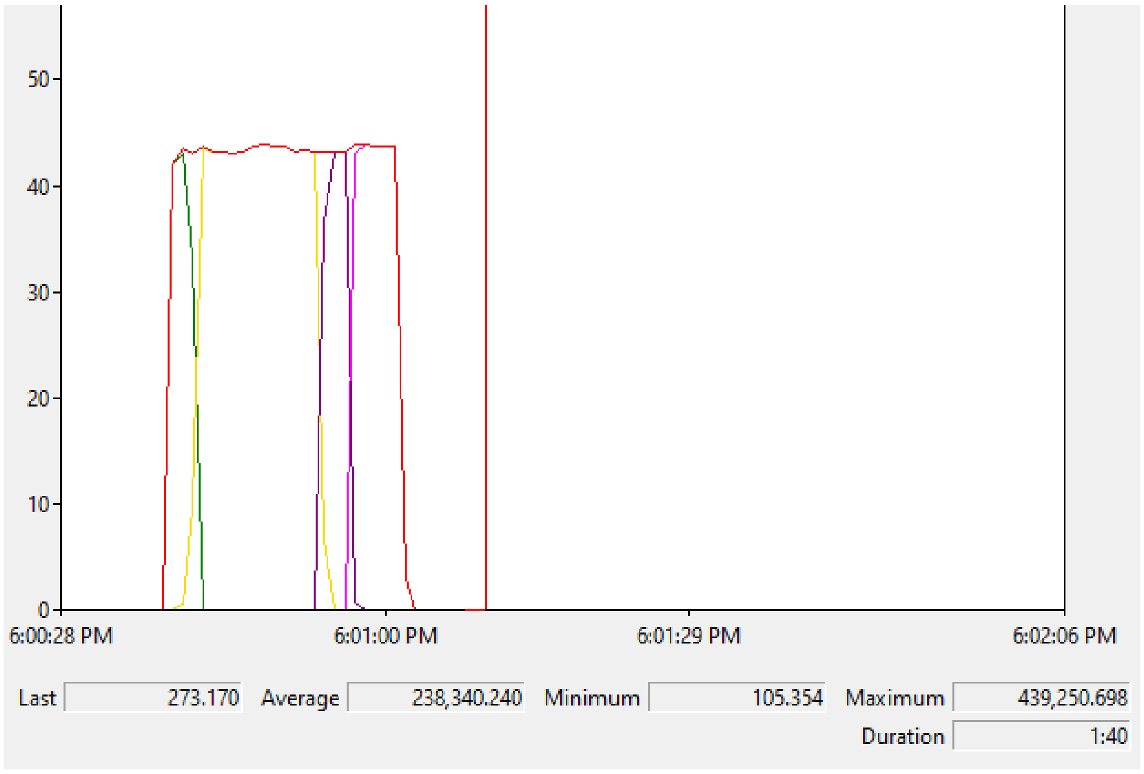
waiting CPU, and does not return control before the requested delay has completed. If such an interruption takes a significant amount of time, the measurement can be far off the supposed value.

To support this assumption, the raw deviations are organized in a scatterplot in Figure 6.3. For a delay of 1 microsecond, all deviations are below 0.7 microseconds. The other three of 10, 100, and 1000 microseconds do show deviations of more than 17 microseconds, but 99.97%, 99.90%, and 99.92%, respectively, of the errors are below 1 microsecond, which supports the assumption.

6.3. Hypercall Injection Performance Evaluation

To determine the theoretical maximum rate of hypercall injection possible, as well as find out at what cost the logging features come, the injection driver is used to issue hypercalls with non-existing call codes. This should result in the shortest possible processing time of hypercalls, as probably one of the first tasks of the hypervisor is to check the call code, which results in it instantly returning an error. The call codes 0x0100 and 0x0101 are not listed in the documentation as valid hypercalls. Executing the hypercalls, while logging the result value, shows the hypervisor returns 0x2, which, according to the specification, is the code for `HV_STATUS_INVALID_HYPERCALL_CODE`. When execution times are logged, it is reported that each call takes between 0.6 and 0.8 microseconds.

The Windows tool perfmon, short for Performance Monitor, when running in the root partition, provides many performance counters related to Hyper-V. One of them measures hypercalls per second, and it can be specified whether to get the counter for a specific logical or virtual processor, or the total of all virtual processors of a partition. After seven minutes of monitoring the test machine, with no guests active, the average value of total hypercalls/sec in the root partition amounts to 175. Figure 6.4 shows a screenshot of perfmon's GUI. The selected performance counters are the hypercalls/sec for every



Last 273.170 Average 238,340.240 Minimum 105.354 Maximum 439,250.698
Duration 1:40

Figure 6.4.: Screenshot of perfmon showing Hypercalls/sec during Campaign

virtual processor of the root partition, as well as their combined total (which is the red graph). Because many different values can be shown simultaneously by perfmon, the y-axis always generically shows the values 0 to 100. Each performance counter has a scale value to bring these values into the correct order of magnitude. To the counters in the screenshot, a scale of 0.0001 is applied. Thus, e.g., 40 on the y-axis is equivalent to 400000 hypercalls/sec. Here, 10 million hypercalls are executed with a fairly consistent rate of up to 439250 calls/sec. The crossing yellow, green, purple, and pink graphs are the counters specific to individual virtual cores. They indicate that the injection's driver campaign execution is not pinned to a particular core, and sometimes switches to another virtual processor. This could be one of the factors resulting in delays longer than specified, which has been observed in Section 6.2.

On the bottom, statistics about one selected performance counter are displayed. In this case, the total hypercall rate is chosen. Beware that the comma is a decimal point. It is questionable how a rate of 273.170 for the last second can be achieved. Probably, 274 hypercalls were counted, the time interval was measured to be 1.003 seconds, and the hypercall count was normalized to 1 second flat.

To find the maximum achievable hypercall rate, the campaign shown in Listing 6.2 is used.

Listing 6.2: HCCDL Campaign to Determine Maximum Hypercall Rate

```
count = 10000000;

proc main() {
    for (_ : range(0, count)) {
        hcall(["name" -> "InvalidHypercallNoInput"]);
    }
}
```

Beforehand, a new hypercall has been added to the JSON-based hypercall expert-knowledge used by the `HyperVCampaignGenerator`. It has the name “InvalidHypercallInInput”, the call code 0x0100, and expects no parameters. This campaign should yield the fastest execution for multiple reasons. Because an invalid hypercall is used, the processing time for the call should be minimal. The call also does not have any parameters, so no input page values have to be copied from the campaign to the actual input page. Finally, because always the identical call is issued, the compiler can use one binary campaign entry to describe 65535 calls. Thus, the driver has to read only one entry and can perform the same number of calls before interacting with the campaign file or memory buffer again.

While not related to the execution speed, it can be mentioned as a side note that the binary campaign file, packing 10 million no-input hypercalls, only takes up 1083 bytes. In comparison, storing only the input page of a single call would already take up 4096 bytes.

However, while this campaign is well suited to determine the maximum hypercall throughput achievable, it probably will not give an accurate representation of performance, when not always the same call is invoked. The homogeneous campaign will probably show less performance difference between memory-based and file-based campaign traversal because the accesses happen only after every 65535th hypercall.

Therefore, another invalid hypercall, with the call code 0x0101 and without parameters, has been added to the Hyper-V hypercall expert knowledge file. Both calls should be processed identically by Hyper-V. If they are executed in alternatingly, the compiler has to generate a binary campaign entry for every individual call. Listing 6.3 presents the adapted campaign.

Listing 6.3: HCCDL Campaign for Memory- and File-based Execution Differentiation

```
count = 10000000;

proc main() {
    for (_ : range(0, count/2)) {
        hcall(["name" -> "InvalidHypercallNoInput"]);
        hcall(["name" -> "AnotherInvalidHypercallNoInput"]);
    }
}
```

Now, encoding the 10 million calls takes up 70,000,012 bytes. 7 bytes for each of the 10 million entries, and 12 bytes for the header, storing byte count, call count, and delay count.

Furthermore, the campaign with varying calls is modified, such that the invalid calls require 8 bytes of input page values. For the file size, this means an additional 8 * 10,000,000 bytes, for a total of 150,000,012 bytes to store the total campaign. For the driver, this means one additional file read, or memory copy, respectively.

The three campaign files are executed with the driver using file-based access, as well as memory-based access. No values were logged. Each scenario-access type combination has been executed 30 times. The maximum total hypercalls/sec achieved during the campaign has been retrieved from perfmon and noted down.

Figure 6.5 displays the results in form of a boxplot. On the x-axis, the scenarios are laid out. The word before the dot specifies the access type: Unsurprisingly, “file” stands for file-based campaign traversal and “mem” for memory-based access. The word after the dot differentiates the campaign used. Using always the same call is referred to as the baseline because it is likely to perform the best. “varied” denotes the scenario with alternating no-input calls, while “varied8” has alternating calls with 8 bytes of input. The y-axis shows the rate of hypercall invocations in calls/sec.

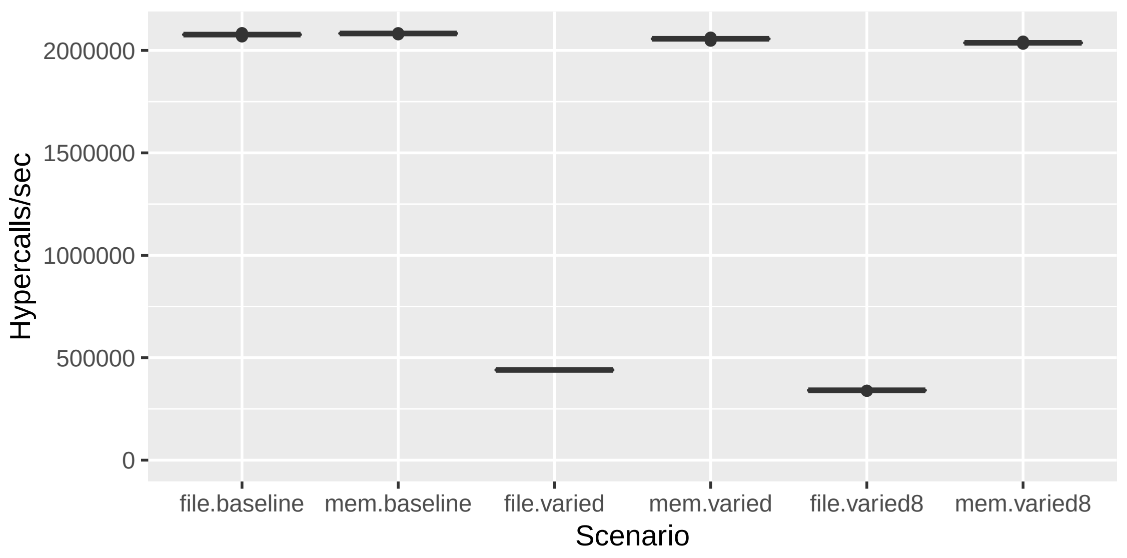


Figure 6.5.: Boxplot of the Maximum Hypercall Injection Rates

In all scenarios, the 30 measured rates show so little deviation compared to the absolute value that the boxes of the boxplot are effectively reduced to lines. Even the dots indicating outliers graphically overlap with the boxes. Both memory- and file-based access achieve their best results in the baseline scenario, as expected. Also, there hardly is a difference in their performance visually detectable. Actually, the file scenario in one case achieves a rate of 2,084,352 calls per second, which is slightly higher than the best memory result of 2,084,055. Comparing the minimal rates, memory only drops to 2,078,687 calls per second, while the file-based minimum is 2,068,454.

However, this drastically changes if entries have to be fetched from the campaign file more frequently. The file-based access performance is reduced significantly, to a maximal throughput of just 442,797 hypercalls/sec. That is only 21.24% of its baseline maximum. Meanwhile, the memory-based variant is affected, but only marginally. It still manages to achieve 2,062,365 calls per second at best, which still is 99.00% of the peak baseline performance.

The additional 8 bytes of input page values for every call, unsurprisingly, reduce the rates further. In the case of the file scenario, a noticeable drop down to 342,799 calls/sec, at best, happens. The memory approach is, again, affected rather marginally. It still is able to execute 2,042,839 hypercalls in one second.

Impact of Logging Values

Now, this paragraph covers the evaluation of how much overhead the logging of the execution time, timestamps, result value, and the whole output page cause, respectively. The campaign executed is in every scenario the one introduced in Table 6.2. For reference, the baseline of the previous experiment has been included. The measurements can be seen in Figure 6.6, again, in form of a boxplot.

Again, the x-axis differentiates the scenarios. “file”, “mem”, and the baseline workload are the same as before. In the “res” scenario, the driver was instructed to log the hypercall result value. In “exect”, there is the expense of taking timestamps, calculating the difference, and storing this value. In “timest”, the timestamps have to be taken as well. Here, no subtraction is performed, but two 8-byte values have to be stored. And finally, the

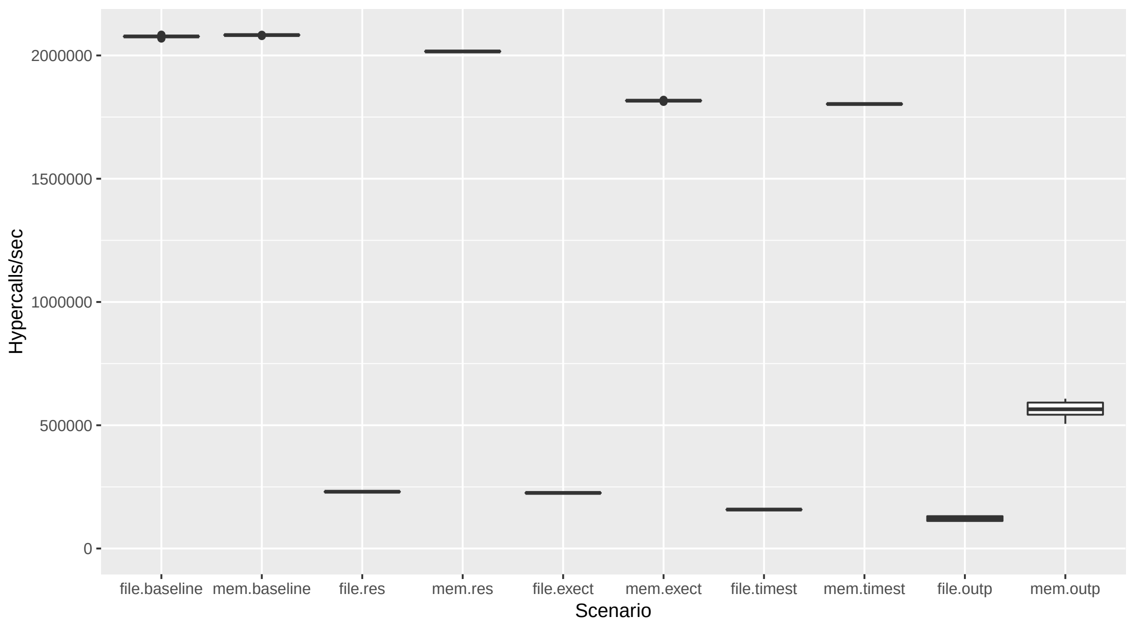


Figure 6.6.: Boxplot Showing the Penalties of Logging Values

4096-byte output page has to be logged for every call in the “outp” scenario. Remaining unchanged, the y-axis shows the number of executed hypercalls per second.

The results are in alignment with the observations in the first experiment. As soon as frequent access to the binary campaign is required, the file-based execution approach’s throughput is decimated. In the case of the file-based approach logging the result value, all the driver has to do additionally, compared to the baseline, is write 8 bytes to the log, after every hypercall. Nonetheless, the hypercall throughput drops from the file’s baseline of 2,084,352 to a mere 231,171 per second, in the best measured case. The memory-based result value logging also has an impact but manages to keep the peek rate over 2 million calls per second, at 2,017,512.

The next scenario, execution time, shows interesting behavior. In terms of log access, there is no difference between the result and the execution time. Both have to store 8 bytes after every call. While the driver gets the result value delivered by the hypervisor without additional effort, taking timestamps and calculating their difference is necessary before the execution time is known. The expense reduces the throughput of the file variant by 5,000, compared to the result value, to 226,781 calls/sec, which is 98.10% of the result of the logging rate. The impact on the memory-based implementation is more significant, this time, though. When logging execution times, it can only invoke 1,820,209 calls/sec, at best. Compared to the rate when logging result values, this is only 90.22%.

Advancing to storing the timestamps, a further decrease in performance has to be expected since now, timestamps have to be taken, as well as two 8-byte values have to be placed in the log file for every hypercall. Using file access, no more than 158,121 calls/sec can be issued. For memory access, taking the timestamps seems to be the more expensive operation than the value storing, as throughput with timestamps is only down marginally from the execution time results, to 1,803,907 hypercalls per second.

The final pair of results shows the measurement for the scenario of putting the whole hypercall output pages into the log. Both logging approaches achieve their lowest performance during this scenario. The file approach performs with 132,363 calls/sec, again, a little bit worse than when logging timestamps. However, the memory approach struggles with performing the measurements, completely. Because a buffer, as large as the total binary log file, has to be reserved in kernel memory, it is not possible to use a campaign file that issues more than 1.4 million invalid hypercalls per second. Executing those, perfmon reports rates between 505,688 and 607,670 calls per second. The unusually high variance could be due to the fact that perfmon might not always measure a 1 sec-long interval in that the driver is active for the whole second. According to the reported rates, the time to issue 1.4 million calls is over 2 seconds, in which case there has to be a 1 sec-interval covered. However, if the rate is higher, the total execution time might fall below the 2 seconds, and get split unequally over two 1 sec-intervals. The consistency of the measured rates suggests that the actual rate is not much higher than the measured maximum.

The results presented throughout this chapter demonstrated that the driver can perform delays and take timestamps accurately. Furthermore, the performance of the driver has been evaluated in various scenarios. Hence, the fourth goal is fulfilled:

G4: *Validate the correctness and evaluate the performance of the Hyper-V injection module.*

7. Hyper-V Hypercall Performance

This chapter presents the case study in that the hypercall testing framework is used to perform load testing campaigns on Hyper-V. The first two sections cover the hypercall information retrieval process: Section 7.1 utilizes the hypercall monitor to learn about which hypercalls occur during regular operation. Section 7.2, on the other hand, consults Hyper-V's specification to provide a survey of documented hypercalls, including their suitability for load testing. Finally, the identified suitable calls are used in load testing campaigns.

7.1. Hypercall Monitoring

This section is supposed to utilize the WinDbg-based hypercall monitoring tool, whose concept has been explained in Section 4.4 and which subsequently was implemented in Section 5.4. Given the testbed's structure, the hypercall characteristics of three scenarios were supposed the be profiled:

- Which hypercalls are invoked by the root partition during idle, when no guest partitions are running?
- Which hypercalls are performed by the root partition during idle, if another partition is active? Is there a difference compared to the first scenario (due to, e.g., inter-partition communication)?
- Which hypercalls happen in the guest partition, during the previous scenario?

Unfortunately, no meaningful results can be obtained. To be able to use the kernel debugger, debug mode has to be enabled, and it has to be defined, at which IP address and port WinDbg waits for the debuggee kernel to connect. This works without any problems with a VM on the control machine, which has been used to test and debug the injection driver during development. However, enabling debug mode on the root partition of the test machine leads to it refusing to boot, which requires completely reinstalling Windows to fix the machine.

