Università di Pisa

Scuola di Ingegneria
Laurea Magistrale in Computer Engineering

Hypervisor-based guest agent protection

Relatori:
Giuseppe Lettieri
Pericle Perazzo

Candidato:
Lorenzo Susini

Anno Accademico 2020/2021

Abstract

Hypervisor-based guest agent protection

Nowadays, cloud computing is gaining more and more popularity. People use cloud-related services every day. Virtualization is one of the most important enabling technology for this new kind of computing. CPUs are often equipped with features aiming to simplify hypervisor's tasks, such as running multiple virtual machines on the same physical one. The isolation guarantees offered by virtualization can be exploited to add a layer of security to a running guest system, both to its operating system and the applications running on it. Researchers in the past tried to achieve the same goal with systems which are completely outside the virtual machine, or totally inside it. The former approach suffers from the semantic gap problem, deriving from the difficulty to reconstruct semantic information from raw and low-level data. The latter, instead, cannot easily protect itself from attackers, since it is difficult to do it from the same privilege level. This work tries to follow a hybrid approach, running security critical and monitoring code inside the guest kernel and at the same time protecting and enforcing its execution from the hypervisor. A secured paravirtualized channel is then used to extract meaningful guest's data, which can then be examined to detect intrusions or enforce further security related policies.

Contents

Chapter 1

Introduction

As stated by NIST (National Institute of Standards and Technology), "*cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [11]. Virtualization is one of the enabling technologies of cloud computing and, in part, it helped cloud computing to become a reality. Indeed, virtual machines can be easily configured and rapidly deployed, like discussed in the definition. Furthermore, these tasks are often automated, opening the possibility of implementing dynamic and on-demand provisioning and to scale horizontally. Thanks to virtualization, utilization of the physical infrastructure is significantly improved by sharing physical resources and allowing virtual servers to move from one physical machine to another.

One of the key tasks of a hypervisor is to keep guest virtual machines isolated, with respect to other virtual machines and the host, and to precisely control the

usage of the underlying physical hardware. The isolation guarantees offered by the hypervisor are of particular interest also from a security standpoint, since the hypervisor can control the execution of a guest and analyze its state at the hardware level.

This thesis attempts to explore ways to use existing virtualization technologies to add a layer of security to a guest OS, possibly detecting and preventing attacks to the guest OS itself or to applications running on it, by executing monitoring code in a secure and stealthy way.

1.1 Problem statement

A system to observe and monitor the guest's state can be deployed both in the guest itself or in the hypervisor. Both these two approaches have advantages and disadvantages.

Considering the *in-guest* approach, the monitoring system is placed in the same OS that it is monitoring. In fact, it is visible to an attacker and this is one of the main drawbacks of this approach. The only line of defense between the monitoring system and the attacker is solely offered by the operating system. If an attacker manages to escalate privileges by exploiting some vulnerability (or a chain of vulnerabilities), the guest OS cannot protect the monitoring agent anymore and it becomes part of the attack surface. As an advantage, instead, given that the system is inside the guest OS, it has access to all the semantic information, such as files, kernel data structures, processes and much more, and can readily use this information to monitor the state of the virtual machine.

For what concerns the *in-hypervisor* approach, the monitoring system has access to the hardware state of the guest. For instance, it can inspect its memory and its registers. Since the monitoring agent is outside the virtual machine, the isolation provided by the virtualization layer ensure that an attacker gaining privileges in the guest cannot tamper with the system anymore, assuming that the software and the underlying hardware that allows the implementation of the hypervisor are safe. From a security point of view, this is of paramount importance, because the system can proceed with its work regardless the state of the guest. However, this approach lacks the semantic information that is available in the former approach. For instance, it is true that memory can be accessed, but in order to extract meaningful information from it, these kind of systems have to interpret bytes into a higher level semantic. In literature, this problem is often referred to as the *semantic gap*. The isolation offered by the virtualization layer is very attractive for the implementation of these kinds of systems, and that is why this approach was used by many researchers, leading to the birth of a set of techniques referred to as Virtual Machine Introspection.

This thesis will attempt to find new and creative ways for executing monitoring code using a hybrid approach, trying to combine the advantages of the two previously discussed approaches.

1.2 Objectives of monitoring agents

Generally speaking, monitoring agents can be used to implement intrusion detection and prevention systems to detect and block attacks. Monitoring the activities of guest systems can also be used to implement:

- forensics tools and malware detection, to derive an in depth view of an attack
- virtual machine based honeypots, as an alternative way to study malware and different kind of attacks
- system recovery, to restore the system in a safe state after an attack

Furthermore, monitoring a guest system is a useful feature to be offered to users by cloud providers. This can help to not only protect users, but also the infrastructures of cloud providers, since many hypervisor related attacks often start from a compromised virtual machine.

1.3 Related work

As noticed earlier, the literature offers a wide variety of systems and projects, which mainly fall in the two categories of *in-guest* and *in-hypervisor* approaches. This section discusses some of the features of two relevant projects, namely, the Linux Kernel Runtime Guard and LibVMI.

The Linux Kernel Runtime Guard [9] is an open source project carried on by OpenWall, a company whose leader is Solar Designer, a famous security researcher. This project represents the in-guest approach, even if it was intended to run not only in virtualized environments. As the name suggests, the LKRG is shipped as a kernel module whose aim is to "post-detect and hopefully promptly respond to unauthorized modifications to the running Linux kernel (integrity checking) or to credentials such as user IDs of the running processes (exploit detection)". Whenever the attacker successfully upgrades its credential through a kernel attack, any further

attempt to use them is denied before the kernel would grant the access, e.g opening a file with those new credentials. When such behaviour is detected, the process is killed. LKRG also tries to provide system integrity by protecting important kernel objects such as the .text section containing kernel code and modules, critical kernel variables and CPU registers. The default action is to make the kernel panic, since any milder response would be not effective. It is interesting to notice that developers of this project are aware that this system is *bypassable by design*, but still raises the bar of difficulty for many kinds of already or not yet known type of attacks. The reason why it is bypassable, though, was already explained: the system is visible to attackers and it is difficult to protect it from the same privilege level.

LibVMI [8], instead, is the de facto standard for the in-hypervisor approach. In general, most of the in-hypervisor based approaches are related to Virtual Machine Introspection techniques. The main reason for using these kinds of techniques is because they allow:

- minimum performance impact on the virtual machine, since the monitoring software is always running outside the VM
- minimum modifications to the hypervisor, which helps to put in place the monitoring system while not increasing the complexity of the hypervisor, thus not enlarging the attack surface
- little modifications to the guest operating system, to let the system be deployable on a wide variety of systems with little effort
- security of the monitoring system, guaranteed by the isolation features already

provided by virtualization.

One of the main techniques used to implement those kinds of system is the memory introspection, and this is what also LibVMI tries to accomplish. Main memory contains several important data structures such as process descriptors, loadable kernel modules, page tables, and others. For this reason, LibVMI is focused on reading and writing memory from virtual machines. For convienence, it also provides functions for accessing CPU registers, pausing and unpausing a VM, printing binary data, and hooking on hardware events. The library also offers Python bindings and can be used together with Volatility, a pretty famous memory forensic tool. However, the preliminary work of any VMI tool is to first bridge the semantic gap, i.e extract meaningful information from memory. This is the most difficult task that this kind of systems have to perform and this is why researchers are trying to find alternative ways to solve it.

Chapter 2

Background

As previously mentioned, all ideas related to this work were implemented making use of existing technologies. The following sections are dedicated to a brief description of each of them.

2.1 Virtualization

Virtualization technologies allow running one or more virtual machines, called *guests*, on the same physical server, called *host*. The core idea is to create virtual servers that can run as if they were running on a real physical machine. The entire operating system and its applications can be run inside a virtual machine. Guest systems are completely managed by a software layer running inside the host, which is referred to as the *hypervisor* or *Virtual Machine Monitor* (VMM). The hypervisor is responsible for virtualizing all the physical resources that are assigned to a virtual machine, such as CPUs, memory and I/O devices. Guest virtual machines are typi-

cally forced to interact only with the virtualized resources, and the hypervisor itself prevents them to deal with the physical ones.

In general, there are mainly two approaches to implement hypervisors: Type-1 and Type-2 hypervisors. Type-1 hypervisors are also called *bare-metal hypervisors*. They are installed directly on the hardware and have direct access to each physical resource. This can be seen both as an advantage and a disadvantage: direct access to physical resources often allows to obtain better performance, but at the cost of being too hardware dependent, thus reducing portability. On the other hand, Type-2 hypervisors run on top of an operating system: they can access hardware resources only through the usage of the underlying operating system, exploiting the portability of the operating system itself and its drivers, at the cost of slightly degrading the performance of virtual machines. However, sometimes the distinction between Type-1 and Type-2 hypervisor is not so clear. In this work, the KVM hypervisor is used: it can be considered as a mix of the two approaches since it makes use of hardware virtualization features directly as if it was a Type-1 hypervisor, but at the same time tries to reuse many of the features of the Linux kernel for its own purposes, as in Type-2 hypervisors. More details on it are in the following sections.

The virtualization problem has been faced in many ways, leading to as many ways of implementing it. In particular, there exist at least three main different solutions to the problem.

2.1.1 Emulation

The first one is the so-called *emulation*. An emulator is a program that is typically built around a main loop. Its purpose is to mimic the actions that would be performed by the target machine processor, such as fetching, decoding and executing instructions. Typically, it is possible to think about the execution phase of the processor as a big switch, having one case for each possible instruction belonging to the instruction set of the target machine. Each of those instructions is then translated into actions, i.e host instructions, performed by the hypervisor, allowing for the possibility to run any software that was originally written for another machine, potentially with a different architecture and instruction set w.r.t the host. Usually, emulators are implemented as userspace programs in the host system and they are in charge of virtualizing every resource used by the guest. To do that, a different data structure for each of the resources available to the guest is used. For the sake of clarity, it is possible to think about the simplest machine as if it was made up of only its CPU and memory. Then, the CPU can be defined as a collection of registers and memory as an array of bytes, allocated in the virtual address space of the process running the emulator. When executing instructions, the emulator will modify the state of these resources according to their semantic. Of course, when it comes to emulating a much more complex machine, many other features have to be taken into account, such as interrupts and I/O devices, but in general the approach to be used is the same for any hardware resource. Also, it has to be noted that the emulator can make use of all the functionalities offered by the host operating system to implement any kind of hardware resource. For instance, considering a virtual disk

used by the virtual machine to store data, it can be easily implemented using a file in the host. The disk will be accessed by the target processor instructions related to I/O and the emulator can implement those instructions using filesystem-related system calls, such as *open*, *read*, *write* and *close*. On the other hand, however, introducing I/O devices is also one of the biggest challenges in emulation. This is because I/O devices often perform operations asynchronously w.r.t to the main loop implementing the CPU. Typically, this problem is solved by using multithreading, assigning one thread to the CPU and one or more threads to I/O devices. This may introduce race conditions and interference problems within the emulator, thus all objects that need to be accessed by multiple threads must be protected using synchronization techniques, increasing the complexity of the emulator itself.

The emulation approach can be considered as a naive approach to virtualization. The emulator is simply scanning guest instructions one by one and translating them into pieces of software running on the host. It is often the case that one guest instruction is translated into many host instructions, leading to a non-negligible overhead. Techniques like binary translation aim at improving the performance of emulation by translating blocks of guest instructions into host instructions. By looking at many instructions at a time, it is possible to optimize, for instance, the number of host instructions needed. Furthermore, binary translation often caches translated blocks to reuse them when the same guest block is hit again. However, even this approach cannot reach near-native performance and thus other approaches to virtualization were introduced to let virtual machines run much faster. Finally, even if this is not the approach used in data centers, it has some concepts that are

to some extent reused in other approaches and that are useful for the purposes of this work.

2.1.2 Trap & Emulate

Trap & Emulate approach tries to overcome the limitations of emulation. The fundamental idea behind this approach was to try to run virtual machines having the same architecture and instruction set of the host as low privileged userspace processes. Anytime a VM is executing, the CPU directly takes orders from the guest. This is possible only because the instruction set of the guest is exactly the same as the one of the host. Of course, there is the need to put limitations on what the guest system can do. Those limitations are introduced by the fact that the virtual machine is running with low privileges: anytime the guest operating system tries to execute a privileged instruction, a trap will be generated. Through this trap, the hypervisor can regain control of execution, taking care of correctly emulating the instruction that caused the trap on behalf of the guest, thus limiting what the guest virtual machine can do. For instance, think about the `IN` and `OUT` instructions on x86 architectures. As these are privileged instructions, all I/O operations will be handled by the hypervisor, preventing the guest to execute operations on real hardware. However, this approach was not completely feasible on some architectures, in particular x86 architectures. This is because the instruction set contains instructions that behave differently if executed at different privilege levels. In particular, those instructions will not trap, leading to inability of the hypervisor to regain control of the execution, even if that instruction should have been emulated. Most of those

instructions are related to memory management and interrupt handling, but their discussion is out of the scope of this work. Anyway, the Trap & Emulate can be considered as the main starting point for all modern techniques used nowadays.

2.1.3 Paravirtualization

Paravirtualization is one of the proposed solutions to alleviate the problem of not being able to trap all the desired instructions, specifically on x86 architectures. This problem mainly arises because one of the requirements that was implicitly taken into account was to run unmodified versions of the guest operating system. Paravirtualization releases this constraint and tries to make the guest operating system aware that is being virtualized and able to communicate with the hypervisor. This is accomplished by replacing all instructions which are hard to virtualize with *hypercalls*. The hypercall notion is very similar to the one of system calls: as system calls are used to let userspace processes communicate with the operating system, hypercalls are used to let guest virtual machines communicate with the hypervisor. With hypercalls the hypervisor can easily regain control of the execution every time it is needed and emulate those that were difficult-to-virtualize instructions.

This is how paravirtualization solves the problem of the Trap & Emulate approach. However, it introduces many other problems. In fact, paravirtualized systems fully rely on the interface exposed by the hypervisor. If that interface changes, they are not able to run, thus portability is drastically reduced. Furthermore, it may be difficult or impossible to have support for closed source operating systems. Other techniques like hardware-assisted approaches virtualization in a different way

and without any guest modifications, as it is explained in the following, but the idea of hypercall is essential to the purpose of this thesis and it will be recycled in the proposed system.

