Karlsruher Institut für Technologie

Tracing Privileged Memory Accesses to Discover Software Vulnerabilities

Masterarbeit
von

Felix Wilhelm

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Dipl.-Inform. Marc Rittinghaus

Bearbeitungszeit: 07. Mai 2015 – 06. November 2015

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 6. November 2015

Abstract

Shared Memory is an important mechanism for efficient inter-process communication. When one side of the communication has higher privileges than its counterpart, the shared memory interface becomes a trust boundary and privileged code operating on it needs to be audited for security vulnerabilities.

In this thesis we present an approach based on memory tracing to discover vulnerabilities in shared memory interfaces. In contrast to other works in this area, the presented implementation is based on hardware-assisted virtualization and uses manipulation of EPT permissions to intercept memory accesses.

We evaluate our implementation against paravirtualized device drivers for the Xen hypervisor, which use shared memory for inter-domain communication. Besides successfully identifying the privileged components responsible for processing untrusted shared memory data, the presented analysis algorithms are used to discover three novel security vulnerabilities in security critical backend components.

Contents

1 Introduction

Memory pages shared between different execution contexts are a fundamental communication mechanism of modern computer systems. In many cases one side of the communication has higher privileges and needs to protect itself against malicious behavior of its counterpart. Examples for this situation include communication between userland and kernel space[20], sandbox implementations of modern web browsers[44] and the inter-domain communication of popular hypervisors[8].

In addition to classic software vulnerabilities, such as missing validation and verification, shared memory interfaces can suffer from a special type of race condition called *double fetch* vulnerability. Bochspwn[20] first demonstrated how these issues can be used for local privilege escalation attacks against the Windows kernel and how memory tracing can be leveraged to identify these vulnerability types automatically. While Bochspwn was successfully applied in the context of user-kernel interaction, its reliance on an instrumented version of the Bochs CPU emulator leads to an extremely high overhead and bad performance. This limits its suitability for the analysis of more complex software environments.

The objective of this thesis is the discovery of software vulnerabilities in the inter-domain communication interfaces of mainstream hypervisors. To achieve this goal, this thesis presents and implements an approach to discover such vulnerabilities by tracing and analyzing all privileged read and write accesses to shared memory pages. We improve upon the research presented in [20], by designing and implementing a toolkit for memory access tracing and pattern analysis using hardware-assisted virtualization and modified page table permissions.

In comparison to approaches based on software emulation, this reduces the passive overhead significantly and allows the targeted tracing of shared memory communication even in very complex environments. The presented implementation is based on the open source Xen hypervisor[3] as platform for nested virtualization and uses Simutrace[34] as highly efficient trace storage, allowing for the collection and offline analysis of even long running traces. Furthermore, large parts of the design and implementation are completely target agnostic, making them reusable for analysis of different hypervisors and even other shared memory interfaces such as sandbox implementations.

The effectiveness of the presented approach is evaluated by analyzing the security aspects of paravirtualized devices in Xen. Besides being able to identify the privileged components that can be targeted by an attacker, our implementation

is able to discover three novel security vulnerabilities affecting the Xen hypervisor. These vulnerabilities were reported to the Xen maintainers and were assigned XSA-155[52].

The remainder of this work is structured as follows: Chapter 2 discussed several core concepts required for this thesis. Besides introducing shared memory communication and double fetch vulnerabilities in general, the different types of virtualization on the Intel x64 architecture are presented. This is followed by an introduction into the concept of virtual machine introspection and a detailed discussion of the overall architecture of three mainstream hypervisors. Chapter 3 highlights the problem of security for inter-domain communication and reviews several different ways for discovering vulnerabilities in these interfaces. After this, the proposed design of our solution is presented in Chapter 4. Important aspects of the implementation are reviewed in Chapter 5, before the results of the performed evaluation are finally presented in Chapter 6. The thesis finishes with a final conclusion and a discussion of further research topics in Chapter 7.

2 Background

This chapter introduces the technical concepts and terminology required for the rest of this thesis. Section 2.1 introduces the idea of shared memory communication and the reasons for its popularity. In comparison to other IPC mechanisms, shared memory can suffer from a special type of vulnerability called double fetch, which is introduced in Section 2.2. The chapter continues with Section 2.3, which describes virtualization on the Intel x64 architecture, concentrating on the Intel VT-x extensions. After an introduction into Virtual Machine Introspection (VMI) in Section 2.4, the chapter concludes with an overview about the architectures of three mainstream hypervisors in Section 2.5.

2.1 Shared Memory

Shared memory is one of most widespread inter-process communication (IPC) methods[43, 41]. The main reason for its popularity is the performance advantage in comparison to other message based IPC mechanisms such as pipes or message queues, which are implemented on top of system calls.

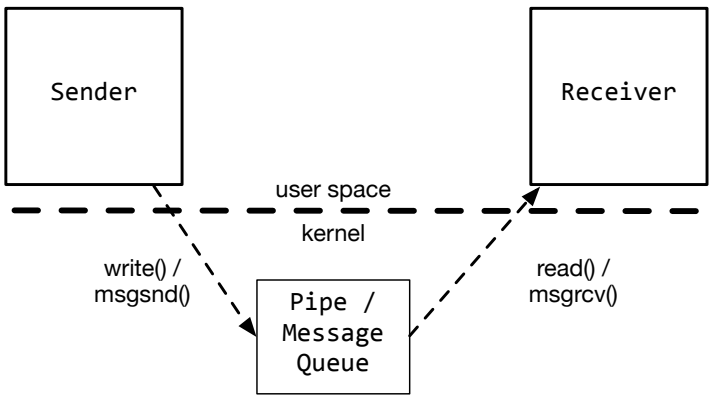

Fig. 2.1: Memory copies during IPC.

As described in [41] and visualized in Figure 2.1, passing data between two processes using a message oriented approach requires at least two additional copies: The sender triggers a copy from user space to kernel, while the receiving side needs to copy into the other direction from kernel back into the user space process.

For shared memory IPC there is no such overhead. Instead, there is a one time setup cost when the shared memory section is created. While the exact APIs to initialize differ between operating systems or hypervisors, the implementation is always the same: One or more physical memory pages are shared by mapping

them into the virtual address space of multiple execution contexts. When talking about operating systems, an execution context normally just corresponds to another user space process, but the mechanism stays the same when talking about different virtual machines. After this page mapping is created, data transfers between two contexts do not require any involvement of the kernel (or hypervisor). Instead, simple memory reads and writes can be used, reducing the need for expensive copy operations. Depending on the exact use case, zero copy protocols are possible, which have very good performance characteristics.

Some kind of synchronization method between the communication partners is required when shared memory is used. To do this, all standard synchronization techniques such mutexes, locks and semaphores can be used on top of shared memory[41]. However, there is an important limitation to note: These synchronization methods require all communication partners to participate, they cannot enforce it. No widespread shared memory APIs include functionality comparable to a mandatory file lock, which is enforced by the underlying layer. This is normally not a problem when all communication participants operate on the same privilege level. While a misbehaving side could interrupt the communication, this cannot be considered a security issue. If, however, the shared memory interface is a trust boundary and one side has less privileges, such issues can become much more interesting from a security perspective. Even though there is a large amount of research concerning the safe use of shared resources, they concentrate on insecure behavior triggered by incorrect use of synchronization primitives. A recent example is ThreadSanitizer[37], an instrumentation based data race detector for C and C++ software. However, this research is only partially applicable, because it does not take the existence of a malicious communication partner into account. High-level synchronization methods are not enforced in shared memory interfaces, which means they can simply be ignored, triggering potential vulnerabilities.

One example for such a vulnerability type is called double fetch, which will be introduced in the following.

2.2 Double Fetches

Double fetches are a special type of *Time-of-Check-to-Time-of-Use (TOCTTOU)* bugs[20]. TOCTTOU bugs exist when data can be manipulated between verification or validation - the time of check - and the time of use.

The probably best known examples of TOCTTOU bugs affect file system accesses[6]: A privileged process, for example a *setuid* binary, checks that a file is owned by an unprivileged user and then performs a modification to this file on behalf of the user. If the permission check and the modification are separate actions, an attacker can replace the file with a symbolic link to a system file. If the timing is right

and this replacement happens right after the check is performed but before the actual modification happens, unauthorized manipulation of important files might be possible.

While, TOCTTOU bugs exist in different software layers and in different environments, the core principle is always the same. A description of this bug class can be found in [1] published in 1976:

> "Whenever there is a "timing window" between the time the control program verifies a parameter and the time it retrieves the parameter from shared storage for use, a potential security flaw is created. This is because contemporary operating systems allow a user to have two or more activities (processes) executing concurrently."

We use the term *double fetch* to describe potential TOCTTOU vulnerabilities where the shared medium is a shared memory region. This terminology was introduced by Fermin J. Serna in a post on the Microsoft Security and Defense blog[28]. One of the main inspirations for this work is *Bochspwn*[20], a Bochs based toolkit to discover double fetch vulnerabilities in the Windows kernel. While Bochspwn uses software emulation to generate memory traces and does not target shared memory communication, it introduces several of the core concepts of this thesis. Besides being the first to try to discover double fetch vulnerabilities using memory access tracing, they also introduce the ability to separate tracing and analysis steps. In addition, the possible extension of the approach with more analysis algorithms and by using hardware-assisted virtualization is mentioned even when no details regarding the implementation of these extensions are given.

Most published examples of double fetch vulnerabilities affect the interface between user space and kernel: Listing 1 shows a vulnerability in the *sendmsg* system call handler of the Linux kernel fixed in 2005[10]. In line 5 the `copy_user` macro is invoked to dereference a pointer into user space and copy the value of the `cmsg_len` field into a local variable `umclen`. `umclen` is used to calculate a length for the final data structure, which is allocated using a call to `kmalloc` in line 15. However, before the data is copied into the allocated structure in line 20, `umclen` is again initialized with the value from user space in line 18.

This is a classic example of a double fetch vulnerability. If an attacker is able to win the race condition and exchange the value of `cmsg_len` between the first and the second access, an exploitable heap overflow can be triggered. While this specific bug can be easily identified in the source code, this is not always the case. Listing 2 shows *CVE-2013-1278* first presented in [20]. The vulnerable code pattern was discovered in multiple system call handlers, this specific example is extracted from the `nt!ApphelpCacheLookupEntry` function. `edi` stores a user space pointer and the `ProbeForWrite` function is used to make sure that the pointer at offset 0x18 of

```
1  int cmsghdr_from_user_compat_to_kern(..)
2  {
3    [...]
4    while(ucmsg != NULL) {
5          if(get_user(ucmlen, &ucmsg->cmsg_len))
6                  return -EFAULT;
7          [...]
8          tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
9                 CMSG_ALIGN(sizeof(struct cmsghdr)));
10         kcmlen += tmp;
11         [...]
12   }
13
14   if(kcmlen > stackbuf_size)
15         kcmsg_base = kcmsg = kmalloc(kcmlen, GFP_KERNEL);
16
17   while(ucmsg != NULL) {
18         __get_user(ucmlen, &ucmsg->cmsg_len);
19
20         if(copy_from_user(CMSG_DATA(kcmsg),
21                 CMSG_COMPAT_DATA(ucmsg),
22          (ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg)))))
23 [...]
24 }
```

Listing 1: Double fetch in sendmsg system call. Two calls to `copy_from_user` create a double fetch vulnerability affecting the `umclen` variable.

`edi` is a writable user space address. When the arguments are passed to `memcpy`, this pointer is fetched a second time from user space memory. If the data is exchanged between these two accesses, arbitrary kernel memory can be corrupted. As shown in [20], this can be used for a local privilege escalation attack against vulnerable systems. Because no source code for `nt!ApphelpCacheLookupEntry` is publicly available, it cannot be evaluated if the double fetch is the result of two C pointer dereferences or of a compiler optimization.

The exploitability of double fetch vulnerabilities is discussed in detail in [20]. On single core systems, races might not be winnable under all circumstances if a context switch never occurs between the time of check and the time of use. However, for multi core systems even very short race conditions can be exploited as long as a loss does not trigger a system crash or a similar irreversible condition. Because modern virtualization environments are always operating in a multi core environment, we consider even short race conditions as exploitable for the purpose of this thesis.

```
1  mov     ecx, [edi+18h]
2  ;[..]
3  push    4
4  push    eax
5  push    ecx
6  call    _ProbeForWrite
7  push    dword ptr [esi+20h]
8  push    dword ptr [esi+24h]
9  push    dword ptr [edi+18h]
10 call    _memcpy
```

Listing 2: Double fetch in nt!ApphelpCacheLookupEntry. An invalid value can be written to `edi+0x18`, between the call to `ProbeForWrite` and the second memory fetch in line 9.

2.3 x64 Virtualization

A core topic of this thesis is virtualization on the Intel 64bit (x64) architecture. The main evaluation targets are the inter-domain communication mechanisms of popular hypervisors and the proposed and implemented solution heavily relies on hardware-assisted virtualization. Therefore, this section introduces the core challenges of virtualization on Intel systems and discusses the hardware virtualization features added in recent processor generations. In order to concentrate on mechanisms relevant for this thesis, several topics such as interrupt virtualization and System Management Mode are ignored in the following.

2.3.1 Virtualization Fundamentals

In a traditional system the operating system has full control over all hardware resources. A virtualized system introduces a new software layer called virtual machine monitor (VMM) or hypervisor. The VMM is responsible for managing access to the hardware for each running virtualized system. Each virtualized system, also called virtual machine (VM), consists of virtual memory, one or more virtual CPUs and virtualized devices. In general, a VMM gives a guest operating system the illusion to be running on real physical hardware. Hypervisor can be separated into type-1 and type-2 hypervisors[43]: type-1 hypervisors run directly on the hardware, while type-2 hypervisors run on top of a normal operating system.

One important requirement in general purpose virtualization is that one VM can not influence the execution of other VMs running on the same physical host. This means virtual memory, CPUs and devices must be isolated from each other and access to privileged operations on the real hardware must be restricted. Privileges on x64 are implemented using a ring model[18]: A processor always operates in a ring between 3 and 0, where ring 0 is the most privileged operation mode. Only code

running in ring 0 has accesses to privileged instructions, the complete memory space and memory mapped or port based IO. Of course normal OS kernels operate under the assumption that they are running in ring 0. However, unrestricted access to all these privileged operations violates the isolation requirement of isolation. There are two practical approaches to solve this problem in software: *binary translation* and *pravirtualization*. Binary translation was pioneered by VMWare[43]. The hypervisor dynamically replaces privileged operations with emulated versions that operate on the virtual hardware. Paravirtualization, first implemented by the Xen hypervisor[3], requires modification of the guest operating system to replace all privileged operations with calls to a hypervisor API. The guest kernel is then moved to a less privileged ring, while the hypervisor is the only code still operating in ring 0. Both approaches are quite successful but they have important downsides. Binary translation does not require modification of the guest operating system and can reach a surprisingly high performance level, but the engineering effort for creating a production ready hypervisor using this approach can not be overestimated. On the other hand, paravirtualization uses the standard hardware protection mechanisms and allows for a very small and simple hypervisor, but requires modification of the guest system. Because of these difficulties with pure software based approaches and the rising demand for virtualization on the x64 architecture, Intel introduced the VT-x extensions[45] in 2005. Nowadays hardware-assisted virtualization using the Intel VT extension or the similar implementation by AMD are by far the most relevant virtualization types in productive use.

2.3.2 Intel VT-x

VT-x adds two additional CPU modes[45]: *VMX non-root* operations and *VMX root* operations. The ring privilege level still exist in both operation modes, so code could be operating in ring 3 in VMX root mode or in ring 0 in non-root mode. The hypervisor runs in root mode, while all guests operate in non-root mode. Context switches between root mode and non-root mode are called *VM entries* and *VM exits*. These transitions and the operation of the processor in non-root mode is managed using a newly introduced data structure called *virtual machine control structure*(VMCS). The VMCS is separated into six logical groups[18]:

Guest-State. Saves the processor state on a VM exit. Is used to restore it on a VM entry.

Host-State. Processor state is loaded from here on a VM exit.

VM execution control fields. These fields control processor behavior when operating in non-root mode.

VM entry control fields. These fields control the VM entry behavior.

VM exit control fields. These fields control the VM exit behavior.

VM exit information fields. These fields contain information about the most recent VM exit.

Management of the VMCS can be performed by using a number of newly introduced instructions that are only available in root mode: They include `VMPTRLD` and `VMPTRST` to load and store pointers to the currently used VMCS. `VMREAD` and `VMWRITE` to read and write VMCS fields and `VMLAUNCH` or `VMRESUME` to trigger a VM entry.

Code executing in VMX root mode behaves the same way as before, but when the CPU is operating in non-root mode, privileged operations can be trapped and handled by the hypervisor. Certain instructions like *WRMSR* or *CPUID* always trigger a VM exit, the behavior of others can be configured using the execution control fields in the VMCS. Interestingly, many privileged instructions do never trigger an VM exit because they transparently operate on VM specific data when executed in non-root mode. This includes all instructions involving interrupt and exception handling[18].

The *trap and emulate* approach enabled by these additions is sufficient to protect the hypervisor and other guests from a misbehaving or malicious virtual machine: All instructions that directly access hardware features can be trapped and emulated safely. Because all accesses to the *CR3* register are intercepted, the hypervisor can enforce a strict separation between its own linear address space and those used by different VMs. In early versions of Intel VT, the hypervisor was required to keep track of the relation between a guest physical and the machine physical address space using a mechanism called *shadow page tables*[39]. When using this approach, a hypervisor is forced to intercept all page faults or page table updates in the VM to keep the shadow page tables in sync with their virtual equivalent. Of course, this triggers a high amount of VM exits, degrading the overall performance. To improve performance, Intel decided to introduce an additional hardware feature called *extended page tables* (EPT).

2.3.3 Intel EPT

Extended page table is Intel's name for a hardware feature also known as *second level address translation* or *nested paging*. EPT introduces the concept of guest-physical address[18]. The guest is in full control of its own page tables and address translation inside the virtual machine works the same as on a non virtualized system. But after the normal address translation has finished, the processor performs an additional translation step going from the guest-physical to the real physical address. As shown in Figure 2.2, EPT translation uses an extended page table pointer (EPTP) stored in the VMCS execution control fields and performs a 4 level deep page-walk through EPT paging structures, very similar to the one performed for normal address translation.

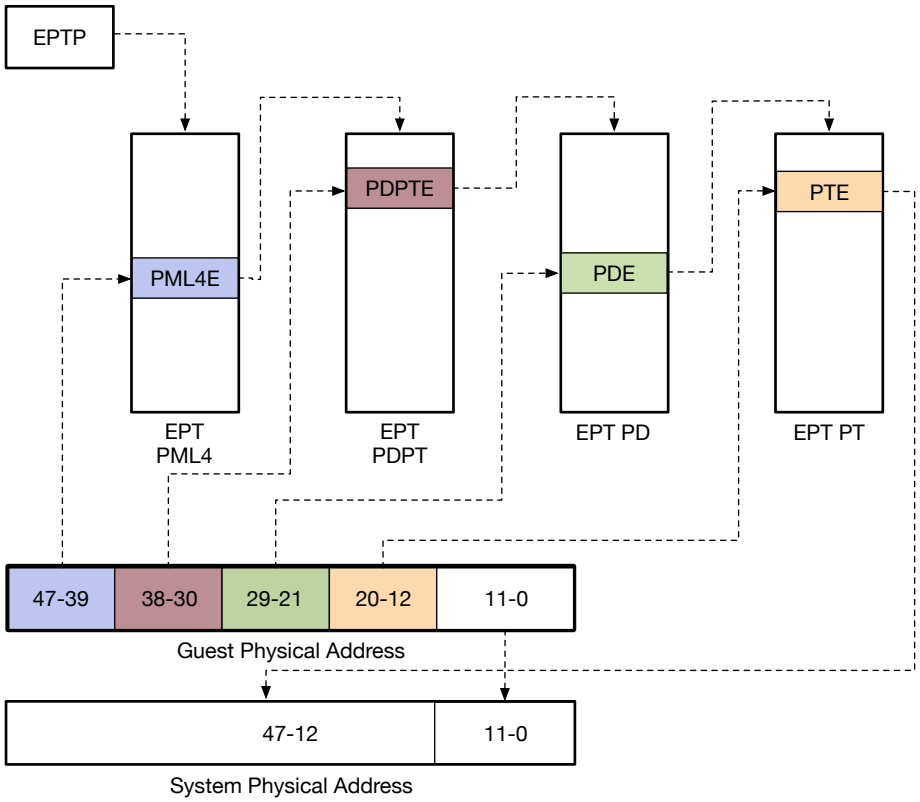

Fig. 2.2: Intel EPT Address Translation. Guest physical addresses are translated into system physical addresses using an additional address translation layer.