Enabling debug mode in a guest partition of the test machine is possible, even though it then takes several minutes to boot. Both, the guest partition tested on the test machine, and the guest partition tested on the control machine, for confirmation, exhibit strange behavior under monitoring. As already described in Section 6.3, the Windows tool perfmon can be used to view Hyper-V performance counters, amongst others the counters for hypercall invocations per second. In the following, measurements shown are taken on the control machine. The guest partition running there only has a single virtual processor. Figure 7.1 shows the rate at which hypercalls are invoked when the partition is idling. The

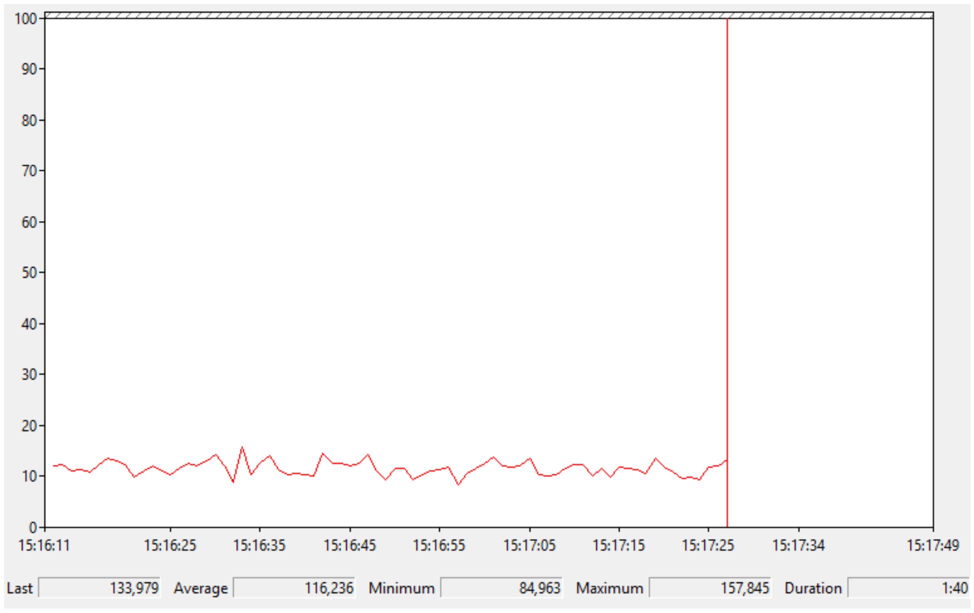


Figure 7.1.: Hypercalls/sec on an Idling Guest Partition

x-axis shows the time, and the y-axis a scaled version of the hypercalls/sec performance counter for the only virtual processor of the guest partition. "10" on the y-axis equals 100 hypercalls/sec.

The gist of Figure 7.1 is that somewhere around 50 to 200 hypercalls per second are invoked in an idling, one-processor guest partition. To find out which calls this baseline workload is composed of, the hypercall monitoring tool is supposed to capture and store their call codes. The performance counter measured while doing this is displayed in Figure 7.2. The

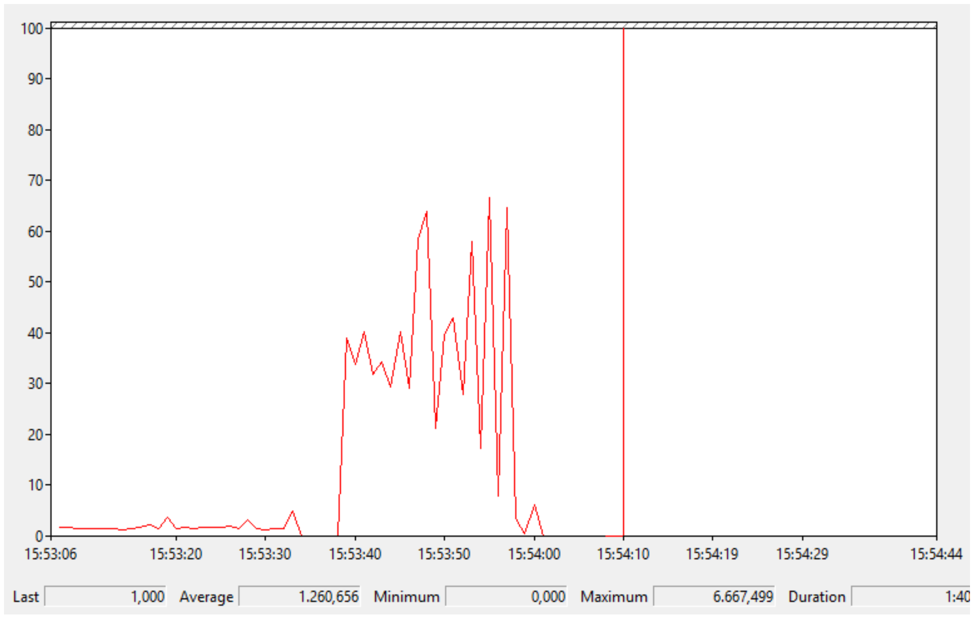


Figure 7.2.: Hypercalls/sec on a Monitored Idling Guest Partition

axes still show time and hypercalls/sec, respectively, the scaling on the y-axis has changed, however. Now, “10” is equivalent to 1000 hypercalls/sec. In the beginning, no monitoring is active, and the performance counter’s value hovers within the expected range, with a few minor peaks. At around 15:53:35 on the x-axis, WinDbg is instructed to halt the execution of the partition. As a paused processor cannot invoke hypercalls, the counter drops to 0. A few seconds later, the monitoring script is executed. Suddenly, there are intervals in that up to 6600 hypercalls occur.

In this case, the monitoring script terminates after 20 seconds, leaving the debuggee in a paused state. Yet, however, it is not clear whether the monitoring script is causing the increase in hypercalls, or if it is due to the execution pause. It is not possible to invoke the monitoring script without shortly pausing the debuggee. But the opposite can be tested: Halting the processor, and continuing after a few seconds, without the monitoring script. Figure 7.3 presents the influence of doing this on the hypercalls/sec performance counter. The axes are the same as in Figure 7.2. The debugger breaks the guest partition once at

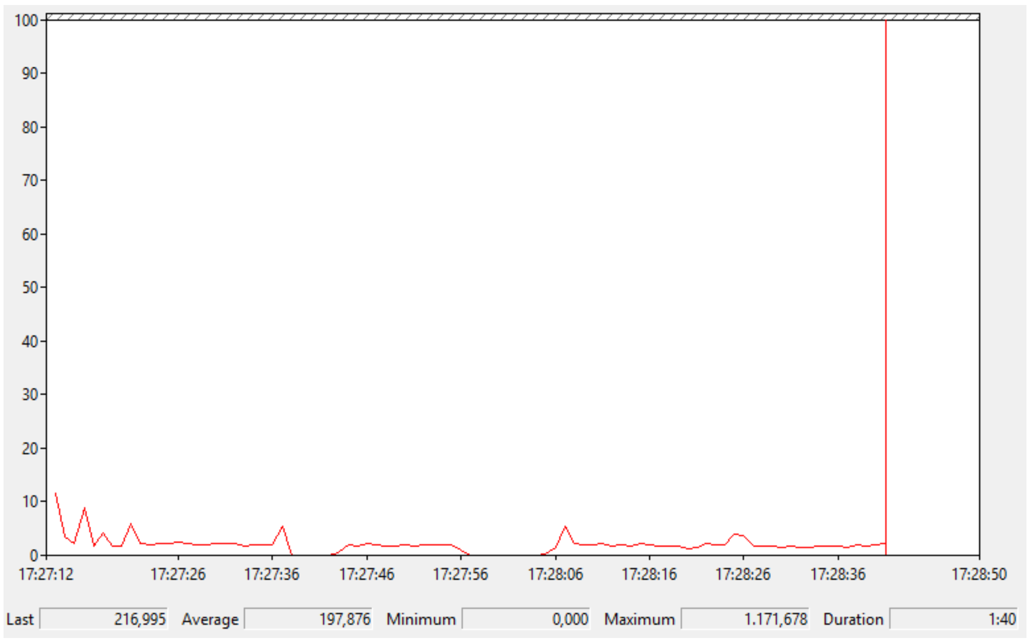


Figure 7.3.: Influence of Debugger Halts on Hypercalls/sec

around 17:27:38, and once at 17:27:57, for a few seconds. When execution is restarted, no significant impact on the hypercall rate can be detected. The behavior shown in Figures 7.2 and 7.3 is repeatable very reliably. Thus, running the hypercall monitor somehow increases the rate at which hypercalls are issued by the guest partition.

Is the hypercall monitor at least capturing all the calls, allowing to further inspect them? The monitor outputs timestamps, which indicate at which system time a hypercall triggered the breakpoint. By rounding the millisecond timestamps down to full seconds, grouping, and counting the entries for each of the 20 seconds, a graph of the captured hypercalls per second can be plotted. It can be seen in Figure 7.4. Unfortunately, there is at least an order of magnitude difference between the calls recorded by the monitor and the performance counter value delivered by perfmon. The data from the monitor indicates that at no point in time more than 100 hypercalls per second trigger the monitoring breakpoint. The invocation rates reported by the monitor appear to be even lower than the hypercalls/sec counter at idle.

If the breakpoint set on the hypercall page is triggered on every call, and the Python script logs the timestamp and call code - which there is no reason not to assume that - and the performance counter provided by perfmon is trusted to be correct, then how can

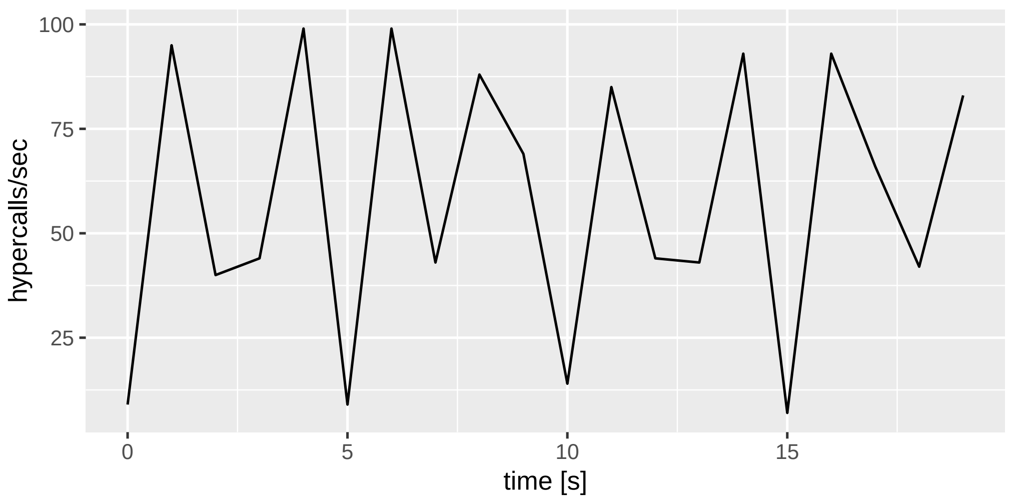


Figure 7.4.: Hypercalls Captured by the Monitoring Tool

this behavior be explained? There would have to be another way for the guest partition to issue hypercalls. One that does not access the first instruction on the hypercall page, but yet is registered by the performance counter. This could explain the discrepancy between the hypercall invocation rates, but still, why is there a multitude of calls more per second when the monitor is running? Could the partition notice that hypercalls take unusually long to processes and try to switch to a different execution mechanism?

All of that is just speculation, and currently, there is no easy way to find the correct answer. As far as the fourth research question is concerned:

RQ4: *Which hypercalls occur during normal operation?*

Unfortunately, due to unexpected, unexplainable system response, profiling the hypercall behavior of normal operation actually changes the behavior, making it currently impossible to reliably state any information about it.

7.2. Hypercall Survey

To perform load testing, it is necessary to be able to execute hypercalls correctly, unlike in Section 6.3, where non-existing hypercalls were issued. Unfortunately, as Section 7.1 described, it was not possible to retrieve trustable information about which hypercalls happen during normal operation, which would have been useful to identify suitable calls. This section provides a survey of the hypercalls documented in Hyper-V's functional specification [14].

Each call is first described and then judged depending on how well it fits the requirements of the hypercall testing framework and the load testing workload. The framework's restrictions rule out rep calls and calls with variably-sized headers. To be able to be used effectively in stress testing, a hypercall has to have either no parameters or the values to be passed have to be known, such that the hypercall does perform the work it is supposed to do. Additionally, the call has to be able to be executed repeatedly, without causing any (intentional) malfunction to the hypervisor.

- **[SUITABLE] HvExtCallQueryCapabilities**: This is one of the extended hypercalls supported by Hyper-V. It does not take any parameter. All it does is output a value called `Capabilities`, which is 8 bytes large. The first bit indicates whether the extended hypercall ”HvExtCallGetBootZeroedMemory” is available to call. The other 63 bits are reserved. This hypercall is suitable for testing, but it probably is very inexpensive for the hypervisor to processes.

- **[UNSUITABLE] HvSetVpRegisters**: This call takes a list of register name-value pairs, and sets those values in the registers of the specified guest partition. During normal execution, this might produce errors or crashes in the guest partition. However, the call is unsuited in the first place because it is a rep call.

- **[UNSUITABLE] HvGetVpRegisters**: This is a pendant to the `HvSetVpRegisters` call. It takes a list of register names, looks up their values in the specified guest partition, and outputs those to the caller. This should be safe to execute because nothing is actively changed, and interesting because of the register-checking workload, however this unsuitable due to being rep call, as well.

- **[UNSUITABLE] HvStartVirtualProcessor**: When a guest partition with multiple cores is booting, it has to start the other processor cores. This hypercall provides a paravirtualized way of doing this. The ID of the partition (-1 can be used to refer to own partition) and the index of the processor have to be passed, along with values for each of the registers. When this call is executed during normal operation, it is probably either going to fail to execute or crash the operating system.

- **[UNSUITABLE] HvGetVpIndexFromApicId**: Usually, virtual processors can detect their virtual processor index by reading an MSR. However, this is not possible when the processor is not yet started. This hypercall allows retrieving the virtual processor index using its ID of the Advanced Programmable Interrupt Controller (APIC). Unfortunately, it is a rep call, taking a whole list of APIC IDs, and thus unsuitable.

- **[UNSUITABLE] HvSwitchVirtualAddressSpace**: This hypercall is the paravirtualized method of writing to the CR3 register: it changes the page table used for memory translation. However, unlike a CR3 write, the TLB is not flushed implicitly. Changing the address space is likely to quickly cause crashes due to illegal or wrong memory accesses, rendering this call unsuitable.

- **[SUITABLE] HvFlushVirtualAddressSpace**: This call provides a paravirtualized way to flush TLBs. So, together with the `HvSwitchVirtualAddressSpace`, it can be used to perform the equivalent of a CR3 write. However, this call cannot only flush the TLB of the processor core executing the call but also of other cores of the same partition. Usually, a page table address, and a bitmask describing the processor to be flushed, have to be specified, which are not known before runtime. It is possible to specify the flag `HV_FLUSH_ALL_PROCESSORS` to flush the TLBs of all virtual processors. Similarly, `HV_FLUSH_ALL_VIRTUAL_ADDRESS_SPACES` can be used to clear all TLB entries. Combining both flags results in the call flushing all values from all TLBs in the partition. This call is suitable to be used in testing campaigns, and interesting because of its TLB-flushing workload.

- **[UNSUITABLE] HvFlushVirtualAddressSpaceEx**: This is a rep call version of the call `HvFlushVirtualAddressSpace`, and allows to define a processor set whose TLBs should be flushed. Due to its rep call nature, and the existence of the non-rep version, this call is unsuitable.

- **[UNSUITABLE] HvFlushVirtualAddressList**: This is yet another version of the `HvFlushVirtualAddressSpace` call, with the possibility to specify virtual address ranges. All TLB entries overlapping with those ranges are flushed. As with the previous call, due to its rep call nature, and the existence of the non-rep version, this call is unsuitable.

- **[UNSUITABLE] HvFlushVirtualAddressListEx**: This is a further variant of `HvFlushAddressSpaceList`, but takes a set of virtual processor IDs. It is a rep call, and thus, unsuitable.
- **[MAYBE SUITABLE] HvTranslateVirtualAddress**: This call can be used by the root partition to translate a GVA of a guest partition to a GPA. It is unknown which address the page table will be mapped when the call is executed by a campaign, but even for unmapped addresses, the hypervisor will have to perform a lookup in the page table, so, this call might be suitable.
- **[SUITABLE] HvExtCallGetBootZeroedMemory**: This call allows a partition to find out which memory pages have been zeroed by the hypervisor to avoid zeroing them twice. This call is suitable to be executed, but will probably not be expensive to process.
- **[UNSUITABLE] HvAssertVirtualInterrupt**: This call allows the root partition to trigger an interrupt in a guest partition. Due to unknown values needed for the guest partition ID, this call is classified as unsuitable.
- **[MAYBE SUITABLE] HvSendSyntheticClusterIpi**: This call is also concerned with interrupts. It sends interrupts with a given interrupt vector to the processors defined by a bitmask. The interrupt vector has to be in the range between 0x10 and 0xff, which could be tried to be executed. Thus, this call is classified as being maybe suitable.
- **[UNSUITABLE] HvSendSyntheticClusterIpiEx**: The same as the previous call `HvSendSyntheticClusterIpi`, but with a variably-sized processor set as input. It is a rep call and thus unsuitable.
- **[UNSUITABLE] HvRetargetDeviceInterrupt**: The documentation states that this call retargets a device interrupt, presumably to another virtual processor. Due to the lack of understanding of the purpose of the call, as well as its parameters, it is unsuitable.
- **[MAYBE SUITABLE] HvPostMessage**: This hypercall posts a message of up to 240 bytes to a connection specified by their ID. The hypercall monitor, if working, could be used to identify the IDs of open connections. However, it is unknown what will happen when random messages are delivered to them. Subsequently, this call is rated to be maybe suitable.
- **[MAYBE SUITABLE] HvSignalEvent**: Similarly to `HvPostMessage`, this operates on a connection ID. No message is specified here, only a flag, which is transported to the other end of the connection, which is then also notified using a synthetic interrupt. Again, it could be tested how the system reacts to random event signaling; thus, the call is maybe suitable.
- **[SUITABLE] HvNotifyLongSpinWait**: If a partition has waited a long time to acquire a spinlock, which another processor core in the same partition holds, it can issue this call to hint the hypervisor that it might adjust its scheduling. Input is a number describing how long the processor has been waiting for the lock. This call is suitable and might be interesting if the hypervisor actually does take action and checks its scheduling.
- **[UNSUITABLE] HvModifyVtlProtectionMask**: It is unclear what this call actually does, and with it being a rep call, it is unsuitable anyways.
- **[UNSUITABLE] HvEnablePartitionVtl**: As the previous call, this one is related to Virtual Trust Level (VTL) management, which is an unclear concept at the point of writing. Subsequently, this called is marked as unsuitable.

- **[UNSUITABLE] HvEnableVpVtl**: The same goes for this hypercall; it is unsuitable due to a lack of understanding.
- **[MAYBE SUITABLE] HvVtlCall**: This call is specified to change from one VTL to the next higher. It is unclear what the result of execution will be. As there are neither parameters nor output values, though, it might be tried.
- **[MAYBE SUITABLE] HvVtlReturn**: The same goes as for `HvVtlCall`, but this call reduces the VTL. No parameters, so it might be suitable.
- **[UNSUITABLE] HvFlushGuestPhysicalAddressSpace**: This call is supposed to be used with nested virtualization. It flushes the TLBs that cache translations from second-layer GPAs to GPAs. An address space ID has to be passed, of which none are known. Thus, the call is unsuitable.
- **[UNSUITABLE] HvFlushGuestPhysicalAddressList**: Similar to the hypercall `HvFlushGuest-PhysicalAddress`, but additionally, a range of GPA pages can be specified. It is unsuitable for the fact that it requires an address space ID, and additionally, for being a rep call.