2.1.4 Hardware-assisted virtualization

Another way to solve the problems related to the Trap & Emulate approach is hardware-assisted virtualization. This kind of approach is based on a simple consideration: while the virtual machine is running and giving orders directly to the CPU [1], the only way to restrict the actions of the guest is by means of the hardware. In recent years, both Intel and AMD added virtualization features to their processors, which aim at offering support to hypervisors. In the following, only Intel's ones will be considered, which are grouped into Intel Virtualization Technology. In general, hardware is modified to specifically and selectively put controls on guest systems. This can be done if the hardware itself is aware of whether it is a virtual machine or the hypervisor that is running. To determine these two states, a simple bit can be used. This idea led to the introduction of the *root* and *non-root* mode in Intel processors. When the processor is in root mode, the hypervisor is executing and the VM is on pause. Contrarily, when a VM is executing, the processor is in non-root mode. It is worthwhile noticing that these two new states of the processor are completely *orthogonal* to the already existing system and user mode (Ring 0 and Ring 3 in protected mode), leading to four different combinations: *root/system, root/user, non-root/system* and *non-root/user*, which are typically associated with the host OS,

[1] the same ISA case is considered again

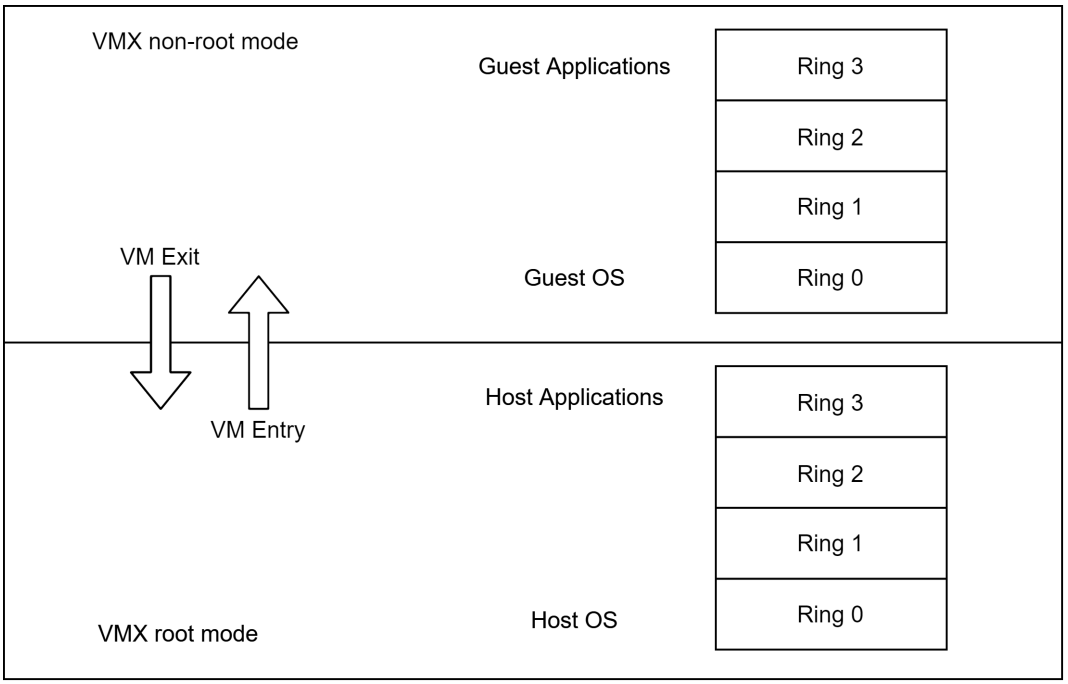

Figure 2.1: Representation of Intel Virtualization Technology

host applications, guest OS and guest applications, respectively, as also shown in Figure 2.1. Thanks to the introduction of this new mode of operation, guest systems can now switch between userspace and kernel space without hypervisor intervention, something that was impossible using the Trap & Emulate approach.

Whenever the guest kernel tries to execute an instruction that needs the intervention of the hypervisor, the hardware forces a transition from non-root to root mode, letting the hypervisor regain control of execution. This event is defined as `VM EXIT`. On `VM EXIT`, the hardware also takes care to let the hypervisor know the *exit reason* of the VM. This information can be used to correctly emulate the instruction that caused the exit on behalf of the guest. After the emulation of such instruction, the hypervisor turns the processor into non-root mode and starts the VM again,

which will continue its execution with the instruction next to the one that caused the exit. To handle this mechanism, Intel has designed and implemented new instructions and defined new data structures. First, since this is a new feature of Intel processors, hypervisors can discover support for Intel Virtualization Technology by executing the `CPUID` instruction. This instruction returns processor identification and feature information to the `EAX`, `EBX`, `ECX` and `EDX` registers. If bit 5 in `ECX` is set, it means that the processor supports Virtual Machine Extensions (VMX), meaning that hardware-assisted virtualization is possible. To enter and exit VMX operation mode it is possible to use the `VMXON` and `VMXOFF` instructions. When VMX is turned on, the hypervisor can make use of all the other newly introduced instructions. In particular, transitions from root to non-root mode are achieved by means of `VMLAUNCH` and `VMRESUME`, used to start and resume virtual machines.

Many of the VMX instructions are related to the handling of the Virtual Machine Control Structure (VMCS). This is one of the most important data structures used to implement the mechanisms briefly introduced before and it is worthwhile being aware of its detail in depth. This data structure is used to manage VM entries and VM exits and to define the processor behavior in non-root mode. It can be manipulated by means of the new instructions `VMCLEAR`, `VMPTRLD`, `VMREAD` and `VMWRITE`. The hypervisor maintains several VMCSs, typically one for each processor of each virtual machine. Out of all the VMCSs, only one can be the current VMCS and that one is pointed by a specific processor register. To make current a specific VMCS, the instruction `VMPTRLD` can be used. All instructions used to manipulate the VMCS have effects only on the current VMCS. Usually, hypervisors reserve 4KB to store

VMCSs, which are organized into six logical groups:

- Guest-state area. The state of the guest processor is stored and loaded using this area on `VM EXIT` and `VM ENTRY`, respectively. The processor is of course saved by using the set of its registers, including its general-purpose registers, control registers, debug registers and so on. The value stored in the instruction pointer can be used to determine the instruction that caused the `VM EXIT` and to locate the first instruction that the guest should execute on `VM ENTRY`.

- Host-state area. This area is used to load the state of the host processor on `VM EXIT`. All the host's registers are saved in this area on `VM ENTRY`. Since the entire state of the host processor is saved here, the guest can freely modify its own registers without affecting the host.

- VM-Execution control fields. This area defines controls on the processor behavior which are put in place when in non-root mode. They determine in part the causes of `VM EXIT`. These fields are used to specify what should happen both on asynchronous events (e.g. interrupts) and synchronous events. For the latter, two 32-bit vectors are used to determine which events or instructions should cause a `VM EXIT`. Examples of these controls are CR3-load and CR3-store exiting, enabling second level address translation (Extended Page Tables in Intel), Descriptor-table exiting (exiting on LIDT or SIDT execution) and many others.

- `VM EXIT` control fields, used to control `VM EXIT`.

- `VM ENTRY` control fields, used to control `VM ENTRY`.

- `VM EXIT` information fields. This area contains data related to the reason that caused the last `VM EXIT`. It is used to understand what has happened and what the hypervisor should do before entering in non-root mode again.

Another important feature offered by Intel Virtualization Technology is hardware support for the virtualization of the guest Memory Management Unit (MMU). The introduction of this new feature heavily improved older implementations of virtualization of virtual memory, such as shadow page tables. In general, the task of MMU virtualization involves translating Guest Virtual Addresses (GVA) into Host Virtual Addresses (HPA). The guest kernel is only able to prepare mappings to perform the translation from GVA to Guest Physical Addresses (GPA), maintaining its own page tables. To go further in the translation and to put controls on the memory accessed by the guest, it is clear that the hypervisor should be involved in some way. When using the shadow page tables approach, hardware will never use the guest page tables directly. Instead, the hypervisor forces a `VM EXIT` whenever the guest tries to modify the CR3 register or its own page tables and performs actions to let the guest access memory only where it is supposed to, by using host page tables. In this way, guest will use the mappings prepared by the hypervisor and the MMU will continuously perform translation from GVA directly to HPA. This is a completely working approach, but it is very expensive: `VM EXIT`s involves very time-consuming operations, such as saving the guest's state and loading the host's one, examining the exit reason, and perform operations accordingly. For this reason, the number of exits should be maintained as low as possible, otherwise, virtual machines will have to pay a great performance overhead.

To overcome these issues, Intel implemented a second-level address translation approach, Extended Page Tables (EPT) in their jargon. The idea behind this approach is simple: the host processor, and specifically the host MMU, is capable of holding two pointers. The translation of a guest virtual address starts by using the guest's CR3 register, which is loaded in the physical CR3 register when the VM is running. Through the usage of its CR3 register, the VM is able to translate any guest virtual address to guest physical address. Then, each guest's physical address needs to be translated into its corresponding host physical address, and this is done with second-level page tables, which are completely managed by the hypervisor and equivalent to the normal page tables. To maintain these second-level page tables, the MMU is equipped with the second pointer, called `EPTP`. Hardware uses these second-level page tables when the processor is in non-root mode. The guest can now modify its own page tables without neither hypervisor intervention nor `VM EXIT`s since the translation now happens in hardware and second-level page tables are prepared by the hypervisor, solving the performance problems of the previous approach. It must be noted, though, that the overhead is paid in another way: a translation of a guest's virtual address now involves up to 24 memory accesses, assuming four-level page tables both for the guest and the host. This can be explained by following all the steps involved in the translation:

- First, the translation of a GVA must start from the guest CR3. This register contains a GPA, which in turn must be translated before actually accessing the guest level-4 table. The translation of the address in CR3 requires a walk in the second-level page tables of the host, thus four memory accesses. Then,

after the translation of CR3, it is possible to access the guest level-4 table with one more memory access, resulting in five memory accesses.

- After gaining access to level-4, a part of the GVA is used to index that table to access the corresponding level-3 table. Again, the pointer to this table is a GPA that must be translated too. Four accesses are needed for the translation and one more access for the level-3 table.

- The previous steps must be performed also for level-2 and level-1.

- In the last step, the translation is retrieved. The obtained GPA corresponding to the translation of the initial GVA has to be translated again into a HPA, and this adds just four memory accesses, resulting in 24 accesses in total before accessing the needed data.

For the sake of clarity, Figure 2.2 shows the steps just discussed. The result of these steps is a 2D page walk, both in the host and guest directions, and that is the reason why this approach is also called Two Dimensional Paging. To avoid performing these long page table walks, the processor is equipped with many TLBs to cache the recently used translations. Furthermore, another way to decrease the number of memory accesses in case of a TLB miss is using bigger-sized pages, to shorten the length of the walk.

This is the most widely adopted approach to MMU virtualization today since it outperforms many of the previous ones, shadow page tables included. Another important aspect of the second level page tables is that both table and page descriptors have their own control bits: in fact, the hypervisor can use these bits to

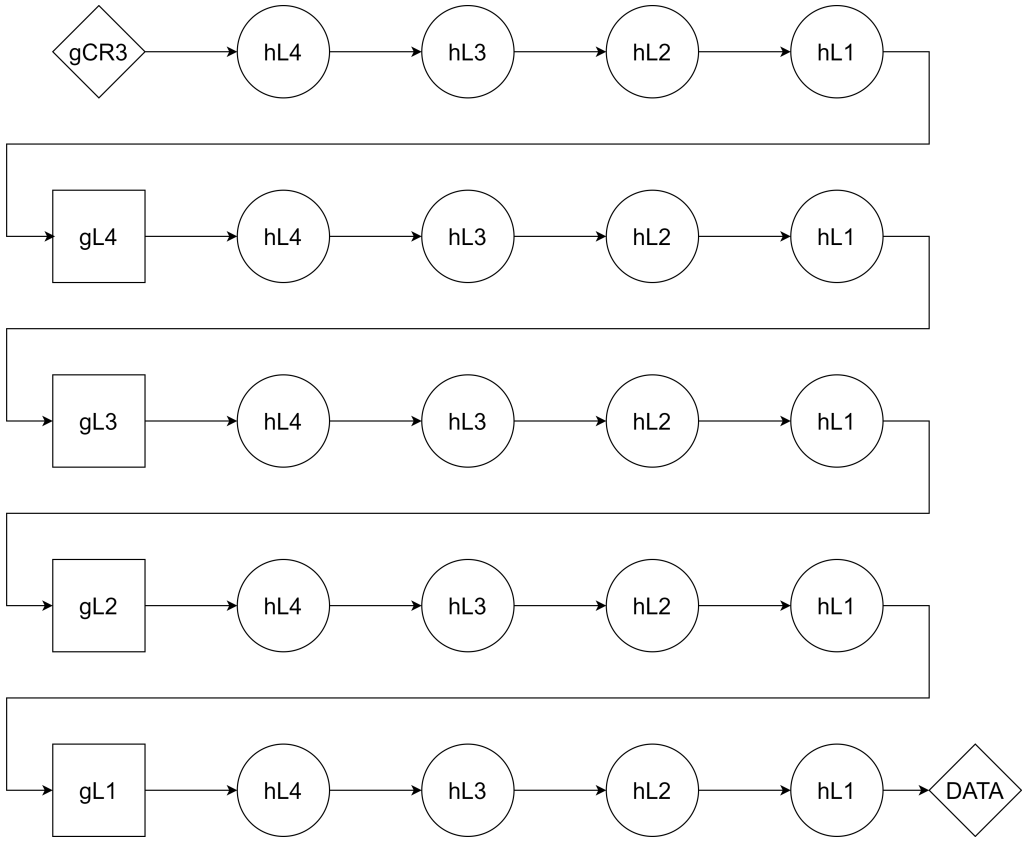

Figure 2.2: The two-dimensional page table walk. Host page tables are walked horizontally, while guest's ones are scanned vertically, resulting in 24 memory accesses before accessing data.

enforce a specific kind of memory access, such as read, write or execute access, or combinations of them. Furthermore, through the usage of the access and dirty bit, it can understand which pages were accessed, written, and, by consequence, read. To use these bits, however, the hypervisor must know which are the memory areas requiring such protections and with which access mode.