The main advantage of EPT is the reduction of VM exits and the offloading of virtualized memory management to the hardware layer. This means the hypervisor code can be significantly simplified and does not have to be concerned with any page table updates performed by the guest. The memory separation is enforced by the hardware as long there is no overlap between the EPT structures used by two virtual machines or the memory pages of the hypervisor itself.

All EPT structures including the EPT page table entry contain fields controlling the access permissions of the referenced physical memory page(s). For example, this can be used by the hypervisor to share a read-only page with his guests. When a VM performs a disallowed access on a guest-physical memory address, an *EPT violation* is triggered leading to a VM exit. This behavior is completely transparent to the virtual machine and can be used for implementing copy-on-write optimizations or to collect data about the behavior of the VM.

2.3.4 Nested Virtualization

Nested virtualization describes the concept of running a hypervisor as a virtual machine on top of another hypervisor. In order to keep the terminology unambitious, we call the outer hypervisor the level 0 (L0) hypervisor and the inner one level 1

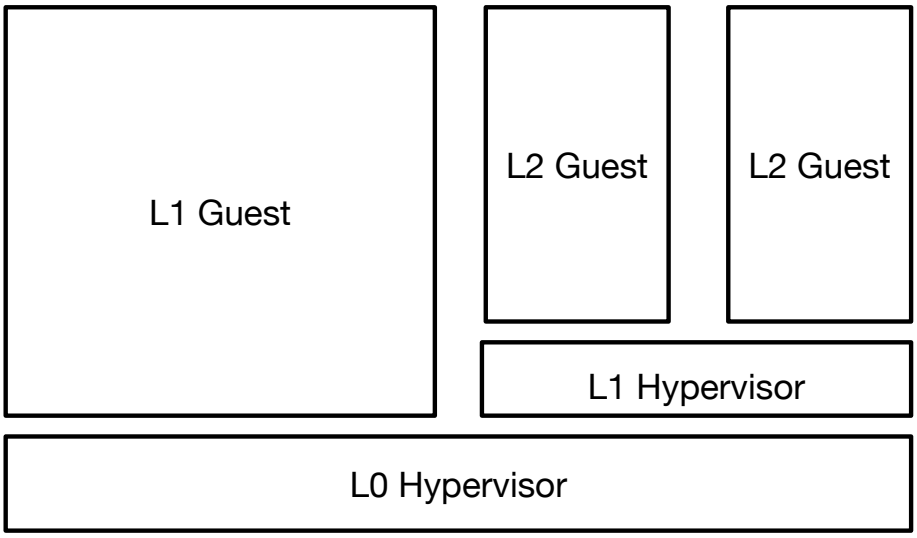

Fig. 2.3: Nested virtualization terminology.

(L1). The L1 hypervisor is just a special type of L1 guest and can run in parallel with other guests and even additional L1 hypervisors. Finally, level 2 (L2) guests run on top of the L1 hypervisor. Figure 2.3 visualizes these connections.

The main use case for nested virtualization is the ability to run a hypervisor in a cloud environment[53]. More recently, Microsoft started to use its Hyper-V hypervisor as a way to isolate security critical components from the normal operating system starting with Windows 10[19]. Because this practically turns the Windows 10 operating system into a Hyper-V VM, support for nested virtualization is required to install additional virtualization software on the system. Currently most mainstream hypervisors only have partial support for nested virtualization, but current development efforts[29, 53] indicate that this will change in the next years.

Mixing two different types of virtualization can often work without any problems. A L0 hypervisor based on Intel VT can host a L1 hypervisor based on binary translation or para-virtualization without any special support. It starts to get more complex when two hypervisors based on Intel VT are nested, which of course is the most relevant use case. The L1 hypervisor operates in non-root mode but stills needs the impression that it is operating in root mode. This means all Intel VT management instructions need to be trapped and emulated by the L0 hypervisor.

Recent extensions of Intel VT try to minimize additional VM exits introduced by nested virtualization as much as possible[53]. For example, *VMCS Shadowing* enables the L1 hypervisor to operate on a shadow VMCS structure without triggering VM exits. Using these features, Intel states a performance loss of only 20% comparing a L1 system to a L2 one[29].

2.4 Virtual Machine Introspection

The concept of Virtual Machine Introspection (VMI) was first introduced in [15] and was defined as an „approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it“. VMI is traditionally used in the context of malware detection and analysis. In this context it has a number of advantages compared to more traditional host based intrusion detection systems (IDS). In a standard host based IDS or sandbox, a software agent is running in the same system as the malware. This requires the agent to rely on the trustworthiness of the operating system, which might be a dangerous assumption if the malware is able to compromise the OS kernel[15, 14]. Furthermore, a hypervisor based inspection can be almost completely hidden from the analyzed system. This means that it is difficult for a malware to simply detect that it is running in a protected or analyzed environment and stop execution[49]. Other features offered by virtualization, like the ability to create and restore snapshots of a running system are also very helpful in the context of malware analysis, making VMI a logical next step.

The hypervisor has complete access to all state of the virtual machine, including CPU registers, memory and the virtual hard drive. This means that at any point in time the current state of the VM can be completely analyzed. In addition, the ability to trap on specific actions of the running malware, is a requirement for efficient analysis. This is quite trivial for software based emulation but more difficult for a hypervisor based on hardware-assisted virtualization. While a very limited form of this trapping could be implemented using software or hardware breakpoints, the authors of [49] describe a more scalable approach by using EPT permissions: By marking specific pages of VM memory as non-executable, the execution of the VM can be traced by analyzing EPT violations. This idea of using EPT permissions as a way to trap on actions performed in the virtual machine is a core concept used in this thesis and will be discussed in-depth in later chapters.

2.5 Hypervisor Architecture

Even though all mainstream hypervisors for the Intel x64 architecture are at least partially based on the Intel VT instruction set and the hardware virtualization support, their overall architecture differs quite strongly. In this chapter the architectures of three of the most popular hypervisors are discussed: Xen, Hyper-V and KVM. These particular hypervisors were chosen for multiple reasons. First of all, all three are widely used and have a mature and feature rich ecosystem. Second, due to the open source nature of KVM and Xen, there architecture is very well documented and implementation details can be easily discovered by reading the available source code. While Hyper-V is a proprietary closed-source hypervisor, the overall architecture is quite similar to the one of Xen. The paravirtualized device drivers used by Hyper-V

are also implemented on top of shared memory[48], making it well suited for this thesis.

In the following discussion, special focus rests on the interfaces used for inter-domain communication as this part of the architecture is the most relevant one for the topic of this thesis.

2.5.1 Xen

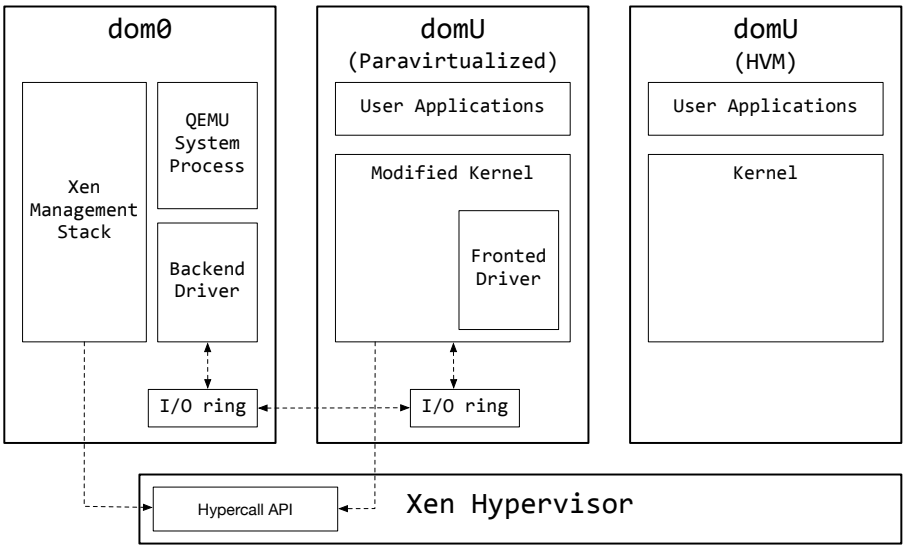

Fig. 2.4: Xen architecture.

Xen[3] is an open source type 1 hypervisor with support for ARM, x86 and x64. Originally a research project at Cambridge University, the first version of the Xen hypervisor was released in 2003. With no Intel VT instruction set available at that point in time, the authors were the first to introduce paravirtualization on the x86 architecture. Instead of software emulation or complex binary translation as performed by other implementations at this time, Xen's paravirtualized virtual machines run modified versions of the guest operating system. The modified kernels do not rely on privileged instructions or direct hardware access and instead communicate with the hypervisor using a set of APIs. Modern versions of Xen also support Intel VT and unmodified guest systems, running as so called hardware virtualized machines (HVM) guests.

Figure 2.4 gives an overview of the Xen architecture and the naming conventions used. The core Xen hypervisor operates directly on top of the hardware and hosts a number of virtual machines called *domains*. The management domain, called `dom0` is a normal linux system running all the management tools required for configuration and operation of the hypervisor and its guests. The management tools communicate with the hypervisor using the *hypercall API*, an interface very similar to the normal system call interface used by operating systems. The decision to put all management

software into a dedicated guest system makes it possible to keep the hypervisor itself relatively simple.

Next to the privileged management domain, two normal unprivileged guests, called `domU` are shown in the Figure. The first `domU` is a paravirtualized guest. It runs a modified guest kernel, that does not interact with the real hardware in any way. Instead, the kernel communicates directly with the hypervisor using the hypercall API. Even though this is the same API that is also used by the management stack, all privileged functionality is restricted to `dom0`, and the `domU` kernel is only allowed to perform actions that affect its own VM.

The paravirtualized guest also requires virtual hardware devices. These are implemented in two parts, the *frontend* and *backend* components: The frontend driver runs in `domU` and plays the role of a normal hardware device driver in the guest OS. When an action is performed on the virtual device, the frontend driver uses a communication mechanism called *XenBus* to send a request to the backend driver operating in `dom0`. Depending on the type of device the backend driver can process the request completely in software or forward it to a real hardware device.

In comparison to paravirtualized guests, HVM domains do not require special support for Xen. CPU and memory are virtualized with the help of Intel VT and EPT, but the domain still needs access to hardware devices. To enable this, Xen uses device emulation offered by the QEMU system emulator[4]. By default each running HVM guest has a corresponding QEMU process running in `dom0`. QEMU emulates old standard devices that are well supported by all mainstream operating systems. Thanks to this, no special drivers are required and a completely unmodified operating system can run in the domain. Still, in practice pure HVM guests are rarely used. Instead of the relatively slow emulated devices offered by QEMU, the HVM guests can use the same frontend drivers as paravirtualized guests. This means that the inter-domain communication between frontend and backend drivers is a potential attack surface irregardless of the domain type, making it particular interesting.

The core mechanism used for inter-domain communication in Xen is shared memory. Sharing memory between two domains is implemented using a data structure called *grant table* and the `grant_table_op` hypercall that operates on it[8]. Using the grant table functionality, two domains can share physical memory pages between each other. This mechanism is used by the paravirtualized drivers to implement I/O rings for performing the actual communication. An I/O ring is a simple ring buffer used for asynchronous communication. The same ring can be used for sending as well as receiving data and a mechanism called *event channel* is used for notification after new data was written into the I/O ring[8]. While the use of I/O rings based on shared memory pages is not a hard requirement for paravirtualized drivers, the protocol has been adopted by all standard Xen drivers. Device drivers that require

large data transfers between domains like block or network devices often implement on demand mapping of shared memory pages for bulk data transfers.

The split driver model used by Xen gives a large amount of freedom regarding the implementation of the backend driver. Depending on performance or security requirements, a backend driver could be implemented as an independent user space process, a QEMU extension or as a Linux kernel module. In some cases this is even configurable by the end user. For example, the backend component of the Xen *blkfront* driver that is responsible for offering virtual block devices to a guest VM can be the `xen-blkback` kernel module, the `xen_disk` implementation of QEMU or one of multiple variants of `blktap`, a user space daemon.

From a security standpoint, the most relevant aspect of the Xen architecture is the privileged role of the management domain `dom0`. Even though it is a virtual machine it has access to the complete state of all other guests and can directly communicate with the hardware. For most environments, this makes a compromise of `dom0` as critical as a compromise of the Xen hypervisor itself. Consequently, attacks on the backend components of paravirtualized drivers are very relevant. Even more so for backend components that are implemented in the kernel, because a vulnerability in one of these can directly lead to a full `dom0` compromise.

2.5.2 Hyper-V

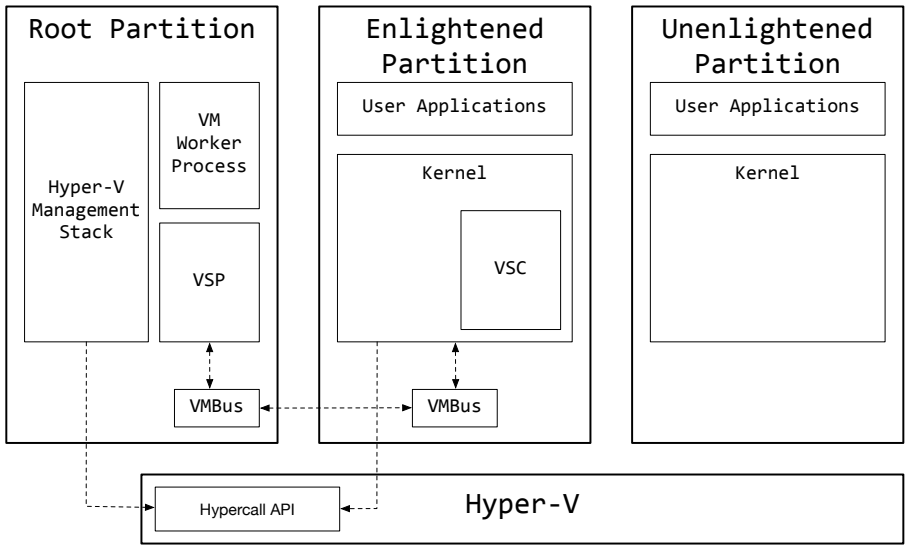

Fig. 2.5: Hyper-V architecture.

Hyper-V is a closed source type-1 hypervisor developed by Microsoft. In contrast to earlier Microsoft virtualization products such as Virtual PC, Hyper-V is completely based on hardware-assisted virtualization with support for Intel VT as well as AMD SVM. Besides being advertised as the main virtualization solution for Windows

servers, Hyper-V is used in the Xbox One console, the Microsoft Azure cloud[48], and as an additional security layer on the client starting with Windows 10[19].

The Hyper-V architecture is strongly inspired by Xen as can be seen in Figure 2.5. Instead of calling the guests domains, they are called *partitions* and the *root partition* has the same role as `dom0`. Accordingly, `domU`'s are called *child partitions*. As in Xen, all management components are running in the root partition, keeping the hypervisor itself as small as possible.

While all partitions use hardware-assisted virtualization for CPU and memory, Hyper-V differentiates between *enlightened* and *unenlightened* partitions, depending on their use of paravirtualized device drivers and the hypercall API. Unenlighted partitions depend on emulated devices and do not know about the hypercall API, while enlightened partitions rely on paravirtualized devices and hypercalls to enable better performance. Instead of using QEMU for device emulation, this functionality is included in the *VM Worker Process* (VMWP). Each running child partition has a worker process assigned, which is heavily restricted using the Windows permission model[48]. The split driver model of Xen for paravirtualized devices is also used by Hyper-V: The backend component is called Virtualization Service Provider (VSP) and the frontend part is the Virtualization Service Client (VSC).

Communication between two partitions occurs with a communication mechanism called VMBus and *guest physical address descriptor lists* (GPADL) used for data transfer. The VMBus interface implements a ring buffer similar to the I/O rings used by Xen. Large data transfers are implemented by mapping the guest pages into the address space of the root partition.

In summary, the Hyper-V architecture is more or less identical with the one used by Xen. Fully paravirtualized domains are not available, but other than that each Xen component has a corresponding replacement in Hyper-V. Consequently, the same security properties that were described in the last section also hold true for Hyper-V.

2.5.3 KVM

KVM, which stands for Kernel-based Virtual Machine, is an open-source hypervisor for Linux systems on the x86 architecture[22]. KVM requires support for hardware-assisted virtualization and supports both the Intel VT and AMD SVM extensions. In comparison to the textbook design of Xen and Hyper-V, KVM is deeply integrated into the Linux kernel leading to a more unconventional architecture as visualized in Figure 2.6. It consists of a Linux kernel module (kvm.ko) that adds virtualization capabilities to a Linux system. While this deep integration with Linux makes the architecture less clean than the previous two examples, it has a number of advantages[21]: First of all, large parts of the kernel code can be reused to implement the hypervisor

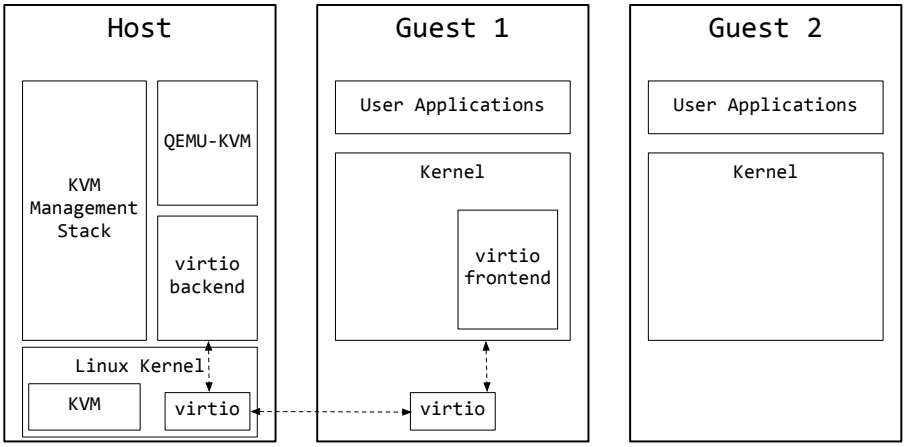

Fig. 2.6: KVM architecture.

functionality. This includes scheduling, memory and power management. In addition, communication involving a guest VM, the host VM, and the hypervisor only requires a single full context switch, because host and hypervisor share a single address space. This can give a better performance than the completely isolated address space of the Xen and Hyper-V hypervisors.

KVM also depends on QEMU for device emulation, similar to Xen. However, the integration between QEMU and KVM goes much further: The complete physical address space of each guest is mapped into its corresponding QEMU process. This makes KVM virtual machines look similar to a normal user space process and allows for easy enforcement of memory limits and swapping[21].

Paravirtualized drivers are implemented on top of the *virtio* mechanism. Virtio is designed to be a hypervisor independent standard for the implementation of paravirtualized devices[46]. The virtio specification describes how device initialization, teardown and configuration of virtual devices are performed and defines the *virtqueue* structure as the main way to transfer data between frontend and backend components. Again, the *virtqueue* is implemented on top of shared memory. Because the guest memory is mapped into the QEMU process, no special way of mapping guest pages is required. Instead, the host can simply access the queue memory using the mapping provided by the QEMU process.

While the exact implementation of the virtio mechanism and the general architecture of KVM differ quite a bit from Xen and Hyper-V, the attack surface and security impact of virtio backend components is identical to the one of the other presented implementations.

2.5.4 Summary

In summary, all of the three presented hypervisors have support for paravirtualized device drivers. All implementations operate with a split driver model, where a back-

end component is running in the management system while a frontend component is executing in the virtual machine. Most importantly, the communication between these two components always involves shared memory pages, making them an apt evaluation target for this thesis. The security boundary enabled by the backend components is well known by the hypervisors' developers. All three discussed implementations offer ways to restrict the privileges of backend components to reduce the impact of a vulnerability: Hyper-V uses the Windows permission model to restrict the worker process responsible for implementing user space backend drivers. KVM uses SELinux for the same purpose and Xen has the ability to move the QEMU process to a single purpose stub domain with restricted privileges. Still, for performance reason many backend components are directly implemented in the kernel of the management system, making full isolation impossible.

3 Analysis

Shared memory, meaning memory pages simultaneously accessible from two different execution contexts, is a core mechanism used for local inter process communication. Data transfers over shared memory pages do not suffer from any significant overhead. In addition, arbitrary complex data structures can be exchanged without the need for serialization. In some cases, the two sides communicating over shared memory have different privileges, making the interface a potential target for attacks. Examples for this situation include the communication between user-space software and the kernel, and sandbox implementations of modern web browsers like Google Chrome [44].

This thesis concentrates on shared memory communication in the context of system virtualization: As discussed in Section 2.5, all mainstream hypervisor use shared memory for high performance inter-domain communication. Most prevalent use cases for virtualization have high security requirements. In many cases, some of the virtual machines running on a physical host have to be considered malicious. This could be because non-trusted consumers operate them like in a public cloud system, the VM is used for malware analysis or simply because the applications running inside the virtual system have a large external attack surface. Of course, this makes the inter-domain communication interface a trust boundary and a particularly interesting attack surface to analyze.