To provide a better overview of which calls are suitable, maybe suitable, or unsuitable to be used for testing with the current framework implementation and knowledge about Hyper-V, the categorizations are presented in Table 7.1.

Hypercall	$\surd$	?	X
HvExtCallQueryCapabilities	X		
HvSetVpRegisters			X
HvGetVpRegisters			X
HvStartVirtualProcessor			X
HvGetVpIndexFromApicId			X
HvSwitchVirtualAddressSpace			X
HvFlushVirtualAddressSpace	X		
HvFlushVirtualAddressSpaceEx			X
HvFlushVirtualAddressSpaceList			X
HvFlushVirtualAddressSpaceListEx			X
HvTranslateVirtualAddress		X	
HvExtCallGetBootZeroedMemory	X		
HvAssertVirtualInterrupt			X
HvSendSyntheticClusterIpi		X	
HvSendSyntheticClusterIpiEx			X
HvRetargetDeviceInterrupt			X
HvPostMessage		X	
HvSignalEvent		X	
HvNotifyLongSpinWait	X		
HvModifyVtlProtectionMask			X
HvEnablePartitionVtl			X
HvEnableVpVtl			X
HvVtlCall		X	
HvVtlReturn		X	
HvFlushGuestPhysicalAddressSpace			X
HvFlushGuestPhysicalAddressSpaceList			X

Table 7.1.: Suitablility of Documented Hypercalls for Testing

This section has provided the details necessary achieve the first part of the fifth goal:

G5: *Identify hypercalls suited for repeated execution and evaluate the impact of different load levels.*

The calls **HvExtCallQueryCapabilities**, **HvFlushVirtualAddressSpace**, as well as **HvExtCall-GetBootZeroedMemory**, and **HvNotifyLongSpinWait** have been found suitable to be used for load testing. Six other documented hypercalls have been judged to be potentially suited as well; however, further evaluation of their behavior will be necessary to clarify.

7.3. Load Testing

After Section 7.3 identified four hypercalls that can be executed in fast repetition, this section will use them to perform campaigns with different load levels, based on included delay durations. The impact of the frequency of hypercall invocation is evaluated based on the metric of hypercall execution time.

HvExtCallQueryCapabilities

The call `HvExtCallQueryCapabilities` has the purpose of reporting which "extended hypercalls" are supported by the hypervisor. It does not take any parameter. However, it writes to the output page. The least significant bit of the least significant byte indicates whether the extended hypercall `HvExtCallGetBootZeroedMemory` is available, which is the only other extended hypercall.

Listing 7.1 shows the HCCDL code used to check whether the call is actually executed successfully and appears to behave correctly.

Listing 7.1: HCCDL Campaign to Validate `HvExtCallQueryCapabilities`

```
proc main() {
        hcall(["name" -> "HvExtCallQueryCapabilities"]);
}
```

The campaign is compiled, transferred from the control machine to the root partition of the test machine (for testbed setup, see Section 6.1), and performed there. Execution time, result value, and output page are logged during execution. Subsequently, the log is transferred back to the control machine, where the `HyperVConsoleReporter` is used to generate the human-readable report shown in Listing 7.2.

Listing 7.2: Validation Result for `HvExtCallQueryCapabilities` on Root Partition

```
Hypercall:
        Name: HvExtCallQueryCapabilities
        Exec time: 1.4us
        Result value: 2
```

The specification defines the result value 0x2 as `HV_STATUS_INVALID_HYPERCALL_CODE`. So, this hypercall is not available for the root partition. The same procedure has been replicated, but this time on the guest partition. Listing 7.3 displays the report:

Listing 7.3: Validation Result for `HvExtCallQueryCapabilities` on Guest Partition

```
Hypercall:
        Name: HvExtCallQueryCapabilities
        Exec time: 24.1us
        Result value: 0
```

The result value is 0x0, which represents `HV_STATUS_SUCCESS`. The hypercall also takes a significantly longer time to execute, compared to the root partition. However, it is not clear whether this caused by the call actually requiring processing or just because hypercalls are generally slower when issued from a guest partition. Also, a sample size of 1 does not provide significant results. Taking a look at the binary log file also reveals that the least significant bit of the least significant byte of the output page is set to 1, indicating that the call `HvExtCallGetBootZeroedMemory` is available.

Having validated that the call actually works at least in the guest partition, it can be used in a load test. Listing 7.4 demonstrates how the HCCDL language can be utilized to construct a campaign that executes different load levels over time.

Listing 7.4: HCCDL Load Test Campaign for `HvExtCallQueryCapabilities`

```
LOAD_LEVEL_TIME = 3000000;
SLEEP_TIME = 2500000;
REP_COUNT = 10;

proc main() {
    for (rep : range(0, REP_COUNT)) {
        for (load_level : [5, 10, 25, 50, 100, 250,
                           500, 1000]) {
            perform_load_level(load_level);
            delay(SLEEP_TIME);
        }
    }
}

proc perform_load_level(del) {
    calls_necessary = LOAD_LEVEL_TIME / del;
    for (i : range(0, calls_necessary)) {
        hcall(["name" -> "HvExtCallQueryCapabilities"]);
        delay(del);
    }
}
```

First, some global values are set. In the test, every load level should be executed for a similar amount of time. The load level variable specifies this time to be 3 seconds. In between load phases, the test should pause and not execute any hypercalls. In this case, a sleep time of 2.5 seconds is used. Finally, the repetition count defines how many times every load level should be executed.

The `main` procedure uses its outer for loop to realize the repetitions, and its inner loop to alternate between performing one of the load levels and sleeping. A load level is defined by the duration delayed between hypercall invocations. The `perform_load_level` procedure assumes that the hypercalls do not need processing time, and calculates, based on the delay between calls, how many calls are needed to fulfill the load level time with calls. So, the 5-microsecond load level should take roughly as long as the 500-microsecond to execute, which also means there are 100 times more calls to be scheduled for the 5-microsecond interval.

The binary campaign is 158,340,572 bytes in size, consisting of 11,310,000 calls and 11,310,080 delays. The file is transferred to the guest partition of the test machine and executed. In the root partition, perfmon is running, plotting the total hypercalls/sec of the guest partition. A screenshot is included in Figure 7.5. Here, the trend of the graph is

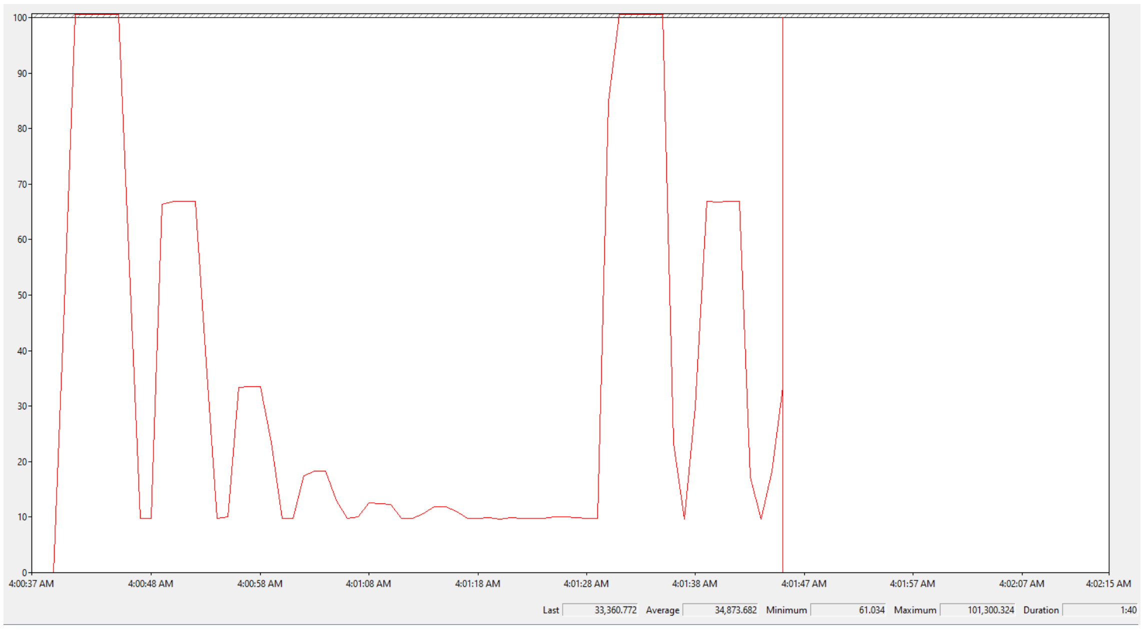


Figure 7.5.: Hypercalls/sec during Load Test

more relevant than concrete numbers. The load testing campaign starts shortly after perfmon began plotting. The first peak is the 5-microsecond level, executing around 100,000 calls per second. It is followed by ever-decreasing peaks, denoting the sequence of load levels that increase in delay. After the 8 different levels are performed, the next repetition starts.

However, the idle hypercall rate in between the load levels is at around 10,000 per second, which is unexpected. The injection driver should not be making calls during the sleep time between load levels. Further testing showed that as long as the injection driver is active, the hypercall rate stays very consistently at around 9,700 hypercalls/sec. This even applies to the extreme case that the whole campaign consists of a single delay of 20 seconds; its execution caused 20 seconds of a steady rate of around 9,700 calls/sec. The load level campaign, as well as the 20-second delay, were tested in the root partition (where the hypercalls fail because of invalid call code, which does not matter for this test). Everything worked as expected. In between load levels, the rate dropped down to a few hundred per second. In the case of the 20-sec delay, no effect on the hypercall rate was detectable. The tests were repeated multiple times in the guest partition, even across reboots of just the VM, as well as the whole test machine, but the behavior persisted.

Regardless of this phenomenon, the execution times logged by the injection driver have been analyzed. To recall, the campaign stated that every load level should be executed 10 times. The mean of the execution times of each phase is calculated. Subsequently, there are 10 mean execution times for every load level. Figure 7.6 shows them in form of a boxplot. Details about this type of plot have been provided is Section 6.2.

The x-axis lays out the different load levels, based on the duration of the delay between hypercalls in microseconds. Beware that the axis does not scale according to the numerical difference of neighboring delays. The differences between the load levels on the right are much larger than those further on the left. The y-axis shows the mean execution times, also in microseconds. It does not start at 0, which exaggerates the differences between values.

In comparison to the resolution of the execution time, which is 0.1 microseconds, all of the IQRs are rather small, suggesting consistency in the measurements. It can be observed that

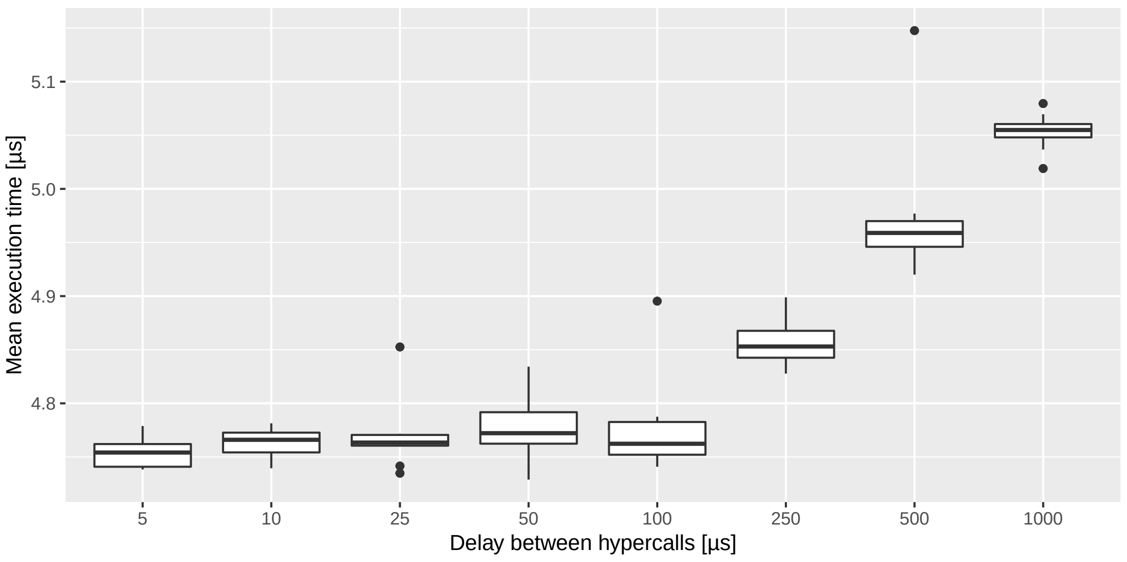


Figure 7.6.: Impact of Load Level on `HvExtCallQueryCapabilities`

the load levels from 5 up to 100 microseconds have very similar means of execution times. However, increasing the delay further, the execution times actually start to increase. To further investigate this behavior, the load test campaign has been changed to test the load levels from a delay of 100 microseconds up to 1000, with a step size of 100 microseconds. Figure 7.7 presents the plot.

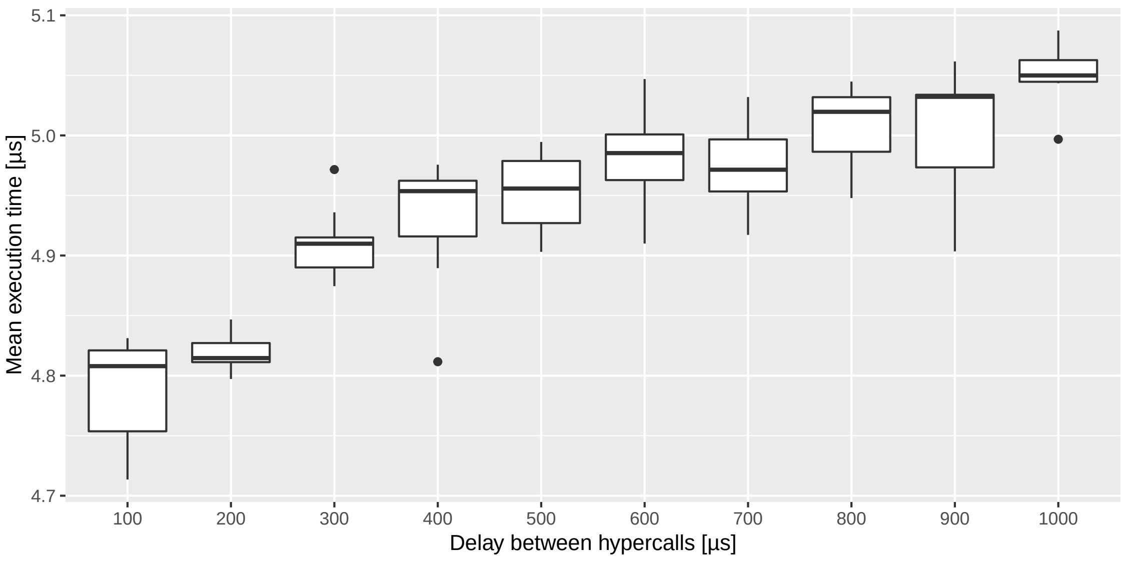


Figure 7.7.: Impact of Load Level on `HvExtCallQueryCapabilities`

The plot is structured identically to the plot in Figure 7.6. But now, the x-axis shows the load levels from 100 to 1000 microseconds, in equidistant, true-to-scale steps. Comparing the values that the two plots have in common, it can be seen that they match up well. For the 100-microsecond load level, the mean execution times are at or right below 4.8 microseconds. The value for the 250-microsecond delay does not exist in the second plot, but interpolating the values of 200 and 300 results in execution times around 4.85, matching the value from the first plot. In both figures, 500 microseconds of delay result in around 4.95 microseconds of execution time, and the 1000-microsecond delay lies at 5.05 microseconds execution time, twice.

Analyzing the trend, while taking both plots into account, suggests that the execution time is consistent for load levels up to 100 microseconds. Then, there is a linear increase from 100 to 400 microseconds. At 400, the behavior changes again to a linear increase with a smaller slope. This might be due to more effective caching when the hypercall rate is higher but that is completely up for speculation.

To make sure that the measurements are consistent, the non-equidistant campaign is executed another 5 times. Before the execution of each campaign, the whole testbed was rebooted to ensure that the hypervisor is in a fresh state. The plots can be found in the Appendix in Section B, in the Figures 8.1, 8.2, 8.3, 8.4, and 8.5. There are notable differences in the absolute execution times, e.g., the IQR for the 1000-microsecond delay measures in at 5.05 microseconds in Figure 8.1, while it is around 5.4 microseconds in Figure 8.3.

However, the trend that higher delays between hypercalls lead to higher mean execution times is observable in all measurements. Interestingly, the repetitions 1 and 4 show consistent times up to the 100-microsecond load level, while in the repetitions 2, 3, and 5, the trend already affects lower delays.

HvFlushVirtualAddressSpace

The next hypercall to assess is `HvFlushVirtualAddressSpace`, which provides the possibility to invalidate TLB entries. Usually, the call expects an address space ID and a mask of processors, to determine which entries have to be invalidated in which virtual processors. However, there is the possibility to pass flags to the call. When using the `HV_FLUSH_ALL_PROCESSORS` and `HV_FLUSH_ALL_VIRTUAL_ADDRESS_SPACES` flags, the hypercall ignores the address space and processor mask parameters and invalidates all entries in all TLBs.

As seen with `HvExtCallQueryCapabilities`, it is necessary to ensure the correct execution of the call, first. Listing 7.5 displays the HCCDL code that is used to test the `HvFlushVirtualAddressSpace` hypercall.

Listing 7.5: HCCDL Campaign to Validate `HvFlushVirtualAddressSpace`

```
HV_FLUSH_ALL_PROCESSORS = 0b1;
HV_FLUSH_ALL_VIRTUAL_ADDRESS_SPACES = 0b10;

proc main() {
    hcall(["name" -> "HvFlushVirtualAddressSpace",
           "AddressSpace" -> 0,
           "Flags" -> HV_FLUSH_ALL_PROCESSORS +
               HV_FLUSH_ALL_VIRTUAL_ADDRESS_SPACES,
           "ProcessorMask" -> 0]);
}
```

This time, the guest, as well as the root partition, get an `HV_STATUS_SUCCESS` (0x0) as the result value. Hence, load testing can be performed for both. Because every call needs to store 24 bytes of input page, the 5-microsecond load level has been removed from the campaign to keep the file size manageable. Figure 7.8 shows the plot for the guest partition, while the results of the root partition are visualized in Figure 7.9.