2.2 QEMU and KVM

As mentioned before, hardware-assisted virtualization offers support for implementing hypervisors. Kernel-based Virtual Machine (KVM) [14] is a Linux subsystem developed to transform Linux into a hypervisor, leveraging the newly added features for virtualization. On the other hand, QEMU is an emulator and it can be used to run target systems with a different architecture with respect to the host, or when the guest's and host's architecture are the same, it can make use of KVM by setting up virtual machines using its API. In this way, QEMU can leverage KVM for creating and managing virtual machines, adding memory and CPUs to them. Then, it has also the responsibility for emulating virtual devices attached to them, such as keyboards, hard drives, PCI devices, and network cards. This is always done in the same way by QEMU, both when it works as a hardware emulator and when it uses KVM with hardware-assisted virtualization.

Both QEMU and KVM are open source software and they are a perfect fit for this work. The source code is readily available on the Internet and it can be studied and modified for different purposes, like the ones discussed here. This work tries to modify the part of QEMU's code using the KVM API and the KVM kernel module itself to achieve its goals, thus it is worthwhile explaining the high-level architecture of both.

KVM is structured as a Linux character device. Its device node is `/dev/kvm` and it can be used from userspace, e.g from QEMU, through a set of `ioctl` system calls. Typical operations that can be issued are:

- Creation of a new virtual machine

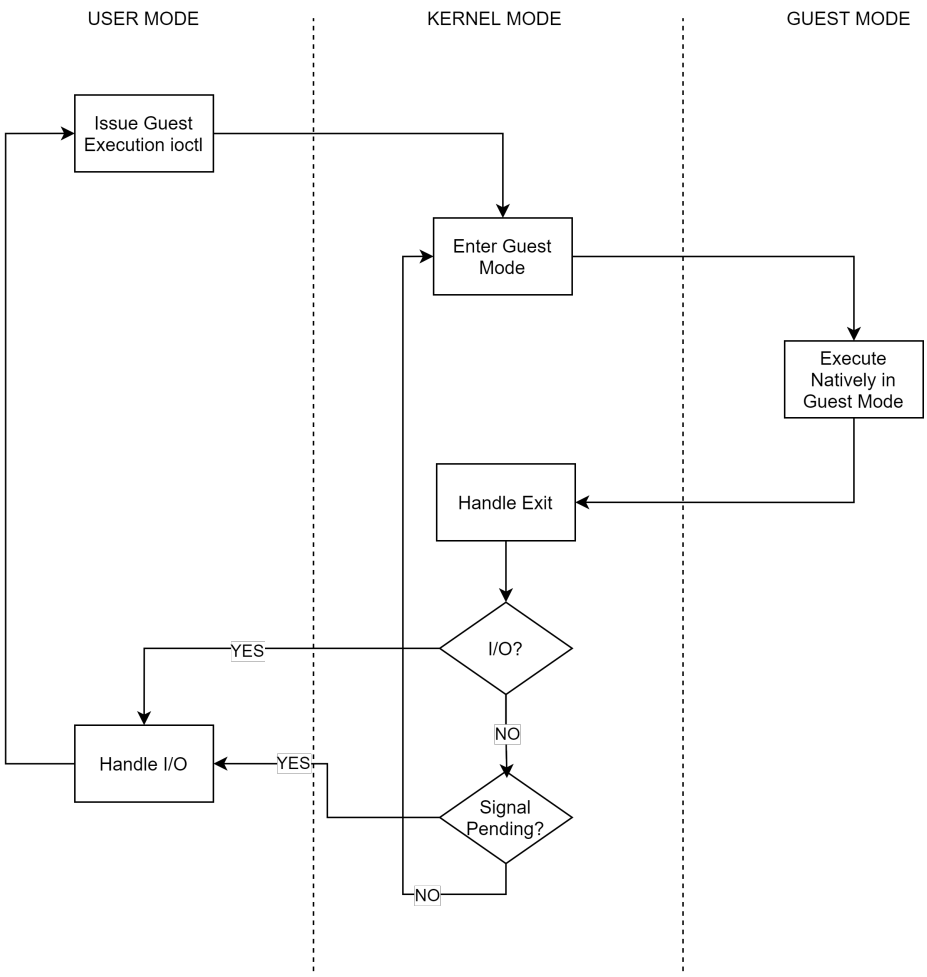

Figure 2.3: KVM guest execution loop

- Allocation of memory to a virtual machine
- Reading and writing virtual CPU registers
- Running a virtual machine

Once the virtual machine is configured (e.g, the userspace process corresponding to QEMU has attached memory and CPUs to the VM through the usage of the KVM API), the guest gets executed in a triply nested loop, as shown in Figure 2.3.

The userspace process asks the kernel to execute the virtual machine, thanks to the `KVM_RUN` ioctl. The kernel performs the actions needed to execute the guest. In particular, in Intel architectures, it will execute a VMLaunch instruction once the corresponding VMCS is correctly set up, forcing the processor to enter the non-root mode. After this step, the guest is executed until a VMExit occurs. If the exit reason is related to an I/O instruction, for instance, it will be completely handled in userspace and, in this case, in the QEMU process. Once QEMU has emulated the instruction, it will issue again the ioctl to run the virtual machine and so on. This is the most simple way to explain how QEMU and KVM cooperate to implement a complete hypervisor.

2.2.1 QEMU threading model

To deeply understand how QEMU works, it is fundamental to be aware of its threading model, which is thoroughly depicted in Figure 2.4. As already mentioned, QEMU is a userspace event-driven multithreaded application. The entire QEMU process is used to model the whole virtual machine. When using QEMU with KVM enabled, its main purpose is to virtualize I/O and interact with the KVM kernel modules for guest code execution. In particular, the part of QEMU regarding I/O is strongly based on events. In fact, QEMU makes use of the I/O thread to let the virtual machine interact with the external world. The I/O thread is centered around a big main loop that makes use of the `select` or the `poll` family of system calls, waiting for new file descriptors to become ready. Whenever a file descriptor is ready, the main loop is responsible for *reacting* to this event by calling the corresponding *callback*

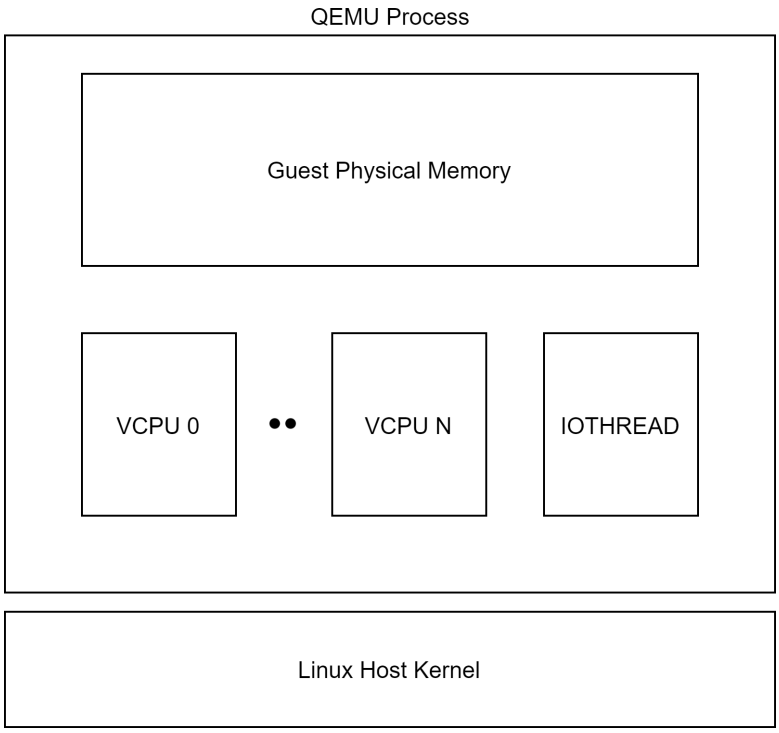

Figure 2.4: QEMU threading model

function. This behavior can be fully summarized using the `qemu_set_fd_handler` function as an example: it allows to add a file descriptor to be monitored by the main loop. File descriptors typically become ready for read and write operations, and that is why the previous function takes as an argument both a read and write callback. Using this logic, file descriptors callbacks will execute sequentially one after the other and in a non-blocking fashion since callbacks will invoke I/O functions only on ready file descriptors.

Another important aspect regards the execution of the guest code itself. Each virtual CPU attached to the VM corresponds to a thread inside QEMU. These threads are responsible for executing the loop shown in the already discussed Figure

2.3. It has to be noted that the vCPU thread continuously switches between guest execution and handling of I/O events. Thus, it is clear that data structures representing I/O devices are accessed concurrently both by vCPU and I/O thread, as already discussed in Section 2.1.1. In particular, it can be noticed how the emulation approach and the one discussed are pretty similar. The only difference consists in the way guest code is executed: in the emulation approach, guest instructions were executed through the use of the execution loop, while in this approach guest instructions are directly executed on the CPU, using all the features of hardware-assisted virtualization. Furthermore, since vCPUs are modeled as threads, they are scheduled as normal threads in the system and the QEMU process can make use of signals whenever it wants to stop the execution of the guest.

2.2.2 KVM kernel modules

The proposed solution will also try to modify the KVM kernel modules to accomplish its goals, and that is why it is relevant to briefly discuss how it is organized internally.

KVM is organized in two kernel modules: one is the `kvm.ko` and the other is architecture-specific. Since the Intel VT-x technology is being used, the `kvm-intel.ko` modules will be used. Specifically, the first module will detect the type of processor of the machine and will load the needed module accordingly. The two main files of KVM are `kvm-main.c` for the generic part of KVM and `vmx.c` for the Intel specific part.

The `/dev/kvm` character device is implemented as a `misc_device` and it is possible to let the kernel handle the ioctls by associating to it the appropriate

`file_operation` struct, as reported in Listing 2.1.

```
static struct file_operations kvm_chardev_ops = {
        .unlocked_ioctl = kvm_dev_ioctl,
        .llseek         = noop_llseek,
        KVM_COMPAT(kvm_dev_ioctl),
};

static struct miscdevice kvm_dev = {
        KVM_MINOR,
        "kvm",
        &kvm_chardev_ops,
};
```

Listing 2.1: KVM character device

The userspace part will also deal with other file descriptors such as the VM and VCPU file descriptors. The latter two are installed in the file descriptor table of the process by means of anonymous inodes, since there are no files or character devices that are backing them. In any case, it is possible to associate a `file_operation` struct pretty much in the same way, using the `anon_inode_getfile` function. The functions that are supposed to be called and respond to ioctls are organized in big switches, having one statement for each possible ioctl.

From the point of view of KVM, a virtual machine is completely described by the `kvm` struct, as reported in Listing 2.2.

```
struct kvm {
```

```
    ...

    struct kvm_memslots __rcu *memslots[
        KVM_ADDRESS_SPACE_NUM];
    struct kvm_vcpu *vcpus[KVM_MAX_VCPUS];

    ...

    struct kvm_arch arch;

    ...

}
```

Listing 2.2: Relevant fields of struct `kvm`

As it can be noticed, this data structure contains an array of pointers to `kvm_vcpu` structs, which is the data structure describing the virtual CPUs. Furthermore, it contains pointers to `kvm_memslots` struct which in turn is used to retrieve all the `kvm_memory_slot` structs associated with the virtual machine. More details on how memory is allocated are described in the following, but until now it is sufficient to notice that the set of `kvm_memory_slots` are used to describe the physical memory of the guest. Lastly, all architecture-dependent features are always encapsulated in a dedicated data structure, such as the `kvm_arch` or the `kvm_vcpu_arch` data structures.

This was a brief description of how KVM is organized internally and will be useful when modifying it to accomplish the objectives of this work.

2.3 Threat modeling

When it comes to building a system for security, it is important to state the relevant assumptions beforehand. In this work, the attacker is modeled as an external one. Typically, virtual machines are used to offer some services to the external world and they have to be accessible in a certain way. It is assumed that the attacker is able to attack the system from the outside and to escalate its privileges gaining Ring 0 access, with the intention of gaining *persistence* as the last step of the attack. Ring 0 access means that the attacker has the ability to arbitrary read and write in the kernel address space. The final objective of the intruder, i.e persistence of the attack, will let her take full control of the virtual machine anytime. What it is important to underline is that this kind of attack scenario is made up of different steps. A system to protect against it can thus be organized in multiple lines of defense and try to mitigate or even eliminate the risk of being attacked or to do the next step, as the proposed system tries to do. Additionally, awareness on how the attacker can gain persistence is crucial. Attackers often achieved this by means of rootkits, which are installed upon successful attacks.

Rootkits are a specific kind of malware that is able to modify the behavior of a running operating system, letting the attacker being able to regain easy access to the root user once the malware is installed. Another rootkit's goal is to hide the fact that the system has been compromised. This is typically done by hiding files, processes, network connections, and so on, so that the attacker's activities are as invisible as possible. Since rootkits have the ability to modify the running operating system, they can do almost everything in the system, such as also spying on users,

clean log evidence, and mislead or tamper with other detection tools on the victim machine. Generally speaking, rootkits can be categorized as userspace rootkits or kernel space ones.