The goal of this thesis is the identification and implementation of an approach for efficient vulnerability discovery in shared memory interfaces with a special focus on inter-domain communication. In the following sections, different approaches to discover vulnerabilities in these interfaces are compared. Following this, the requirements of the memory tracing based approach chosen for this thesis and its suitability for finding different vulnerability types are discussed.

3.1 Security of Inter-domain Communication

The discussion of hypervisor architectures in Section 2.5 already introduced the concept of inter-domain communication: Besides offering a way to communicate directly with the hypervisor, all discussed solutions also have a way to enable direct communication between different virtual machines. These mechanisms are used for the implementation of paravirtualized devices. In contrast to the traditional emulation approach discussed in the last chapter, paravirtualized devices require the installation of special drivers in the virtual machine. However, they compensate

for this by offering a bigger feature set and much higher performance. For example, [27] demonstrates a bandwidth improvement of more than 50% when comparing a paravirtualized virtio device to an emulated network device.

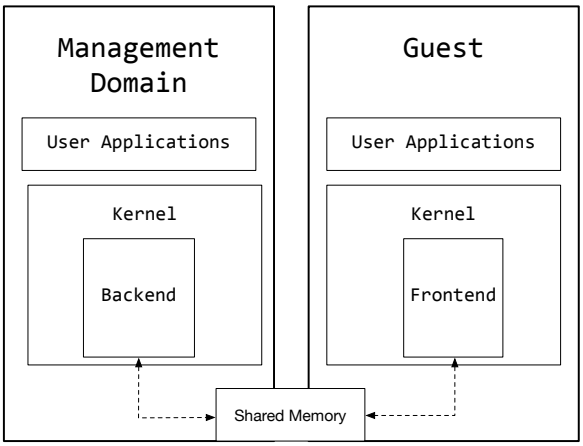

Fig. 3.1: Paravirtualized device architecture. The implementation is split into two components: A backend running in the management domain and a frontend running in the guest.

Paravirtualized devices are implemented using two components as shown in Figure 3.1:

1. A *backend* driver in the management domain is responsible for translating virtualized requests like disk writes or network packets to actual actions. In some cases this can be a simple as forwarding a buffer to the real hardware devices, in others the resulting logic might be completely implemented in software. Backend drivers can run in both user and kernel space.
2. A *frontend* driver in the guest plays the role of a normal device device driver. Instead of communicating with actual hardware, requests send to the driver are instead relayed to the backend driver using a shared memory interface.

Of these two main components, the backend driver is the security critical one. Vulnerabilities in the backend driver that can be triggered from the frontend can allow a malicious virtual machine to influence the execution of the management domain. Depending on the vulnerability and the design of the backend the impact of such vulnerabilities can range from information leaks over denial of service to a complete compromise of the management domain. As our discussion in Chapter 2.5 demonstrates, full access to the management domain is practically equivalent to a full compromise of the hypervisor. Due to their low-level nature, backend drivers are generally implemented in C or C++ making them prime targets for classic vulnerabilities like buffer overflows, out-of-bounds accesses and integer overflows. Examples for such vulnerabilities in backend drivers are *CVE-2011-1750* [11], a heap-based buffer overflow in the disk backend driver of KVM and *CVE-2015-2361* [12], a unspecified buffer overflow in the Hyper-V storage backend. Because

the communication between the two components needs to be as fast as possible, shared memory regions are used for data transfers. This means that in addition to the classic issues highlighted above, bug classes that are specific to shared memory communication such as double fetches, which were introduced in Section 2.2, have to be kept in mind. However, no such vulnerabilities in paravirtualized devices were published until now, which leads to the impression that the underlying inter-domain interfaces were not heavily audited for this type of vulnerability before.

In summary, inter-domain communication opens a significant attack surface in virtualized environments. From an attackers point of view, the backend driver is not too different from a remote network daemon with the added risk of using shared memory as communication medium. The next section discusses different approaches that can be used to discover vulnerabilities in these interfaces, as well as their advantages and disadvantages. The lack of any public research about double fetch vulnerabilities in inter-domain communication makes them a focus of our thesis.

3.2 Approaches for Vulnerability Discovery

The standard approaches for discovering security vulnerabilities such as manual source code review, static analysis and fuzzing are also applicable to inter-domain communication. In this section the three most popular techniques are evaluated and an alternative approach based on memory access tracing and pattern analysis is presented. Besides evaluating their general advantages and limitations, their suitability to discover double fetch vulnerabilities is a main decision criteria.

3.2.1 Source Code Review

The classic approach for finding vulnerabilities in software is manual source code review. While a skilled auditor can often discover vulnerabilities that are very hard to identify using other techniques, a completely manual approach suffers from several downsides: In-depth source code review is a very time-consuming and slow process. This makes it almost impossible to get full coverage of a large application without a significant resource investment. In addition, software as complex as a virtualization solution includes many different components of which only some have a relevant attack surface. Without an advanced understanding of the overall architecture, even identifying these relevant components can be a difficult process. For example, backend drivers in Xen can be implemented as Linux kernel modules, as QEMU extensions or as independent user-space applications.

Certain types of vulnerabilities are very hard to detect using source code analysis. Wang et al.[47] demonstrate multiple examples of so called *unstable* code that incorrectly depends on undefined behavior of the C language. Because the compiler has a high amount of freedom in the presence of undefined behavior, seemingly valid

security checks can disappear depending on the optimization level used. Without a full understanding of the C language reference, such issues will be missed by most security reviewers. As described in Section 2.2, double fetch vulnerabilities can be introduced by compiler optimization hiding them from an auditor doing pure source code based analysis. Finally, source code might not even be available to a security researcher. Proprietary applications like *Hyper-V* are only available in binary form, making source code review impossible in practice. While a manual security review of the compiled application is possible in theory, the difficulty and time requirements rise significantly in comparison to a source code review.

Keeping these downsides in mind, manual source review is not an ideal first step to identify vulnerabilities in inter-domain communication. The large amount of involved components makes it hard to identify the relevant attack surface manually and some interesting vulnerability types, such as the ones described in [47], are very hard to detect on a source code level. In particular, source code review does not seem to be sufficient to detect double fetch vulnerabilities introduced by compiler optimizations. Still, code review is often needed to gain a better understanding of a vulnerability or to discover more complex vulnerabilities that cannot be triggered by other approaches. The identification of interesting attack surfaces by automated means followed by a complementary source code review seems to be a good approach. The two most prevalent automated techniques are fuzzing and static analysis, which are presented in the next sections.

3.2.2 Static Analysis

An alternative to manual code review is the use of static analysis algorithms. In *Principles of Program Analysis*, the authors characterize program analysis as "static [..] techniques for predicting safe and computable approximations to the set of values or behaviors arising [..] at run-time" [31]. While mainly used by compilers for performing safe optimizations of source code, the same techniques can also be used to discover security vulnerabilities. In theory, static analysis can be performed on either source code or the compiled binary. In practice, the information loss involved in the compilation process and the complexity of binary code makes it hard to perform analysis on large binaries without additional information sources like debugging symbols [40]. Even if source code is available, static analysis of virtualization related code is difficult in comparison to high-level user space applications: For example even parsing the source code of relevant functions, which is a prerequisite for any further analysis, is difficult due to the heavy use of compiler specific extensions or inline assembly [5].

In comparison to a dynamic approach, static analysis can get a much higher code coverage. Because no execution is required, code paths that only trigger under rare circumstances can still be covered. However, even ignoring classic problems such

as the state explosion issues[31], this complete coverage is only possible when all involved components are identified correctly. If the user of the static analysis tool does not know that a certain user space application is part of the attack surface, it will not be analyzed leading to potential false negatives. When using source code based static analysis, vulnerabilities that are introduced by compiler optimizations can also not be discovered.

There are a number of examples for sophisticated and security oriented static analysis tools targeting C software[5, 38]. However, they are either commercial products that are not freely available[5], do not have any available implementation [38] or are not well suited for large software stacks such as hypervisors [9]. In addition, these solutions generally operate on source code, making them unusable for analysis of proprietary software. The development of a static analysis framework specialized for this thesis would require a significant implementation effort. Furthermore such a tool needs a correct model of the language semantics, which is non-trivial for high level C code and much more difficult when low level implementation details like Intel VT are involved.

In summary, static analysis requires correct identification of the involved components and significant implementation effort. Source code based static analysis is not usable for proprietary target systems and can miss vulnerabilities created by compiler optimizations such as double fetches. On the other hand, binary static analysis is still an open research area without significant results for system security. For these reasons, static analysis is not the best approach for this thesis, which makes investigating techniques based on dynamic analysis a logical next step.

3.2.3 Fuzzing

Fuzzing can be defined as a

> "highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities"[32].

The relative simplicity of fuzz testing, the availability of powerful fuzzing tools like *sulley* [42] or the more recent *american fuzzy lop*(AFL) [2] and their surprising efficiency in discovering software vulnerabilities make fuzzing by far the most popular automated vulnerability discovery technique. Fuzzers targeting webbrowsers, javascript engines and multi-media files are responsible for a majority of publicly disclosed bugs in these types of software. Fuzzing is nowadays considered an important part of the software development cycle by vendors such as Microsoft[2, 16].

Fuzzers can be separated into two main categories: *Black-box* fuzzers are not interested in the inner-working of their target and just feed input until it misbehaves

or crashs. In contrast, *white-box* fuzzer try to optimize their coverage of the tested application using various techniques. SAGE[16], a white-box fuzzer developed by Microsoft, uses symbolic execution based on a SMT solver to generate input that triggers as many code paths as possible. Besides the differentiation between black and white-box testing, the method used to generate inputs categorizes fuzzer. Generative fuzzer generate samples based on a specification[32] that describes the structure of valid inputs in a parseable way. The alternative is mutation based fuzzing that works by manipulating a known set of good sample inputs. Both approaches have their advantages, but the lower implementation effort leads to a higher prevalence of mutation based fuzzing. Recently, AFL has shown the high success rate of fuzzing by combining mutation based fuzzing guided by detailed code coverage and has discovered a high number of critical vulnerabilities in a wide range of popular software [2].

These results make it seem like fuzzing is well suited to the problem of discovering vulnerabilities in inter-domain communication. However, there are several important downsides:

Stateful interfaces. The communication between frontend and backend drivers often requires correct initialization and notifications to occur. Without a full understanding of these requirements, a fuzzer will not be able to generate requests that are considered valid. While this problem can be bypassed by making sure the fuzzer behaves like a valid frontend driver, this requires development time for each analyzed interface.

Fragility. The targeted paravirtualized drivers play a critical role in the stability of the virtual machine[26]. Simply sending invalid data to the backend will lead to an invalid state and crash the virtual machine almost immediately. Even worse, if such an invalid state involves the corruption of persistent data, for example when fuzzing a virtualized hard disk, a simple reboot is not sufficient to get back to a valid state. This means that some mechanism for fast restoration of a VM state is a requirement.

Unsuitable for certain vulnerability types. Fuzzer are not the best tool to find race condition vulnerabilities such as double fetches, which where introduced in Section 2.2. To discover such an issue, the fuzzer has to generate multiple suitable requests in a very constrained time-frame and actually trigger the race condition. For short races, this is pretty much impossible.

In summary, fuzzing is a promising approach to vulnerability discovery, but does not seem to be well suited to our objective.

3.2.4 Memory Access Tracing and Pattern Analysis

Memory access tracing is widely used for development, debugging and performance evaluations[34]. In addition, full system traces including memory accesses as well as executed instructions can be used to identify and analyze malicious software

or exploits[13]. Memory access tracing as a technique to discover vulnerabilities was first presented in [20]. As discussed in Section 2.2, the authors use the Bochs CPU emulator to generate traces of all virtual memory addresses accessed by a running virtual machine. They analyze these traces to identify potential double fetch vulnerabilities. As the authors mention, this approach can be generalized to identify other types of vulnerabilities by performing different analysis algorithms on the collected data. A related but not identical approach is the use of execution traces to aid in vulnerability discovery, using dynamic taint analysis or concolic execution as described in [36].

We define *Memory Access Tracing and Pattern Analysis* as a two step technique for discovering vulnerabilities: First, a detailed memory trace is collected during execution of the target application or system. This trace is then processed by one or more analysis algorithms to discover potential vulnerabilities, privileged code working with attacker controlled data or other information that can indicate the existence of a vulnerability. The types of data stored in a memory trace depends on the requirements of the analysis algorithm and limitations introduced by the tracing approach. A useful separation can be created by discerning between algorithms that require access to the actual memory content and those that only need meta data like the accessed address and the accessing instruction. The simplest example for the second type of analysis is an algorithm that extracts all privileged instructions accessing attacker influenced memory address and uses these information to identify the overall attack surface of a complex environment. On the other hand, a trace that contains memory contents could be used to identify address leaks from a privileged to an unprivileged context or the direct use of user controlled pointers. Of course only a small subset of potential vulnerabilities can be directly identified by using pattern analysis. However, the other discussed approaches can profit from insights generated, making the approach more generally useful.

We consider memory access tracing as a suitable approach for this thesis due to two main reasons: A limited implementation effort and the effectiveness in discovering double fetch vulnerabilities. In comparison to the development of a full static analyzer for hypervisor communication, a memory tracing and analysis toolset only requires a moderate implementation effort. Additionally, double fetch vulnerabilities are very well suited for discovery by memory access tracing as demonstrated by [20]. A potential double fetch vulnerability can be detected by searching the trace log for at least two memory fetches from the same address in a single *context*. In comparison, the other vulnerability discovery techniques presented above are less suitable for this vulnerability type: Manual source code analysis does not discover double fetches introduced by compiler optimization, which is also the case for source code based static analysis. As already discussed, fuzzing is not a reliable way to discover race conditions which only leaves static analysis of binary code as a sufficient alternative.

However, statically identifying all references to shared memory regions is non trivial, making memory access tracing a simpler alternative.

In summary, memory access tracing followed by pattern analysis is the most practical approach for discovering double fetch vulnerabilities in the course of this thesis. Still, the goal to trace hypervisor communication adds a number of requirements that need to be kept in mind. The next sections discuss these requirements in depth.

3.3 Requirements for Memory Access Tracing

In general there are plenty of methods we could use to generate memory traces. However, the use case of analyzing inter-domain communication has special requirements that limit the set of suitable approaches, as discussed in the following:

Low-level Communication. A fundamental requirement to use memory access tracing for our purpose is the ability to collect low level communication. Inter-domain communication can involve kernel modules and user space applications in all participating domains. Furthermore, depending on the exact implementation even hypervisor code running in root mode might operate on the exchanged data. This makes approaches like METRIC[24] or PIN tools[23] that are restricted to user space tracing unsuitable.

Versatility. The chosen approach should be usable to analyze different hypervisors. This discards all approaches that require significant patches or modifications to the target software. In particular the existence of source for the target hypervisor should not be a requirement to allow for the analysis of software such as Hyper-V or VMWare ESXi.

Scalability and Performance. While most hypervisors can be configured in a very minimal configuration, the goal to find vulnerabilities with dynamic analysis requires us to execute as much of the existing functionality as possible. This requires that the system can continue to execute with a manageable performance overhead, even when tracing is performed. In addition tracing should not be limited to short time-frames or small data amounts to identify vulnerabilities in time and memory intensive functionality. In general we consider every approach that prevents normal interactive use of the system as unfit.

Configurable. For our use case, only a very small subset of memory accesses is interesting. Every access that does not operate on a shared memory region can be safely ignored. Approaches that allow to only trace accesses to a number of configured memory traces are therefore preferable to an approach that forces indiscriminate processing of all memory accesses

As discussed in the last section, the data collected during memory traces varies based on the requirements of the later analysis step. However certain data is required for

almost all useful analyses. In the following, we list the mandatory data points that need to be collected for each memory access:

Address. The accessed physical memory address. Because different virtual machines will access the same memory address using different virtual addresses, storing the physical address is required for correlation.

Type. The type of access: read, write or execute.

Instruction data. The instruction triggering the memory access. Full access to the instruction bytes is preferable to the storage of only the instruction address, because it allows a complete offline analysis without access to the system memory or binaries.

Size. On x64 memory can accessed with different byte granularity. To correctly identify overlapping accesses and the accessed data we need to store this information in the trace.

Context. Information that describe which virtual machine and which component is responsible for the access. This can be a VM name and a process identifier or a more low level information such as the address of the page directory.

In addition to these required information, approaches that allow the collection of the transferred data are especially interesting. While not required to discover double fetch bugs, several other vulnerability types can be detected when memory data is available. If the chosen approach is able to collect this data, an extension of the developed tool to include such algorithms is feasible for the future.

3.4 Conclusion

This chapter evaluated different approaches to discover vulnerabilities in shared memory interfaces in the context of inter-domain communication. Based upon the discussion of hypervisor architectures presented in the last chapter, the suitability of different analysis methods were compared. Besides having a realistic implementation effort a main decision criteria was the ability to discover double fetch vulnerabilities, which were introduced in Section 2.2. For this reasons, memory accessing tracing followed by pattern analysis was chosen as the approach used for this thesis. Following this decision, the requirements for memory access tracing of inter-domain communication were enumerated. This leads up to the next chapter, where the overall design of our proposed solution is introduced.

4 Design

Based on the analysis performed in Chapter 3 we consider memory access tracing the most promising approach for discovering vulnerabilities in inter-domain communication. In this chapter the proposed design of our toolkit for performing memory access tracing and vulnerability analysis on these communication interfaces is presented. A particular emphasis is laid on the efficient discovery of double fetch vulnerabilities.

In the next Section, two analysis algorithms that operate on memory access traces are highlighted. Based upon their requirements and the general requirements for tracing inter-domain communication presented in Section 3.3, different approaches to full system memory tracing will be compared. This is followed by a description of the proposed design of our memory tracing toolkit and an introduction into the different components involved. The chapter finishes with a walkthrough of the tracing, storage and analysis of a single memory access.

4.1 Analysis Algorithms

Analysis algorithms operate on a collected memory trace. They should not require access to the running target system, which makes it possible to perform the analysis even after the target system is shut down or reconfigured. The algorithms work by iterating over the collected memory access traces and searching for interesting patterns. When needed, additional data like instruction bytes can be passed as input to supplement the analysis. The final output of an analysis algorithm is a human readable representation of results or a machine readable output suitable for processing by other tools.

To validate the approach chosen for this thesis, we propose two analysis algorithms: *attack surface* and *double fetch*. The attack surface algorithm simply iterates through all logged read accesses and maps them to the responsible process or kernel module. The double fetch algorithm tries to identify double fetch vulnerabilities in privileged components. The design of both algorithms is highlighted below.

4.1.1 Attack Surface

The core idea of this analysis is very simple. By identifying all code segments that operate on shared memory regions, the attack surface can be mapped. For the purpose of this thesis, we define attack surface as all code that operates on attacker controlled input. One of the main insights of the vulnerability discovery discussion

in Section 3.2 was the problem of identifying all privileged components that are involved during execution of a virtualized system. While not all of these components will directly operate on shared memory, every component that does is an interesting target for further analysis.

On its own the output of the attack surface analysis does not indicate the existence of vulnerabilities, but it can support other analysis steps such as manual source code analysis. In addition, the results can be used to compare different tracing runs and their code coverage, indicating ways to trigger as much backend code as possible.

4.1.2 Double Fetches

The double fetch algorithm works similar to the one presented in [20]: Two or more read accesses to the same memory address, that are performed in a single privileged *execution context* can indicate the existence of a double fetch vulnerability. While this approach sounds simple, there are a two potential issues that must be addressed: Overlapping reads and the definition of an *execution context*. Overlapping reads can happen due to the different memory access sizes supported by the `x64` architecture. A 4-byte read from the address `0x1008` and a 8-byte read from the address `0x1004` would both access the bytes at `0x1008` to `0x100C`. This means that both the accessed address and the access size needs to be known to perform the double fetch analysis. Otherwise, potential double fetches could be missed when only matching addresses are taken into account, introducing false negatives.