The results from the guest partition show a very similar trend to the one observed previously. The 10-microsecond load level achieves mean execution times slightly above 1.25

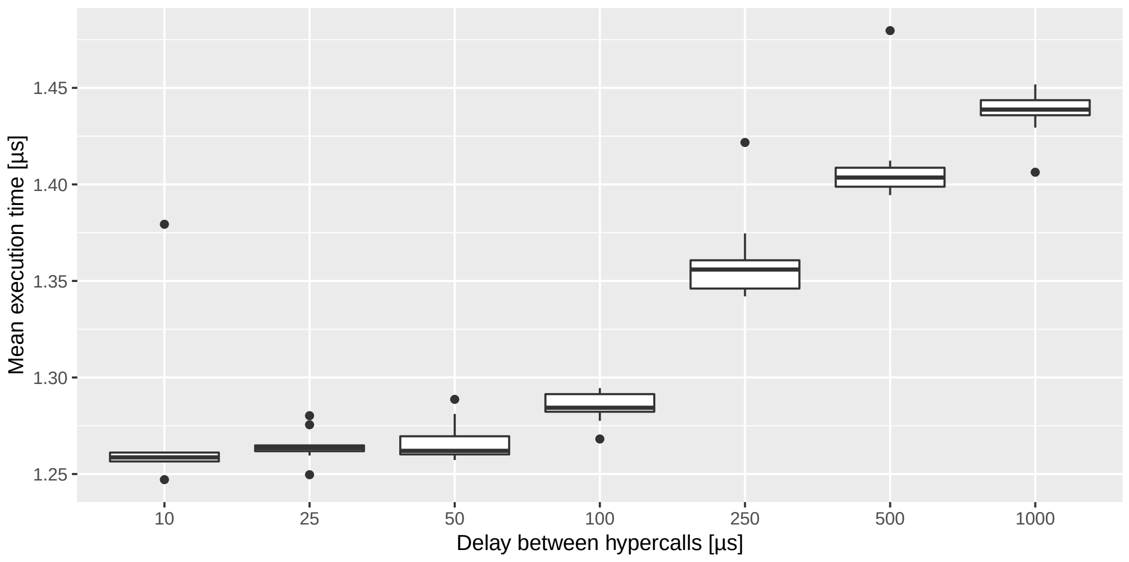


Figure 7.8.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest)

microseconds. The 25- and 50-microsecond delays result in marginally higher durations. Again, starting at the 100-microsecond level, mean execution times begin to rise, up to almost 1.45 microseconds for the 1000-microsecond delay.

Unfortunately, there is an outlier measurement at the 500-microsecond load level in the root scenario, which results in an unpreferable scaling of the y-axis. The execution times for the load levels from 10 to 100 microseconds float between 1.55 and 1.6 microseconds. Then, execution times rise close to 1.7 for the 1000-microsecond load level. Comparing the absolute results for the guest and the root partition shows that the call is faster in the guest. A possible explanation for this observation could lie in the fact that the hypervisor has to flush the TLBs of 16 virtual processors in the root partition, compared to only 4 in the guest partition. Regarding the trend, the root partition shows similar behavior to the measurements in the guest partition, as well as the measurements of the `HvExtCallQueryCapabilities` call.

As with the previous call, repeated 5 more executions of each scenario are performed to confirm the results. Again, the whole testbed is rebooted before every campaign. The Appendix Section C contains the plots for the guest partition in the Figures 8.6 to 8.10. All of them confirm the observations stated for Figure 7.8. Section D of the Appendix provides the plots for the root partition repetitions in Figures 8.11 to 8.15. Except for the first repetition, outliers modify the y-axis such that the IQRs appear only very small. However, the trend of increasing execution times can still be recognized.

Comparing the overall execution times between the `HvExtCallQueryCapabilities` and `HvFlushVirtualAddressSpace` calls raises the question of why the hypercall that does actual work, i.e., flush all TLBs, only needs about a third of the execution time that a call returning, effectively, a single bit, which never changes, requires. The hypervisor specification states that `HvExtCallQueryCapabilities` and `HvExtGetBootZeroedMemory` are extended hypercalls, which can be used by the normal hypercall interface but are handled differently internally. If the internal handling is expensive, this could explain the observation.

Overall, the flushing hypercall does show very similar behavior regarding the increased execution times at lower hypercall load, which could indicate this a behavioral pattern happening with all hypercalls.

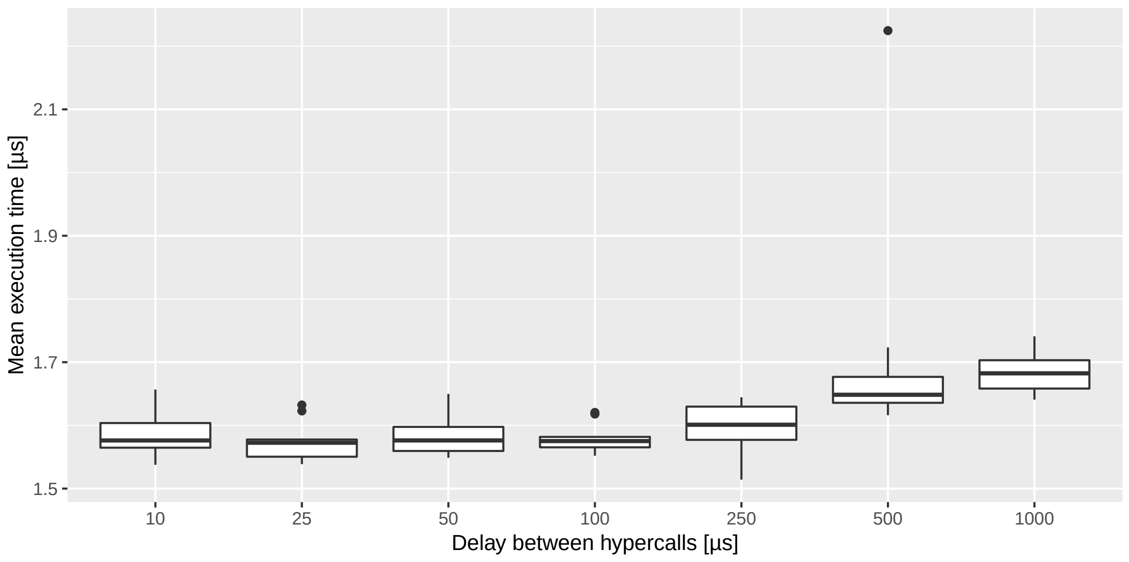


Figure 7.9.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Root)

HvExtCallGetBootZeroedMemory

This is the second existing extended hypercall, next to `HvExtCallQueryCapabilities`. It can be used by a partition to retrieve from the hypervisor, which memory in its GPA address space was already filled with zeros by the hypervisor. It does not take any parameters, and delivers the address of the first zeroed page, together with a number, which indicates how many of the following pages are zeroed as well.

The availability of this hypercall is reported by `HvExtCallQueryCapabilities`. On the root partition, that call was unknown, so this will probably be as well. In the guest partition, which could execute `HvExtCallQueryCapabilities` without errors, it was reported that `HvExtCallGetBootZeroedMemory` is available. The HCCDL campaign shown in Listing 7.6 is used to execute the call nonetheless, to see if the assumptions made hold true.

Listing 7.6: HCCDL Campaign to Validate `HvExtCallGetBootZeroedMemory`

```
proc main() {
    hcall(["name" -> "HvExtCallGetBootZeroedMemory"]);
}
```

As expected, the hypervisor returns a result code of 0x2 (`HV_STATUS_INVALID_HYPERCALL_CODE`). However, also the guest partition receives an error code. Interestingly, the given code, 0x5, is described in the specification as `HV_STATUS_INVALID_PARAMETER`, which does not make sense, considering the specification also states that the hypercall does not take any parameters.

In any way, no possibility is currently found to execute the hypercall successfully, thus it was not included in any load testing.

HvNotifyLongSpinWait

If a virtual processor recognizes that it has been waiting for an extended period of time to acquire a spinlock that is held by another virtual processor of the same partition, it can use the `HvNotifyLongSpinWait` hypercall to inform the hypervisor that changes to the scheduling of virtual processors might have to be reconsidered. The call takes one parameter, which apparently is the number of failed acquirement tries that happened before the call.

The HCCDL campaign designed to validate that the call can be successfully performed is noted in Listing 7.7.

Listing 7.7: HCCDL Campaign to Validate `HvNotifyLongSpinWait`

```
proc main() {
    hcall(["name" -> "HvNotifyLongSpinWait",
           "SpinwaitInfo" -> 1000]);
}
```

As there is no notion of which numbers for the parameter are high and which are low, and because the hypercall essentially is just a suggestion or hint to the hypervisor, the value of 1000 for the parameter is chosen arbitrarily.

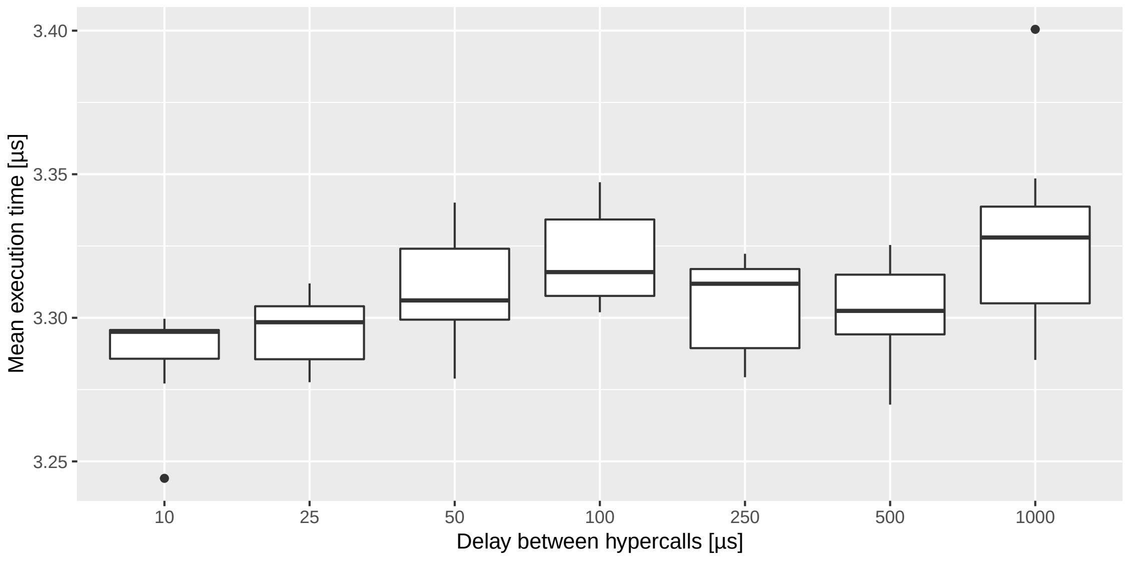


Figure 7.10.: Impact of Load Level on `HvNotifyLongSpinWait` (Param = 0)

Executing the `HvNotifyLongSpinWait` call in the root partition crashes its Windows operating system, and with that, essentially, the whole hypervisor. This has been test repeatedly with various values for the parameter. A single invocation of the call causes a crash every time. This seems like a robustness problem of Hyper-V's hypercall interface. If the hypercall is not supposed to be executed from the root partition, the hypervisor should handle that gracefully by indicating an error in the hypercall result value. There is no reason why this call should cause the system to crash entirely.

The call can be successfully executed in the guest partition, though. The load test is executed twice for this call. One time, 0 is chosen as the value for the parameter (for results, see Figure 7.10), and the other time, the biggest value fitting in the 4 bytes of the parameter is used (see Figure 7.11).

First, comparing the values of two variations, a significant difference cannot be claimed. All mean execution times scatter closely around the 3.3 microsecond mark. The execution times vary slightly, as the load levels are changed. The previously detected pattern cannot be found here. Instead a possibly wave-shaped pattern is present. The absolute change in value is very small, but exaggerated by the scale of the y-axis. However, both scenarios show a similar pattern, only showing some difference when it comes to 250- and 500-microsecond delay load levels.

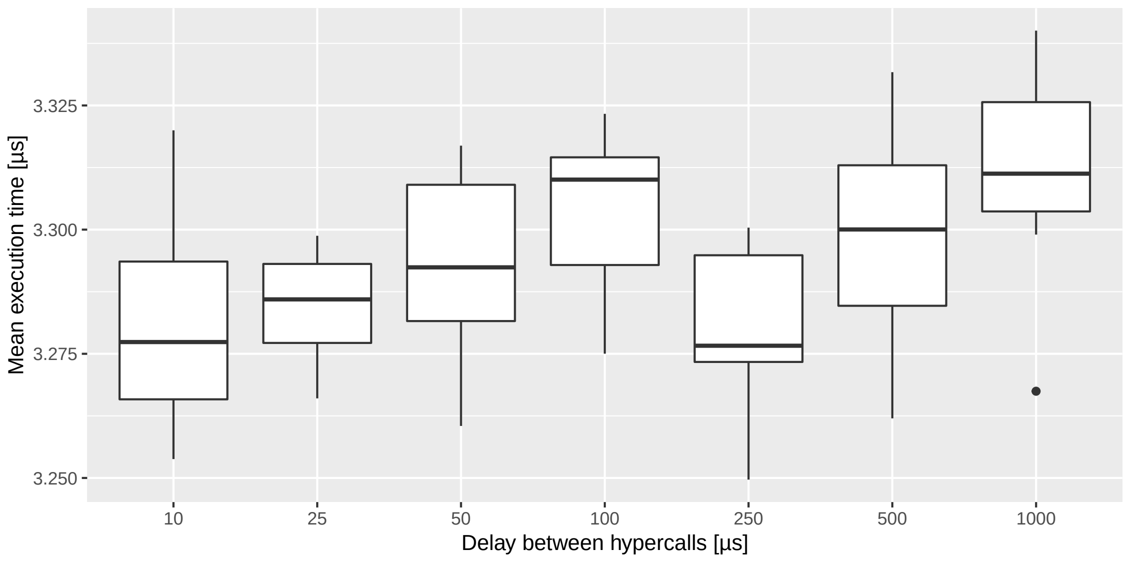


Figure 7.11.: Impact of Load Level on `HvNotifyLongSpinWait` (Param = max)

For both of the parameter values, 3 additional repetitions are executed. The Figures 8.16, 8.17, and 8.18 in Section E display the plots for the parameter equal to 0. The plots for the maximum value parameter are presented in the Figures 8.19, 8.20, and 8.21 is the Appendix Section F. They all show mean execution times hovering between 3.3 and 3.4 microseconds, with minor fluctuations in a possibly wave-shaped pattern. This means that the calls `HvExtCallQueryCapabilities` and `HvFlushVirtualAddressSpace` show similar behavior, which does not apply to all hypercalls, though.

Discussion

G5: *Identify hypercalls suited for repeated execution and evaluate the impact of different load levels.*

With the load testing campaigns performed, and the measured execution times evaluated, the second part of the fifth goal can be noted as achieved.

The results obtained in the process also provide an answer to the fifth research question:

RQ5: *How does Hyper-V react to increasing hypercall loads?*

No universally valid answer can be given; the behavior is dependent on the specific hypercall. The calls `HvExtCallQueryCapabilities` and `HvFlushVirtualAddressSpace` exhibit a similar pattern. With load levels that have delays of 100 microseconds or lower, hardly any variability between hypercall execution times can be detected. However, increasing the delays beyond 100 microseconds results in longer mean execution times. It can be argued that this phenomenon is due to caching being more effective when calls are repeated with a higher frequency.

This observed pattern can not be generalized to all hypercalls, though. HvNotifyLongSpin-Wait's mean execution times do not follow the pattern. They instead fluctuate between neighboring load levels, possibly suggesting a wave-shaped pattern.

8. Conclusion

This chapter concludes the thesis. Section 8.1 provides a summary of the content presented throughout this work. Then, Section 8.2 assesses which of the research questions and goals proposed in Section 1.4 have been answered or achieved, respectively. Also, a summary of the achievements regarding the stated goals is given. Finally, Section 8.3 presents an outlook of possible future work based on the contributions of this thesis.

8.1. Summary

This thesis started off with Chapter 1 giving a brief explanation of the concept of virtualization, followed by a motivation that stated that hypervisors are critical entities of modern infrastructure, whose performance, isolation, reliability, robustness, and security properties have to be ensured. The design and implementation of a testing framework have been proposed. Microsoft's Virtualization-Based Security (VBS) lead to the choice to focus on testing hypercall interfaces, as well as to the selection of Hyper-V as a case study.

Chapter 2 then explained the necessary technical background regarding the x86_64 processor architecture and the virtualization of it. Also, the basic architecture and the available hypercall calling conventions of Hyper-V were introduced.

An overview of related work was presented in Chapter 3. Most notably, there are publications about performing robustness testing of hypercall interfaces. However, both papers were using Xen-specific tools and focused more on mutations of input parameters and classification of crashes.

Chapter 4 analyzed the workflow of performing hypercall test campaigns and subsequently proposed a framework architecture. A kernel module or driver is part of the framework and performs hypercalls based on a binary campaign file. If required, the injection driver can provide binary logs containing information about the execution of the campaign. A compiler is responsible for translating between human-readable and binary driver files. A Domain-Specific Language (DSL) called Hypercall Campaign Description Language (HCCDL) was designed. Furthermore, the behavior of the Hyper-V injection driver, as well as its binary input and output formats, have been specified. Also, the concept of a hypercall monitor has been explained.

The implementation of the framework and hypercall monitor was covered by Chapter 5. The HCCDL compiler is a Java application utilizing ANTLR to parse campaign files. The

Hyper-V hypercall injection driver was implemented as a Windows driver using the KMDF framework. The hypercall monitor was realized using WinDbg and pykd.

The evaluation of the injection driver performed in Chapter 6 validated that the driver is capable of accurately waiting for the durations specified by delays, as well as precise timing measurements. Also, the maximum hypercall throughput has been tested for various scenarios. As soon as frequent accesses to the campaign or log file are required, the file-based implementation experienced a lot of overhead, while there was little effect on the memory-based approach. However, the file-based technique is not dependent on being able to store both campaign and input files in kernel memory.

Chapter 7 was devoted to the Hyper-V case study. At first, we tried to identify hypercalls issued by regular Windows operating systems during normal operation. Due to the hypercall behavior changing when monitoring it, no results were obtainable. Hypercalls documented in Hyper-V's Top-Level Functional Specification were assessed regarding their suitability for load testing. Out of 26 hypercalls, 4 have been found suitable, and 6 have been classified as potentially suitable. 3 of the 4 suited calls could be used in the testbed to measure execution times during different load levels. 2 of the hypercalls exhibit similar behavior; with longer delays between calls, the execution times increase as well. The most probable cause for this effect might be due to caching mechanisms being more effective at higher load levels. Also, a robustness problem has been identified. Executing the hypercall `HvNotifyLongSpinWait` in the root partition leads consistently to a crash.

8.2. Assessment

Table 8.1 provides an overview of the assessment of the research questions and goals that this thesis set out to answer and achieve, respectively.

Research Question/Goal	Assessment
RQ1 Hypercall Testing Workflow	√
RQ2 Workflow Abstraction	√
RQ3 Hyper-V Hypercall Injection	√
RQ4 Default Hypercall Load	X
RQ5 Load Testing Effects	√
G1 Hypercall Interface Comparison	√
G2 Framework Implementation	√
G3 Hyper-V Specific Implementation	√
G4 Injection Driver Validation	√
G5 Load Testing Execution	√

Table 8.1.: Assessment of Research Questions and Goals

Except for **RQ4**, all research questions have been answered, and all goals have been achieved successfully.

RQ4 stated the question, which hypercalls occur during normal operation. Following the details in Hyper-V's documentation, an approach has been proposed that utilized kernel debugging to trigger a breakpoint every time a hypercall is invoked. Thus, it should be able to monitor all hypercalls. The idea was implemented using WinDbg and pykd. However, it was not possible to enable kernel debugging on the root partition of the test machine. While it was possible to use the monitoring tool on guest partitions, intercepting the hypercalls lead to a substantial increase in hypercalls that apparently use an invocation method not specified by Hyper-V's documentation.