Userspace rootkits commonly focus on replacing specific system programs used to extract information from the system. Depending on the goal, it is possible to replace different programs. For instance, the `login` command was historically replaced to place a backdoor to root, or instead, if the objective was to hide malicious processes, `ps` or `top` were targeted. Also, the standard C library was a frequently targeted binary in the past. This type of approach is easy to detect if those files are continuously checked against a hash, computed when the system was known to be in a safe state. Nowadays, most of the userspace rootkits use the `LD_PRELOAD` approach. Basically, the Linux dynamic linker offers the possibility to define which libraries must be preloaded. The attacker can create a malicious shared library and load it before the other normal system libraries are loaded. If the malicious shared library uses some of the symbols used by other libraries, e.g. the standard C library, it can override them and completely redefine their behavior. Examples of targeted symbols is the `readdir` C function, which is a wrapper for the `getdents` system call. By overriding the behavior of this function, it is possible to hide any file inside the system. Furthermore, since most utilities like `ps` use the `/proc` filesystem to extract information, this same method can also hide malicious processes in the system.

Kernel space rootkits operate in a completely different way. They achieve their goals by directly modifying the running operating system, attacking it at its core, the kernel. Usually, kernel-mode rootkits are injected by means of Loadable Kernel

Modules (LKM). LKMs were introduced to give system administrators and developers a way to expand the running Linux kernel with new functionalities. This can be done without neither recompiling the kernel nor rebooting the system because the code contained in a LKM is dynamically linked to the running kernel. This code will end up running at the system level, i.e code will run with no restrictions, giving the attacker the possibility to be as creative as possible to achieve their goal, such as hiding the fact that the system has been compromised. For the same reason, kernel-level rootkits are very hard to detect. There are a wide variety of techniques that are used by kernel rootkits, but typically they all try to operate at the interface between userspace and kernel, modifying the normal execution flow by adding their own logic. More details on these techniques are in the following.

2.4 Work environment

For reproducibility of the results, all the software and versions used in this work are reported below.

- Host operating system is a Ubuntu 20.04 equipped with a Linux 5.11.12 kernel
- The virtual machine runs on a single vCPU with a Linux 5.12.0-rc2 kernel, which was directly downloaded and compiled from GitHub [10]
- QEMU version 5.2.50, forked and compiled from GitHub [13].
- Root file system was generated using [4]
- Intel Core i7-8565U CPU

Furthermore, GDB and the QEMU monitor were used to inspect the virtual machine state for debugging purposes.

Chapter 3

Design and implementation

3.1 System architecture

The core idea of the proposed solution is to implement a part of the system both in the guest and in the hypervisor, to combine the best features of the two approaches. To achieve this, the monitoring code must run inside the guest, so that it is possible to have access to the semantic information of the guest operating system itself. Certainly, there is the need to protect the part of the system running in the guest, otherwise, like already mentioned, it becomes an easy target for attackers, and this is done by the hypervisor.

The overall system is based on the assumption that an attacker cannot deactivate the part of the system implemented in the hypervisor. This is a reasonable assumption because virtualization technologies have to provide isolation by design. However, in the past, attackers managed to escape from the guest virtual machine, gaining arbitrary code execution on the hypervisor's side. This kind of attack is

often referred to as VM escape. In such a case, not only the monitoring system can be compromised, but also other virtual machines and the host itself. An example of these kinds of attacks can be found in CVE-2015-3456 [2], an exploit commonly known as VENOM, which allowed an attacker to execute code with the privileges of the QEMU process. It has been noticed that most of these kinds of attacks exploit bugs related to the I/O emulation service of QEMU, and recently, to prevent them, it has been proposed a multi-process version of QEMU, instead of the current monolithic one. The idea is to completely separate the services for I/O, having one process for each emulated device. This allows to introduce security measures with finer granularity: for instance, if a process is used to emulate a disk device, the set of the system calls it can use can be reduced using the `seccomp` system call and it can be forced to access only a limited set of files, i.e the disk image file and a few more, by using a Mandatory Access Control system (AppArmor, SELinux). An attacker escaping a VM and gaining the privileges of such a secured process cannot compromise the host system and other VMs with the same ease as in the monolithic version. The multi-process QEMU would only introduce benefits also for the proposed system, making its assumption even more reasonable.

The system has to fulfill the following requirements:

1. The monitoring code has to be executed in the guest at given time intervals, and the start of its execution must be ensured by the hypervisor.

2. An attacker must not be able to prevent the execution of the monitoring code and cannot predict when its execution starts.

3. The overall system must be designed with a defense in depth approach in

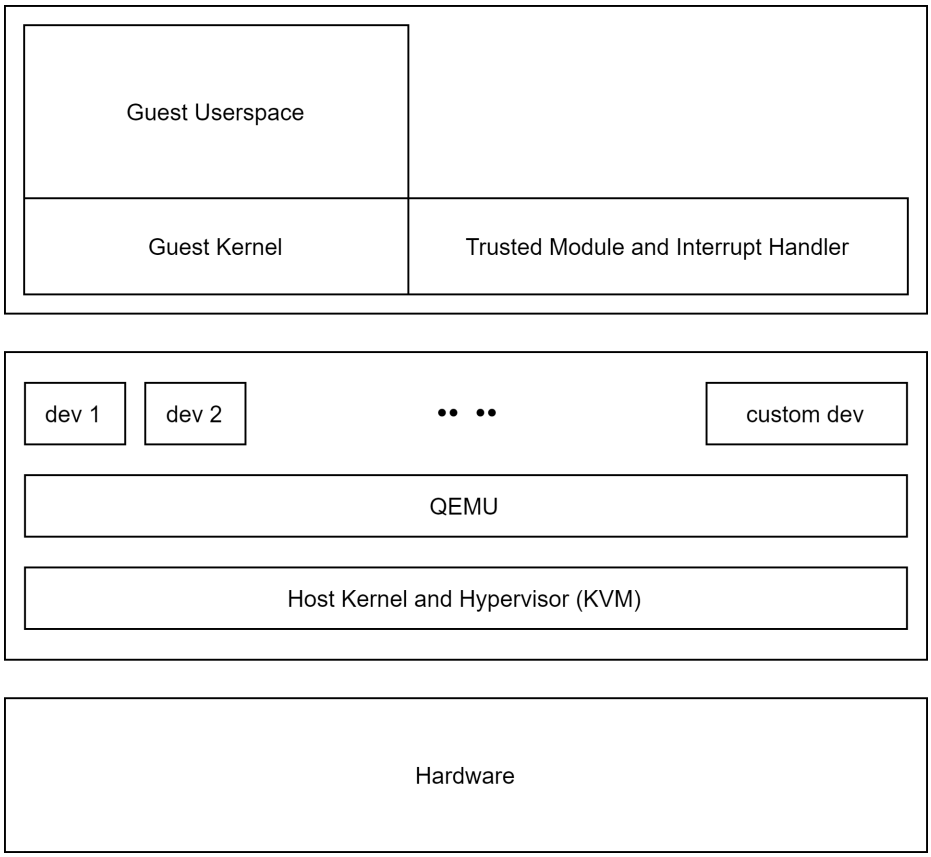

Figure 3.1: Overall system architecture.

mind.

The system's architecture is illustrated in Figure 3.1 and aims at meeting these requirements.

One way to fulfill them is by leveraging the interrupt handling mechanism. In fact, a custom virtual device is added to the QEMU source code and an interrupt handler is installed inside the guest thanks to a Loadable Kernel Module. This module is inserted in the system when it is considered to be in a safe state, e.g just after its boot and right before starting any network activities. The interrupt handler

Figure 3.2: The defense in depth approach.

is responsible for encapsulating the monitoring code. It gets executed whenever the virtual device sends an interrupt, thus forcing the kernel to stop the actions that it was performing and executing the routine associated with the interrupt. Furthermore, the injection of the interrupt is completely up to the hypervisor, which can decide when to inject it regardless of what it is happening in the guest.

The hypervisor is also responsible for protecting the monitoring code inside the guest. In particular, it must take maximum effort to protect the interrupt handling mechanism in the Linux kernel, including code and data structures. To do that, it can both prevent the guest to write to important kernel objects or restore their state just before the injection of the interrupt, so that a certain degree of stealthiness is reached.

The Stealthiness of the system is one of the many defenses that the system has to put in place. The proposed system tries to comply with a defense in depth approach like depicted in Figure 3.2. The rationale behind this method is to implement a series of security mechanisms that are thoughtfully layered to protect the system against possible attacks. This makes sense also because of the model of the attacker that is being considered. In fact, since the attacker has to perform multiple steps to conduct a successful attack, the system attempts to always detect or prevent the attacker's next step. These layers of security measures are implemented in both the

monitoring code, which tries to detect possible attacks in the guest system at the application level, and the hypervisor, which, like already said, must protect the guest agent. By extension and by implementing a genuine hypercall-based API between the guest and the hypervisor, the latter can also introduce security measures to protect other kernel objects, such as kernel code, read-only data structures, page tables, as will be discussed in depth in the following sections.

3.2 System implementation

The following sections describe the implementation details of the system. The biggest part of the hypervisor side of the system was implemented in QEMU and using the KVM API. To implement the features needed by QEMU or the guest that were not yet implemented in KVM, even the KVM kernel module was modified.

3.2.1 The QEMU PCI device

Adding a custom device to the QEMU source code is one of the most common operations. In fact, in the source tree, there is the implementation of a virtual PCI device, just for learning purposes. By studying its source code it is possible to understand some of the internals of QEMU. The virtual device source code can be found in `hw/misc/edu.c` and its documentation is in `docs/specs/edu.txt`.

QEMU uses an object-oriented approach for device emulation. Furthermore, devices are organized into a hierarchy, so that QEMU can call the initialization code of the parent object before calling the one related to the specific device. Any

PCI device has as a parent object a PCIDevice object, which holds most of the information that are common to PCI devices in general, which are also mandatory as dictated by the standard. Of course, real PCI devices use registers to store this information; these registers are modeled by QEMU as simple variables since QEMU is emulating them. These registers are mapped into the PCI configuration space and the parent object exposes functions to let the device set these parameters in its initialization function. In fact, it is possible to set the Device and Vendor ID of the device, to let a driver recognize the device, the interrupt pin (i.e specifying that the device will send interrupts), and of course BARs (Base Address Register), which are initialized as explained in the following.

One of the main responsibilities of the custom PCI device is sending interrupts to the guest virtual machine, forcing the execution of the interrupt handler, and for this reason, it is called FX (Force eXecution) device in the following. The FX device maps its device-specific registers in memory. This can be achieved by making use of the Memory API of QEMU [12] and properly setting BAR registers. As reported by the documentation, memory in QEMU is modeled as an acyclic graph of MemoryRegions objects, where each leaf of the graph represents RAM or Memory Mapped I/O (MMIO) regions. To initialize a MemoryRegion that can be used for MMIO, it is sufficient to call the function in Listing 3.1.

```
void memory_region_init_io(MemoryRegion *mr,
                           struct Object *owner,
                           const MemoryRegionOps *ops,
                           void *opaque,
                           const char *name,
```

```
uint64_t size);
```

Listing 3.1: Initialization function of a MMIO memory region

This function allows initializing a MemoryRegion, pairing it with callback functions. These functions are grouped together thanks to the MemoryRegionOps object, which basically contains function pointers containing the address of the function that must be invoked on a read or write I/O operation on that region. Once the MemoryRegion is initialized, it is possible to finally call `pci_register_bar` function, passing it the newly created memory region. The initial value of any BAR register when power is applied to the device is the size of the memory region it needs to work. Typically, configuration registers are accessed by the firmware (or the Linux kernel if configured to do so) at boot time, the initial size is read from the BARs and a safe place for each requested region is allocated, paying attention to not overlap memory regions associated to different devices. When the device driver accesses the device for the first time, memory regions are already mapped in main memory and BARs register will contain the starting address of the associated region.

To send interrupt periodically, the initialization function of the device also create a thread and a condition variable through the use of `qemu_thread_create` and `qemu_cond_init` functions. The thread executes a never-ending while loop in which:

1. It sleeps for a random amount of time
2. When it wakes up, it sends an interrupt
3. It blocks its execution on the condition variable. It will wake up again when

the device driver has acknowledged the interrupt.

Another important task of the virtual FX device is to expose a register that can be used for creating a paravirtualized communication channel between the guest's trusted module and the hypervisor, and it will be better described next.

The virtual device can be attached to a virtual machine using the `-device` option when invoking QEMU from command line. It is also possible to verify that the device is correctly attached to the virtual machine by verifying the presence of the MemoryRegion using the `info mtree` command of the QEMU Monitor and issuing the command `lspci` from the virtual machine.

3.2.2 The PCI driver

The Loadable Kernel Module inside the guest is responsible to act as a driver for the FX device. The Linux kernel offers a rich API for handling PCI devices, abstracting most of the low-level details of the PCI specification.

First of all, any PCI drivers must initialize an array of `pci_device_id`s. This data structure is used to define a (NULL-terminated) list of the PCI devices the driver supports. Typically, to initialize it, PCI drivers make use of the `PCI_DEVICE` macro, passing it the Vendor and Device ID. In this case, these parameters were written in the configuration space of the FX device by its initialization function and, of course, the driver must match them, since they are used to let the driver recognize the device and to start interacting with it. Once the array of `pci_device_id`s is initialized, the driver can make use of the `pci_driver` struct and the `pci_register_driver` function.

```
struct pci_driver {
 struct list_head        node;
 const char              *name;
 const struct pci_device_id *id_table;
 int  (*probe)(struct pci_dev *dev, const struct pci_device_id *
    id);
 void (*remove)(struct pci_dev *dev);
 int  (*suspend)(struct pci_dev *dev, pm_message_t state);
 int  (*resume)(struct pci_dev *dev);
 void (*shutdown)(struct pci_dev *dev);
 int  (*sriov_configure)(struct pci_dev *dev, int num_vfs);
 const struct pci_error_handlers *err_handler;
 const struct attribute_group **groups;
 struct device_driver    driver;
 struct pci_dynids       dynids;
};
```

Listing 3.2: `pci_driver` struct

As illustrated in Listing 3.2, the `pci_driver` struct contains a set of variables and function pointers that describe the PCI driver to the PCI core implemented in the kernel. As most of the Linux kernel data structures, also PCI drivers are maintained in a doubly-linked list, which is the purpose of the `node` field. The `name` field, instead, is used to associate a unique name to the PCI driver, in order to initialize an entry in the `sysfs` filesystem, showing information about the driver. The `id_table` field is used to let the kernel understand which devices the driver can handle, and this must be initialized with the array of `pci_device_id`s mentioned before. The `probe` function pointer is one of the most important fields. It points to a function

that is executed whenever the kernel recognizes that the driver is interested in a specific device. This happens, for instance, when calling the `pci_register_driver` function: since already existing (plugged) PCI devices are described by `pci_dev` structure in the kernel, when the latter function is called passing as argument the newly initialized driver, the kernel will check for a match between the Vendor and Device IDs between the PCI device descriptors and the entries in `id_table` field of the driver. If such a match occurs, the kernel calls the probe function, which can be used to take ownership of the device. It has to be noticed that a pointer to the PCI device descriptor is also passed as an argument to the probe function. This pointer is particularly important for the driver to later interact with the device. In fact, the FX device driver saves the `pci_dev` pointer to:

- Retrieve the interrupt line used by the device. This is useful to register a handler for the interrupts generated by the device, and it can be done through the use of the function `request_irq`, which is part of the high-level API for interrupt handling
- Retrieve a pointer to the memory region associated with a Base Address Register, through the use of the `pci_iomap` function. This is important, of course, to perform I/O operations using the device-specific registers mapped in memory.