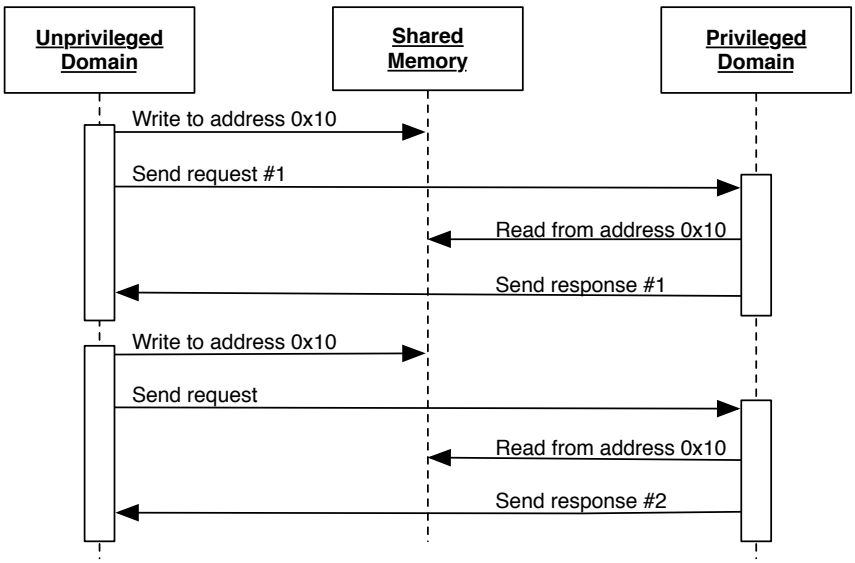

Fig. 4.1: Double fetch: False positive. The reuse of a single shared memory address for multiple requests can mislead a naive double fetch analysis.

A second difficulty is the definition of a single *execution context*. When backend and frontend drivers reuse the same shared memory pages for more than one request,

multiple accesses to the same address will happen sooner or later. However, they do not necessarily indicate a double fetch vulnerability and instead can happen when multiple frontend requests are handled by the same backend function. Figure 4.1 shows an example for such behavior. The two read accesses to the shared memory address `0x10` are triggered by two unique requests and do not have anything to do with each other, but they still access the same memory address triggering a false positive by a naive approach.

The proposed algorithm only considers multiple read accesses when no memory accesses by the unprivileged domain happen in between. This is related to the methodology used by Bochspwn, where only reads that occur during the handling of a single system call are correlated[20]. The described approach removes the mentioned false positives but can theoretically introduce false negatives. An example for this is shown in Figure 4.2: When scheduling stops the execution of the privileged domain right between two read accesses, the unprivileged domain starts to run and performs some kind of unrelated operation on the shared memory page. Because the two read accesses to `0x10` do not seem to happen in a single execution context, they would be missed. However, chances for this behavior are quite low. The risk

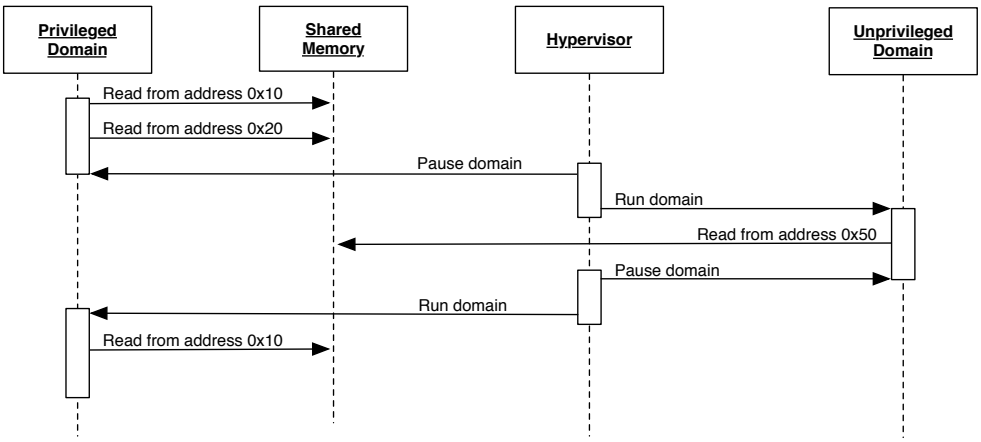

Fig. 4.2: Double fetch: False negative. Scheduler interrupts can lead to context switches that hide double fetch vulnerabilities from the proposed analysis algorithm.

of false negatives becomes acceptable when keeping in mind that the described scheduling must happen every time a vulnerable function is executed. Because tracing is done over longer periods of time, most relevant functions will be triggered multiple times.

Not every discovered double fetch can be assumed to indicate a vulnerability. For example, a function could be repeatedly checking for a mutex, fetch a non security critical value multiple times or perform sufficient validation after every fetch. This means manual analysis is still required. To facilitate this, the double fetch analysis

should print all instructions accessing a memory address, as well as the involved module or process names.

4.2 Approaches for Full System Memory Tracing

The requirement to be able to analyze low level communication, as discussed in Section 3.3, limits the number of approaches suited to our objective. We need the ability to trace memory accesses on all software layers running on the system. Because modification to the target software were ruled out due to the goal of supporting multiple targets, performing some kind of system virtualization is the only way to intercept all memory accesses. In the following three, virtualization approaches are compared: The Bochs x86 CPU emulator used in [20], QEMU used by [13] and similar tools and a hypervisor based on hardware-assisted virtualization.

4.2.1 Bochs

Bochs[25] is a highly portable x86 emulator entirely implemented in software. While most other emulators focus on offering the best performance possible, Bochs' main goal is portability. To support running on as many host architectures as possible, it does not use any advanced hardware features or dynamic recompilation and instead relies on a pure emulation based approach. This makes it possible to run Bochs even on embedded devices with a low amount of available memory.

The Bochs developer take great care to make the emulation as exact as possible, allowing the execution of many different operating systems, including Windows 8 in 32- and 64bit versions. In particular the CPU emulated by Bochs includes hardware virtualization features as discussed in Section 2.3. This means hypervisors such as Hyper-V or Xen can be executed inside a Bochs VM making it a possible target platform for our research.

Tab. 4.1: Tracing requirements: Bochs

Requirement	Bochs
Low-level Communication	✓
Versatility	✓
Scalability and Performance	
Configurable	

Bochs offer a feature rich instrumentation API, which is used by [20] to trace memory accesses. The biggest downside of Bochs is its slow performance in comparison to other approaches. The memory access instrumentation added in [20] further slows down the emulation by a factor of 5. A main reason for this overhead is the fact that every single memory access has to be analyzed by the add-on, because the

instrumentation API does not allow the targeted interception of a small sub set of memory accesses.

Table 4.1 summarizes the advantages and disadvantages of Bochs. Thanks to full system emulation and the capability to emulate Intel VT instructions, Bochs fulfills the first two requirements: Low-level communication can be traced and Bochs supports the emulation of all relevant hypervisors. The slow performance in general and the missing capability of targeted memory interception means the requirements for Performance and Configurability are not satisfied. Still, Bochs seems to be a valid choice if the low performance can be accepted.

4.2.2 QEMU

QEMU is a fast system emulator with support for multiple architectures including x86, ARM and MIPS as emulation targets and host platforms[4]. When emulating x64 code on a x64 host, QEMU can operate in two modes: Software emulation using a dynamic binary translator called *Tiny Code Generator (TCG)* or by using hardware-assisted virtualization with the help of the KVM [22] hypervisor.

TCG operates by dynamically translating blocks of instructions. Privileged instructions are rewritten to safe alternatives as discussed in Section 2.3.1: Privileged instructions are translated into a number of unprivileged ones that operate on the virtual machine state. Because this translation process happens in software, it is possible to add arbitrary instrumentation code that gets executed whenever certain types of instructions are executed. This can be used for memory tracing[34] or execution traces[13] and makes QEMU in TCG mode a popular implementation target for these kind of software. A downside inherent to TCG, is a lower speed in comparison to native or hardware-assisted virtualization. Even though, TCG is much faster than Bochs it still adds a significant overhead. This overhead gets noticeably larger when tracing instrumentation is added as documented in [35] and [13] While the instrumentation capabilities of QEMU are very powerful, they adds a general overhead to each instrumented instruction. For example, an instrumentation of memory accesses can not simply be disabled or enabled for specific memory addresses but will be triggered for every memory access. Of course, this overhead can be partially reduced by keeping the added instrumentation as fast as possible, but this is not trivial.

More importantly, TCG is not suitable for the use case of this thesis due to missing support for modern CPU features: Because of the rising prevalence of hardware virtualization, most of the current development effort for the x64 platform is concentrating on QEMU in combination with KVM. This means that emulation support for modern CPU features is limited in TCG. Initial experiments showed that QEMU/TCG was not able to install a 64bit version of Windows Server 2012, required as a base system for the Hyper-V hypervisor, and that a Xen hypervisor running as a TCG guest

did crash when starting level 2 guests. These first results triggered the decision to not rely on QEMU for this thesis. However, it is important to note that compatibility improvements are regularly added to TCG, making it potentially more suitable in future versions.

Tab. 4.2: Tracing requirements: QEMU

Requirement	QEMU
Low-level Communication	✓
Versatility	
Scalability and Performance	✓
Configurable	

Table 4.2, shows the summarized advantages and disadvantages of QEMU in TCG mode: Whole system emulation and the possibility to add instrumentation code makes it possible to trace low level communication. In addition, the offered performance is sufficient for the described use case. Still, missing support for modern CPU features restricts the systems that can be emulated using TCG and the instrumentation code is executed for each memory access adding a general overhead that can only be partially mitigated.

4.2.3 Hardware-Assisted Virtualization

The final virtualization approach that could be used for this thesis is hardware-assisted virtualization. The core concepts of hardware-assisted virtualization were introduced in Section 2.3: Processors supporting Intel VT add the possibility to run virtual machines natively on the hardware in a special operation mode called *non-root* mode. All unprivileged instructions execute at full speed, whereas privileged operations trigger a VM exit, which can be handled by the hypervisor.

Because only certain privileged instructions trigger a VM exit, hardware-assisted virtualization does not offer as much instrumentation possibilities as the previous two approaches out of the box. Still, memory access tracing is possible using *Extended Page Tables (EPT)*: As described in Section 2.3.3, EPT adds a second layer used during address translation. By restricting the permissions of specific memory pages using EPT entries, each memory access to these pages triggers an EPT violation and a VM exit. The VM exit is handled by the hypervisor which can log it, revert the page permissions for a single instruction and continue execution.

In comparison to the other proposed approaches, this has one important advantage: Memory interception can be enabled and disabled dynamically on a page granularity. This means that all normal system operation can execute natively and only instructions operating on traced memory regions suffer from an overhead due to the EPT violation and corresponding VM exit. While this overhead is quite significant, it only occurs when an application uses the shared memory region. No large passive

overhead is introduced. This EPT based approach is also suitable for a lot of diverse shared memory interfaces in different types of software. The only requirement is the possibility to extract information about the shared memory pages using virtual machine introspection or a software agent running inside the VM.

Tab. 4.3: Tracing requirements: Hardware-assisted virtualization

Requirement	Hardware-Assisted
Low-level Communication	✓
Versatility	✓
Scalability and Performance	✓
Configurable	✓

Still, the use case of tracing inter-domain communication between virtual machines requires support for nested virtualization. The idea of nested virtualization, running a hypervisor inside another one, was presented in Section 2.3.4. Because several major hypervisors include support for nested virtualization, hardware-assisted virtualization fulfills all our proposed requirements as shown in Table 4.3

4.2.4 Comparison

As Table 4.4 shows, memory tracing based on hardware-assisted virtualization is the only approach that fulfills all our requirements. In particular, it allows for configurable tracing, which adds overhead only to accesses to the traced memory accesses while not significantly slowing the rest of the system down. For these reasons, **hardware-assisted virtualization** was chosen as the approach for this thesis.

Tab. 4.4: Tracing requirements: Comparison

Requirement	Bochs	QEMU	Hardware-Assisted
Low-level Communication	✓	✓	✓
Versatility	✓		✓
Scalability and Performance		✓	✓
Configurable			✓

4.3 Proposed Architecture

Figure 4.3 gives a high level overview about the proposed architecture. All involved components are running on top of the level 0 (*L0*) hypervisor. The hypervisor runs two virtual machines: A privileged management domain called `dom0` and a unprivileged domain running a nested hypervisor called *L1*. The *L1* hypervisor is our target system. Because we want to analyze inter-domain communication, the *level 1* hypervisor needs to host at least two *L2* virtual machines: An unprivileged `domU` running frontend drivers for paravirtualized devices and a privileged `dom0` running

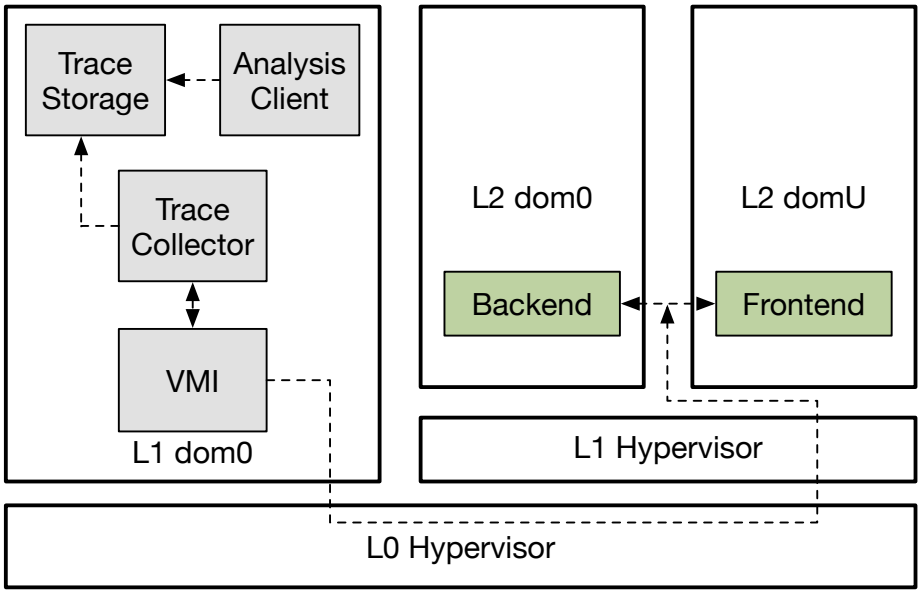

Fig. 4.3: The proposed architecture. All components run in the L1 management domain and communicate with the target system using APIs provided by the L0 hypervisor.

the corresponding backend drivers. The shared memory communication between these drivers can be seen in Figure 4.3 and is the one that needs to be traced and analyzed by our toolkit. The *L1* `dom0` hosts all self developed parts of our toolkit. The *trace collector* is the core component of the proposed design. It needs to interact with the virtual machine introspection (VMI) library to extract information about shared memory ranges from the L1 hypervisor and to enable and disable memory intercepts using EPT permissions of the L0 hypervisor.

When an EPT violation is triggered, the *trace collector* is notified. It extracts all needed access information and stores them into the *trace storage*. After tracing is finished, the *analysis client* can operate on the storage to identify potential malicious traces. In theory, the analysis client does not require access to the VMI component, allowing for a complete offline analysis when the trace stores all needed information. Intercepted EPT violations are completely hidden from the L1 hypervisor. The virtual machine is paused while a memory access is traced.

Due to the low overhead of hardware-assisted virtualization, even for nested environments, all operations that do not involve the traced memory regions can operate at almost native speed. However, traced memory accesses are very expensive in comparison because they will trigger a complete VM exit and multiple context switches. This means the presented architecture is only feasible when the percentage of shared memory accesses is a small part of the overall system activity. Of course, this is the case for inter-domain communication but it makes this approach less fit for tracing all memory accesses of a single process or even of the whole system.

An important detail of the architecture is the fact that the level 1 hypervisor only has a single (virtual) CPU. This might seem surprising due to the fact that Section 2.2 considers multiple cores as a requirement for reliable double fetch exploitation. However, simply identifying these vulnerabilities does not require a multi core system and by restricting the analysis target to a single core the implementation effort is significantly reduced. Otherwise, EPT permissions and access tracing would need to be managed on a per CPU basis while keeping the possibility of rescheduling to different CPU cores in mind.

In the following sections, the requirements for the different involved components will be discussed in more detail.

4.3.1 Hypervisor

For reasons described in Section 4.2, we choose to implement our toolkit on top of a Intel VT based hypervisor. As discussed in the last section, the hypervisor needs to be able to virtualize a second hypervisor, a concept called nested virtualization, as described in Section 2.3.4. Nested virtualization is not in widespread production use and is not supported by all major hypervisors.

The ability to run the L1 hypervisor is not sufficient for our use case, the proposed design requires at least two hypervisor APIs usable by the VMI library: Read access to the memory space and CPU state of the L1 hypervisor, and a way to manipulate its EPT permissions. Furthermore, EPT violations triggered by our modifications should be passed to VMI layer so they can be analyzed and logged by the trace collector. When these APIs are available, no direct modifications to the hypervisor are required. This indicates that even a proprietary hypervisor might be usable in the proposed architecture, as long as nested virtualization is supported and sufficient APIs are available.

4.3.2 Virtual Machine Introspection

The concept of VMI was introduced in Section 2.4. When looking at the requirements of the proposed analysis algorithms and the overall architecture, the requirements for the used VMI library are quite limited:

Memory Access. Read access to the VM memory is required to extract information about the memory pages used for inter-domain communication. Depending on the exact architecture of the target system, the easiest way to find this data might differ but it generally involves identifying and traversing data structures kept in the memory space of the L1 hypervisor or the involved L2 guests. Furthermore, our proposed algorithms profit from access to the instruction bytes because it allows better insight into which operation triggered the EPT violation. While non the proposed algorithms require memory access traces

that include the written content, such a feature would also be implemented on top of this functionality when needed.

CPU State. Read access to the CPU state at the point of the EPT violation. Most relevant is the address of the page table hierarchy base address, which specifies the used page tables, and the current instruction pointer that will point to the instruction accessing the shared memory pages.

Address Translation. As documented in Section 2.3.3, EPT violations are based on guest physical addresses. This means that translation between virtual addresses and physical addresses needs to be performed during trace collection and initialization.

Breakpoints. Breaking on target specific management functions allows an efficient handling of newly added or removed shared memory pages. While, breakpoints can be implemented directly using EPT permissions, direct support by the VMI library is preferable.

All other required features can be implemented on top of these features and the aforementioned hypervisor API for manipulating EPT permissions and handling violations. For example the memory access size, which is needed by the proposed double fetch algorithm, can be extracted out of the disassembled instruction bytes. On the other hand, the name or id of the domain responsible for the memory access can be learned by extracting it out of hypervisor specific data structures stored in memory. While a standalone hypervisor API for manipulation of EPT permissions would be sufficient, a VMI library that already includes EPT events is more convenient because it reduces the coupling to a certain hypervisor version and simplifies the implementation effort.

4.3.3 Trace Collector

The trace collector is the core component of the proposed architecture. It is running as a standard user space process in the management domain of the L0 hypervisor. The collector uses the VMI library to extract information about the shared memory pages out of the L1 hypervisor guests and subsequently removes read and write permissions from these pages using the VMI library or a direct API offered by the hypervisor. When an EPT violation is triggered, the trace collector is responsible for extracting all required state information out of the target VM and storing this data in the trace storage. While the trace collection functionality could be completely implemented in the hypervisor itself, but this would increase the implementation effort significantly because bugs would directly lead to a crash of the L0 hypervisor. In addition, user space libraries can not be directly used from the hypervisor context. By using an API from the privileged `dom0`, all needed functionality can instead be implemented as standard user space utilities.

The trace collector is designed to be as general as possible. The only target specific component that is required by the trace collector is the code that is responsible for identifying the physical addresses of shared memory pages. As we will discuss in the next chapter, the difficulty of this step differs strongly depending on the target architecture. A related functionality is the detection algorithm to decide if a memory access was performed by the privileged level 2 domain or by the unprivileged one. Because only vulnerabilities in the backend driver are a relevant security risk, only memory accesses performed by the backend should be analyzed. This means that some mechanism needs to identify which level 2 VM performed a memory access by analyzing the state of the virtual CPU at the time of the EPT violation. Because most VMI libraries were not developed with the use case of nested virtualization in mind and hypervisors don't expose the state of the simulated VT environment as an API, this is not trivial and requires target specific code. The information which domain performed a memory access, can either be stored inside the memory trace or all memory accesses by unprivileged domains are simply dropped.

The final task of the trace collector is the relaunch of the instruction that triggered the EPT violation. Simply restarting it without relaxing the EPT permissions would result in an endless loop of violations, so single stepping can be used to enable access to the memory address for only a single instruction. This ensures no accesses are missed.

4.3.4 Trace Storage

In Bochspwn[20] all traced accesses are stored in a text file for later analysis. However, the authors note that this simple manner requires large amount of disk space and is limited by the IO performance of the disk backend. In order to minimize the additional overhead introduced by storage, a partially memory backed storage seems preferable. Additional actions like compression and persistent storage should be performed independently and in a different thread than the actual trace entry, so the trace collector can resume the virtual machine as fast as possible, without waiting for these post processing steps to finish.

The tracing storage is the only component to be used by the analysis algorithms. This means it has to store all data required by the algorithms and it should offer a easy to consume library to iterate through trace entries. Furthermore, support to store different data types should be available. In addition to normal trace entries, information about the responsible instruction has to be stored. Storing this information inside the actual trace entries is not optimal, because a single instruction potentially triggers a large number of memory access making this approach inefficient.