While hypercall monitoring information could have helped to identify more hypercalls suitable to load testing, the documentation provided enough calls to achieve **G5** and answer **RQ5**.

8.3. Future Work

There is lots of potential to build and extend upon this work. This section provides ideas for future work, organized in different categories:

Improve Existing Implementation

- Currently, the campaign listener `HyperVCampaignGenerator` builds the binary campaign file entirely in memory and only writes it to file when the campaign is complete. During load testing, it has been noticed that larger campaigns fail to compile because the compiler runs out of memory. This can be fixed by writing the current campaign buffer to file every time it reaches a threshold value.
- The Hyper-V injection driver implements a specialized execution function for every possible combination of requested log values and file-access approach. This has been done to eliminate branch instructions and thus reduce overhead. However, making changes to the campaign execution requires doing the same change in 24 different functions. It should be evaluated how significant the overhead of a single function that makes use of branching is.
- Add support for register-based calling conventions in the Hyper-V injection module.
- Develop a hypercall monitoring tool that does not affect the behavior of the system under test.

Hyper-V Testing Scenarios

- **Software Aging:** Hyper-V's documentation mentions the existence of the hypercalls `HvCreatePartition` and `HvDeletePartition`. Unfortunately, they are not further documented. If it can be figured out how to use those hypercalls, testing campaigns could repeatedly execute the calls to potentially accelerate the aging process of the hypervisor. An effect could be noticeably reduced performance. perfmon also provides performance counters about memory management of the hypervisor, which can be used to identify memory fragmentation.
- **Robustness:** With the execution of the `HvNotifyLongSpinWait` call in the root partition, a robustness problem of Hyper-V was found. The execution of other calls with varying parameters can be used to possibly detect further issues.
- **Performance Isolation:** Performing high-intensity workloads in one guest partition should not lead to performance degradation in a co-located partition. This can be tested using the hypercall testing framework together with CPU benchmarks.

Expand the Framework

- Implement the hypervisor-specific interfaces of the framework for the Xen hypervisor. As a kernel-based injection module, the hInjector [44] can be utilized.
- Implement the hypervisor-specific interfaces of the framework for the KVM hypervisor.

Other:

- Chapter 1 mentioned that vision papers about this work were published, with a focus on software aging [12] and performance [13]. Now that the thesis is finished, the next step is to publish the results in a full research paper.

List of Figures

2.1. Visualization of the x86_64 Paging [17] . . . 7
2.2. Virtualization . . . 9
2.3. Nested Virtualization . . . 12
2.4. Hyper-V High-Level Architecture [32] . . . 13

4.1. Inputs, Outputs, and Side Effects of the Hypercall Testing Framework . . . 21
4.2. Architecture of the Hypercall Testing Framework . . . 22
4.3. Architecture of the Campaign Compiler . . . 24
4.4. Contents of Hyper-V's Hypercall Overlay Page (Intel) . . . 32
4.5. Hypercall Campaign Entry Structure . . . 34
4.6. Hypercall Campaign Structure . . . 35

5.1. HCCDL Lexer Rules Applied to Sequence of Characters . . . 40
5.2. Parser Rule Ambiguity Example . . . 42
5.3. HCCDL Parse Tree Example . . . 42
5.4. HCCDL Compiler Java Project Structure . . . 43
5.5. HCCDL Compiler Java Package Overview . . . 43
5.6. UML Class Diagram of the `parameter` Package . . . 43
5.7. Structure of the `hyperv` Package . . . 46

6.1. Structure of the Hypercall Testing Testbed . . . 55
6.2. Boxplot of the Measured Delay Deviations . . . 57
6.3. Scatterplot of the Measured Deviations . . . 58
6.4. Screenshot of perfmon showing Hypercalls/sec during Campaign . . . 59
6.5. Boxplot of the Maximum Hypercall Injection Rates . . . 61
6.6. Boxplot Showing the Penalties of Logging Values . . . 62

7.1. Hypercalls/sec on an Idling Guest Partition . . . 66
7.2. Hypercalls/sec on a Monitored Idling Guest Partition . . . 66
7.3. Influence of Debugger Halts on Hypercalls/sec . . . 67
7.4. Hypercalls Captured by the Monitoring Tool . . . 68
7.5. Hypercalls/sec during Load Test . . . 74
7.6. Impact of Load Level on `HvExtCallQueryCapabilities` . . . 75
7.7. Impact of Load Level on `HvExtCallQueryCapabilities` . . . 75
7.8. Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest) . . . 77
7.9. Impact of Load Level on `HvFlushVirtualAddressSpace` (Root) . . . 78
7.10. Impact of Load Level on `HvNotifyLongSpinWait` (Param = 0) . . . 79
7.11. Impact of Load Level on `HvNotifyLongSpinWait` (Param = max) . . . 80

8.1. Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 1) . . 102
8.2. Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 2) . . 103
8.3. Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 3) . . 103
8.4. Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 4) . . 104

8.5. Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 5) . . 104
8.6. Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 1) . . 105
8.7. Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 2) . . 106
8.8. Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 3) . . 106
8.9. Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 4) . . 107
8.10. Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 5) . . 107
8.11. Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 1) . . 108
8.12. Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 2) . . 109
8.13. Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 3) . . 109
8.14. Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 4) . . . 110
8.15. Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 5) . . 110
8.16. Impact of Load Level on `HvNotifyLongSpinWait` (Param = 0, Rep 1) . . . 111
8.17. Impact of Load Level on `HvNotifyLongSpinWait` (Param = 0, Rep 2) . . . 112
8.18. Impact of Load Level on `HvNotifyLongSpinWait` (Param = 0, Rep 3) . . . 112
8.19. Impact of Load Level on `HvNotifyLongSpinWait` (Param = max, Rep 1) . 113
8.20. Impact of Load Level on `HvNotifyLongSpinWait` (Param = max, Rep 2) . 114
8.21. Impact of Load Level on `HvNotifyLongSpinWait` (Param = max, Rep 3) . 114

List of Tables

2.1. x86_64 General-Purpose Registers 6
2.2. Hyper-V Hypercall Input Value 13
2.3. Hyper-V Hypercall Result Value 14
2.4. Hyper-V Default Hypercall Calling Convention 14
2.5. Hyper-V Fast Hypercall Calling Convention 15
2.6. Hyper-V Fast Hypercall Calling Convention 15

4.1. HCCDL Operator Precedence 28

6.1. Test Machine Hardware Specifications 56
6.2. Guest Partition Configuration Parameters 56

7.1. Suitablility of Documented Hypercalls for Testing 71

8.1. Assessment of Research Questions and Goals 82

Listings

4.1. Example: JavaScript Object Notation (JSON) 26
4.2. Example: YAML ain't Markup Language (YAML) 26
4.3. Example: Python 27
4.4. HCCDL For Loop Example 29
4.5. HCCDL Procedure Example 29
4.6. HCCDL Global Variable Example 30
4.7. Course of Action of the Hyper-V Hypercall Injection Driver 33

5.1. HCCDL Lexer Rules 40
5.2. HCCDL Program Structure Parser Rules 41
5.3. HCCDL Statement Parser Rules 41
5.4. HCCDL Program Structure Parser Rules 41
5.5. HCCDL Campaign for Parse Tree Example 42
5.6. Interface `hycallparser.builtinProcedures.Procedure` 44
5.7. Interface `visitors_and_listeners.campaignListeners.CampaignListener` 45
5.8. `HyperVConsoleReporter` Output 47
5.9. `HyperVCSVTimingReporter` Output 48
5.10. Calculation of Log Buffer Size (in byte) 50
5.11. Injection Driver `struct campaign_entry` 50
5.12. Injection Driver Hypercall Invocation 51
5.13. Windows Debugger (WinDbg) Scripting Language: Factorial Computation [76] 52
5.14. Minimal pykd Script to Monitor Hypercalls 52

6.1. HCCDL Campaign to Test Delay Accuracy 56
6.2. HCCDL Campaign to Determine Maximum Hypercall Rate 59
6.3. HCCDL Campaign for Memory- and File-based Execution Differentiation . 60

7.1. HCCDL Campaign to Validate `HvExtCallQueryCapabilities` 72
7.2. Validation Result for `HvExtCallQueryCapabilities` on Root Partition . . 72
7.3. Validation Result for `HvExtCallQueryCapabilities` on Guest Partition . . 72
7.4. HCCDL Load Test Campaign for `HvExtCallQueryCapabilities` 73
7.5. HCCDL Campaign to Validate `HvFlushVirtualAddressSpace` 76
7.6. HCCDL Campaign to Validate `HvExtCallGetBootZeroedMemory` 78
7.7. HCCDL Campaign to Validate `HvNotifyLongSpinWait` 79

8.1. Example: Extensible Markup Language (XML) 101

Acronyms

ANTLR	ANother Tool for Language Recognition
AMD	Advanced Micro Devices
AMD-V	AMD Virtualization
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
AST	Abstract Syntax Tree
CAPEC	Common Attack Pattern Enumeration and Classification
CPU	Central Processing Unit
CSV	Comma-Separated Values
DMA	Direct Memory Access
DOS	Denial of Service
DSL	Domain-Specific Language
EPT	Extended Page Tables
GNU	GNU's Not Unix
GPA	Guest Physical Address
GPU	Graphics Processing Unit
GUI	Graphical User Interface
GVA	Guest Virtual Address
HCCDL	Hypercall Campaign Description Language
IaaS	Infrastructure as a Service
I/O	Input/Output
IOMMU	I/O Memory Management Unit
IP	Internet Protocol
IEEE	Institute of Electrical and Electronics Engineers
IQR	Interquartile Range
IRP	I/O Request Packet
JSON	JavaScript Object Notation
KMDF	Kernel-Mode Driver Framework
KVM	Kernel-based Virtual Machine
LKM	Loadable Kernel Module
MITM	Man In The Middle
MMU	Memory Management Unit
MSR	Model-Specific Register
NIC	Network Interface Card

NVM	Non-Volatile Memory
NVM	NVM Express
OC	Overclock
OS	Operating System
PCI	Peripheral Component Interconnect
PD	Page Directory
PDE	Page Directory Entry
PDP	Page Directory Pointer
PDPE	Page Directory Pointer Entry
PIC	Programmable Interrupt Controller
PML4	Page Map Level 4
PML4E	Page Map Level 4 Entry
PnP	Plug and Play
PT	Page Table
PTE	Page Table Entry
RAM	Random Access Memory
RegEx	Regular Expression
RVI	Rapid Virtualization Indexing
SIMD	Single Instruction, Multiple Data
SLA	Service Level Agreement
SLAT	Second Level Address Translation
SLO	Service Level Objective
SPA	System Physical Address
SR-IOV	Single-Root I/O-Virtualization
SSE	Streaming SIMD Extensions
TLB	Translation Lookaside Buffer
TLFS	Top-Level Functional Specification
UD	Undefined Operation
UMDF	User-Mode Driver Framework
UML	Unified Modeling Language
VBS	Virtualization-Based Security
VM	Virtual Machine
VMM	Virtual Machine Monitor
VT-x	Virtualization Technology for x86
VT-d	Virtualization Technology for Directed I/O

VTL	Virtual Trust Level
WDF	Windows Driver Foundation
WDM	Windows Driver Model
WinDbg	Windows Debugger
XML	Extensible Markup Language
YAML	YAML ain't Markup Language

Bibliography

[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*, pp. 255–260, ACM, 2007.

[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)*, pp. 189–198, IEEE, 2014.

[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] “Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.3 Percent in 2019.” `https://www.gartner.com/en/newsroom/press-releases/2018-09-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2019`. Accessed: 2019-06-11.

[7] N. Elhage, “Virtunoid: Breaking out of kvm,” *Black Hat USA*, 2011.

[8] K. Kortchinsky, “Cloudburst: Hacking 3d (and breaking out of vm-ware). url: https://www. blackhat. com/presentations/bh-usa-09/kortchinsky,” *BHUSA09-Kortchinsky-Cloudburst-SLIDES. pdf (vid. pág. 13)*, 2009.

[9] “Virtualization-based Security (VBS).” `https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs`. Accessed: 2019-12-18.

[10] “Microsoft snags hotly contested $10 billion defense contract, beating out Amazon.” `https://www.cnbc.com/2019/10/25/microsoft-wins-major-defense-cloud-contract-beating-out-amazon.html`. Accessed: 2019-10-27.

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

[12] L. Beierlieb, L. Iffländer, A. Milenkoski, C. F. Goncalves, N. Antunes, and S. Kounev, “Towards Testing the Software Aging Behavior of Hypervisor Hypercall Interfaces,” in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 11 2019.

[13] L. Beierlieb, L. Iffländer, S. Kounev, and A. Milenkoski, “Towards Testing the Performance Influence of Hypervisor Hypercall Interface Behavior,” in *Proceedings of the 10th Symposium on Software Performance 2019 (SSP’19)*, 11 2019.

[14] K. van Surksum, “Hypervisor top-level functional specification: Windows server 2012,” 2013.

[15] “Work Package 6: Virtual Secure Mode.” `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Cyber-Sicherheit/SiSyPHus/Workpackage6_Virtual_Secure_Mode.pdf?__blob=publicationFile&v=2`. Accessed: 2019-12-08.

[16] “Intel 64 and IA-32 Architectures Software Developer’s Manual.” `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`. Accessed: 2019-12-04.

[17] “Paging in x86 and x64.” `http://www.renyujie.net/articles/article_ca_x86_5.php`. Accessed: 2019-12-03.

[18] “PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer’s Manual.” `https://pdos.csail.mit.edu/6.828/2018/readings/hardware/8254x_GBe_SDM.pdf`. Accessed: 2019-12-04.

[19] 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.

[20] U. 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*, p. 129, 2000.

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

[22] 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.

[23] J. P. Walters, V. Chaudhary, M. Cha, S. Guercio Jr, and S. Gallo, “A comparison of virtualization technologies for hpc,” in *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*, pp. 861–868, IEEE, 2008.

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *ACM SIGOPS operating systems review*, vol. 37, pp. 164–177, ACM, 2003.

[25] 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.

[26] A. Virtualization, “Amd-v nested paging,” *White paper.[Online] Available: http://sites. amd. com/us/business/it-solutions/virtualization/Pages/amd-v. aspx*, 2008.

[27] I. AMD and O. Virtualization, “Technology (iommu) specification,” 2007.

[28] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, “Intel virtualization technology for directed i/o.,” *Intel technology journal*, vol. 10, no. 3, 2006.

[29] A. Thongthua and S. Ngamsuriyaroj, “Assessment of hypervisor vulnerabilities,” in *2016 International Conference on Cloud Computing Research and Innovations (ICCCRI)*, pp. 71–77, IEEE, 2016.

[30] Y. Dong, Z. Yu, and G. Rose, “Sr-iov networking in xen: Architecture, design and implementation.,” in *Workshop on I/O Virtualization*, vol. 2, 2008.

[31] A. Whitaker, M. Shaw, S. D. Gribble, *et al.*, "Denali: Lightweight virtual machines for distributed and networked applications," tech. rep., Technical Report 02-02-01, University of Washington, 2002.

[32] "Microsoft snags hotly contested $10 billion defense contract, beating out Amazon." `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture`. Accessed: 2019-11-25.

[33] 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*, pp. 261–270, IEEE, 2000.

[34] 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)*, pp. 231–235, IEEE, 1998.

[35] 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)*, pp. 30–37, IEEE, 1999.

[36] 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*, pp. 72–79, IEEE, 1997.

[37] 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)*, pp. 131–136, IEEE, 2007.

[38] M. Vieira, N. Laranjeiro, and H. Madeira, "Benchmarking the robustness of web services," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, pp. 322–329, IEEE, 2007.

[39] 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*, pp. 65–66, IEEE Computer Society, 2007.

[40] 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.

[41] 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)*, pp. 161–166, IEEE, 2018.

[42] 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.

[43] T. Ormandy, "An empirical study into the security exposure to hosts of hostile virtualized environments," 2007.

[44] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, and S. Kounev, "Hinjector: injecting hypercall attacks for evaluating vmi-based intrusion detection systems," in *Poster Reception at the 2013 Annual Computer Security Applications Conference (ACSAC 2013)*, Applied Computer Security Associates (ACSA), 2013.

[45] R. Krebs, C. Momm, and S. Kounev, "Metrics and techniques for quantifying performance isolation in cloud environments," *Science of Computer Programming*, vol. 90, pp. 116–134, 2014.

[46] B. Verghese, A. Gupta, and M. Rosenblum, "Performance isolation: sharing and isolation in shared-memory multiprocessors," in *ACM SIGPLAN Notices*, vol. 33, pp. 181–192, ACM, 1998.

[47] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments," in *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 51–58, IEEE, 2010.

[48] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, and H. H. Huang, "Understanding the effects of hypervisor i/o scheduling for virtual machine performance interference," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pp. 34–41, IEEE, 2012.

[49] Q. Huang and P. P. Lee, "An experimental study of cascading performance interference in a virtualized environment," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 43–52, 2013.

[50] G. report, "Gao - patriot missile software problem." `http://fas.org/spp/starwars/gao/im92026.htm`, 1991. Accessed: 2014-09-07.

[51] 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.

[52] 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*, pp. 178–187, 2013.

[53] 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*, pp. 135–144, 2017.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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), pp. 224–233, IEEE Computer Society, 2011.

[60] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, pp. 107–109, Feb. 2007.

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

[62] K. Vaidyanathan and K. Trivedi, "A comprehensive model for software rejuvenation," *Dependable and Secure Computing, IEEE Transactions on*, vol. 2, pp. 124–137, April 2005.

[63] 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.

[64] 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.

[65] 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*, pp. 84–93, Nov. 1999.

[66] 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.

[67] 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.

[68] 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.

[69] C. H. H. Le, *Protecting xen hypercalls: Intrusion detection/prevention in a virtualization environment.* PhD thesis, University of British Columbia, 2009.

[70] C. Dall, S.-W. Li, and J. Nieh, "Optimizing the design and implementation of the linux {ARM} hypervisor," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pp. 221–233, 2017.

[71] P. Tran-Gia, *Analytische Leistungsbewertung verteilter Systeme: Eine Einführung.* Springer-Verlag, 2013.

[72] T. Parr, *The definitive ANTLR 4 reference.* Pragmatic Bookshelf, 2013.

[73] T. Parr, P. Wells, R. Klaren, L. Craymer, J. Coker, S. Stanchfield, J. Mitchell, and C. Flack, "What's antlr," 2004.

[74] "Getting started with Windows drivers." `https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/`. Accessed: 2019-12-09.

[75] "Writing a Hyper-V "Bridge" for Fuzzing — Part 1: WDF." `http://www.alex-ionescu.com/?p=377`. Accessed: 2019-12-09.

[76] "Introduction to WinDbg Scripts for C/C++ Users." `http://www.dumpanalysis.org/WCDA/WCDA-Sample-Chapter.pdf`. Accessed: 2019-12-09.