Registering an interrupt handler simply means indicating the kernel which function should be called when the device raises an interrupt request. To verify whether the interrupts are correctly received, the interrupt handler can simply print a message in the kernel log. The implementation details about the actions performed by the interrupt handler are discussed in the following.

Now that an interrupt handler is executed whenever an interrupt is raised, it is worthwhile to understand how the hypervisor can protect the overall Linux interrupt handling mechanism from attackers. This can be done by understanding how memory is allocated to a virtual machine and how Linux and the underlying hardware manage the interrupts.

3.2.3 Virtual machine memory allocation

When using hardware-assisted virtualization, QEMU uses the KVM API to allocate memory to the virtual machine. In particular, the QEMU process will allocate pages in its virtual address space that will be used by the virtual machine as its own physical memory. The physical memory of the guest can be thought of as an array of slots, each of them represented by a `kvm_userspace_memory_region` and defined as shown in Listing 3.3. It has to be noticed that the memory allocated to the virtual machine will be shared with the QEMU process. To really attach memory to the virtual machine, the QEMU process can fill a `kvm_userspace_memory_region` structure and issue a `KVM_SET_USER_MEMORY_REGION` ioctl.

```
struct kvm_userspace_memory_region {
 __u32 slot;
 __u32 flags;
 __u64 guest_phys_addr;
 __u64 memory_size; /* bytes */
 __u64 userspace_addr; /* start of the userspace allocated
    memory */
};
```

Listing 3.3: `kvm_userspace_memory_region` struct

It is worthwhile noticing and explaining the fields of the `kvm_userspace_memory_region` struct:

- The `slot` field is used to specify a slot ID. It has to be a number less than the maximum number of user memory slots supported per VM, which can be queried using the `KVM_CAP_NR_MEMSLOTS` ioctl.
- The `flags` field is particularly relevant for this work. It allows specifying flags related to this slot. For now, only a few types of flags are available, one of which is the `KVM_MEM_READONLY` flag. This kind of flag can be used to mark the current slot as read-only. Any attempt to write the memory slot will result in a VM exit, with `KVM_EXIT_MMIO` as exit reason. This kind of flag is a perfect fit to put controls on specific memory areas of the virtual machine and it will be used in the following.
- The `guest_phys_addr` field can be used to specify the starting guest physical address of the slot. It should be noticed that slots cannot overlap in guest physical address space.
- As the name suggests, the `memory_size` field is used to specify the size of the slot, in bytes.
- Since the physical memory of the virtual machine is allocated by the QEMU process, also the corresponding `userspace_addr` must be set, allowing to

specify where the physical memory of the virtual machine is mapped in the userspace process. It should be now clear how the hypervisor can access all the physical memory of the virtual machine.

This data structure is used to interact with the kernel using the already discussed ioctl. However, from userspace, the QEMU process encapsulates all this information in a `KVMSlot` struct, which holds exactly the same data, plus other fields not relevant to this discussion.

Another detail related to memory that should be underlined and that is useful in the following, is how the hypervisor can translate a Guest Virtual Address (GVA) of a process, to its Guest Physical Address (GPA) and then to the corresponding Host Virtual Address (HVA), to let the hypervisor access the physical memory of the virtual machine when needed. The overall translation can be divided into two parts, which are GVA to GPA and GPA to HVA. To perform the first translation, the `KVM_TRANSLATE` ioctl and its related data structure can be exploited: in this way, the current address translation mechanism is used, starting from the `CR3` register, and the GVA is translated into its corresponding GPA. Now, to perform the rest of the translation, it is sufficient to:

1. Find the `KVMSlot` in QEMU containing the obtained GPA. `KVMSlot`s are maintained in an array and it is possible to iterate over them. In particular, to find the corresponding slot, it can be checked if the GPA previously obtained is within the range of addresses between the $[start\_addr, start\_addr + memory\_size]$. If this is the case, the slot was successfully found.

2. Now that the slot was found, it is possible to compute the offset of the address

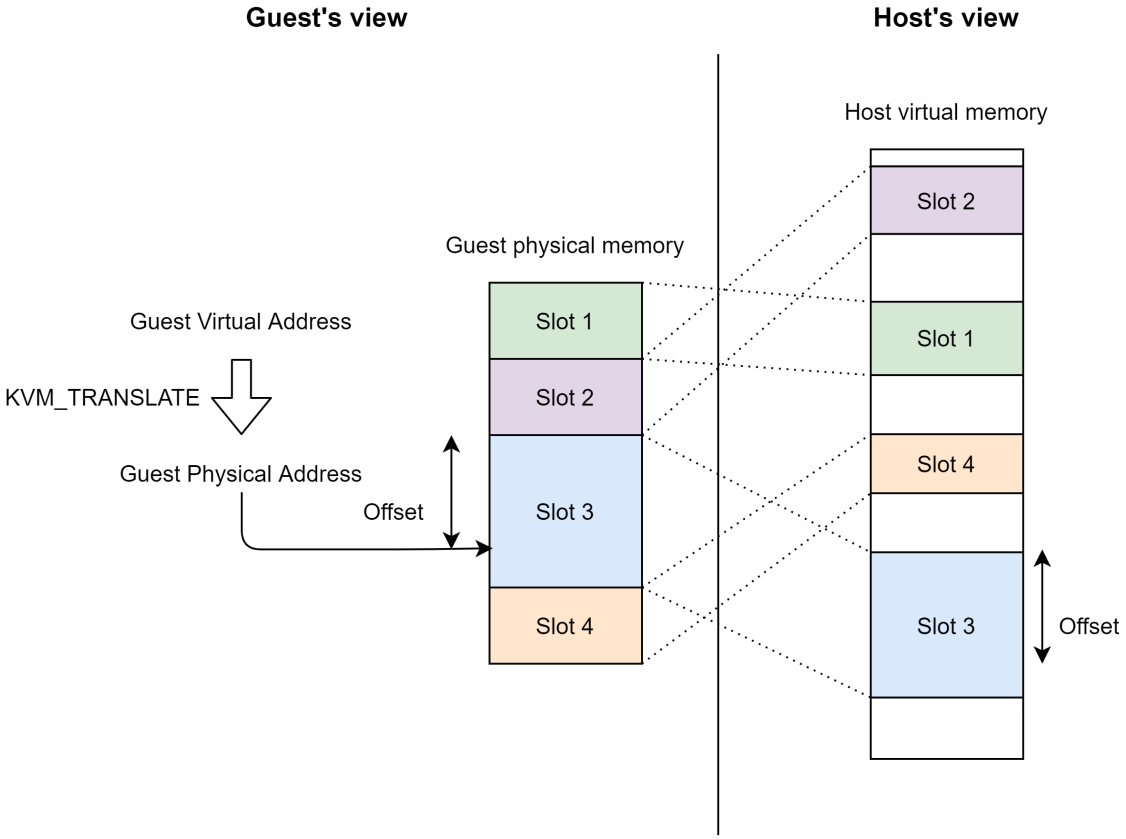

Figure 3.3: Translation of a GVA to a HVA

within the slot. This offset will still be valid in the host virtual space since it is true that slots can be anywhere in the host virtual memory, but the memory corresponding to the slot is contiguously allocated both in the host virtual space and in the guest physical space.

3. The offset and the field corresponding to the host virtual address of the slot (i.e the one with the same meaning as the `userspace_addr` in the KVM memory region structure) can be added together to obtain the HVA.

Figure 3.3 summarizes the steps needed for the translation and explain it graphically.

This translation from GVA to HVA is useful to let the hypervisor perform operations directly in the physical memory of the guest OS and will be used as an important building block throughout this work.

3.2.4 Interrupt handling in Linux

The interrupt handler plays a crucial role in the overall system. It is the core of the system implemented in the guest and it must be protected by the hypervisor. Before actually protecting it, it should be better explained how interrupt handling works in Linux, focusing on the data structures the Linux kernel relies on. GDB, QEMU Monitor and the Linux kernel source code were used to study the interrupt handling flow, with particular attention to the flow that leads to the execution of the interrupt handler of the FX device.

Certainly, the handling of external interrupts takes place both in hardware and software. For what concerns the hardware, on Intel architectures, devices cannot send interrupts directly to the CPU: they are instead connected to a specific device, the I/O APIC (Advanced Programmable Interrupt Controller). Each device is connected to the I/O APIC through an IRQ line and, in turn, the I/O APIC multiplexes the interrupts to the CPU, which has only one special PIN connected to the interrupt controller. The main purpose of the APIC is to monitor the IRQ lines and to notify the CPU of the occurrence of an interrupt. If more than one interrupts occurs simultaneously, the APIC also takes care of ordering the interrupts according to a given priority. To let the CPU execute a special routine to handle the interrupt, an interrupt vector is sent by the APIC to the CPU on interrupt notification. This

vector is used as an index of a particular table prepared by the kernel, the Interrupt Descriptor Table (IDT). The base address (and the size) of this table is maintained in a CPU register, the Interrupt Descriptor Table Register (IDTR). The IDT table is prepared by the Linux kernel and described by the symbol `idt_table`, which is defined as an array of `gate_desc` (which in turn is a `typedef` for `gate_struct`) as shown in Listing 3.4.

```
static gate_desc idt_table[IDT_ENTRIES] __page_aligned_bss;

struct idt_data {
        unsigned int    vector;
        unsigned int    segment;
        struct idt_bits bits;
        const void      *addr;
};

struct gate_struct {
        u16             offset_low;
        u16             segment;
        struct idt_bits bits;
        u16             offset_middle;
#ifdef CONFIG_X86_64
        u32             offset_high;
        u32             reserved;
#endif
} __attribute__((packed));

typedef struct gate_struct gate_desc;
```

Listing 3.4: Definition of the idt_table

The memory layout of the IDT entries is completely described by the `gate_struct` struct. Since it is a particular memory layout, for convenience the Linux kernel uses the `idt_data` struct, and then when it needs to actually copy the information in one of the IDT entries, it uses conversion functions to transform it into the corresponding `gate_struct`. The segment selector field is an index into the Global Descriptor Table and, together with the offset, can be used for starting the execution of the interrupt handling part implemented in software.

The Linux kernel tries to split interrupt handling into three different levels of abstraction:

- a part of the code must be architecture-dependent and must be aware of the details of the interrupt controller
- on top of the hardware-specific part of code, a generic IRQ handling is implemented, abstracting the underlying hardware details and separating the chip details and the real IRQ flow
- the third level of abstraction is then implemented to offer device driver writers an API for interacting with the interrupt handling subsystem. This is also useful to use the same driver on different platforms without code changes. One of the functions belonging to this level of abstraction was already used: it is the `request_irq` function used by the FX driver to register the interrupt handler.

Of course, all these three levels of abstraction contribute to the handling of an interrupt and must be protected. The adopted strategy is to extract the trace

of function calls that leads to the execution of the handler registered through the `request_irq` function. To achieve this, GDB can be leveraged. Upon registration of the handler, the module inside the guest uses the `printk` function to output the address of the handler function (notice that the `%px` format specifier is actually needed). Then, by setting a breakpoint on it, the chain of function calls can be easily obtained using the `backtrace` command. This is a starting point for studying the Linux Kernel source code following the obtained trace of functions. Listing 3.5 shows the trace of function calls.

```
[#0] 0xffffffffc0000205   mov $0xffffffffc00010ad , %rdi
[#1] 0xffffffff810b9518   __handle_irq_event_percpu
[#2] 0xffffffff810b965c   handle_irq_event_percpu
[#3] 0xffffffff810b96d3   handle_irq_event
[#4] 0xffffffff810bd231   handle_fasteoi_irq
[#5] 0xffffffff8101fba9   generic_handle_irq_desc
[#6] 0xffffffff8101fba9   handle_irq
[#7] 0xffffffff8101fba9   __common_interrupt
[#8] 0xffffffff81b57f26   common_interrupt
```

Listing 3.5: Output of the backtrace command. The first entry is the one corresponding to the interrupt handler registered with the driver

One of the core data structures of the generic IRQ handling is `irq_desc`. This data structure represents the descriptor for a IRQ. IRQ descriptors are kept in an array of descriptors, which is also called `irq_desc`, as shown below. This array can be indexed with the same vector used in the lowest level of interrupt handling. For this reason, it is worthwhile to retrieve it for inspecting the interrupt descriptor

associated with the FX device. This can be done as in the following steps:

1. The IRQ line associated with the device can be obtained by issuing the command `cat /proc/interrupts`

2. Once the IRQ line is known, the command `info pic` of the QEMU Monitor shows the interrupt vector.

Then, making use of the GDB `print` command it is possible to index the `irq_desc` array to show the content of the interrupt descriptor.