An advantage of the proposed architecture is the low coupling of the different components. In particular, the analysis clients only operate with the tracing storage making them completely independent from the trace collector and the VMI interface.

As long as the trace storage offers a standardized API, other methods for memory access tracing can be used with the analysis clients.

4.4 Walkthrough

This section describes an exemplary tracing session with the proposed design, starting with the initial page table parsing over the interception of EPT violations to the final analysis.

1. The target L1 hypervisor is started, which in turn starts execution of the L2 management domain (`dom0`). The L2 `domU` is still stopped and no inter-domain communication can occur.
2. The trace collector is started and uses the VMI interface to identify memory pages that are shared between the L2 `dom0` and other partitions. Because no guest domains are running this will not return any results. The trace collector sets breakpoints in the target hypervisor to get notified when new shared pages are configured.
3. The L2 `domU` is started. When the operating system boots, para-virtualized devices are initialized. This triggers initial hand shakes between `domU` and `dom0` and the configuration of shared memory pages.
4. The breakpoints registered in step 2 are triggered and the trace collector extracts the (L1) physical addresses of the shared pages. It removes read and write access from these pages to trigger an EPT violation whenever they are used.
5. System activity in the L2 `domU` triggers the use of the para-virtualized device. Depending on the device type this might happen automatically or manually, for example by triggering a network connection.
6. The frontend driver in `domU` and the backend driver in `dom0` try to exchange data via shared memory. When the virtual CPU tries to access one of the memory pages an EPT violation is raised and control is transferred to the L0 hypervisor. The L1 hypervisor and all its virtual machines are stopped.
7. The L0 hypervisor notifies the trace collector of the EPT violation. The trace collector uses the VMI library to extract all required information out of the paused VM and stores a trace entry in the tracing storage.
8. By relaxing the EPT permissions, single-stepping over the triggering instruction and removing the permission again, the trace collector makes sure the target system is not triggering the same EPT violation over and over again. Instead execution can continue normally with the next instruction until the next memory access occurs.

9. Steps 5. till 8. repeat until the target system shuts down or the trace collector is closed manually. Step 4 is triggered whenever a new shared memory page is configured.
10. In the final step an analysis algorithm is started to iterate over the trace storage. The output can be used for manual analysis or as an input into other tools.

An important advantage of this design is that step 10 can be executed at any time tracing was finished. As long as the trace storage is not deleted, improvements in the analysis algorithms can be directly tested on already collected data.

4.5 Limitations

There are several limitations that need to be kept in mind when comparing the presented approach to different designs and when evaluating the discovered potential vulnerabilities. These limitations and their impact on the analysis results are discussed in the following.

Tracing overhead. Every access to a monitored memory region triggers at least 2 VM exits, page table modifications, and multiple context switches. The overhead for active tracing is therefore quite large. However, in comparison to regular system activity, inter-domain communication occurs only rarely. Due to that, a high overhead for active tracing is preferable in contrast to a lower permanent overhead introduced by other approaches like software emulation. This makes sense for the presented use case, but might not be the right choice for analyzing shared memory interfaces with a high number of accesses. For example, analyzing kernel-user space communication can be ruled out due to the extremely high number of memory pages involved and the fast rate of context switches.

Single core virtualization. Introducing support for more than one core in the target system would significantly increase the implementation effort as highlighted in Section 4.3. In theory, this can lead to problems when vulnerable code is only executed on multi core systems. For example, a frontend driver could optimize for the number of available cores by choosing a different communication method. Still, we consider the risk for missing vulnerabilities due to this behavior to be acceptable in comparison to the greater implementation effort needed for supporting multiple cores.

Target coverage. Dynamic analysis in general is limited to the code that is actually executed by the target system. If a certain functionality is not used during tracing, no vulnerabilities in it will be discovered. Code coverage can be improved by triggering as much system activity as possible during tracing. However, this is not a bullet proof approach, because some code might only be triggered in special configurations or under unlikely circumstances

Reliance on nested virtualization support. The proposed design relies on working support for nested virtualization. None of the presented hypervisors considers this feature production ready, and bugs and instabilities have to be excepted. While this might have an negative impact on the results of this thesis, better support for nested virtualization will reduce the impact of this limitation in the near future.

4.6 Conclusion

The design of our memory access tracing toolkit is built on top of hardware-assisted virtualization and the use of Intel EPT to dynamically modify page table permissions. By running a target hypervisor as a nested virtual machine and removing access permissions from memory pages used for inter-domain communication, all accesses to these pages can be logged. We use virtual machine introspection library to access VM memory, identify the shared pages and to extract the state of the virtual CPU whenever a memory access is detected. In order to keep the active overhead as low as possible and to allow offline analysis, collected traces are stored in a dedicated trace storage. The two proposed analysis algorithms operate directly on this storage, leading to a largely decoupled architecture that allows for the replacement of most components.

5 Implementation

In this chapter the implementation of the architecture proposed in Chapter 4 is presented. The Xen[3] hypervisor was chosen as the hosting hypervisor using the libvmi[33] library as the interface between hypervisor and trace collector. The Simutrace[34] tracing framework is used as a trace storage, which only required the trace collector and analysis algorithms to be developed from scratch. All used third party components offer a C API, giving us as wide range of possibilities for our implementation language. Due to the ease of integration and high performance requirements C++ was chosen as implementation language.

Thanks to the decoupled design, large parts of the implementation are completely target independent. As discussed in Section 4.3.3, only the trace collector requires target specific code. For this thesis, support for three hypervisors was implemented: Xen, Hyper-V and KVM, with Xen having the most mature implementation. In all cases, the inter-domain communication mechanisms used by paravirtualized devices, which were highlighted in depth in Section 2.5, were targeted. The following section concentrate on the code paths that are target independent, the target specific functionality is documented separately at the end of the chapter.

5.1 Components

The proposed design was split into five main components. Three of those could be implemented by using off-the-shelf components: (a) The L0 hypervisor responsible for hosting the management domain, the target system and offering APIs for introspection and EPT manipulation. (b) The VMI library that sits between the trace collector and hypervisor and (c) the trace storage for persistent and efficient storage of memory traces.

5.1.1 Hypervisor

The Xen hypervisor was chosen as L0 hypervisor for our implementation. For an introduction to the general architecture of the Xen hypervisor see Section 2.5.1. Xen is one of the two mainstream open source hypervisors (the other one being KVM). While being open-source is not a requirement in itself, none of the available commercial hypervisors offers an API that fulfills the requirements detailed in Section 4.3.1. In comparison to KVM, Xen offers a more feature rich API out of the box, including support for EPT based memory interception using the *memaccess* API. All

APIs can be used from user space applications running in the management domain `dom0` removing the need to perform direct modifications to hypervisor code.

Nested virtualization is considered to be a *tech preview* feature not suitable for production use but supported for most configurations. The official Xen wiki[30] lists Xen itself, KVM, Vmware and Hyper-V as working targets for nested virtualization. While we were not able to replicate all of these results during implementation, the main evaluation requirement of running Xen on Xen is well supported.

Most of the development was performed on Xen version 4.5, the current stable version at the time of writing. However, API calls to Xen are wrapped using the libvmi library for introspection, which offers a stable API, supporting all recent Xen versions and hides Xen API changes from our toolkit. In addition, the libvmi interface is less complex than the direct Xen API, reducing the implementation effort even further.

5.1.2 Virtual Machine Introspection

libvmi is a open source C library for virtual machine introspection (VMI)[33]. It offers a mostly hypervisor independent API to read and write memory of a virtual machine, intercept hardware events and accessing the virtual CPU state. In addition, utility functions that provide easy access to semantic information, such as the list of running processes or a map from CR3 registers to process IDs, are available for Linux and Windows guest systems. libvmi supports the Xen and KVM hypervisors and can also operate on physical memory dumps.

```
addr_t read_ptr(vmi_instance_t vmi, addr_t dtb, addr_t va) {
    auto phys_address = vmi_pagetable_lookup(vmi, dtb, va);
    addr_t value;

    if (vmi_read_64_pa(vmi, phys_address, &value) != VMI_SUCCESS)
    { /*... */}
    return value;
}
```

Listing 3: Using libvmi to extract a pointer out of VM memory.

Listing 3 shows an example of using the libvmi API to extract an 8byte pointer value out of the VM memory: The `read_ptr` function first translates the virtual address `val` into a physical address using the `vmi_pagetable_lookup` function and the address of the used page table structure `dtb`. The `vmi_read_64_pa` function is then used to extract the bytes out of the VM memory and store them in the returned variable `value`. The interesting aspect of this code is that it is entirely implemented in standard user-space C++ code and works with all hypervisors that are supported

by libvmi. This is more preferable than the potential alternative of adding code to the hypervisor itself or interacting with a number of potentially unstable APIs.

The most useful feature of libvmi is its support for Xen's memaccess API. This feature is part of a more general functionality offered by libvmi, its event API. This API can be used to trap on certain register writes, as well as on memory accesses. While trapping on registers is limited to those where a write access triggers an VM exit, memory traps use the Xen memaccess API, which itself is based on EPT permissions.

5.1.3 Trace Storage

Simutrace[34] is used for the storage and retrieval of memory access traces. It is based on a client server architecture that allows for fast and asynchronous writing of trace entries. The client component, which is running as part of the trace collector communicates with the server component using shared memory. The server is responsible for compression and storage of the collected data, reducing the work that needs to be performed by the trace collector.

Simutrace was designed for ease-of-use and has a simple C API that can be easily integrated into both the trace collector and analysis clients. In particular, reading and writing of trace entries uses an almost identical API. A core concept of Simutrace are *streams*. Each stream consists of a number of ordered entries of a single type and streams can be created by the client whenever required. The separation of semantically different trace entries into streams, allows for a number of useful optimizations[34]: Because all entries in a single stream have the same size, unique entries can be directly addressed by offset. Additionally, custom compressions methods optimized for specific trace types can be implemented. This feature helps our implementation to reduce the space requirements of long running traces.

5.2 Trace Collector

The trace collector is responsible for the identification of shared memory pages, the tracing of memory accesses and the subsequent data extraction and communication with the trace storage. It uses libvmi to communicate with the hypervisor and stores the traces using Simutrace.

5.2.1 Identification of Shared Memory Pages

The first task of the trace collector is the identification of shared memory pages used for inter-domain communication. This task could be done completely target independent by walking through the extended page tables of the target system and searching for physical pages that are mapped by different guests. However, this approach is hard to implement correctly and very error prone. For example, as discussed in Section 2.5.3, the KVM hypervisor maps the whole memory of each

of its guest into the address space of the corresponding QEMU process. Simply iterating over the page tables would indicate that all pages of the guest are shared with the host system. Of course, almost none of these pages are ever used for shared memory communication making the general approach unsuitable in the case of KVM. Furthermore, without target specific code all updates to the EPT tables managed by the L1 hypervisor need to be intercepted to make sure they do not create a new shared memory mapping. This would create an large overhead, not acceptable for our use case. For these reasons, our implementation requires target specific code to identify the set shared pages and to intercept all updates to this set.

Regardless of the target hypervisor, the result of this step is an updated set of guest physical pages of the L1 hypervisor memory. Every one of these pages is shared between two virtual machines, which for our case normally means it is shared between the management domain and an unprivileged guest. It is important to note, that these characteristics are not important for the rest of the trace collector implementation. As long as the page set is valid and updated regularly, tracing could also be performed on a page that is shared between two user space processes or used for kernel communication.

5.2.2 Tracing of Memory Accesses

```
1  event_ptr new_memevent(State *s, addr_t paddr,
2                         vmi_memevent_granularity_t granularity,
3                         vmi_mem_access_t access,
4                         event_callback_t callback) {
5      auto event = new vmi_event_t();
6      event->type = VMI_EVENT_MEMORY;
7      event->mem_event.physical_address = paddr;
8      event->mem_event.npages = 1;
9      event->mem_event.granularity = granularity;
10     event->mem_event.in_access = access;
11     event->callback = callback;
12
13     if (vmi_register_event(s->vmi, event) != VMI_SUCCESS)
14      { /*... */}
15
16     return event;
17 }
```

Listing 4: Creating a memory event in libvmi.

The tracing of memory accesses is implemented on top of the event API offered by libvmi. Listing 4 demonstrates how this API can be used to intercept accesses to VM memory. The `event` variable specifies the details about the registered event. This includes the physical memory address that should be trapped, whether the whole

page or only the exact address should trigger an interception and which types of access should be handled. The `callback` function will be called whenever the event is triggered. After `event` is initialized, it is registered using the `vmi_register_event` function.

Even though the libvmi API hides a lot of the underlying complexity from the developer, the underlying implementation uses the EPT based approach outlined in the last chapter. The `vmi_register_event` call triggers the use of Xen's memaccess API to modify the EPT permissions of the physical page corresponding to `paddr`. When an EPT violation on this page is triggered, Xen notifies libvmi, which passes execution to the specified callback function. The generation and storing of a trace entry is then performed inside this callback function.

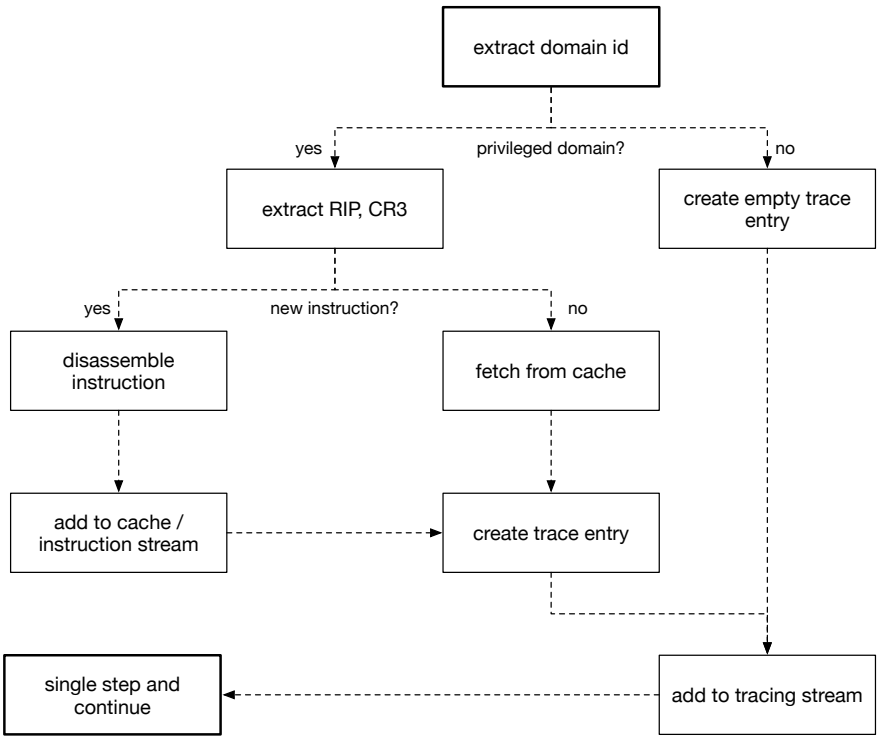

Fig. 5.1: Decision tree for the callback handler function.

The trace collector uses this API by registering a memory event that triggers on read and write accesses for every shared memory page. All these events call back to the `xen_trace_event` function when triggered. Listing 5.1 shows the layout of this callback function. When the callback is executed, the target VM is paused. This makes it possible to access the complete state of the virtual machine, which is used to extract the id of the currently active L2 guest. By knowing the domain id, the code can distinguish between memory accesses performed by the privileged backend and the ones done by the frontend running in an unprivileged domain. When the unprivileged domain performs the memory access, no further data extraction is

performed. Instead, a fake trace entry with all fields set to zero is generated. These fake entries can be later used by the analysis algorithms to detect context switches between unprivileged and privileged domains.

If the privileged domain did perform the memory access, the trace collector needs to collect all information used by the analysis algorithms. The accessed physical memory address and the type of memory access is communicated by the triggered EPT violation and automatically provided to the callback function. In addition, the virtual `RIP` and `CR3` register values are extracted using libvmi. Using these information, the bytes of the accessing instruction can be fetched from VM memory. As discussed in Section 4.1.2, it is important to store the size of a memory access to perform a precise double fetch analysis. This information is not included in an EPT violation and needs to be extracted out of the instruction properties. To do this, the Capstone[7] disassembly library is used. Capstone is a multi-architecture disassembly library with a powerful and easy to use C API. By using Capstone to disassemble the instruction, its operand sizes and therefore the size of the memory access can be learned easily. Fetching and disassembling instruction is relatively expensive in comparison to the other performed actions. Initial evaluation showed that almost all memory accesses are performed by only a small set of instructions, with an even smaller subset of instructions accessing shared memory hundred of times during even short traces. To reduce the overhead of superfluous fetching and disassembling, a caching layer was introduced. In addition, the instruction bytes itself are not stored directly in the trace entries but are instead stored in a specialized instruction stream, which only uses a single entry for each unique instruction.

After all necessary data is fetched from the caching layer or the instruction itself, a trace entry is created, which is then written to a dedicated tracing stream provided by Simutrace. If the callback function would simply return after this, without modifying the EPT permissions, the target VM would be stuck in an endless loop triggering an EPT violation over and over again. Instead, the EPT permissions responsible for the violation are relaxed temporarily and a single step is triggered in the target VM. After this, EPT permissions are restricted again. This approach ensures that no memory accesses are missed.

5.2.3 Trace Entries

As discussed in the last section, the presented implementation uses two separate Simutrace streams to store memory access traces and instruction data. The first stream is responsible for storing the actual memory access trace. To do this, it uses the data type visualized in Figure 5.2, which is provided by Simutrace. By using this pre-defined data type, SimuTrace is able to use an optimized compression algorithm specialized on memory traces. This leads to an improvement in the compression

ratio and reduced space requirements during long tracing sessions, as well as a faster compression speed. The following data fields are stored in the trace:

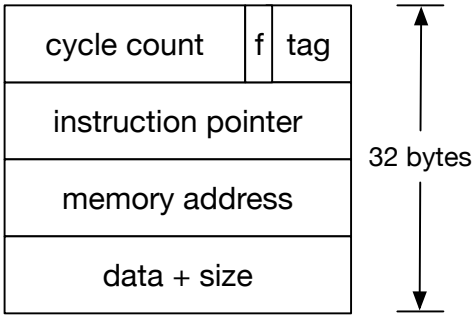

Fig. 5.2: Layout of a memory trace entry.

Cycle count. A 48 bit steadily increasing time value. This can be used to correlate events stored in different streams, but this is currently not required by the implementation. Therefore, the trace collector just stores an incrementing value in this field.

Full size flag. A 1 bit flag to indicate whether the memory access size is 64bit. This is required for correct parsing of the combined data/size field at the end of the entry.

Tag. A 15 bit value for storing arbitrary data which is not interpreted by Simutrace. The trace collector uses this field to store whether a memory access was a read or write.

Instruction pointer. The address of the instruction that performed the memory access.

Memory Address. The accessed virtual memory address.

Data and Size. Simutrace uses a single 64bit field for storing the access size as well as optional memory contents. A 64bit access can be indicated by using the full size flag. Smaller accesses use the last 32bits of the field to encode the access size and the first 32bits to store the data content.

Because the implemented analysis algorithms do not require access to memory contents, the trace collector simply zeroes the data field of every trace entry. While this makes the entry type more complex than needed, it allows the simple addition of memory content when required. If a new analysis algorithm would require access to the memory content, all existing algorithms could still be used without being rewritten to support a new format. Furthermore, compression makes the storage cost of the addition field negligible.

In addition to this memory access stream, a second stream is used to store semantic information about the instructions that triggered a memory access. Because a single instruction can be executed hundred of times during a single tracing session, there

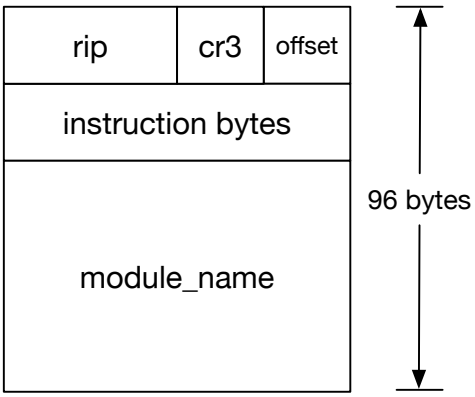

Fig. 5.3: Layout of an instruction trace entry.

is no one to one mapping between instructions and memory accesses. This means storing the instruction data inside the previously discussed entry type would be extremely inefficient. This second stream stores entries of the format shown in Figure 5.3. Besides including the virtual `RIP` and `CR3` registers, the raw instruction bytes are stored. In addition, the human readable name of the kernel driver containing the instruction, and the instruction offset relative to the driver start address is added when possible. This data is later used by the analysis algorithms to ease manual analysis.