Appendix

A. Approach

Listing 8.1: Example: XML

```
<0>
    <instruction>loop</instruction>
    <loopcount>1000</loopcount>
    <loopbody>
        <instruction>hycall</instruction>
        <rcx>3425</rcx>
        <rdx>8</rdx>
    </loopbody>
    <loopbody>
        <instruction>delay</instruction>
        <duration>50</duration>
    </loopbody>
</0>
```

B. Repeated Measurements: HvExtCallQueryCapabilities

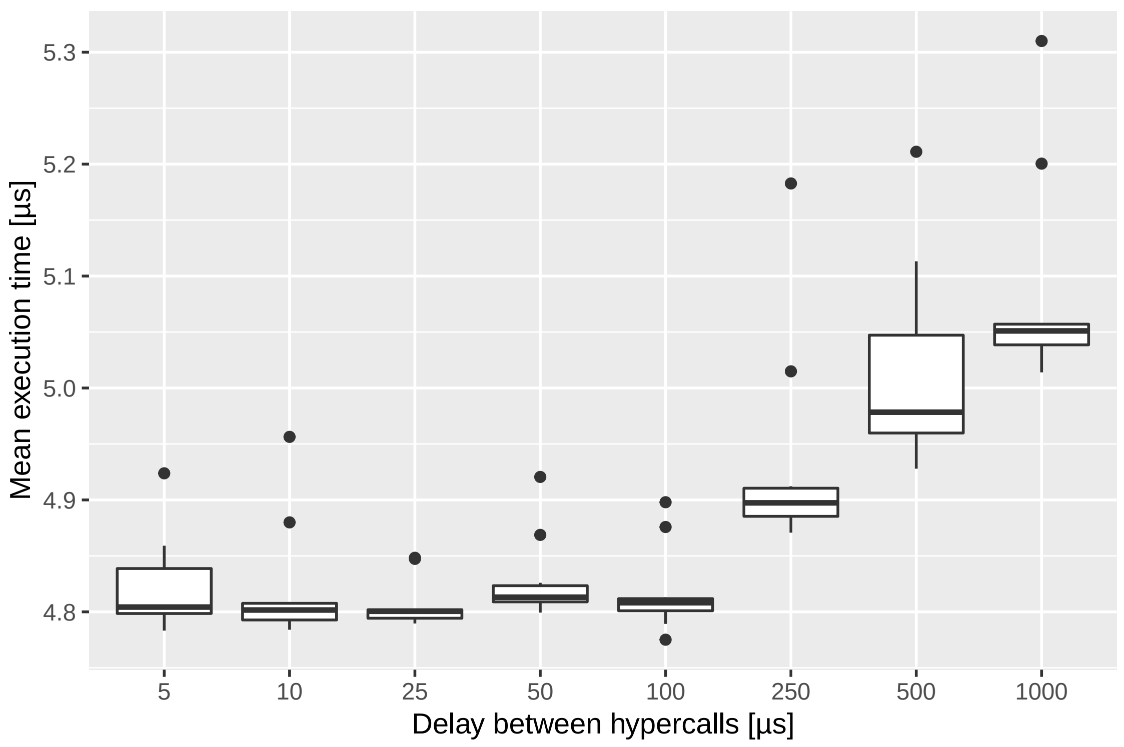


Figure 8.1.: Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 1)

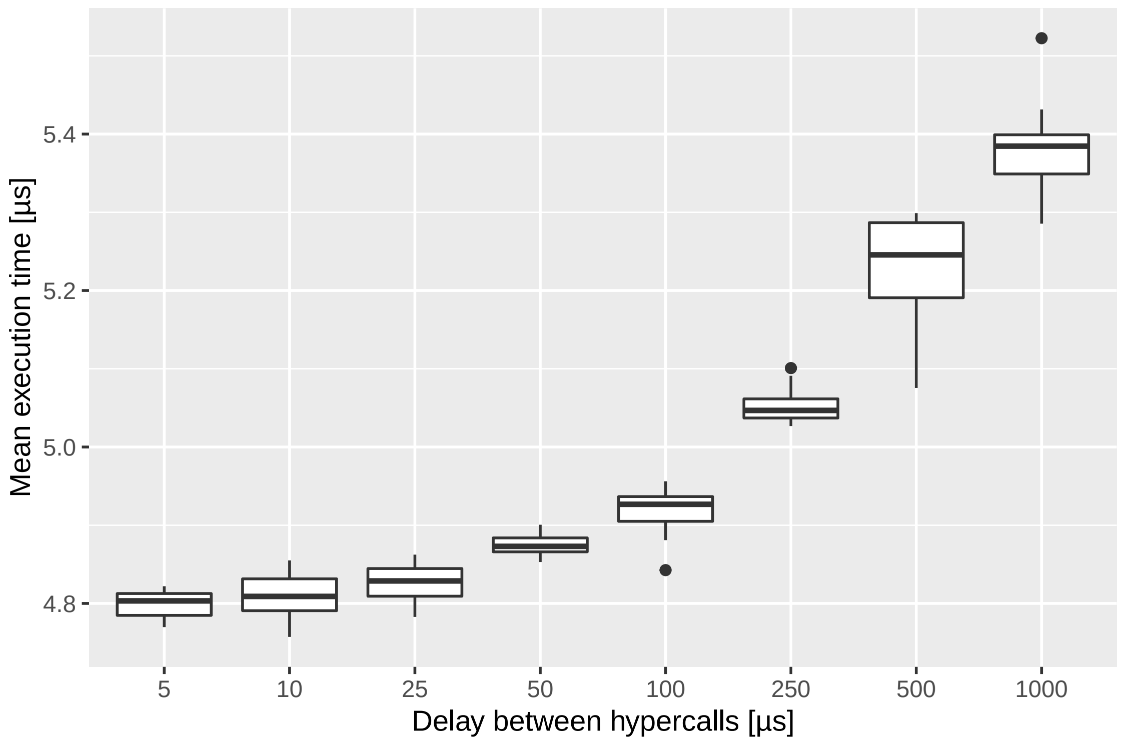


Figure 8.2.: Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 2)

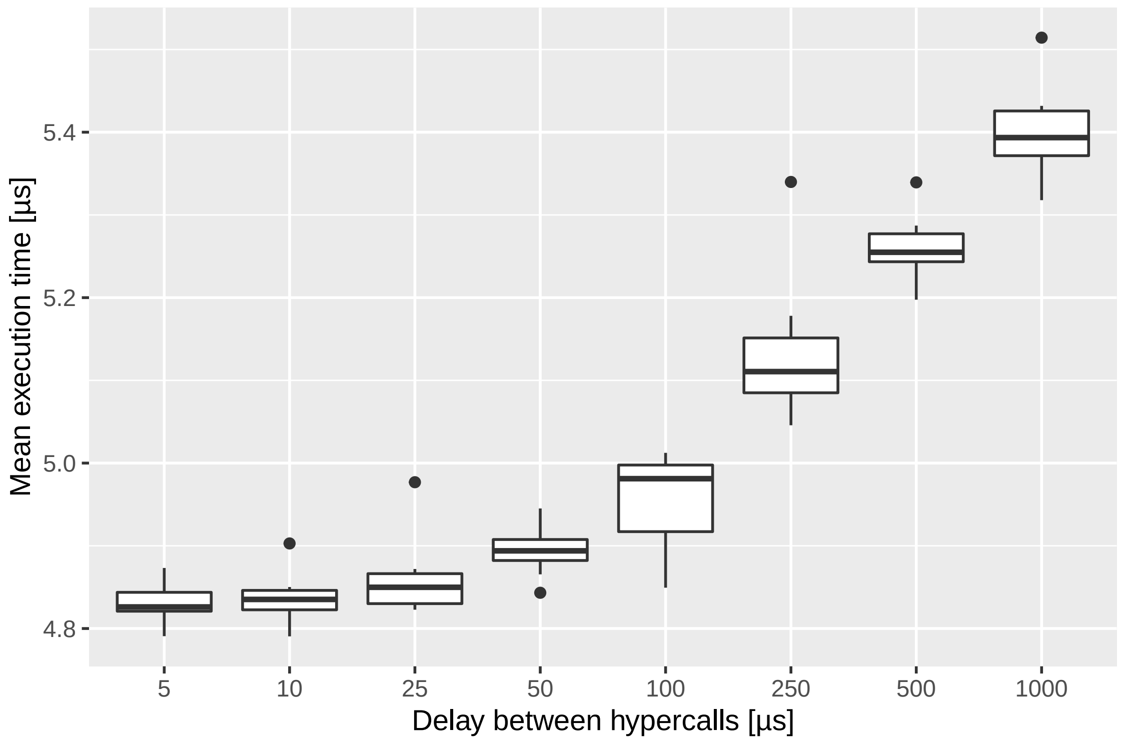


Figure 8.3.: Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 3)

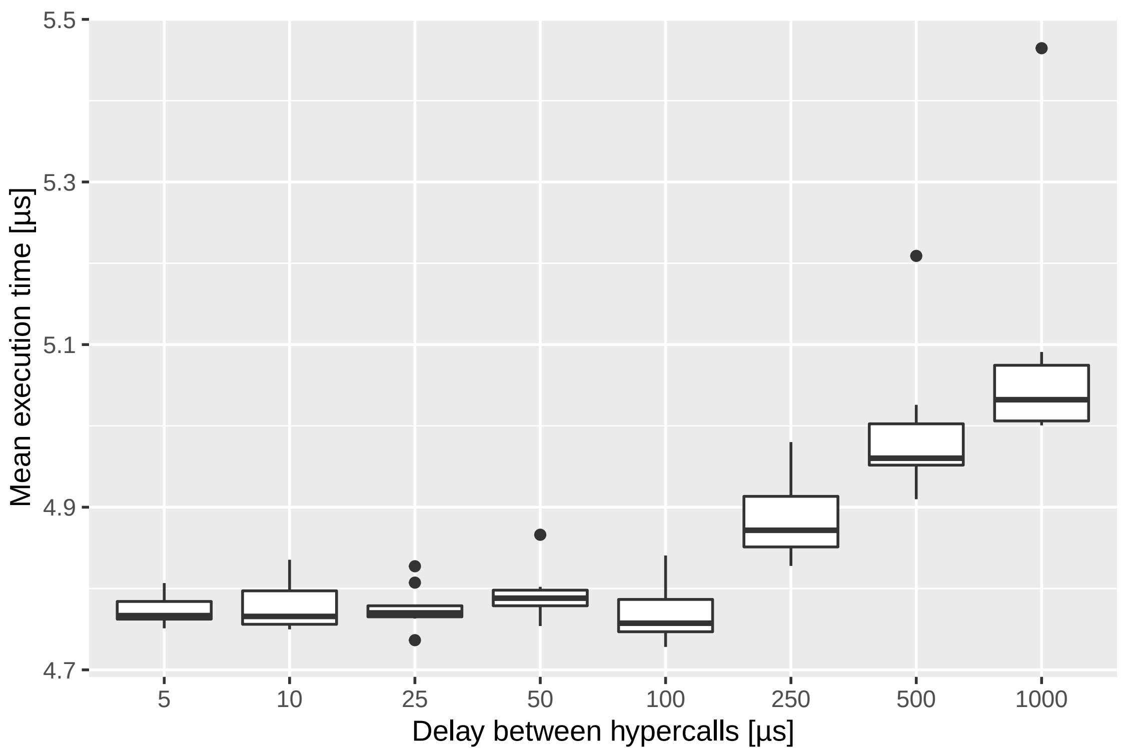


Figure 8.4.: Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 4)

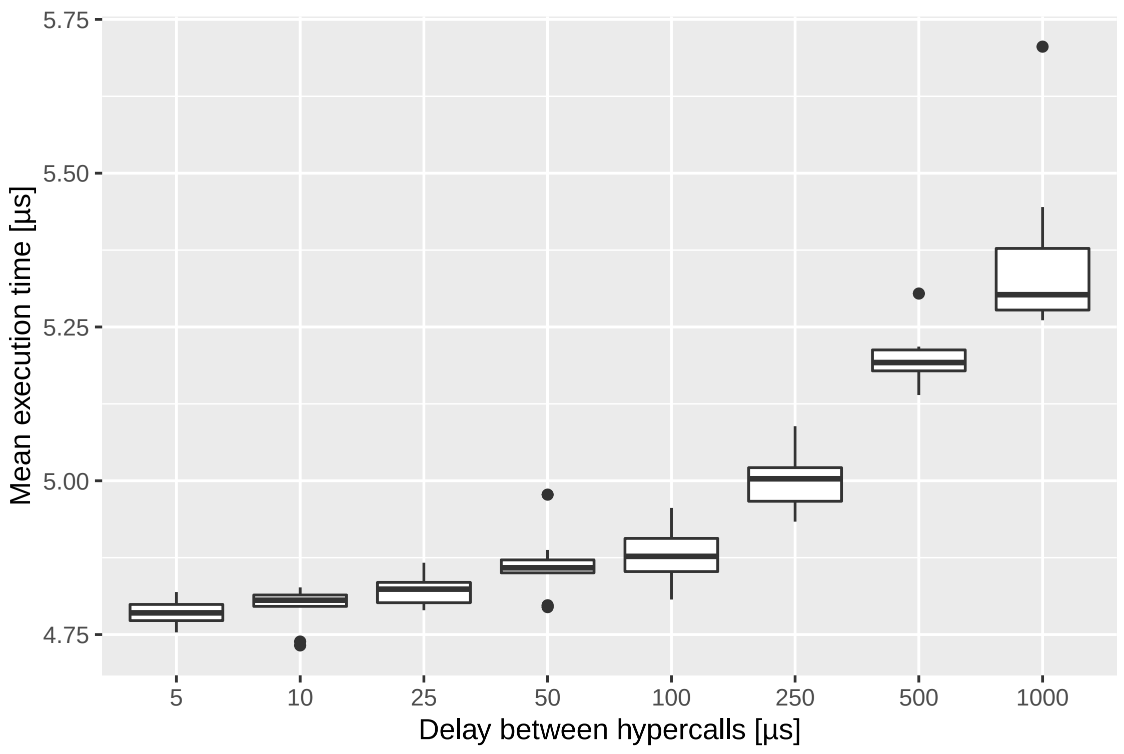


Figure 8.5.: Impact of Load Level on `HvExtCallQueryCapabilities` (Repetition 5)

C. Repeated Measurements: HvFlushVirtualAddressSpace (Guest)

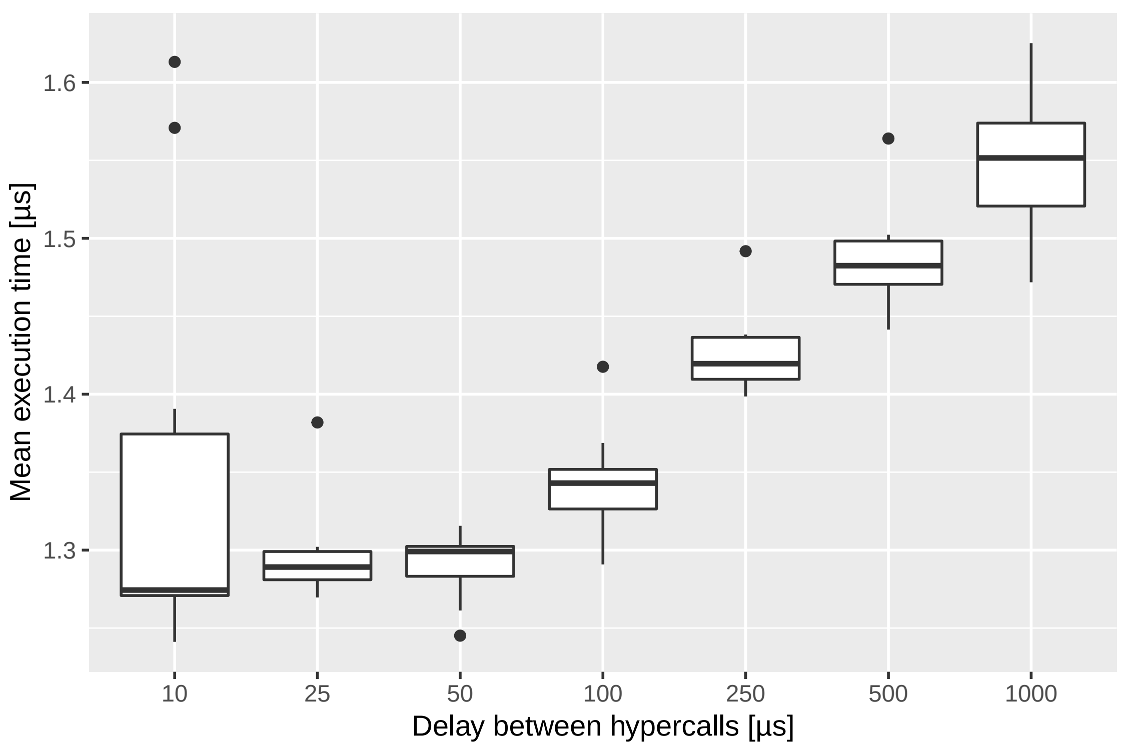


Figure 8.6.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 1)

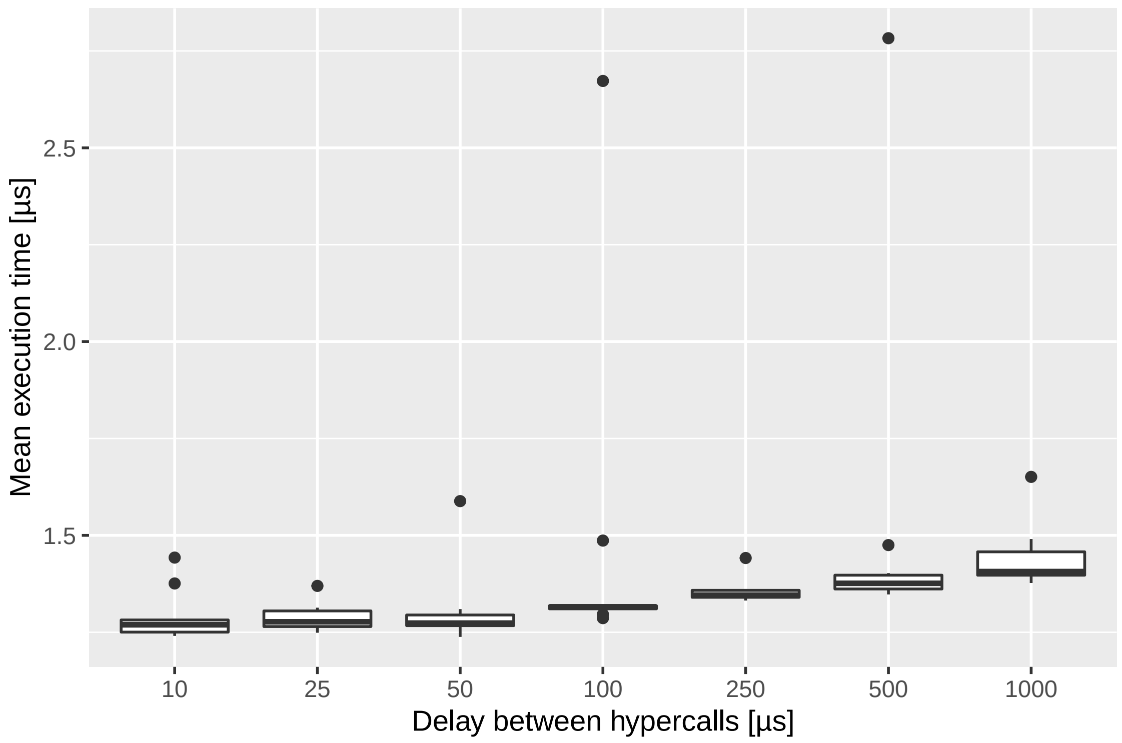


Figure 8.7.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 2)