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
        [0 ... NR_IRQS-1] = {
        ... /* initialization redacted */
        }
};

struct irq_desc {
    ... /* redacted */
        irq_flow_handler_t      handle_irq;
        struct irqaction        *action; /* IRQ action list */
    ... /* redacted */
} ____cacheline_internodealigned_in_smp;
```

Listing 3.6: `irq_desc` data structure (redacted) and array definition

Once the fields are printed on the screen, the analysis can proceed further by examining the two of them reported in Listing 3.6. These are the most relevant fields of the `irq_desc` data structure and also for the overall interrupt handling mechanism. The main reason is that they allow crossing the three level of abstractions. In fact,

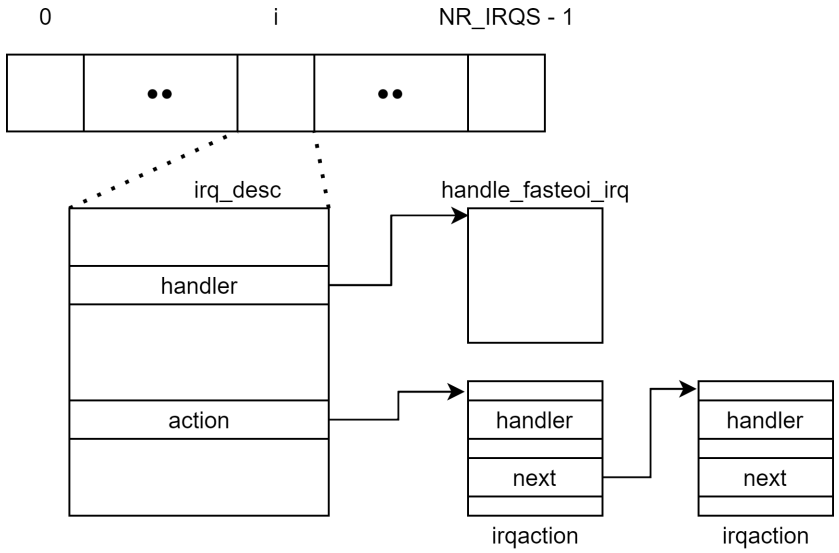

Figure 3.4: Interrupt handling main data structures

- The `handle_irq` is a function pointer. It points to the function that must be called by the lowest level of abstraction to start the generic IRQ handling. For what concerns the interrupt related to the FX device, `handle_irq` points to `handle_fasteoi_irq`.

- The `action` field, instead, is the head of a list of `irqaction`s. This list is used to call the handlers registered through the `request_irq` function and, for this reason, it is used to reach the highest level of abstraction of the interrupt handling mechanism. It has to be noticed that this list is maintained to allow the sharing of an IRQ. On interrupt, the kernel calls all the handlers stored in the list of `irqaction`s. It is up to the interrupt handler to understand if it was its device that originated the interrupt.

For the sake of clarity, these important data structures and their connections are

shown in Figure 3.4

These were the most important concepts on how the hardware and the Linux kernel cooperate to handle an interrupt. The next section will make use of this useful information to try to protect the overall mechanism.

3.2.5 Memory protection from the hypervisor

One way to protect the guest agent is by enforcing memory write protection from the hypervisor. Most of the background knowledge needed to protect the interrupt handler was already discussed and this section explains how the protection was implemented.

In general, two strategies can be implemented by the hypervisor:

- Monitoring. Some of the kernel objects are invariant from the time the handler is installed to the time the virtual machine is shut down. The integrity of these objects can be periodically checked by the hypervisor. This is a sort of passive protection.

- Write protection. The hypervisor can ensure that certain parts of memory, in particular corresponding to sensitive data structures and code, cannot be written. This is referred to as active protection.

Whenever it is possible, active protection must be used. This is because it is better to readily stop not allowed operations instead of periodically monitoring the important kernel object. In fact, it could be the case that monitoring is not enough if the monitored object is checked periodically: the attacker could modify the ob-

ject to complete the attack and then restore its state. If this happens between two consecutive checks of the monitoring code, the attacker can bypass this protection.

In this case, active protection can be implemented thanks to the `KVM_MEM_READONLY` flag of KVM memory slots. This flag is important also because it allows to implement the protection from userspace, thus avoiding writing any code inside the KVM kernel module and not enlarge the attack surface by introducing bugs inside the host kernel. As already explained, this flag prevents any write operation to a particular memory slot, but not reads. Also, code contained in such a memory slot can be executed. If a write is issued in a slot marked as read-only, a VM Exit occurs and the hypervisor regains control of the execution. The associated exit reason will be `KVM_EXIT_MMIO`. To protect a specific part of memory, the following steps can be performed:

1. The slot containing the chunk of memory to be protected can be temporarily removed. This operation can be done by issuing the same ioctl used to allocate the slot but passing zero as size. As a result, the virtual machine will now miss a part of its physical memory. Notice that this part of physical memory is still reachable using the virtual address space of the hypervisor so that anything is lost because it will be remapped soon.

2. The removed slot is then split into three parts: the chunk of memory that has to be protected is put in a slot whose size is the minimum amount of bytes needed to contain it. This slot is inserted again by computing the right physical address and associating it with the right corresponding host virtual address. The remaining part of memory belonging to the previously removed

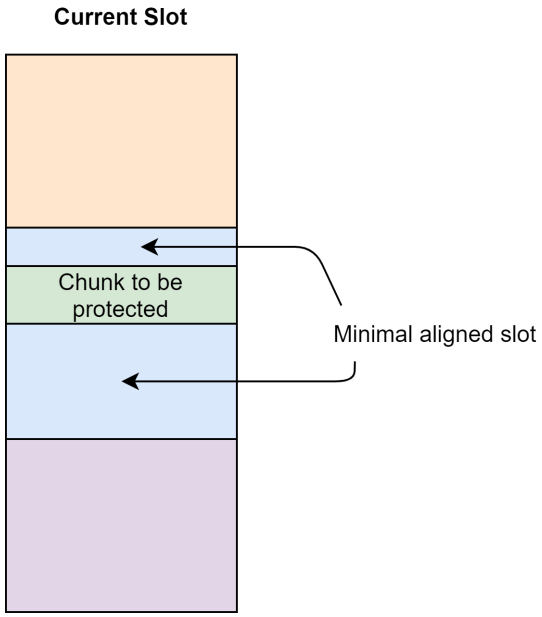

Figure 3.5: Slot-splitting technique. The current slot is represented by the biggest rectangle. Colors illustrate how it will be split. The green part is taken into account by the PMC data structure.

slot is allocated again, without any kind of flags.

Figure 3.5 summarizes these operations. This is a completely working approach. However, it has to be noted that this slot splitting technique can operate only with page granularity, but the chunk of memory to be protected can be smaller. This is due to the fact that Intel VT-x requires that memory allocated to the guest is page aligned, so that it is possible to make use of Extended Page Tables (EPT) to virtualize the guest's MMU, as already pointed out in the introduction.

This problem is solved by introducing a new data structure in the hypervisor, called `protected_memory_chunk` (PMC). The only purpose of this data structure is to protect only the specific part of memory containing sensitive information with a finer granularity. To do this, whenever the guest asks to protect a specific data

structure or part of kernel code, it passes its address and size to the hypervisor. The latter will initialize a new `protected_memory_chunk` and will add it to a linked list. Using the data kept into each `protected_memory_chunk` the hypervisor can easily discriminate when the guest is trying to write in any of them, by just adding additional checks to the `KVM_EXIT_MMIO` exit reason. In particular, the following checks must be put in place:

1. If the write address is within a PMC, the write operation is ignored. This is an event that deserves to be logged: depending on what the PMC is containing, it could be a clue for a possible attack.

2. If the write operation is not inside a PMC, but it is inside a slot previously created with the purpose of protecting another chunk of memory, the hypervisor performs the write on behalf of the virtual machine. To do this, the PMC struct contains a pointer to the KVMSlot crated for containing the PMC itself.

3. All the other cases are related to normal MMIO operations, so the default behavior is used unchanged.

Another detail has to be highlighted: how the guest module can pass parameters to the hypervisor. To solve this problem without any modification to the core of KVM, a sort of custom hypercall was implemented. This is done thanks to two key observations:

- Firstly, the QEMU process can use the `KVM_GET_REGS` and the `KVM_GET_SREGS` ioctls to read all the registers of the virtual CPU.

- Secondly, the QEMU process already handles the I/O emulation, thus a VM exit is triggered whenever an I/O operation needs to be executed.

Combining these two observations, a hypercall can be implemented. The idea is to reserve a particular area of memory to let the guest call the hypervisor: this is done by reserving offset `0x80` for the BAR register already set up for the FX device. Furthermore, the guest can put variables, addresses, and sizes inside registers using extended assembly. Listing 3.7 shows a simple way to do that. After putting the parameters of the hypercall inside registers, the guest triggers a VM exit by writing into the BAR plus the offset designed for the hypercall. The hypervisor will recognize that the I/O operation is related to that particular port, so it will read the parameters that were previously put inside the registers and will satisfy the request.

```
__asm__ volatile("movq %0, %%r8"
                 :
                 :"r"(THIS_MODULE->core_layout.base));
__asm__ volatile("mov %0, %%rax"
                 :
                 :"r"(mmio + HYPERCALL_OFFSET));
__asm__ volatile("movq $1, (%%rax)"::); /* triggering the
    hypercall */
```

Listing 3.7: Example of hypercall: putting the base address of the loaded kernel module inside register `r8`, then triggering the VM exit

Using the slot splitting technique together with the protected memory chunks, the hypervisor can apply protections to the overall kernel code and data structures responsible for interrupt handling and it can do it with the granularity needed, thus

preventing the attacker to disable or tamper with the system residing in the guest, meeting one of the initial requirements stated in the design phase. Furthermore, it is worthwhile noticing that this memory protection can be applied to many other kernel objects. For instance, the entire .text and .rodata sections of the kernel can be protected using this approach, thus making more difficult attacks which tries to patch the kernel to execute malicious code or attacks to important read only data structures, such as the IDT or the system call table.

3.2.6 Generalizing the hypercall handling

In the previous section, it was described how the guest can ask the hypervisor to perform some operations. The result is a paravirtualized channel that can be exploited for many different purposes. For this reason, it helps to add a new parameter that can be used to distinguish different requests. This is done by introducing a *type* parameter. As a consequence, the hypervisor can read this parameter by simply reading the guest register that was assigned to it and then performs actions accordingly. Furthermore, since hypercalls are similar to system calls but involves guest and host, also a custom Application Binary Interface (ABI) must be introduced. To do that, all the parameters of the hypercall are assigned to registers from `R8` to `R15`. The type parameter is of course assigned to `R8` and depending on that, the hypervisor knows which and where are all the other parameters.

Most of the introduced hypercalls are related to the protection of the agent itself or to the protection of the running kernel. It is important to notice that this newly created channel represents the opportunity to introduce many different policies and

techniques to add a layer of security to the guest system, and that is the reason why it is an important achievement for this work.

One of the most important hypercall is the already discussed memory protection hypercall. Two other different approaches that were put in place are the *Save and Compare* and the *Save and Reload* techniques.

The *Save and Compare* approach can be seen as a tool offered to the monitoring code and it is mainly used to let the latter analyze the state evolution of kernel data structures. This is because sometimes it could be hard to enforce only certain kind of accesses to them (such as write protection), and further analysis is needed. To implement this approach, two hypercalls are implemented. The first one is used to save a given data structure, passing its address and size to hypervisor like already described. The second one, instead, is used to compare the current state of a data structure against its saved state. The hypervisor will return back to the monitoring code the offsets in which it has detected changes in the data structure so that the guest agent can use them to semantically understand which fields were changed and to understand if something bad has occurred within the system.

The *Save and Reload* approach, instead, is used to protect the guest agent itself in an alternative way or together with the memory write protection hypercall. The key observation of this approach is that both code and data structures related to the interrupt handling mechanism must not be always protected. In fact, since the interrupts come from the virtual device in the hypervisor, the latter might need to only reload kernel objects that were previously saved when considered in a safe state. This reloading operation will be performed every time upon interrupt injection, thus

forcing the guest kernel to use safe objects. This approach was also explored to reach a degree of stealthiness of the system: an attacker may try to hook function pointers or patch kernel code to execute malicious code and prevent the handler to be executed, but at the moment the interrupt is injected and managed all the kernel objects are restored in their safe state.

It has to be noted that both these two approaches require a hypercall to save pieces of guest memory. To do that a new data structure was introduced in the hypervisor, the Saved Memory Chunk (SMC). This data structure contains a pointer to an area of memory placed in the heap of the QEMU process which is not allocated to the VM and containing a copy of the guest memory that has to be saved. To specify if the SMC has to be used with the Save and Reload approach, a boolean flag parameter was added to the hypercall. This way, every time an interrupt of the FX device has to be injected, the hypervisor will also scan the list of SMC, checking if the flag for reloading is set. If so, the guest memory will be replaced using the previously saved memory chunk.

Another important aspect that has to be underlined is related to the paravirtualized channel itself. This channel allows the guest agent and the hypervisor to communicate in order to implement most of the important features of the system. An attacker who already gained high privileges on the system may try to execute hypercalls to subvert the system. This, of course, needs to be denied. A simple solution is found by implementing the Finite State Machine illustrated in Figure 3.6. The communication channel starts in a *closed* state, in which hypercalls are simply discarded. Even more, attempts to call hypercalls in this state are also logged be-

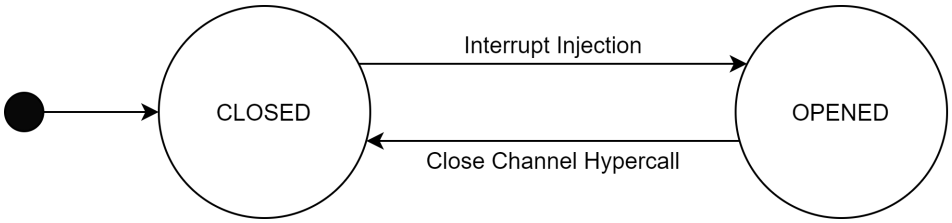

Figure 3.6: Paravirtualized channel Finite State Machine

cause they could be a clear sign of a compromised guest system. In fact, legitimate hypercalls are performed only when the communication channel is in the *opened* state. The transition between these two states occurs every time the FX device raises its interrupt. When using the Save and Reload approach, all the important data structures and code of the kernel will be reloaded at this point in time, then the interrupt handler will run safely, leading to the execution of the function defined in the FX device driver. As already pointed out, the monitoring code is encapsulated in this function. From this moment, it can make use of hypercalls to communicate with the hypervisor. Since this function is executed atomically in interrupt context, no attackers can issue malicious hypercalls while the monitoring code is executing. This is true also because the system was implemented for a virtual machine running on a single vCPU[1]. When all the needed monitoring activities are completed, the monitoring code issues a hypercall to close the channel, forcing the transition back to the closed state.

There are also other features that the monitoring code may need to perform its work. For instance, the monitoring code may have to be stateful and return

[1]To run the system with a multiple vCPUs virtual machines the hypervisor may need to stop all but one vCPU: at the end of the day vCPU are just threads and the hypervisor already has mechanisms to kick out vCPU and regain control

results back to the hypervisor. Furthermore, results may be too big to fit into registers. To solve these problems, the FX device drivers allocate pages in the guest system at initialization. The addresses of these pages are made known to the hypervisor through the usage of another dedicated hypercall. A part of these pages is reserved for the state of the monitoring code. This could be useful to implement, for instance, checkpointing, meaning that the monitoring code may remember where it arrived in performing its operations when it ran the last time. Checkpointing is also important because the monitoring code is running in atomic context and for performance reasons, it must try to not steal too much time from the virtual machine. The pages reserved for the state of the monitoring code are saved every time the communication channel is closed and reloaded on its opening so that an attacker cannot modify the state of the guest agent. The other part of the pages is instead reserved for results. The monitoring code makes use of these pages to pass information back to the hypervisor. The results are trusted only when the close channel hypercall is issued. In this way, results may be passed to other software tools to perform deeper analysis outside of the virtual machine. An example use case of this feature will be discussed in the following.

3.2.7 Accessing all kernel symbols: the double kprobe technique

Another important aspect that has not been considered until now is how the monitoring code can access all the symbols of the running kernel. It has to have access to all of them because it must protect the kernel internals as needed, indicating the

hypervisor where important data structures reside in guest memory. This is the particular way of solving the semantic gap problem proposed in this solution.

The monitoring code is registered as an interrupt handler when the module is loaded. The reason why this part of the system is implemented as a module is that this solution must be usable for already compiled kernel: usually, cloud service providers install new virtual machines on demand by getting off-the-shelf images and operating systems. For this reason, the only way to add the part of the system running in the guest is by means of LKM, which are compiled independently of the core kernel. The main point is that kernel modules are supposed to use symbols that were explicitly exported. This is done to carefully select which are the variables and the functions that are considered strictly necessary to use in modules and that allow them to perform their normal operations, such as implementing device drivers. In the past, a typical workaround used by rootkits against this exporting system was found by using the `kallsyms_lookup_name` function. This function takes the name of a symbol as input and returns the address of it, thus allowing to programmatically know where all the symbols reside. This function is appealing also for the monitoring code, which can use it for good causes, for sure. Unfortunately, since `kallsyms_lookup_name` completely broke up the exporting system, it was recently unexported to make it more difficult to be used in this way. However, the unexporting of the function is not sufficient to avoid its usage, because another workaround was found, which is based on kprobes. This technique aims at using the `kallsyms_lookup_name` function, even if it is not exported anymore. When the address of this function is retrieved, all the other kernel symbols can be resolved by

using it and that is why the target of the following method is precisely that function.

As reported in the documentation, kprobes allow to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. A kprobe can be registered in the system by means of the `register_kprobe` function, which takes as input a kprobe data structure, defined in Listing 3.8