Even though the described usage of Simutrace is quite simple, it is sufficient for our normal use case of tracing the communications between two virtual machines. In theory, dedicated streams could be used for different shared memory pages or paravirtualized devices. However, this added complexity does not have any clear benefits as long as the size of the main stream does not get too large. On the other hand, adding more streams to store more semantic information might be necessary when implementing additional analysis algorithms. Due to the design of Simutrace this is easily possible, without breaking backwards compatibility.

5.2.4 Attaching and Detaching

An optional feature that proved to be very useful during normal usage is the ability to attach and detach the trace collector at arbitrary times. This allows to only trace memory access during a certain time frame and to update the trace collector without restarting the target virtual machine. Having the ability to safely detach the trace collector is also a useful feature to handle exceptions: A goal of the collector implementation was to not crash the target hypervisor because of premature exits of the trace collector.

To enable this, one important assumption must always hold: All registered memory events need to be deregistered, before the trace collector process exits. Otherwise, a memory access to one of the traced memory pages will trigger a hypervisor intercept,

which however is not able to pass the event further to the trace collector, leading to a hang of the target system. To ensure correct behavior, the trace collector always keeps a list of all currently active memory events in a global state object. The destructor of this object is responsible for deregistering all active events. Enabling interactive attaching and detaching only requires capturing user invoked signals send to the process using the `sigaction` function and letting them trigger a controlled exit. This will automatically call the state destructor, letting the target virtual machine run unrestricted.

5.3 Analysis Algorithms

As previously discussed in Section 4.1, two algorithms were implemented for this thesis: Attack surface and double fetch analysis. Both algorithms only communicate with Simutrace, allowing for full offline analysis even if the target system is not running anymore. This also means that the algorithms are independent of the exact implementation of the trace collector. Switching from an EPT based trace collector to a different approach based on software emulation would not require a rewrite of the analysis components, as long as the same data is collected.

Both implemented analysis algorithms were developed as standalone C++11 tools. They have no external dependencies besides the Simutrace library and communicate with Simutrace using a small wrapper around the default API. The wrapper provides a type-safe lambda based interface to iterate over streams and entries while not performing any superfluous copies.

5.3.1 Attack Surface

The attack surface algorithm is very simple. We consider every function in a backend driver that performs a read access to a shared memory region to be part of the attack surface. This is because all code that operates on attacker controlled data can have vulnerabilities and should be analyzed further.

To identify all instructions working on shared memory, the algorithm iterates of the memory access stream until it finds a read access. Using the stored `RIP` instruction pointer, the corresponding instruction is fetched out of the `instruction_entry` stream and stored in the result set. This process continues until the whole stream is enumerated.

The analysis tool supports two output modes: The first mode lists all discovered instructions in a human readable output format. The second mode outputs in a machine readable format that can be easily imported into other tools. A proof-of-concept script was developed to import this output into a database file used by the IDA[17] disassembler, allowing for efficient manual analysis of closed source backend components.

5.3.2 Double Fetches

The main analysis algorithm implemented for this thesis identifies double fetch vulnerabilities by searching for potentially vulnerable access patterns. An overview about the design of this analysis was already given in Section 4.1.2. Double fetches can be discovered by finding multiple fetches from an address in a privileged single execution context. Context switches, meaning a switch between the privileged and unprivileged domain, are detected by looking for memory access performed by the unprivileged domain. As described in the last section, when the trace collector sees a memory access by the unprivileged domain an empty trace entry will be submitted. The double fetch algorithm uses these artificial events to split the memory trace into chunks that correspond to a single execution context. The algorithm can be further configured in two ways:

Overlapping memory accesses. Depending on a configuration flag, the algorithm identifies only multiple accesses to a memory address with an identical start address, or also considers overlapping accesses with potentially different sizes to be a sign for a double fetch vulnerability. Disallowing overlapping memory accesses can be used to reduce the number of false positives, while at the same time increasing the chance to miss a potential vulnerability. Only considering accesses with the same start address reduces the noise level, because copy operations that operate on blocks of data are filtered in most cases. Of course, this setting raises the risk of false negatives.

Interweaved read and writes. Until now, our discussion of double fetch issues mostly ignored the handling of privileged writes to the same address. Interweaved reads and writes of a memory access often indicate a synchronization primitive or a reuse of a memory area. There are two ways they can be handled: Either they are ignored completely, or they reset the access count back to zero. A reason for the second behavior is the fact that synchronization primitives, such as mutexes, will be repeatedly read and written and might make the analysis algorithm output more noisy. On the other hand, an application could mistakenly use the shared memory region as temporary storage and removing these accesses from the output can therefore lead to false negatives.

Figure 5.4 shows the code flow of the double fetch analysis in its most conservative setting: Interweaved reads and writes are forbidden and only accesses with the same starting address are considered as potential double fetches. The algorithm stores the set of instruction pointers, that accessed a certain address, in a hash map which is initialized to be empty. The code iterates over every trace entry and checks whether it is an empty entry generated by an unprivileged memory access. Privileged accesses are divided into reads and writes. A read triggers the addition of its instruction pointer to the map entry of the accessed address. A write clears the map entry of the address, as long as interweaved writes are forbidden. Otherwise, it is just ignored.

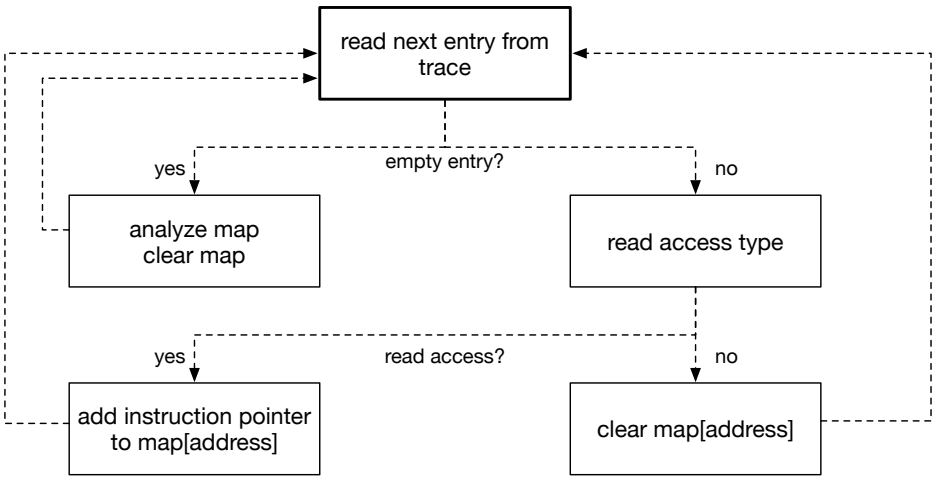

Fig. 5.4: Conservative double fetch analysis. Interweaved reads and writes and overlapping memory accesses are ignored.

Unprivileged accesses indicate that a context switch occurred and trigger analysis of the hash map: Every map entry that consists of more than a single instruction pointer, is added to the list of double fetch candidates. After that the map is cleared again and the analysis continues with the next entry. When the stream ends, the map is analyzed a last time and the list of double fetch candidates is returned.

Before printing this list to the user, entries that occur multiple times are removed. In order to not miss potential interesting variants involving three or more memory accesses, only entries that contain identical set of instruction pointers are considered identical. The discussed configuration settings have a large impact on the number of double fetches discovered, as well as their security relevance. Chapter 6 evaluates the effect of these settings against real world targets.

5.4 Target Specific Code

As discussed in Section 5.2 our implementation requires target specific code in three places:

Identification of shared pages. In order to trace memory access to shared memory pages, these pages need to be discovered first. This step normally requires parsing and traversing of hypervisor data structures and is only feasible if such a global data structure exists. The advantage of finding all shared memory pages at a certain point in time is better support for attaching and detaching of the trace collector. If this approach is not feasible, interception of shared page creation as discussed below can also work without this mechanism.

Interception of shared page updates. Even if the feature to identify all shared pages is implemented, doing so for every context switch would incur an

unacceptable performance overhead. Instead, updates to the set of shared pages, meaning their creation and destruction should be intercepted. This makes it possible to keep a current set of shared pages without performing unnecessary work. If shared pages are not stored globally, this mechanism can also be used as a partial replacement. All pages that are created while the trace collector is attached can be extracted and traced correctly. Of course, this has the down side that shared pages might be missed if the trace collector does not attach to a target system immediately during startup.

Domain identification. The trace collector requires the ability to differentiate between privileged and unprivileged memory accesses. This can be done by identifying the currently active domain in the EPT violation handler. For L2 guests that are virtualized using hardware-assisted virtualization, this information can be extracted by analyzing the currently active VMCS (see 2.3.1). Unfortunately, Xen and libvmi do not provide an easy way to access this data for nested hypervisors. This requires the use of target specific code.

Interestingly, the first two mechanisms have no explicit relationship to inter-domain communication. The same functionality could also be implemented for two user space processes performing shared memory IPC or for user space to kernel communication. The same holds true for the concept of domain identification, which is only used as a mechanism to distinguish between privileged and unprivileged memory accesses. Instead of identifying the domain responsible for the memory access and deciding the handling of the access based on its privileges, the same could be done with process privileges.

Still, the focus of this thesis lies on inter-domain communication, and the following three targets were chosen as evaluation targets: Xen, KVM and Hyper-V. For reasons discussed in the next section, the Xen implementation is by far the most mature one and is the core focus of our evaluation. However, the outlined approaches for the two other hypervisor architectures demonstrate that our general design is not target specific and can be used to search vulnerabilities in different target software.

5.4.1 Xen

Identification of shared pages

Xen's primary mechanism for inter-domain shared memory communication are grant tables, introduced in Section 2.5.1. By using a special hypercall named `grant_table_op`, domains can share their own memory pages with other domains. With this knowledge, the code to extract shared pages and to get notified of possible page changes is quite simple: In the first step, a list of all active domains running in the target hypervisor is extracted by traversing through a global Xen data structure named `domain_list`. For each of these domains, the location of the `grant_table` is

read and all grant entries are processed. While the exact structure of a grant entry is quite complex, the only relevant attribute for our implementation is the guest physical frame number.

Interception of shared page updates

The described mechanism alone is sufficient for finding all shared memory pages at a certain point in time. However, additional grant entries can be created on demand by paravirtualized drivers. In order to get notified of changes to the grant tables, we use libvmi to create a breakpoint at the end of the `grant_table_op` hypercall handler. By breaking at the end, the new grant entries are already inserted into the grant table and can be extracted as described before.

Due to the strict separation of memory spaces in the Xen architecture, all shared memory spaces need to be implemented using the grant table functionality. This ensures that the described approach does not miss any shared pages that are established using other means.

Domain identification

The aforementioned steps work regardless of the virtualization type used for the L2 guest, because both paravirtualized guests and guests using hardware-assisted virtualization rely on grant tables. In contrast the implemented approach for identifying the currently active domain is specific to paravirtualized guests. This is valid, because the management domain `dom0` is always paravirtualized and we can freely choose the virtualization type for the unprivileged guest. Furthermore, several paravirtualized device frontend do not support hardware-assisted virtualization based guests. This makes paravirtualization the logical choice for the domU.

Paravirtualized guests share their address space with the hypervisor, which is globally mapped at the high end of the address space. Every virtual CPU has its own hypervisor stack specified in the MSR register `SYSENTER_ESP`. At the bottom of the stack, a `cpu_info` structure is stored that contains a pointer called `current_vcpu` that points to another management structure describing the state of the virtual CPU. This structure has a pointer to the domain that is currently active in the `domain` field, which in turn contains the domain id. Listing 5 shows how the trace collector extracts this data by reading the `SYSENTER_ESP` and `CR3` registers. After this the described data structures are traversed by repeatedly fetching the memory of the target system.

```
uint16_t get_domid() {
    reg_t rsp, cr3;
    vmi_get_vcpureg(state->vmi, &rsp, SYSENTER_ESP, 0);
    vmi_get_vcpureg(state->vmi, &cr3, CR3, 0);

    const int stack_size = 4096 << 3;

    addr_t current = (rsp & (~(stack_size - 1))) + stack_size - 24;
    addr_t vcpu = vmi::read_ptr(state->vmi, cr3, current);
    addr_t domain = vmi::read_ptr(state->vmi, cr3, vcpu + 16);

    return vmi::read_word(state->vmi, cr3, domain);
}
```

Listing 5: Identification of the currently active Xen domain using management data structures stored by the hypervisor.

5.4.2 KVM

As described in Section 2.5.3, the complete address space of a KVM guest is mapped into its corresponding QEMU process. This means that in theory every guest page can be considered shared. In practice, only a small subset of these pages is accessed by the management domain during the lifetime of the VM and tracing accesses to all pages would introduce an extreme performance overhead. Instead a potential trace collector implementation has to rely on trapping on the creation and destruction of *virtqueue* data structures which are used by virtio drivers. This can be done by intercepting calls to the QEMU virtqueue initialization and destruction functions, and parsing the passed arguments.

Differentiating between the KVM host and unprivileged guests is easy to do in KVM, because the KVM hypervisor is running in the same address space as the rest of the host operating system. This means the privileged host domain can be recognized by simply checking for a running KVM.

5.4.3 Hyper-V

As discussed in Section 2.5.2, the main mechanism used for shared memory communication in Hyper-V are GPADLs. Mapping GPADLs into the address space of a partition requires the partition to perform a hypercall. By intercepting this hypercall shared memory pages can be identified.

Domain identification in Hyper-V can be implemented by identifying a unique and constant physical memory address for all domains. While this requires some manual analysis in the beginning, it allows fast and stable differentiation between the different systems.

6 Evaluation

In this chapter, the presented approach to discover software vulnerabilities in inter-domain communication is evaluated against a real world target. The goals of this evaluation are threefold: (a) Analyze and discuss the performance overhead introduced by the presented implementation. (b) Gain a better understanding of the characteristics of inter-domain communication in Xen and most importantly (c) discover vulnerabilities in the privileged components involved in this communication.

In Section 6.1, the methodology chosen for this evaluation is presented. This is followed by a description of the evaluation setup, including the used hardware, software versions and configuration settings in Section 6.2. Section 6.3 describes the results of our evaluation, including performance numbers, instruction statistics and the results of our attack surface and double fetch analysis algorithms. Following this, two of the more interesting results of our evaluation are discussed in greater depth in Sections 6.4 and 6.5, before the chapter concludes in Section 6.6.

6.1 Methodology

The evaluation is split into two parts. In the first part, benchmarks for CPU, disk and network performance were executed to gain a better understanding of the passive and active overhead of nested virtualization in general and our tracing toolkit in particular. In the second, more important part, the two implemented analysis algorithms are executed on multiple collected traces and the results are analyzed.

As discussed in the last chapters, the following hypervisors were chosen as potential target systems: Xen, KVM and Hyper-V. Unfortunately, evaluation of KVM was heavily restricted due to instabilities of the Xen L0 hypervisor when running L2 guests virtualized by KVM. In the same vein Hyper-V did not start when running as virtualization guest. Even though some time was spent trying to identify and patch bugs in Xen's nested virtualization support, this was not successful. Therefore, our evaluation was only performed against a nested **Xen** hypervisor and its paravirtualized devices.

One of the inherent problems of dynamic analysis is the fact that only code that gets executed can be analyzed for vulnerabilities. This means that as much functionality as possible needs to be used in order to get useful results from the two analysis algorithms. While no reliable automatic way for triggering all functionality of the frontend driver was developed, device activity was triggered manually in several ways: Tracing was active during the boot process and shutdown process. This

means all actions performed during device initialization and destruction were traced. During runtime of the L2 `domU`, the functionality of the device was used as varied as possible. For block devices this includes the reading, writing, creation and deletion of files and directories, whereas a network device was used for network communication using different protocols and traffic patterns. In addition, device configuration was queried and modified when possible. To ensure that the performed activity lead to an acceptable code coverage, the output of the attack surface algorithm was compared to the source code of the backend driver. These comparisons indicated that our approach was successful in reaching a good code coverage.

All discussed performance benchmarks were executed four times with the presented results being the averaged results of the last three runs.

6.2 Evaluation Setup

Our evaluation setup consists of a single physical system running all components of our architecture. Table 6.1 shows the configuration of this system and the version numbers of all relevant components.

Tab. 6.1: Evaluation setup.

Component	Model/Specification
CPU	Intel Xeon E3-1271 v3 @ 3.60GHz
Memory	32GB DDR3-1600
L0 Hypervisor	Xen 4.5.0
L1 dom0 OS	Ubuntu 15.04
L1 dom0 Kernel	3.19.0-18-generic
Simutrace	3.2.2-1
libvmi	Commit eeca74fe..

In theory, the version of Xen used does not have an impact on the implementation of paravirtualized devices. Instead the frontend and backend components are part of the virtualized guests. Still, we have chosen to use two different Xen systems as L1 hypervisors in order to get full support for all supported paravirtualized devices: With version 4.5 Xen removed support for its traditional management stack *xend* and only supports the new *xl* management utility. However, several of the more exotic paravirtualized devices such as SCSI and USB devices are only supported using the older *xend* based management stack.

Tab. 6.2: Target systems.

Component	Xen-Ubuntu	Xen-SLES
L1 Hypervisor	Xen 4.5.0	Xen 4.4.2_08-1.7
L2 dom0 OS	Ubuntu 15.04	SLES 11 SP4
L2 dom0 Kernel	3.19.0-18-generic	3.0.101-63-xen
Management Stack	xl	xend

This means efficient testing requires at least two target systems with different L2 management domains. Table 6.2 shows the configuration of these two target systems: The first called *Xen-Ubuntu* is running the Xen hypervisor in version 4.5 using an Ubuntu 15.04 system as management domain. The second system *Xen-SLES* is running Xen in version 4.4.2, which is one of the last versions with support for *xend*. The management domain is running Suse Linux Enterprise Server in version 11 SP4. SLES was chosen as management domain because of its extensive support for some of the lesser known paravirtualized device types.

A paravirtualized guest in Xen uses a number of paravirtualized devices under normal circumstances. This includes devices required for normal operation such as a block device representing the virtual hard drive, a virtual network interface and a frame buffer. In addition, the following devices were explicitly added to the target systems:

PVUSB. Paravirtualized USB Support enables the passthrough of USB devices to a virtual machine. Xen's implementation is implemented in the `xen-usbback` (backend) and `xen-usbfront` (frontend) kernel modules. To enable testing of these modules, the level 2 `domU` was configured to use a USB device accessible from the L2 `dom0`. Support for paravirtualized USB devices was only available on *Xen-SLES*.

PVSCSI. Paravirtualized SCSI allows the direct use of a SCSI device in a virtual machine. The functionality is implemented in `xen-scsiback` and `xen-scsifront`. Only the older *xend* based management stack has support for this feature making it only available in *Xen-SLES*.

PCI Passthrough. Allows the use of PCI devices in a virtual machine. PCI passthrough is well supported in both management stacks and could be tested on both *Xen-Ubuntu* and *Xen-SLES*.

Disk Backends. Frontend support for paravirtualized block devices is implemented by the `xen-blkfront` kernel module. For the backend, there are multiple options: A kernel based backend called `xen-blkback`, a separate user space daemon named `blktap` and the `xen_disk` backend included in QEMU. All of these backend devices were tested in separate tracing rounds.

6.3 Results

This section describes the results of the performed evaluation. In the first part the performance characteristics of our approach are evaluated by comparing the results of two benchmarks testing CPU and paravirtualized device performance. After this, several data points concerning the characteristics of inter-domain communication in Xen are highlighted. This includes the number of memory accesses performed during our traces, as well as statistics about the types of instructions that operate on

the shared memory regions. The section continues with an analysis of the output of the attack surface algorithm, describing the different components that can be potentially targeted by an attacker. Finally, the results of the double fetch analysis algorithm are presented and the discovered vulnerabilities are discussed.

6.3.1 Performance

Two benchmarks were performed to assess the overhead introduced by our implementation: CPU and memory performance was measured using the `sysbench` benchmark utility. The assumption for this benchmark was that a small overhead is introduced by nested virtualization, but no significant additional overhead should be added when active tracing is performed. The reason is that the benchmark does not directly interact with shared memory pages, so any additional slow down is triggered by background activity of the paravirtualized devices. In addition, the write performance to a paravirtualized device was evaluated by using `dd` to write a 1GB file to a virtual hard drive. Because every data transfer is passed through shared memory, a very large overhead introduced by active tracing was expected. All of the tests were performed on *Xen-Ubuntu* running the previously discussed configuration.

CPU/Memory

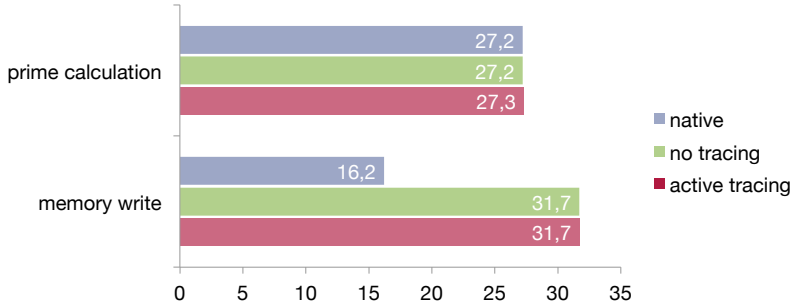

Fig. 6.1: Sysbench CPU and memory benchmarks. Average runtime in seconds.