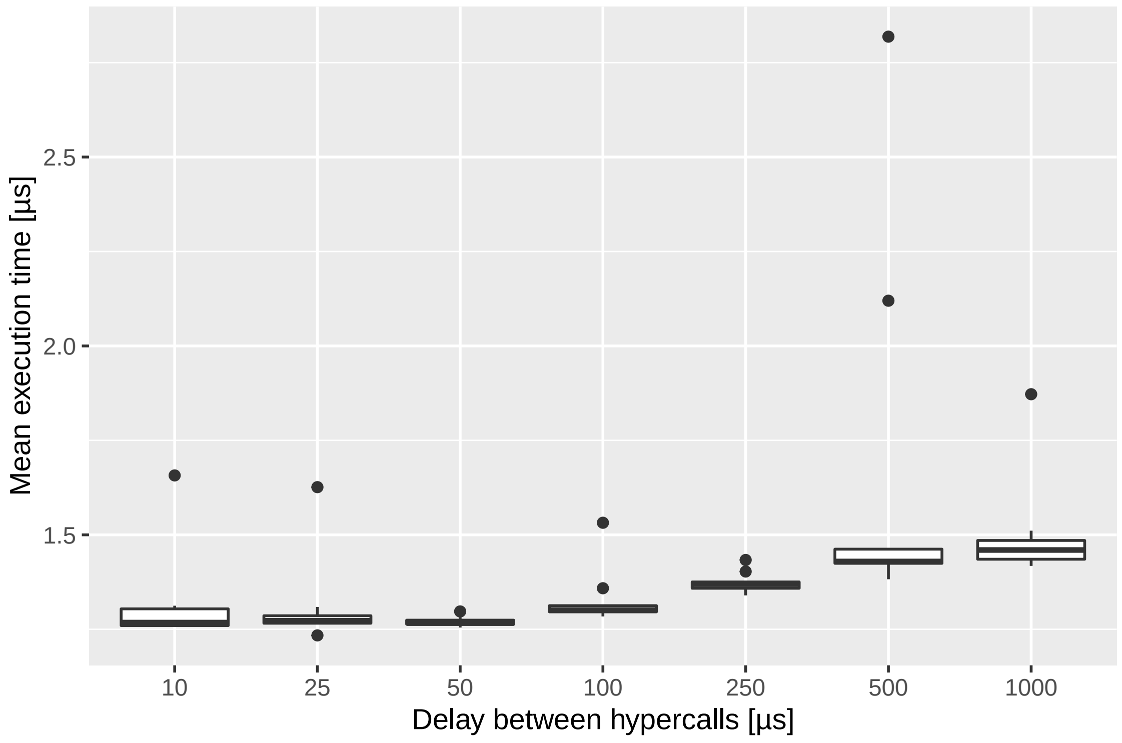


Figure 8.8.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 3)

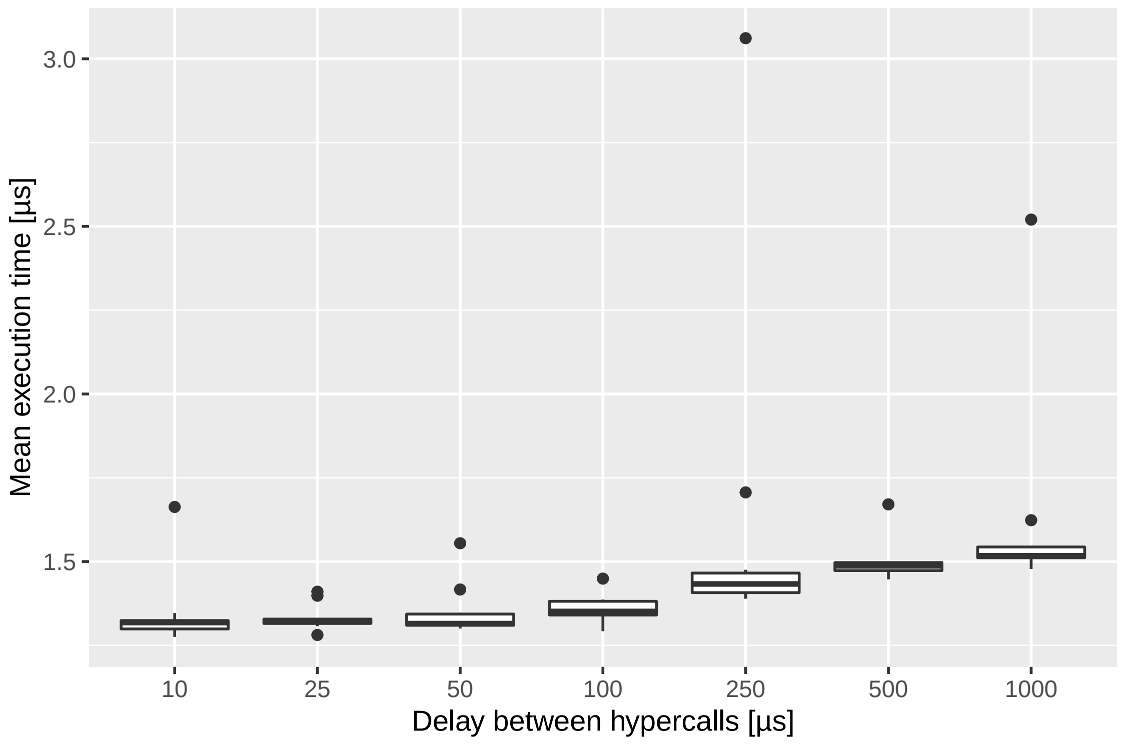


Figure 8.9.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 4)

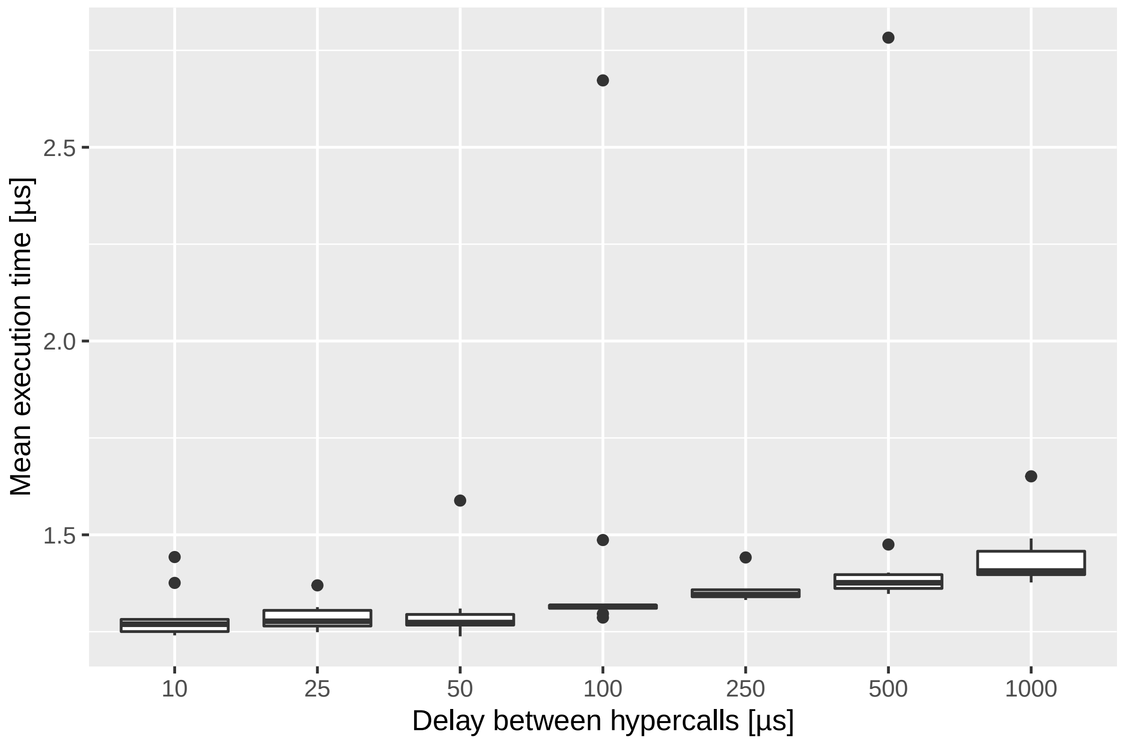


Figure 8.10.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Guest, Rep 5)

D. Repeated Measurements: HvFlushVirtualAddressSpace (Root)

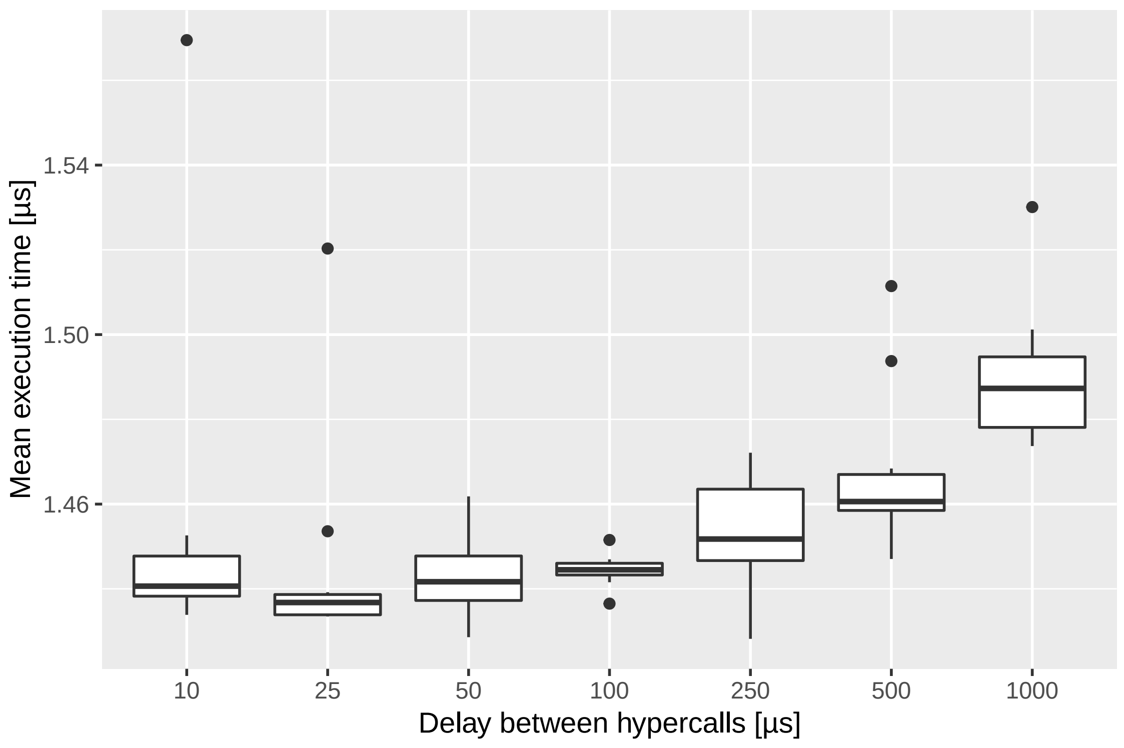


Figure 8.11.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 1)

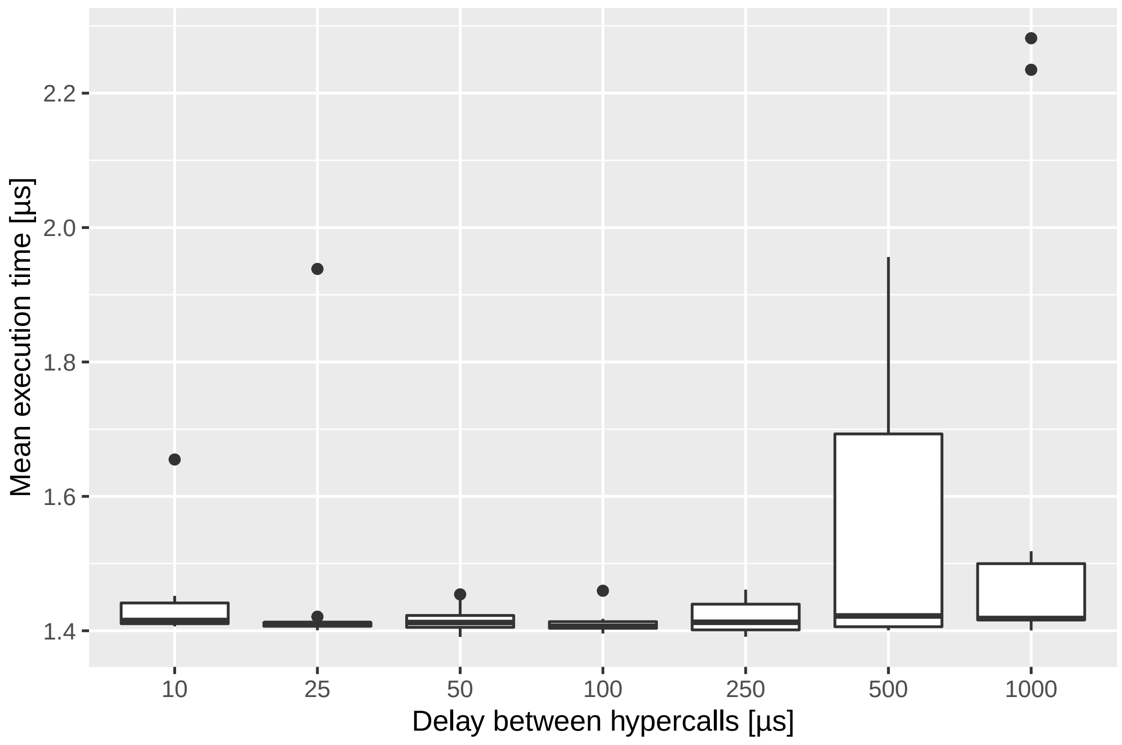


Figure 8.12.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 2)

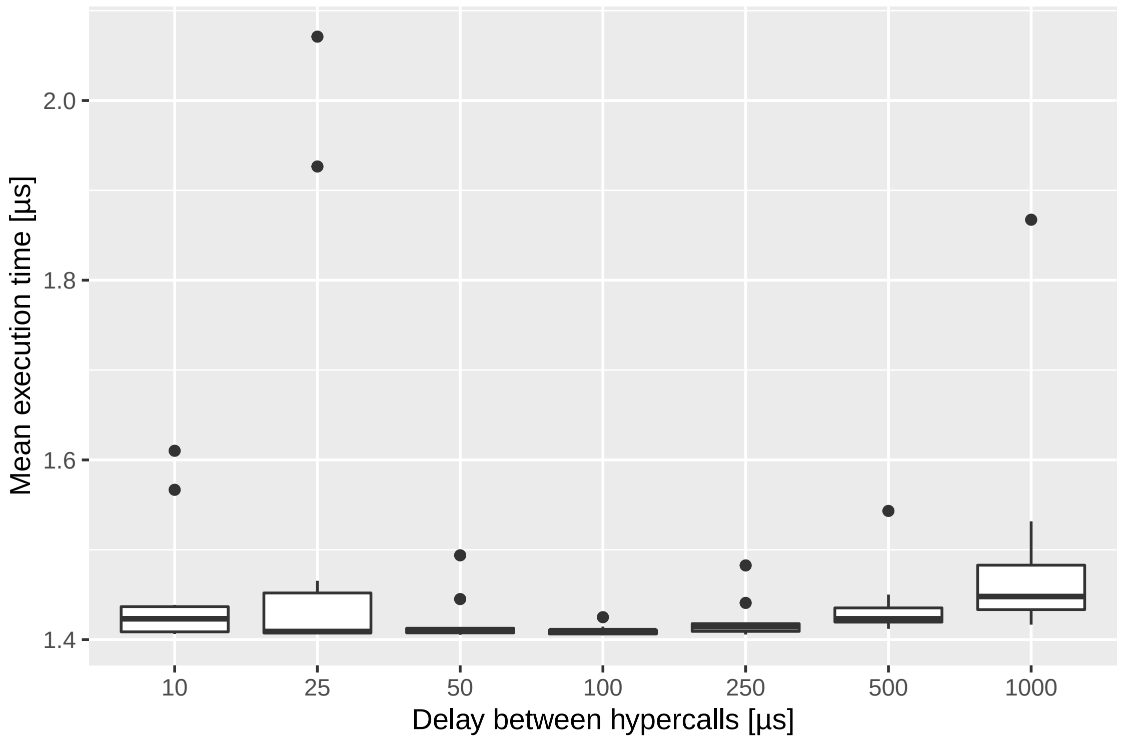


Figure 8.13.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 3)

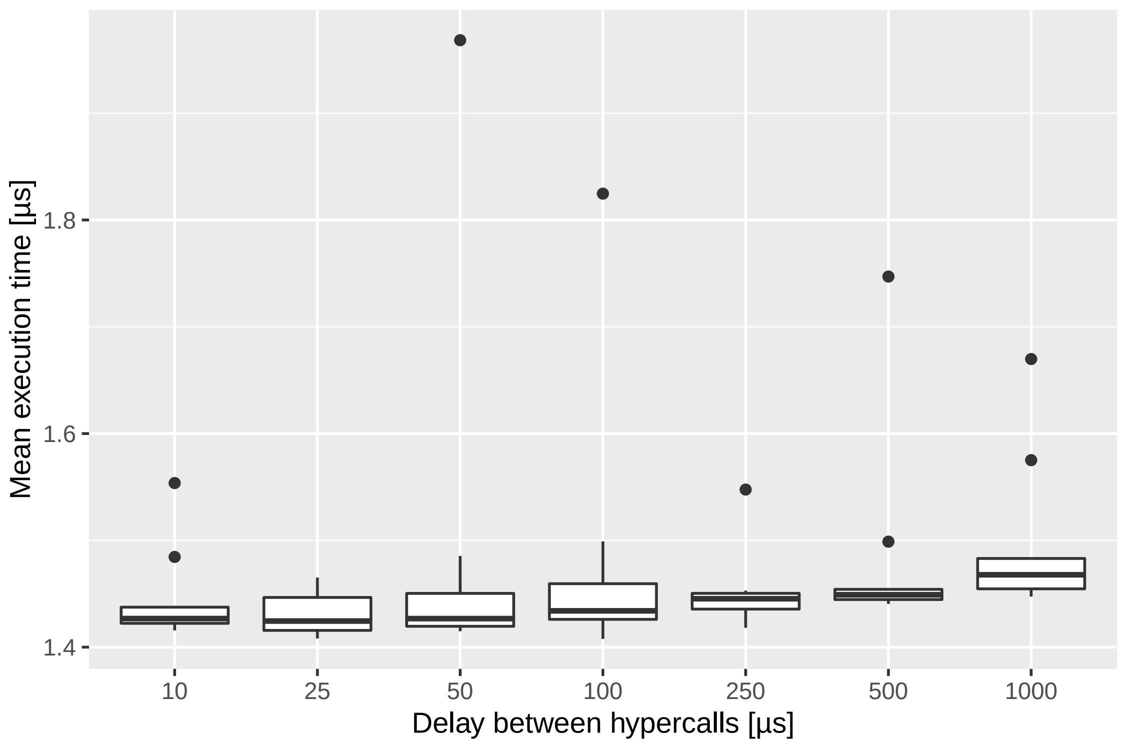


Figure 8.14.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 4)