```
struct kprobe {

    ...

    kprobe_opcode_t *addr;
    const char *symbol_name;
    ...

    kprobe_pre_handler_t pre_handler;
    kprobe_post_handler_t post_handler;
    ....

    kprobe_opcode_t opcode;
    struct arch_specific_insn ainsn;

    ...
};
```

Listing 3.8: Relevant fields of struct kprobe

In particular, kprobes can be attached directly to kernel addresses or symbols. To do that, the `addr` and the `symbol_name` members can be used, respectively. It has to be noted that if the symbol's name is specified, the registering function makes

use of the `kallsyms_lookup_name` function to retrieve the address of the symbol, and this is why kprobes become interesting. Internally, kprobes work by saving the instruction at the address specified using the `opcode` and `ainsn` members and replacing it with the `INT3` instruction [2]. The `INT3` instruction causes a breakpoint exception, resulting in a call of the debug exception handler. When execution control is taken by the exception handler, all the CPU's registers are saved. The handler is then responsible for passing the state of the processor to the function pointed by the `pre_handler` pointer, which is a function defined by the user of the kprobe. This function can examine the state of the processor for its own purposes. When it returns, the exception handler is responsible for emulating the instruction that was replaced by `INT3`. When it finishes, it calls the function pointed by `post_handler`, another user-defined function. After, the exception handler let the CPU run again at the instruction next to the probed one. Now, to reach the goal of retrieving the address of the function `kallsyms_lookup_name`, two considerations can be made:

1. A kprobe can be placed at the address of the `kallsyms_lookup_name` symbol.

2. As already pointed out if a symbol is specified instead of an address, the address of that symbol is retrieved internally by means of `kallsyms_lookup_name`.

The technique to retrieve the target address can be named *double-kprobe* technique: it consists on inserting two kprobes. The first one is used to replace the first instruction of the `kallsyms_lookup_name` with the `INT3` instruction. Now, whenever that function gets executed, the `pre_handler` function is called, which

[2] on x86_64 architectures

contains code defined in the kernel module. Then, the kernel module tries to register a second interrupt by using a second symbol name, which can be again the `kallsyms_lookup_name` symbol. In this way, the module indirectly calls the target function and the `pre_handler` of the first kprobe gets executed. Now, since the state of the processor is passed to the `pre_handler` function, it is possible to read the value of the instruction pointer, which will contain the address of the instruction next to the `INT3` instruction. Since the opcode of the latter is only one byte long, the `pre_handler` obtains the address of `kallsyms_lookup_name` by just subtracting one from the value of the instruction pointer. This is the address of the target function and now, by means of that, the kernel module is now capable of accessing all kernel symbols and it is free to show the hypervisor where important data resides in memory.

3.2.8 Module hiding

The part of the system placed in the guest also tries to be invisible, at least when trying to detect its presence using simple tools. The main point is to let any user not being able to remove the kernel module with commands like `rmmod` and to not show any the entry of the module in the list of loaded modules when using `lsmod`. To do that, the main observation to do is kernel modules are kept in a list of `module` structs. By simply removing the current module from this list, the module will not appear anymore in `/proc/modules`, the directory is used for instance by `lsmod` to list all the loaded modules. Moreover, modules are also visible in the `sysfs`. To hide them there, the corresponding `kobject` could be deleted, and that is how the

module was hidden inside the guest system. Moreover, by using the Save & Reload approach, the kernel module does not need to be in memory every time: when receiving the close channel hypercall, the hypervisor could also decide to completely delete all the code and data of the module, and reinsert them later on interrupt injection.

3.3 Customizing KVM kernel module

The previous sections described how the system was implemented in the userspace part of QEMU and the guest operating system. To improve it, however, some work has been carried out also in the KVM kernel module. The following sections describe two new features of the system that are made possible only through the usage of the part of the hypervisor residing in the host kernel. These new features are both an improvement of the current system and a starting point for generating data for further inspecting the state of the virtual machine for security purposes. It is important to say that all the modifications of the KVM kernel module are done and thought to be relatively easy and short in terms of lines of code. This is important to not increase the attack surface of the hypervisor, and, as a consequence, to still have the assumption that the attacker cannot directly attack the hypervisor itself.

3.3.1 IDTR and Control Registers active protection

As already explained, the Interrupt Descriptor table is one of the key data structures for interrupt handling. It is prepared by the Linux kernel and used by the hardware,

which accesses it through the IDTR register. If the entry of the IDT that corresponds to the pin to which the FX device is attached or the IDTR are modified by the attacker, the overall system can be compromised. Until now, to solve this problem, the IDT table was write-protected by means of the related hypercall. The IDTR register, instead, in the first prototype of the system was checked from time to time in QEMU. In particular, the system registers are read by means of the `KVM_GET_SREGS` ioctl, then it was checked if the IDTR register contained a value that is equal to the one that it had when the FX device driver was installed in the system. The check was initially performed every time on VM Exit, before executing guest code again. This approach is too simplistic: the attacker can perform its operations and modify IDTR inside the window of time consisting in two VM Exit, and the hypervisor will never notice a change in the IDTR value. This is one example of why active protection is usually way more effective than monitoring.

To actually pass from a monitoring to an active protection approach, the KVM kernel module needs to be modified. Two are the key observation to implement this:

1. As briefly discussed in the introduction, there exists a secondary execution control of the VMCS called *Descriptor-table Exiting*. It lets the hypervisor regain control whenever the IDTR is read or written.

2. There must be a way to let the guest agent tell the hypervisor to enable the active protection.

Point 1 is already a solution on its own. In fact, by simply using the `SECONDARY_EXEC_DESC` macro it is is possible to enable that secondary execution control. For point 2, the situation is different. The idea could be to implement a new hypercall, in the same

way as before. However, if the same mechanism of the implemented hypercall is reused, the userspace part will handle it and then it cannot enable the active protection directly, because this feature is more suitable to do in the kernel part. A workaround could be to implement also a new ioctl to specifically enable this new protection. However, this is not the followed approach, since a new one was explored. In fact, another way to let the guest agent communicate with the hypervisor[3] is to use custom MSR registers. As reported in the documentation [7], KVM offers the opportunity to define some custom MSRs which are completely emulated by KVM itself. The idea is to define a new MSR register and initializing it with a zero value. Then, when the FX device driver is loaded, it can enable the active protection of IDTR by simply writing a non-zero value to that register. To write to specific MSR registers one can use the `native_write_msr` Linux kernel function, which is simply a wrapper for the `WRMSR` instruction. When a non-zero value is written into that register, the active protection is enabled and this is all the FX kernel module must do. Internally, the write is directed to a variable called `idtr_pinned`, and precisely to a newly defined field of the `kvm_vcpu_arch` struct. When this variable is set to a non-zero value, all the subsequent writes are ignored, to avoid to turn the protection off by rewriting a zero value in the MSR register. Then, by simply adding an `if` statement to the `emulator_set_idt` function defined in the x86.c file, it is possible to check whether `idtr_pinned` contains a value different from zero. If so, the function returns without setting the IDTR register. Since all attempts to write the IDTR register end up in a call to this function because of the activation of the

[3]this time directly with the kernel part

Descriptor-table Exiting secondary execution control, the guest cannot modify the IDTR register anymore during its execution, thus achieving what was the initial goal.

It is interesting to notice that this mechanism can also be applied to other relevant registers to implement further protections, not strictly related to the guest agent but to the running guest kernel. For instance, other protections that were implemented involve the CR0 and CR4 control registers. A control register is a processor register which changes or controls the general behavior of a CPU. Usually, each bit of the register are used to enable or disable certain features, and that is the difference with the IDTR register, which instead contains an address (and the size) of the IDT. To deal with them, the pinning of certain bits was implemented. It consists in forcing that only certain bits, once pinned, cannot be flipped. To implement this mechanism a slightly different approach was used. In particular, for each control register two custom MSRs are defined. The first one only supports reading operations and is used to tell the device driver which bits can be pinned. The other one, instead, is used to actually pin the bits. As in the example of the IDTR register, this latter MSR starts with a value of 0 and only allows writes that set the bits allowed, specified in the read register as just discussed. As usual, to complete the active protection there is the need to control the guest exiting and to add some logic to the functions that KVM internally uses to set bits in these registers. To be precise, the way the hypervisor regains control is by means of the Guest/Host Masks and Read Shadows for CR0 and CR4 as reported in the Intel documentation. To add the new logic concerned with pinning certain bits of these control registers,

the related functions are `kvm_set_cr0` and `kvm_set_cr4`. This allowed to introduce a paravirtualized version of the control register pinning inserted in the Linux kernel by Kees Cook and briefly described here [17]. There are many other registers and bits that can use these ideas to protect the running kernel. Tests were performed with the CR0.WP and CR4.SMEP bits, which are a typical target of rootkits and exploits, but a similar protection could be implemented to protect the entry point of system calls (LSTAR MSR) and the bit that turns on the No-Execute protection (EFER.NXE), for instance.

3.3.2 Implementation of an Access Log in KVM

Many other features of the system could be implemented by modifying the KVM kernel module, but there is one in particular that deserved to be explored and implemented. Also, implementing this allows to pave the way for implementing many other approaches in a fairly similar way.

As mentioned earlier in Chapter 2, in Intel hardware-assisted virtualization, the processor makes use of a second level page table to virtualize the guest MMU, which is called EPT in Intel terminology. The idea is to use the bits of the entries of the second level page tables to enforce certain kind of policies in the guest system or to monitor its behaviour. To explore this opportunity, a deep dive in the KVM code concerned with the management of the second level page tables was needed. Recently w.r.t the time of writing, Google researchers have introduced in the mainline KVM a new way to make use of the hardware features for virtualizing the guest MMU, and they called it Two Dimensional Paging (TDP) MMU [6]. This newly introduced

part of code is much more cleaner than the old implementation and that is why the implemented system makes use of it. One of the main problems to solve is to use the EPT bits is to iterate over the second level page tables. To do that, the key data structure to use is the `tdp_iter` struct, as shown in Listing 3.9. When used together with the `for_each_tdp_pte` it is possible to access down to the 4K page table entries, where it is possible to set or unset bits like Read, Write, Execute, Accessed and Dirty bits, as defined by the Intel documentation.