Figure 6.1 shows the results of the two performed `sysbench` benchmarks. In both cases, native performance was compared to a system running under nested virtualization without active tracing, as well as a nested guest whose inter-domain communication was actively traced. The prime calculation benchmark was performed using `sysbench -num-threads=1 -test=cpu -cpu-max-prime=25000 run`, which involves the repeated calculation of all primes till 25000. As expected, there is no significant overhead introduced by nested virtualization itself or active shared memory tracing.

The memory write benchmark used `sysbench -num-threads=1 -test=memory -memory-total-size=10G run` to calculate the memory performance by writing 10GB of data into memory. In this case, there is a clear overhead introduced by

nested virtualization. Still, the active tracing of shared memory communication does not introduce additional overhead as long as the written data does not touch the watched memory pages.

Paravirtualized Device I/O

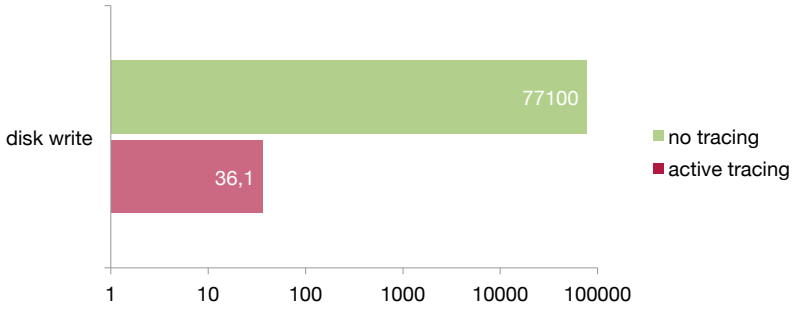

Fig. 6.2: Paravirtualized disk benchmark. Write speed in KB/s.

Figure 6.2 shows the performance of a `dd` write of a 1GB file to a paravirtualized hard disk. Because the complete 1GB file content is transferred over the traced shared memory pages, write speed crawls down to 36 KB/s when active tracing is performed. This shows the high active overhead introduced by our approach and its limitation in tracing heavily used memory segments.

6.3.2 Inter-domain communication characteristics

A dedicated tracing run was performed using the *Xen-Ubuntu* target to gain a better understanding about general characteristics of inter-domain communication in Xen. Ten minutes of simulated system activity was traced, which includes paravirtualized disk activity by searching through the file system, network traffic generated using `curl` and `ping`, as well as interactive shell usage via SSH and the builtin Xen console.

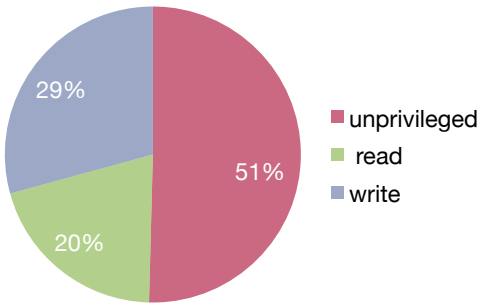

Fig. 6.3: Ratio of different memory accesses to shared memory.

During the trace, about 6.3 million memory accesses were logged. Figure 6.3 shows the ratio of the different memory accesses. Half of the accesses were performed by the unprivileged domain, while two thirds of the privileged memory accesses were writes. The almost exact 1:1 ratio between privileged and unprivileged accesses makes sense when thinking about the way data is transferred over shared memory: It is written by one side and fetched by the other. The higher ratio of privileged writes in comparison to reads can be explained with the performed system activity. Because the performed file search and network download are read heavy activities, the backend needs to transfer more data to the frontend than in the other direction.

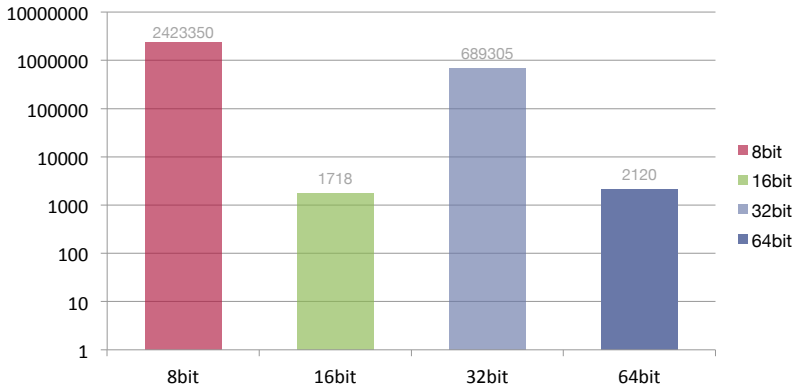

Fig. 6.4: Memory access sizes (logarithmic scale).

Figure 6.4 shows the count of the different access size using a logarithmic scale. Because only privileged memory accesses are logged with these details, unprivileged accesses are not included in this statistic. Surprisingly, more than 77% of all memory accesses have a 8 bit size, with 22% accesses of size 32 bit and only a few 64bit or 16bit accesses. The reason for these statistics becomes clear when looking at the most frequently executed instructions shown in Table 6.3. Nearly all of the single bytes memory accesses, are triggered by a single instruction in the `copy_user_enhanced_fast_string` function, which is a kernel helper function to copy an ASCII string from or to user space memory. Because this function operates one byte at a time, it triggers a high number of memory accesses when copying large strings. The second and third most frequent instructions are both parts of the

Tab. 6.3: Most frequent instructions operating on shared memory.

Hits	Instruction	Function
2420739	`rep movsb byte ptr [rdi],byte ptr [rsi]`	`copy_user_enhanced_fast_string`
630387	`mov esi, dword ptr [r8 + rsi + 0x400]`	`handle_io` (xenconsoled)
14074	`mov ecx, dword ptr [rax + 0xc00]`	`handle_io` (xenconsoled)

xenconsoled daemon responsible for providing a virtual console. The reason for this high ranking is the heavy use of the virtual console during the tracing run. Again

the more frequent instruction is part of a copy loop that moves data between the shared memory page and a private data structure.

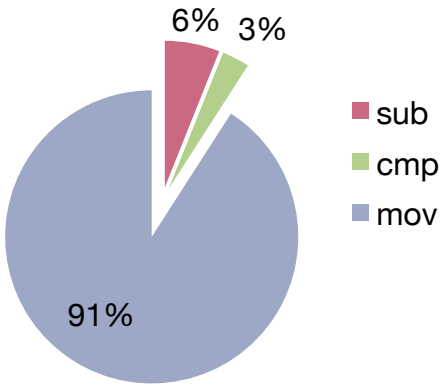

Fig. 6.5: Ratio of instruction opcodes accessing shared memory.

Finally, Figure 6.5 shows the ratio of the different opcodes used to access shared memory. 91% of all unique instructions that operated on shared memory are a variant of the `mov` instruction with 6% being a type of subtraction (`sub`) and 3% comparisons (`cmp`). While the high prevalence of `mov` instructions was expected, the existence of `sub` and more importantly `cmp` instructions are a potential indicator for double fetch problems: A `cmp` operating on shared memory, followed by a `mov` from the same address is a clear indicator for a potential double fetch vulnerability.

In summary, the collected statistics validate our initial assumptions about inter-domain communication. Both frontend and backend operate heavily on the shared memory pages, and while most of the accesses are simple copy operations there are a number of occurrences where more complex operations are directly executed on these shared addresses.

6.3.3 Attack Surface Analysis

The attack surface analysis algorithm was executed on two traces, collected on *Xen-Ubuntu* and *Xen-SLES*. *Xen-SLES* was configured to run a L2 guest using paravirtualized USB and SCSI devices in addition to the default configuration. The *Xen-Ubuntu* L2 guest had access to a paravirtualized PCI device and used two separate paravirtualized hard drives, one corresponding to a raw file and the second to a block device on the management domain. In both cases, tracing was performed over 60 minutes of active system usage.

Xen-Ubuntu

The *Xen-Ubuntu* trace triggered 146 unique instructions accessing shared memory. These instructions were part of the following components:

xen-netback. The `xen-netback` kernel module is responsible for handling network traffic sent and received by our virtual machine over its paravirtualized interface. Even though the backend driver and its corresponding frontend `xen-netfront` communicate using a quite complex and feature rich protocol, the `xen-netback` driver is actively developed and under heavy scrutiny, making it a hard target to find vulnerabilities in.

xen-blkback. The `xen-blkback` kernel module is used for accesses to the paravirtualized hard drive that corresponds to a block device on the management domain. This is in contrast to the paravirtualized hard drive represented by a simple file, which is handled by the QEMU process discussed below. This difference in the responsible backend components is an interesting example to show the use case for the attack surface algorithm: A seemingly trivial configuration change completely replaces a security critical backend component with a different one. The `xen-blkback` code is heavily integrated into the Linux block I/O layer, making in-depth source code review quite difficult. Nevertheless, the code is not as actively developed as the `xen-netback` code and is an interesting target for further analysis.

xenconsoled. The xen console daemon is responsible for providing a virtual console to a paravirtualized guest. The xenconsoled code base is quite small, making a full source code review possible. Still, on the *Xen-Ubuntu* management domain, the daemon is running with full root privileges and without security measures such as position-independent code (PIC). This is an unfortunate lack of hardening for such a security critical component.

xenstored. This daemon provides the *XenStore* service to all domains running on the system. *XenStore* is used an storage space shared between the domains and can be described as an inter-domain key value store[8]. *xenstored* shares the lack of defense in depth mechanisms like PIC with *xenconsoled* but has much larger functionality. This makes it an interesting target for further research.

xen-pciback. The *xen-pciback* kernel module provides the backend for the paravirtualized PCI device running in the guest domain. Support for PCI passthrough is becoming more relevant due to the support for GPU acceleration in popular cloud environments. This makes this functionality a relevant target.

QEMU. While the QEMU system process is mostly for providing access to emulated devices, it also includes a backend component to the *xen-blkfront* frontend driver. As mentioned above, the QEMU backend is used when the paravirtualized disk is represented by a single file in the management domain. Due to the varying quality of QEMU's emulated driver code, the QEMU process is a

traditional target for attacks against Xen[50, 51]. In our evaluation QEMU is running as root on the management domain, but uses position independent code for its own executable, making Address Space Layout Randomization (ASLR) quite effective. In addition, QEMU can be moved into a dedicated stub domain as discussed in [8]. In comparison to the backend components implemented in kernel space and the lesser protected *xenstored* and *xenconsoled* processes, vulnerabilities in QEMU are generally much harder to exploit.

Xen-SLES

As expected, the tracing on *Xen-SLES* had large overlaps with our results for *Xen-Ubuntu*: Only the QEMU disk backend and *xen-pciback* were not executed on this system. Instead the following three new components were discovered:

xen-scsiback. This kernel module is the backend for the paravirtualized SCSI device. With almost 2000 lines of code, this kernel module is one of the more complex backends and is an interesting target for large scale enterprise environments, where the high performance offered by direct SCSI access might be preferred to a more standard approach.

xen-usbback. The *xen-usbback* kernel module offers paravirtualized USB devices to a guest domain. In comparison to the other kernel based backend components, this module is not included in the mainline Linux kernel. This indicates that is is only rarely used in practice and makes it a less interesting research target.

blktap. The blktap kernel module and user space daemon are an alternative block based backend that is used instead of *xen-blkback* or QEMU for guests running on *Xen-SLES*. Again, this shows that small configuration changes can have significant impact on the existing attack surface.

In summary, 9 separate privileged components working on shared memory could be identified using the attack surface algorithm. Due to Xen's open source nature, these components could also be identified manually by reading source code and documentation. However, the same algorithm also works on proprietary hypervisors such as Hyper-V, where a manual analysis would be much more difficult.

6.3.4 Double Fetch Vulnerabilities

The double fetch algorithm was executed on the same traces used for attack surface analysis in the last section. This resulted in 39 potential double fetch issues. In the following, these results are analyzed and discussed.

False Positives

A large percentage of the discovered double fetches can be considered false positives, because they do not indicate any type of security vulnerability or software bug. For

this purpose we define *false positive* as a double fetch that happened but does not cause incorrect behavior. False positives can be again separated into two overlapping classes: The vast majority of false positives are repeated accesses to synchronization variable ssuch as mutexes. The second case are double fetches from variables that always include the necessary security checks after the fetch.

```
;Double Access for ffffc90000af8000
ffffffffc0367016 63f26000 (xen_netback): mov edx, dword ptr [rax]
ffffffffc0367955 63f26000 (xen_netback): mov eax, dword ptr [rdx]
ffffffffc03679c3 63f26000 (xen_netback): mov eax, dword ptr [rax]
ffffffffc0368218 63f26000 (xen_netback): mov eax, dword ptr [rdx]
ffffffffc036adaf 63f26000 (xen_netback): mov eax, dword ptr [rax]
```

Listing 6: Suspected double fetch in xen-netback. The report generated by the double fetch algorithm shows repeated accesses to a single memory address.

Listing 6 shows a false positive reported in the *xen-netback* kernel module. Output from the double fetch analysis always follows the same output format: The first line lists the memory address that was accessed multiple times. After that, the first row lists the virtual address of the instruction that performed the memory access followed by the value of the `CR3` register at that point in time. The third row lists a human readable name of the responsible process or kernel module before the disassembled instruction is printed at the end of the line.

When matching these trace entires to the source code of *xen-netback* it becomes clear that the accesses are triggered by repeatedly querying for new requests on the shared ring buffer. Of course, this does not lead to any kind of security issue.

A second example for a false positive is shown in Listing 7. The double fetch was triggered by the `handle_io` function of the *xenconsoled* process. When looking at the

```
;Double Access for 7fa6f13c7c00:
403d7c 12b3d000 (xenconsoled): mov ecx, dword ptr [rax + 0xc00]
404123 12b3d000 (xenconsoled): mov edx, dword ptr [r14 + 0xc00]
```

Listing 7: Suspected double fetch in *xenconsoled*.

source code of this function it becomes clear that these memory access are triggered by an inlined function whose simplified code is shown in Listing 8. The function reads two values `cons` and `prod` from shared memory and correctly uses a memory barrier to make sure the values are stored into registers. Listing 7 shows the double fetch report for `cons`, while a second almost identical report was generated for `prod`. After both values are stored in a register the unsigned *size* value is calculated and validated against an upper limit. This code is safe, even when executed multiple times. A vulnerability would only exist when one of the later accesses to `out_cons`

or `out_prod` would not include the validation, but this is not the case making the report a false positive.

```
1 cons = intf->out_cons;
2 prod = intf->out_prod;
3 xen_mb();
4
5 size = prod - cons;
6 if ((size == 0) || (size > sizeof(intf->out)))
7         return;
```

Listing 8: Safe size calculation in *xenconsoled*. The unsigned `size` variable is correctly checked against an upper limit.

QEMU xen_disk

One of the more interesting findings returned by the double fetch algorithm affects the block backend implementation in QEMU, also called `xen_disk`. QEMU defines two more or less identical helper functions named `blkif_get_x86_64_req` and `blkif_get_x86_32_req` for parsing and copying frontend requests from shared memory to a private buffer. Listing 9 shows a simplified version of the first function. Knowing that the `src` variable points into shared memory, it is easy to see that the three accesses to the `nr_segments` field in line 7, 12 and 13 are a typical example for a double fetch vulnerability. The two last accesses are the most interesting ones, because they could potentially allow for a controlled heap overflow: The if condition in line 12 tries to enforce that `n` never becomes larger than `BLKIF_MAX_SEGMENTS_PER_REQUEST`, but this could be bypassed by modifying the value of `nr_segments` between the two accesses. This can be used to trigger a heap overflow in the final for loop.

As it turns out, this code is not exploitable in the evaluated system: The reported double fetch lists an access triggered by line 7 and a second one triggered by the if condition in line 12. The assignment operation in line 13 is optimized by the compiler and reuses the already fetched value instead of performing another costly memory operation. Even though this bug does not have any security impact on our target system, this might change if a compiler optimizes the code in a different way. Therefore, this potential vulnerability was reported to the Xen maintainers and is planned to be fixed as part of XSA-155[52]. This result validates our argument from Section 3.2, that source code analysis is not sufficient to reliably identify double fetch vulnerabilities. In this case an analysis based only on source code would rate this vulnerability more critical as it is in most real world environments.

```
1  void blkif_get_x86_64_req(blkif_request_t *dst,
2                            blkif_x86_64_request_t *src)
3  {
4          int i, n = BLKIF_MAX_SEGMENTS_PER_REQUEST;
5
6          dst->operation = src->operation;
7          dst->nr_segments = src->nr_segments;
8          // ...
9          if (src->operation == BLKIF_OP_DISCARD) {
10                 //..
11         }
12         if (n > src->nr_segments)
13                 n = src->nr_segments;
14         for (i = 0; i < n; i++)
15                 dst->seg[i] = src->seg[i];
16 }
```

Listing 9: Double fetch issues in QEMU block backend. `src->nr_segments` is fetched multiple times.

xen-blkback

Another vulnerability was discovered in the `xen-blkback` kernel module. Listing 10 shows parts of the vulnerable function `xen_blkbk_parse_indirect`. In this case the `segments` array is stored in the shared memory region. The if conditions in line 9 and 10 perform validation of the `last_sect` and `first_sect` attributes of the current index. If this validation fails processing of the whole array is stopped. However, both of the validated values are already used before the check and all of these uses are translated into dedicated memory accesses. This means that an attacker can write malicious values into `seg[n].offset` and `seg[n].nsec` and then modify `last_sect` and `first_sect` back to sane values before the check executes. An exact analysis of the impact of this vulnerability is difficult to perform due to the interdependency of this code with the Linux block I/O layer. Still, this vulnerability was reported to the Xen maintainers and is planned to be fixed as part of XSA-155[52].

xen-pciback

The most critical vulnerability discovered during our evaluation affects the backend driver for paravirtualized PCI devices: `xen-pciback`. Listing 11 shows the output generated for this vulnerability by the double fetch algorithm: Two memory accesses to a single address are performed one is a comparison with the constant 5 and the second access is a normal read.

Manual analysis shows that both accesses are part of the `xen_pcibk_do_op` function, which mostly consists of a big switch statement as shown in Listing 12. `op` is stored

```
1  for (n = 0, i = 0; n < nseg; n++) {
2      //...
3      i = n % SEGS_PER_INDIRECT_FRAME;
4      seg[n].nsec = segments[i].last_sect -
5              segments[i].first_sect + 1;
6
7      seg[n].offset = (segments[i].first_sect << 9);
8
9      if ((segments[i].last_sect >= (PAGE_SIZE >> 9)) ||
10         (segments[i].last_sect < segments[i].first_sect)) {
11             rc = -EINVAL;
12             goto unmap;
13     }
14     //...
15 }
```

Listing 10: Double fetch in xen-blkback. The `segments` array is stored in shared memory, making the repeated accesses to `last_sect` and `first_sect` insecure.

```
;Double Access for ffffc90000afa004:
ffffffffc0343fe3 12b3c000 (xen-pciback): cmp dword ptr [r13 + 4], 5
ffffffffc0343ff1 12b3c000 (xen-pciback): mov eax, dword ptr [r13 + 4]
```

Listing 11: Double fetch in *xen-pciback*.

in shared memory, but looking at the source code alone does not show any signs of a double fetch vulnerability.

However, the compiled code highlighted in Listing 13 quickly shows the root cause of this issue: The switch case was compiled into an optimized jump table, which incorrectly accesses the switch condition twice. Line 1 shows the first access to the `op->cmd` variable as discovered by the double fetch analysis. The value is compared to the constant 5 and if it is larger, a jump to the default case of the switch statement is triggered in line 3. If this is not the case, `op->cmd` is fetched from memory a second time and is used as an offset into the jump table in line 5. This is highly problematic, because the second fetch can result in an arbitrary value giving an attacker complete control over the indirect jump target.