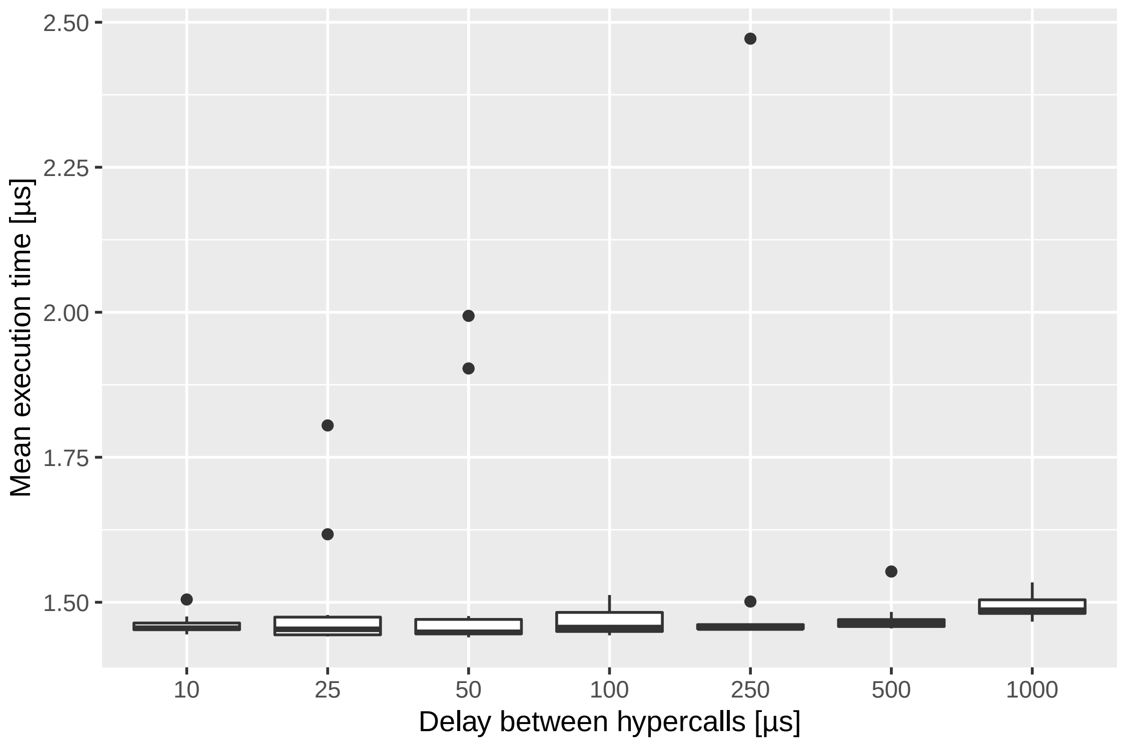


Figure 8.15.: Impact of Load Level on `HvFlushVirtualAddressSpace` (Root, Rep 5)

E. Repeated Measurements: HvNotifyLongSpinWait (Param = 0)

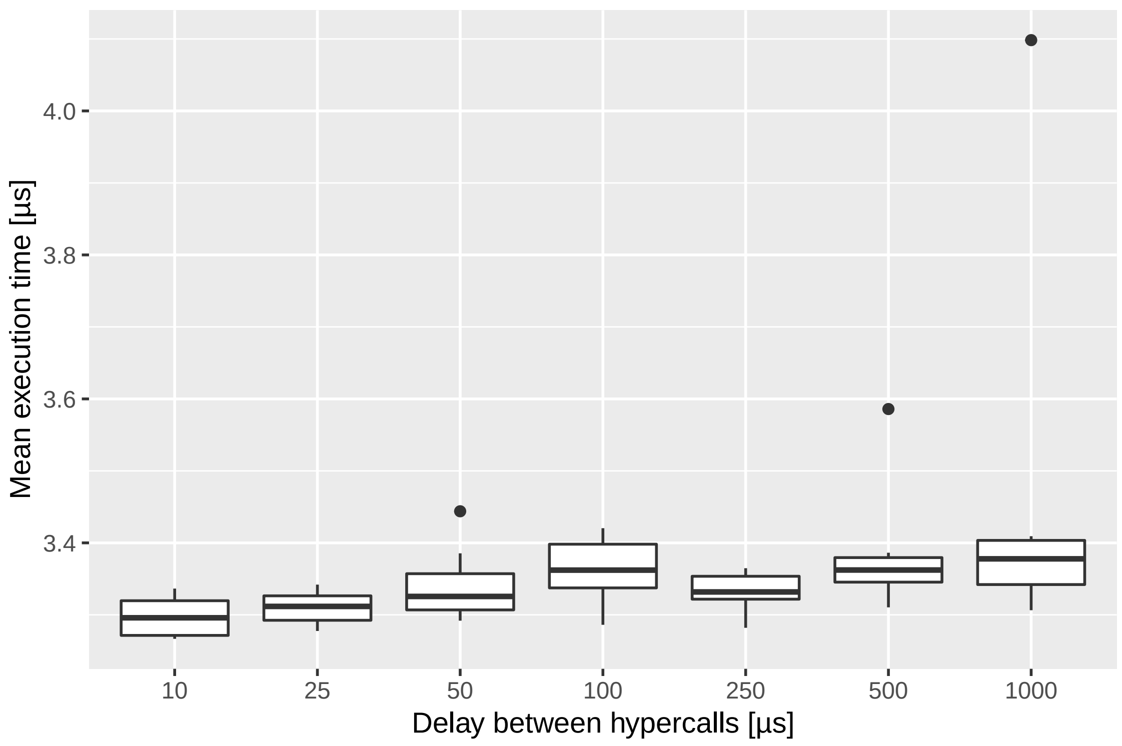


Figure 8.16.: Impact of Load Level on `HvNotifyLongSpinWait` (Param = 0, Rep 1)

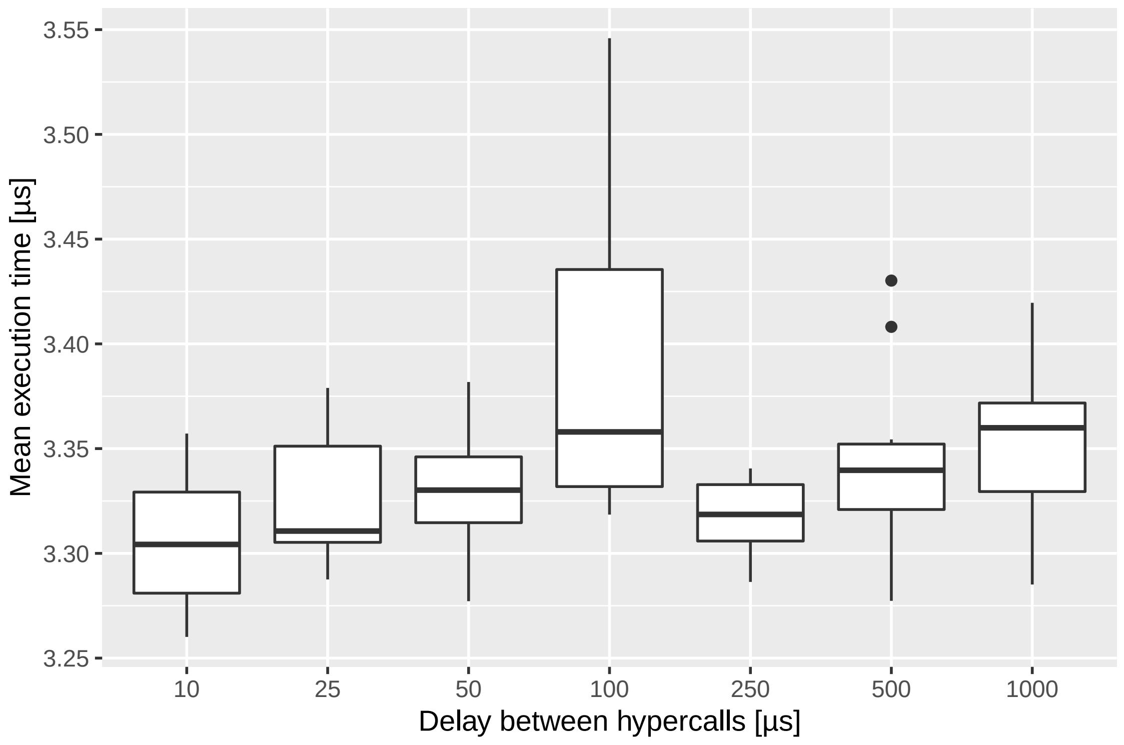


Figure 8.17.: Impact of Load Level on `HvNotifyLongSpinWait` (Param = 0, Rep 2)

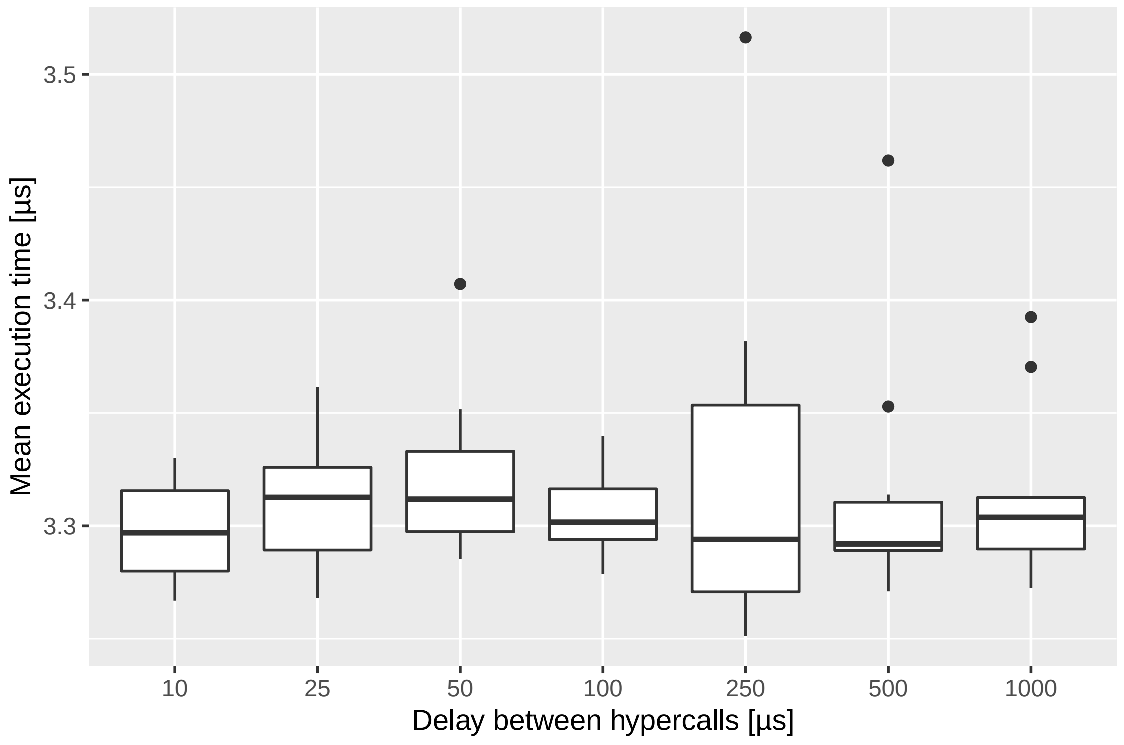


Figure 8.18.: Impact of Load Level on `HvNotifyLongSpinWait` (Param = 0, Rep 3)

F. Repeated Measurements: HvNotifyLongSpinWait (Param = max)

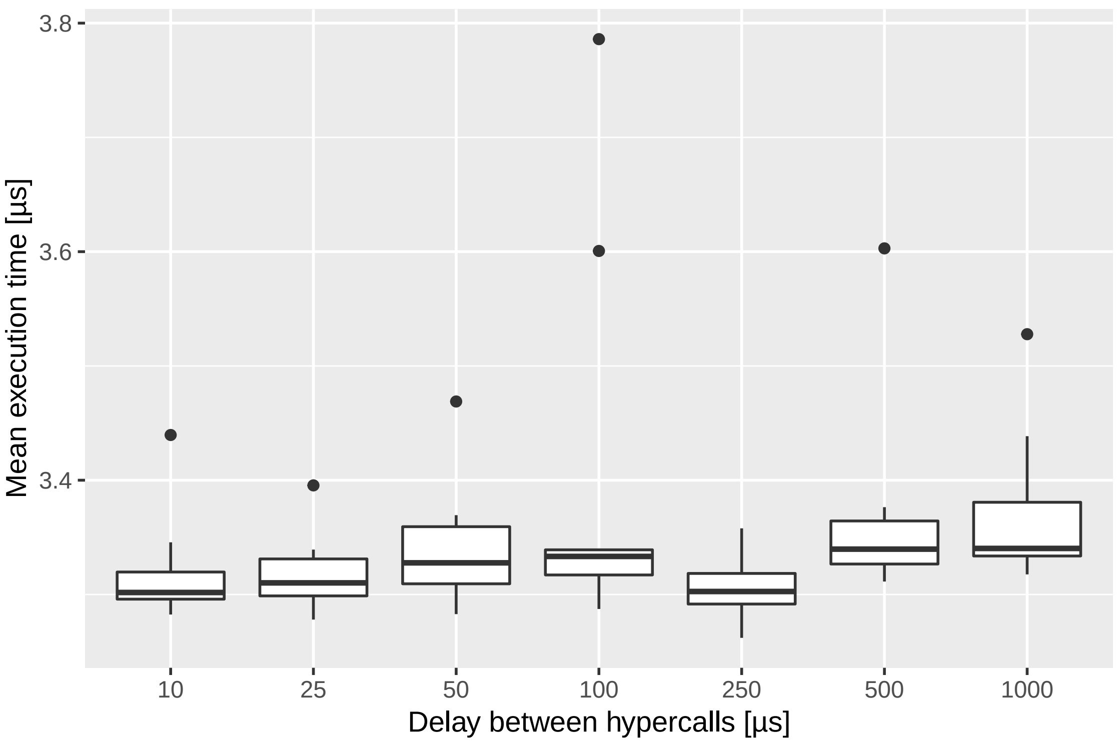


Figure 8.19.: Impact of Load Level on `HvNotifyLongSpinWait` (Param = max, Rep 1)

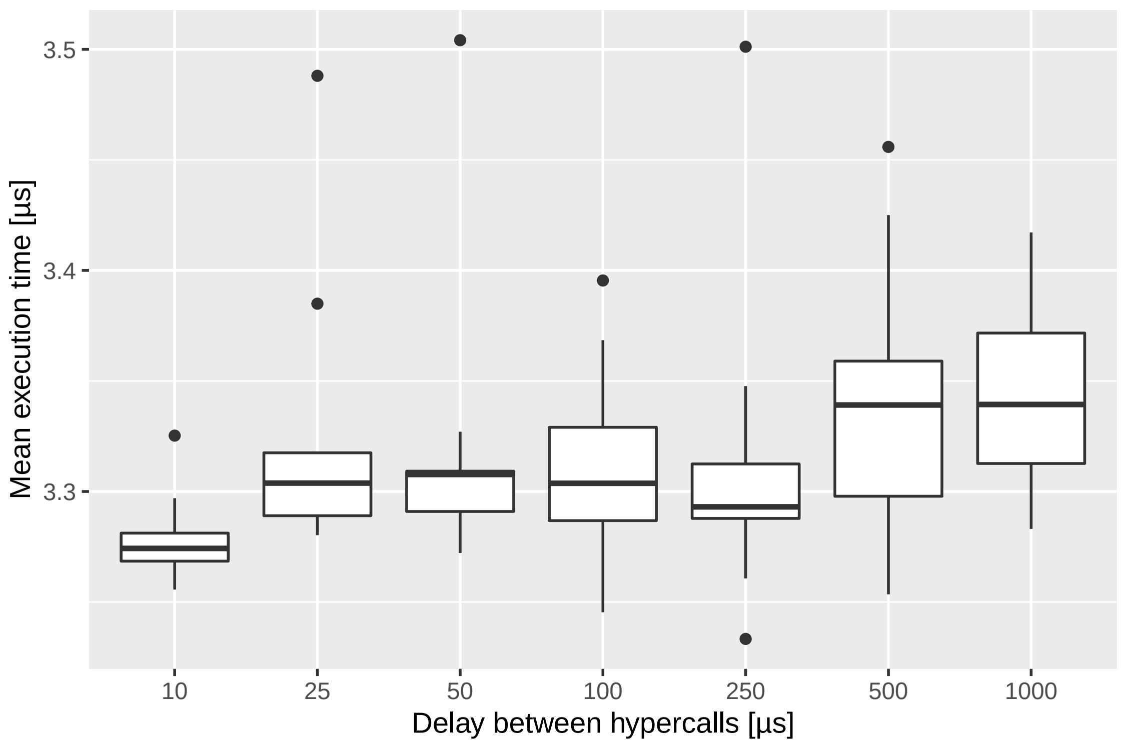


Figure 8.20.: Impact of Load Level on `HvNotifyLongSpinWait` (Param = max, Rep 2)

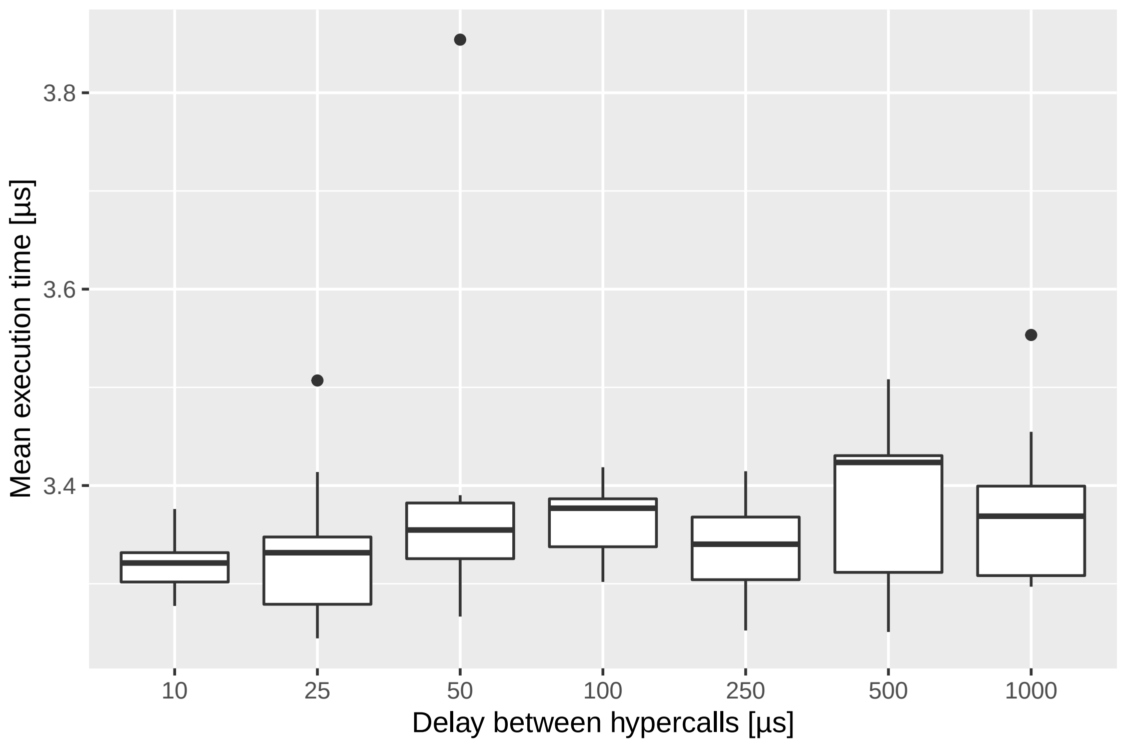


Figure 8.21.: Impact of Load Level on `HvNotifyLongSpinWait` (Param = max, Rep 3)