This vulnerability was reported to the Xen security team and is planned to be patched as part of XSA-155[52]. The next section gives an introduction about how this vulnerability can be triggered and exploited to achieve arbitrary code execution on the management domain.

```
1  switch (op->cmd) {
2       case XEN_PCI_OP_conf_read:
3               op->err = xen_pcibk_config_read(dev,
4                         op->offset, op->size, &op->value);
5               break;
6       case XEN_PCI_OP_conf_write:
7               //...
8       case XEN_PCI_OP_enable_msi:
9              //...
10      case XEN_PCI_OP_disable_msi:
11             //...
12      case XEN_PCI_OP_enable_msix:
13                 //...
14      case XEN_PCI_OP_disable_msix:
15             //...
16      default:
17              op->err = XEN_PCI_ERR_not_implemented;
18              break;
19 }
```

Listing 12: Vulnerable switch statement in xen-pciback. `op->cmd` is stored in shared memory.

```
1 cmp    DWORD PTR [r13+0x4],0x5
2 mov    DWORD PTR [rbp-0x4c],eax
3 ja     0x3358 <xen_pcibk_do_op+952>
4 mov    eax,DWORD PTR [r13+0x4]
5 jmp    QWORD PTR [rax*8+off_77D0]
```

Listing 13: Assembly of the vulnerable switch statement in xen-pciback. The jump table implementation fetches the case value twice. This allows an attacker to control the jump destination in line 5.

6.4 Notes on Exploiting xen-pciback

The *xen-pciback* double fetch vulnerability discussed in the last section is particularly interesting for multiple reasons: First of all it cannot be easily detected using source code review. Even knowing that the `op->cmd` value is stored in shared memory does not directly lead to the discovery of the vulnerability. In addition, the bug gives an attacker immediately indirect control over the instruction pointer making it highly probable that arbitrary code execution can be achieved. Lastly, the race condition can be triggered as often as needed and does not cause any system instability. If the race is lost, the PCI request will be considered invalid but this should not have any impact on the overall guest system.

Still, the vulnerability has one relevant downside: The time between the two memory accesses is very small, because only two instructions are executed in between. Even though one of them is a potentially slower branching instruction, the time span in which the value has to be manipulated is quite small.

As discussed in Section 2.2, we only consider guests with at least 2 virtual CPUs. Keeping this requirement in mind the first approach to trigger the vulnerability is quite simple: The exploit starts two processes scheduled on different CPU cores which both start executing an infinite loop. The first process is responsible for triggering requests to the *xen-pciback* module, which is easily possible by generating some activity on the PCI device. Due to the way the *xen-pcifront* driver is implemented, these requests will always reuse the same shared memory area making *op->cmd* always stay at the same address. By knowing this address, the second user process can repeatedly iterate between the original harmless value for *op->cmd* and a malicious value that triggers a jump to a different instruction pointer. As discussed by [20], the easiest and fastest way to switch between these two variable states is by using the `xor` instruction with a constant value depending on the chosen target value.

Testing the presented approach demonstrates that the short race is no problem in practice. In general, the race was won after less than ten PCI requests demonstrating the effectiveness of the described approach.

By getting an invalid value past the upper limit check of the jump table implementation, an attacker has complete control over the lower 32 bits of the `RAX` register in the `jmp QWORD PTR [rax*8+0x0]` instruction. This instruction performs an indirect jump, meaning the pointer at the address `rax*8+off_77D0` is fetched and written into the `RIP` register. Successful exploitation depends on the ability of an attacker to identify an offset which points to an attacker controlled value or a valid function pointer. While a complete description of an exploit for this vulnerability is out of scope for thesis, one possibly approach is outlined in the following.

On a modern Linux system the ordering and address ranges of kernel modules is almost completely randomized. This means that the search for potentially interesting offsets is restricted to the *xen-pciback* module itself. In addition the attacker only controls the lower half of the *rax* register, making it impossible to insert a negative value and search *before* the jump table at `off_77D0`. Still, there are several interesting possibilities: Almost immediately after the jump table used by the `switch` statement in the vulnerable `xen_pcibk_do_op` function, there is a second jump table used by the `xen_pcibk_frontend_changed` function shown in Listing 14. Listing 15 shows how the first of this switch statement is translated into assembly. The code copies the value of the `r13` register into `rdi` making it the first argument for the subsequent call to `xen_pcibk_attach`. When this code is normally called, `r13` points to a structure of type `xen_pcibk_device`, but when it is instead executed as part of our exploit,

```
void xen_pcibk_frontend_changed(struct xenbus_device *xdev,
                                enum xenbus_state fe_state)
{
        struct xen_pcibk_device *pdev = dev_get_drvdata(&xdev->dev);

        switch (fe_state) {
        case XenbusStateInitialised:
                xen_pcibk_attach(pdev);
                break;

        case XenbusStateReconfiguring:
                xen_pcibk_reconfigure(pdev);
                break;
        //..
        //..
}
```

Listing 14: "Reusable" switch statement in xen-pciback. The jump table generated for this switch statement can be used to trigger a type confusion after exploiting the xen-pciback double fetch vulnerability.

```
mov rdi, r13
call 0x3720 <xen_pcibk_attach>
```

Listing 15: Assembly of a reusable switch case. When exploiting the xen-pciback double fetch vulnerability, `r13` points to an attacker controlled location.

`r13` points to the attacker controlled shared memory region. This means we can call the function `xen_pcibk_attach` that would normally operate on trusted internal input with an fake structure completely under our control. This opens up a significant number of further approaches to reach the final goal of arbitrary code execution in the management domain.

6.5 Restricting the Impact of Compiler Optimizations

Besides the vulnerabilities presented above, the large impact of compiler optimization on double fetch vulnerabilities is a very interesting result of the double fetch analysis. To the best of our knowledge the *xen-pciback* double fetch is the first published vulnerability that is triggered by an (incorrectly) optimized switch statement. On the other hand, the impact of the potential double fetch vulnerability discovered in QEMU *xen_disk* is hard to assess without knowing exactly which combinations of compiler, compiler versions and flags lead to a vulnerable or non vulnerable result.

An interesting aspect of this is the existence of code that could potentially become vulnerable due to seemingly irrelevant changes to the rest of the function or the

compiler itself. For example, Listing 16 shows a switch case from the *xen-scsiback* backend. Even though it is very similar to the vulnerable one in *xen-pciback* and also operates on a variable stored in shared memory, the compiler generated code does not contain a double fetch. However, this could change when a new case is added,

```
1  switch (ring_req.act) {
2      case VSCSIIF_ACT_SCSI_CDB:
3          //...
4          break;
5      case VSCSIIF_ACT_SCSI_ABORT:
6          scsiback_device_action(pending_req, TMR_ABORT_TASK,
7              ring_req.ref_rqid);
8          break;
9      case VSCSIIF_ACT_SCSI_RESET:
10         scsiback_device_action(pending_req, TMR_LUN_RESET, 0);
11         break;
12     default:
13         //....
14         break;
15     }
```

Listing 16: Potentially vulnerable switch statement in xen-scsiback. `ring_reg.act` is stored in shared memory but the compiler does not generate an insecure jump table.

or even if the register allocation of the overall function changes due to modifications. This is of course not acceptable for such security critical code.

Code that seems vulnerable when looking at the source code, but is compiled correctly due to unenforced compiler decisions, should be considered insecure and must be fixed. In the case of the code shown in Listing 9, this is as easy as adding a temporal variable for `src->nr_segments` and enforcing a single access to it using a memory barrier.

For code such as the two discussed switch statements that only becomes vulnerable due to compiler optimizations, there are two viable alternatives: First, all variables stored in shared memory could be marked as `volatile`, which enforces a 1:1 mapping between variable and memory accesses. The other, more preferable approach is to restrict the primitives performed on shared memory variables to two secure ones: Direct accesses that copy a value into a local variable and which are protected by memory barriers, and the use of byte based copies that move whole structures from shared to private memory. This ensures that the compiler does not have the possibility to generate double fetch vulnerabilities by accident, and also makes it harder for a developer to introduce such vulnerabilities.

6.6 Conclusion

The presented evaluation validates several assumptions stated in the earlier parts of this thesis: The used memory tracing approach based on hardware-assisted virtualization and EPT permissions is well suited for the purpose of tracing shared memory communication. One of the main advantages to alternative approaches based on software emulation is the very low passive overhead. However, the chosen method introduces a very high active overhead when traced memory pages are heavily used. For use cases where a lot of memory activity needs to be traced, other approaches that try to improve the performance of software emulation are more feasible.

The attack surface algorithm correctly identified privileged backend components that operated on the traced memory regions. However, the evaluation demonstrated an important limitation of this approach. Because the algorithm does not collect a stack trace, only the immediate function that accesses shared memory can be identified. This is a problem for cases where these memory addresses are only accessed using generic copy functions, which makes it harder to identify the component responsible for the access. A potential improvement of the algorithm could try to extract the call stack when a memory address is performed. However, reliable detection of stack frames is not trivial in all cases making this quite difficult in practice.

The double fetch algorithm was able to identify three novel security vulnerabilities in popular backend components of the Xen hypervisor. This shows the feasibility of our memory traced approach for vulnerability discovery and indicates that our assumption about the lack of research in this area holds true. While the evaluation was limited to the Xen hypervisor, these results imply that research on the inter-domain communication of other hypervisors might be a good idea for further research.

Finally, the evaluation demonstrated the big impact of compiler optimizations on double fetch vulnerabilities. This shows that even seemingly secure source code can be compiled into vulnerable code and that developers have to be very careful when writing code that operates on shared memory.

7 Conclusion

Shared Memory is an important mechanism for efficient inter-process communication. In many cases the shared memory interface is a trust boundary separating privileged and unprivileged components. Examples for this include sandbox implementations and the paravirtualized device architecture of mainstream hypervisors. This makes research on security vulnerabilities affecting these interfaces important, especially because issues such as double fetches make implementing safe shared memory communication non-trivial.

In this thesis an approach to discover vulnerabilities in hypervisor inter-domain communication using memory tracing was presented, implemented and evaluated. In contrast to previous work in this area the presented approach is based on hardware-assisted virtualization and uses manipulation of EPT permissions to intercept and analyze memory accesses. This enables targeted tracing of shared memory communication with a very low passive overhead. The presented implementation is also largely target independent. Support for analyzing a new hypervisor or more generally a different shared memory interface can be easily added without a large implementation effort.

The effectiveness of the presented approach was proven by performing an evaluation against the paravirtualized device drivers of the Xen hypervisor. The evaluation demonstrated that our implementation fulfills the performance requirements for analyzing a real world hypervisor and that memory tracing can be used to map the attack surface available to an attacker targeting shared memory communication. Most importantly, the implemented double fetch analysis algorithm was successfully used to discover three novel security vulnerabilities in backend components of the Xen hypervisor. This demonstrates that the presented approach is capable of finding security issues in well audited software and indicates that the currently used approaches to secure hypervisor related code are not sufficient.

7.1 Future Work

One of the most promising areas for further research is the adaption of our implementation to support more hypervisors. Currently only the Xen hypervisor is fully supported as a target, but this is mainly due to compatibility problems concerning the nested virtualization of other hypervisors. Due to the rising significance of nested virtualization these issues will be hopefully fixed in the near future, allowing for analysis of these products. In addition, adding target support for popular sandbox

implementations and other security critical shared memory interfaces seems to be a promising extension of the presented work.

If the reliance on nested virtualization turns out to be a big road block for supporting other hypervisors, an alternative implementation of the trace collector based on software emulation could be evaluated. While this removes the advantages of our implementation that depend on the use of hardware-assisted virtualization, the decoupled nature of our architecture allows the reuse of all analysis components even if the actual trace collection is implemented completely in software. Current research such as Simuboost[35] tries to significantly improve the performance of software based emulation and might be a well suited target for such an implementation.

An alternative extension of the presented approach is the implementation of other analysis algorithms. While the presented attack surface and double fetch algorithms are very effective for analyzing inter-domain communication, other algorithms might be more suited for other use cases. In particular, it should be evaluated if the addition of memory contents to the memory trace could allow the implementation of more sophisticated algorithms enabling the discovery of other vulnerability classes.

Finally, future work should evaluate how memory access tracing can be used in combination with other automated approaches for vulnerability discovery. For example, the ability to identify code segments that operate on shared memory using the presented attack surface algorithm could be combined with static binary analysis to identify missing validation checks and other security issues. At the same time mechanisms used for measuring and increasing code coverage during fuzz testing could improve the performance of the double fetch algorithm by ensuring that all interesting code paths are executed.

Bibliography

[1]Robert Perry Abbott, Janet S Chin, James E Donnelley, et al. *Security analysis and enhancements of computer operating systems*. Tech. rep. DTIC Document, 1976 (cit. on p. 5).

[2]*American Fuzzy Lop*. `http://lcamtuf.coredump.cx/afl/`. Accessed: 2015-10-22 (cit. on pp. 23, 24).

[3]Paul Barham, Boris Dragovic, Keir Fraser, et al. „Xen and the Art of Virtualization“. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 164–177 (cit. on pp. 1, 8, 13, 43).

[4]Fabrice Bellard. „QEMU, a Fast and Portable Dynamic Translator.“ In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46 (cit. on pp. 14, 33).

[5]Al Bessey, Ken Block, Ben Chelf, et al. „A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World“. In: *Commun. ACM* 53.2 (Feb. 2010), pp. 66–75 (cit. on pp. 22, 23).

[6]Matt Bishop, Michael Dilger, et al. „Checking for race conditions in file accesses“. In: *Computing systems* 2.2 (1996), pp. 131–152 (cit. on p. 4).

[7]*Capstone*. `http://www.capstone-engine.org/`. Accessed: 2015-10-22 (cit. on p. 48).

[8]David Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008 (cit. on pp. 1, 14, 64, 65).

[9]Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, et al. „Frama-c“. In: *Software Engineering and Formal Methods*. Springer, 2012, pp. 233–247 (cit. on p. 23).

[10]*CVE-2005-2490*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-2490`. Accessed: 2015-10-22 (cit. on p. 5).

[11]*CVE-2011-1750*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1750`. Accessed: 2015-10-22 (cit. on p. 20).

[12]*CVE-2015-2361*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2361`. Accessed: 2015-10-22 (cit. on p. 20).

[13]Brendan F Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. „Repeatable Reverse Engineering for the Greater Good with PANDA“. In: (2014) (cit. on pp. 25, 32, 33).

[14]Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. „Virtuoso: Narrowing the semantic gap in virtual machine introspection“. In: *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 297–312 (cit. on p. 12).

[15]Tal Garfinkel, Mendel Rosenblum, et al. „A Virtual Machine Introspection Based Architecture for Intrusion Detection.“ In: *NDSS*. Vol. 3. 2003, pp. 191–206 (cit. on p. 12).

[16]Patrice Godefroid, Michael Y Levin, and David Molnar. „SAGE: whitebox fuzzing for security testing“. In: *Queue* 10.1 (2012), p. 20 (cit. on pp. 23, 24).

[17]*IDA: About*. `https://www.hex-rays.com/products/ida/index.shtml`. Accessed: 2015-10-22 (cit. on p. 51).

[18]Intel. *Intel® 64 and IA-32 Architectures Software Developer Manual*. 2015 (cit. on pp. 7–9).

[19]Alex Ionescu. „Battle of SKM and IUM: How Windows 10 Rewrites OS Architecture“. In: *Blackhat USA 2015* (2015) (cit. on pp. 11, 16).

[20]Mateusz Jurczyk and Gynvael Coldwind. „Identifying and exploiting Windows kernel race conditions via memory access patterns“. In: (2013) (cit. on pp. 1, 4–6, 25, 30–32, 39, 71).

[21]Jan Kiszka, CT T DE IT, and Corporate Competence Center Embedded Linux. „Architecture of the Kernel-based Virtual Machine (KVM)“. In: 2010 (cit. on pp. 16, 17).

[22]Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. „kvm: the Linux virtual machine monitor“. In: *Proceedings of the Linux Symposium*. Vol. 1. 2007, pp. 225–230 (cit. on pp. 16, 33).

[23]Chi-Keung Luk, Robert Cohn, Robert Muth, et al. „Pin: building customized program analysis tools with dynamic instrumentation“. In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 190–200 (cit. on p. 26).

[24]Jaydeep Marathe, Frank Mueller, Tushar Mohan, et al. „METRIC: Memory tracing via dynamic binary rewriting to identify cache inefficiencies“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.2 (2007), p. 12 (cit. on p. 26).

[25]Darek Mihocka and Stanislav Shwartsman. „Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure“. In: *1st Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA-35, Beijing*. 2008 (cit. on p. 32).

[26]A. Milenkoski, B.D. Payne, N. Antunes, M. Vieira, and S. Kounev. „Experience Report: An Analysis of Hypercall Handler Vulnerabilities“. In: *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. Nov. 2014, pp. 100–111 (cit. on p. 24).

[27]Gal Motika and Shlomo Weiss. „Virtio network paravirtualization driver: Implementation and performance of a de-facto standard“. In: *Computer Standards and Interfaces* 34.1 (2012), pp. 36–47 (cit. on p. 20).

[28]*MS08-61: The case of the kernel mode double-fetch*. `http://blogs.technet.com/b/srd/archive/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch.aspx`. Accessed: 2015-10-22. 2008 (cit. on p. 5).

[29]Jun Nakajima. „Making Nested VIrtualization Real by Using Hardware Virtualization Features“. In: *LinuxCon Japan* (2013) (cit. on p. 11).

[30]*Nested Virtualization in Xen*. `http://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen`. Accessed: 2015-10-22 (cit. on p. 44).

[31]Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999 (cit. on pp. 22, 23).

[32]P. Oehlert. „Violating assumptions with fuzzing“. In: *Security Privacy, IEEE* 3.2 (Mar. 2005), pp. 58–62 (cit. on pp. 23, 24).

[33]Bryan D Payne. „Simplifying virtual machine introspection using libvmi“. In: *Sandia Report* (2012) (cit. on pp. 43, 44).

[34]Marc Rittinghaus, Thorsten Groeninger, and Frank Bellosa. „Simutrace: A Toolkit for Full System Memory Tracing“. In: (2015) (cit. on pp. 1, 24, 33, 43, 45).

[35]Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. „SimuBoost: Scalable Parallelization of Functional System Simulation“. In: *Proceedings of the 11th International Workshop on Dynamic Analysis (WODA 2013)*. Houston, Texas, Mar. 2013 (cit. on pp. 33, 76).

[36]Edward J Schwartz, Thanassis Avgerinos, and David Brumley. „All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)“. In: *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. 2010, pp. 317–331 (cit. on p. 25).

[37]Konstantin Serebryany and Timur Iskhodzhanov. „ThreadSanitizer: data race detection in practice“. In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM. 2009, pp. 62–71 (cit. on p. 4).

[38]Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. 1st ed. Springer Publishing Company, Incorporated, 2008 (cit. on p. 23).

[39]Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005 (cit. on p. 9).

[40]Dawn Song, David Brumley, Heng Yin, et al. „BitBlaze: A New Approach to Computer Security via Binary Analysis“. English. In: *Information Systems Security*. Ed. by R. Sekar and ArunK. Pujari. Vol. 5352. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 1–25 (cit. on p. 22).

[41]W Richard Stevens, Bill Fenner, and Andrew M Rudoff. *UNIX network programming*. Vol. 2. Addison-Wesley Professional, 2004 (cit. on pp. 3, 4).

[42]*Sulley*. `https://github.com/OpenRCE/sulley`. Accessed: 2015-10-22 (cit. on p. 23).

[43]Andrew S Tanenbaum. *Modern operating systems*. Pearson Education, 2009 (cit. on pp. 3, 7, 8).

[44]*The Chromium Project: Sandbox*. `https://www.chromium.org/developers/design-documents/sandbox`. Accessed: 2015-10-22 (cit. on pp. 1, 19).

[45]R. Uhlig, G. Neiger, D. Rodgers, et al. „Intel virtualization technology“. In: *IEEE Computer* 38.5 (May 2005), pp. 48–56 (cit. on p. 8).

[46]„Virtual I/O Device (VIRTIO) Version 1.0 - Committee Specification Draft 01 / Public Review Draft 01“. In: (2013) (cit. on p. 17).

[47] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. „Towards optimization-safe systems: Analyzing the impact of undefined behavior“. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 260–275 (cit. on pp. 21, 22).

[48] Felix Wilhelm and Matthias Luft. „Security Assessment of Microsoft Hyper-V“. In: (2014). Accessed: 2015-10-22 (cit. on pp. 13, 16).

[49] Carsten Willems, Ralf Hund, and Thorsten Holz. „Cxpinspector: Hypervisor-based, hardware-assisted system monitoring“. In: *Ruhr-Universitat Bochum, Tech. Rep* (2013) (cit. on p. 12).

[50] *XSA-135: Heap overflow in QEMU PCNET controller, allowing guest->host escape*. `http://xenbits.xen.org/xsa/advisory-135.html`. Accessed: 2015-10-22 (cit. on p. 65).

[51] *XSA-139: Use after free in QEMU/Xen block unplug protocol*. `http://xenbits.xen.org/xsa/advisory-139.html`. Accessed: 2015-10-22 (cit. on p. 65).

[52] *XSA-155: Multiple vulnerabilities in paravirtualized devices*. `http://xenbits.xen.org/xsa/advisory-155.html` (cit. on pp. 2, 67–69).

[53] Xiantao Zhang and Eddie Dong. „Nested Virtualization Update from Intel“. In: *Xen Summit* (2012) (cit. on p. 11).