```
struct tdp_iter {

        gfn_t next_last_level_gfn;
        gfn_t yielded_gfn;
        tdp_ptep_t pt_path[PT64_ROOT_MAX_LEVEL];
        tdp_ptep_t sptep;
        gfn_t gfn;
        int root_level;
        int min_level;
        int level;
        int as_id;
        u64 old_spte;
        bool valid;
};

#define for_each_tdp_pte_min_level(iter, root, root_level,
    min_level, start, end) \
        for (tdp_iter_start(&iter, root, root_level, min_level,
            start); \
              iter.valid && iter.gfn < end;                    \
              tdp_iter_next(&iter))
```

```
#define for_each_tdp_pte(iter, root, root_level, start, end) \
        for_each_tdp_pte_min_level(iter, root, root_level,
           PG_LEVEL_4K, start, end)
```

Listing 3.9: `tdp_iter` struct and related macros

The `tdp_iterator` will perform a pre-order traversal of the paging structure towards the mapping of the `next_last_level_gfn` guest frame number. In particular, it is possible to set the minimum level the iterator should iterate over thanks to the `min_level` field. The `level` field, instead, contains the level in the paging structure where the iterator is currently on. Other two particularly relevant fields are `sptep` and `old_spte`, which are a pointer to the current descriptor and a snapshot of the value contained in it, respectively. Through the usage of the `spte` pointer it is possible to actually set or unset bits in the descriptor. To reach the goals of this work, only 4KB page descriptors were modified. Finally, another important field is the `valid` one, which can be used to assess whether the iterator walked off the end of the paging structure.

For the purposes of this work, a good candidate bit to use as an example is the Access bit. This bit is automatically set by the hardware when the corresponding guest physical page is accessed, both in read and write operations. Currently, the KVM API only supports a set of ioctls for dirty logging, but no access log is implemented. The idea is to actually implement it, so that other software tools can use it to implement further monitoring activities in the system. The design of the new ioctls used to start logging and to retrieve the log are pretty similar to the ones related to dirty logging. Specifically, the two new ioctls are:

- `KVM_CLEAR_ACCESS_LOG`: this ioctl is used to iterate over all guest page descriptors to reset the access bit.

- `KVM_GET_ACCESS_LOG`: this ioctl takes as input a memory slot id and then it returns a bitmap array, where each of the bits correspond to a page and it is set to one if and only if that page was accessed.

There are many use cases that may be implemented using this new feature. For instance, when used in conjunction with the dirty log ioctls, the access log ioctls can be used to determine which are the pages that were accessed by read operations. Moreover, the access log was really useful to understand which pages were accessed when handling the interrupt of the FX device and to make sure that everything was actually protected. Another relevant example could be the implementation of a sort of *verifier* for kernel modules: the FX kernel modules could ask the hypervisor to clear the access log when a new module is being inserted in the system. The hypervisor will then be told to retrieve it after the module was loaded. By inspecting the set of accessed pages, the latter can try to understand and decide whether to trust or not the loaded module. These are just few basic examples and use cases related to the access log, but were not implemented, because the main objective of this part of work was to try to make use of the second level page tables bits.

The same reasoning and almost the same code can be reused to make use also of the Execute bit of the second level page tables, which can be used to impose further constraints to the guest system also for security purposes.

3.4 A simple use case: processes, files and networking monitoring

This section is dedicated to a brief description of what the monitoring code can actually do. As already described at the beginning of this chapter, this system tries to comply with a defense in depth approach. The monitoring code is responsible for the security of userspace applications. To monitor what is going on inside the guest system, one of the most used forensics tools can be reimplemented also in the proposed solution: process monitoring. To do this in a trustworthy way, the guest agent will use kernel information to monitor userspace processes. This is because the guest userspace cannot be trusted at any time, since compromising it is the first step of the attacker. In addition, retrieving information from the kernel and not from userspace tools help to deal with userspace rootkit. This helps to comply with the defense in depth approach. Furthermore, the system tries to protect itself and to be resilient to kernel side attacks, as discussed in the following.

Monitoring processes inside the guest system is one of the most explored ways to understand if everything is running as expected within the system and to discover malware or signs of intrusion. Many different approaches were implemented in the past. Virtual Machine Introspection techniques deserves particular attention to this discussion. Many of the VMI approaches relies on the knowledge of the offsets of the `task_struct`'s fields, which is the Linux process descriptor. However, since the Linux kernel is constantly evolving, it may be the case that systems developed with this idea in mind are not properly working anymore after a new kernel release.

Furthermore, in modern Linux kernels, important data structures have a randomized layout [15]: typically, the compiler lays out in memory the fields of a C structure in order of declaration, but in modern kernels the layout in memory of these structs is automatically randomized. This is done to make exploits that rely on overwriting an exact field of a C structure much more difficult. As it could be imagined, this is also an important complication that VMI techniques have to take into account to retrieve the list of processes of the running virtual machine.

In the proposed system, instead, things are much more easier. As long as the guest agent is protected, the monitoring code can use all kernel functions and data structures to retrieve not only the list of processes, but also which files were opened and which sockets are in use. Figure 3.7 briefly describe how that is achieved. As already mentioned, the process descriptor in Linux is described by the `task_struct` structure. Process descriptors are kept in a doubly linked list, thus it is easy to iterate over it and retrieve useful data on each process. To actually iterate over it, the `for_each_process` macro can be used, starting the iteration from the `init_task`. From there, it is possible to generate a first simple output by using the `pid` and the `comm` field of each process descriptor, containing the process identifier and the command name associated with the process. Digging deeper in the fields of the `task_struct` it is possible to identify those that are the key data structure to inspect the file descriptors of each process. Particular attention must be paid to `file` and `inode` structs. From the process descriptor it is possible to follow the `files` and the `fdt` pointers, which are pointers to `files_struct` and `fdtable`, respectively. The actual table of opened file can be accessed through the `fd` field of the `fdtable`

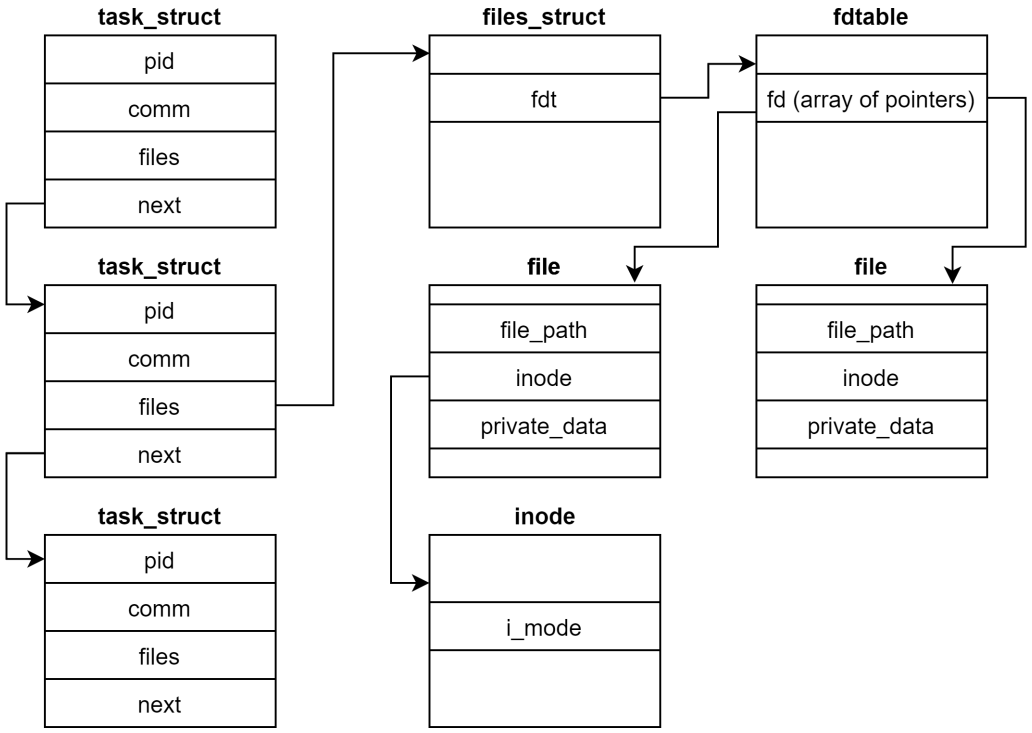

Figure 3.7: Main data structures for process, files and networking monitoring

struct, which is an array of pointers to `file` data structures. The latter is one of the most important data structure in the Linux kernel. It represents an open file in the system. In fact, every opened file has an associated structure of this type in kernel space. The `file` structure has many important fields, among which the `f_path`, containing the path in the file system of the opened file, and a pointer to the corresponding `inode` structs. An `inode` represents the actual file on the disk and it is important to retrieve the associated data. For the purposes of this work, the `i_mode` can be used to understand if the file is a socket. If this is the case, the corresponding `private_data` field inside the `file` struct pointing to that inode contains pointers to networking related data structure (specifically the `socket` and

`sock` data structures). From there, it is possible to understand, for instance, if the socket is used for a TCP connection and which port is using. Listing 3.10 shows an example of the results obtained extracting data from the discussed data structures.

```
apache2 [205]
    fd 0        /dev/null
    fd 1        /dev/null
    fd 2        /var/log/apache2/error.log
    fd 3        socket, source addr 0.0.0.0, source port 80
    fd 4        pipe:[10944]
    fd 5        pipe:[10944]
    fd 6        /var/log/apache2/other\_vhosts\_access.log
    fd 7        /var/log/apache2/access.log
bash [211]
    fd 0        /dev/ttyS0
    fd 1        /dev/ttyS0
    fd 2        /dev/ttyS0
```

Listing 3.10: Processes, files and networking listing

As it can be noticed from the above Listing, for instance, there is an Apache 2 web server and a bash running inside the virtual machine, with PID 205 and 211, respectively. The web server is listening for incoming connections on port 80, as it can be seen from file descriptor 3. The bash process, instead, has its stdin, stdout and stderr directed to a terminal.

A few other aspects need to be highlighted. First of all, as already discussed, all the results and data collected by the monitoring system are passed to the hypervisor through the pages of guest memory allocated at initialization. The hypervisor will

trust and collect the content of those pages only at the moment the communication channel goes to the closed state. Moreover, it has to be noted that the monitoring code is using internal kernel functions and symbols to perform its work. To be sure that it is using code and data structures that are not maliciously modified by an attacker within the guest system, all of them are either protected through the memory write protection hypercall or reloaded using the Save and Reload approach. Last but not least, it has to be considered that also the guest page tables and the entries related to the symbols used by the monitoring code must be protected. Again, this problem was solved thanks to the Save and Reload approach: all the translations of the kernel symbols needed by the guest agent are retrieved at the initialization through the usage of `pgd_offset`, `p4d_offset`, `pud_offset`, `pmd_offset` and `pte_offset_kernel` functions, then they are saved and reloaded as already discussed. This is crucial to ensure that the monitoring code is not misled by attackers that tries to manipulate the guest page tables with malicious intentions. As a result, the monitoring code is safe and its result are reliable even if the attackers has already attacked the system.

Chapter 4

Evaluation

The most effective way to test the system is to attack it. To do so, some of the most used kernel rootkits techniques were explored to better understand how attacks are typically implemented. In particular, great material for attack techniques was found in two GitHub repositories which are meant to show rootkits methods for learning purposes. These two rootkits are Diamorphine [3] and Reptile [16] (which comes with particularly interesting slides [1] from the authors).

To actually attack the system, a Loadable Kernel Module was implemented, as it is the de facto standard way to implement rootkits. The Loadable Kernel Module has a parameter that can be used to specify the attack type it has to perform. The following section goes through a possible attack scenario, describing in detail the steps performed by an attacker and how the proposed system can detect or even prevent them from being performed successfully.

4.1 An attack scenario

As discussed in the threat modeling section, the attacker modeled in this work is an external one. Typically, when attacking a system, the attacker tries to gain access to a shell to execute commands inside the victim machine. This is called Remote Code Execution, or RCE for short. Most of the time, it is achieved by exploiting vulnerabilities in the services offered by the virtual machine, e.g a web server, and forcing it to execute a reverse shell, where the victim machine connects to the attacker one, waiting for incoming commands. Usually, a web server runs as a particular user inside the system and the attacker has to upgrade its credentials running commands as this user. The proposed system may detect this situation by checking two different aspects:

- Usually, shell processes do not run with the credentials of the user reserved for the web server. For this reason, it may be possible to use this as a sign of intrusion.

- In addition, also network connections can be used. Since the attacker forced the web server to execute a reverse shell, an unexpected outbound connection can be noticed and treated as a sign of intrusion.

The proposed system can easily implement these kinds of checks. This could be done by stating a set of rules and checking data that was retrieved from the virtual machine, like the process list described earlier, against them. For the moment, the system limits itself to simply detect this kind of situations, but it could also try to block the attack by killing the process, for instance. However, this is not the major

concern of this work and it was not implemented, even if it could be implemented in the future.

The attacker's next step is to upgrade its credential. Typically, the aim is to gain *root* access, and this can be achieved step by step by upgrading the credentials to other users of the system first. There is a multitude of ways to accomplish this, but in general attackers have three different ways of doing that:

- Attacking root processes and trying to let them execute code to the benefit of the attacker
- Attacking executable files owned by root with the `SUID` bit enabled and again trying to let them execute code to the benefit of the attacker
- Attacking the underlying kernel

The activities carried out by the attacker inside the system are still monitored through the use of the guest agent, and there are multiple other ways to detect signs of intrusion. Supposing that the attacker is gaining root access anyway in some way, she will then try to obtain persistence of the attack, usually by inserting a Loadable Kernel Module, i.e a rootkit, and running code at Ring 0. There is a multitude of actions the attacker can take to try to subvert the proposed system or to easily regain root access whenever it is needed. Many of these attack steps where implemented to test the robustness of the system.

One of the first objectives could be to prevent the monitoring code to execute. This can be done in at least two ways:

- The first one could be the attacking of the function pointer pointing to the

handler function that encapsulates the monitoring code. As the the careful reader may remember, that pointer resides in the `irqaction` data structure. The attacker can try to make it point to an empty function, thus completely disabling the overall system.

- Another kind of attack could be modifying the entry of the IDT that leads to the execution of the FX interrupt handler to not call that function anymore. This can be done by modifying that entry in the current IDT or creating a fake IDT and making IDTR point it, or changing the page tables to point to the new fake one.

These kind of attacks were proved unsuccessful against the proposed system. This is mainly due to the memory write protection hypercall and the Save & Reload approach. The IDT is successfully marked as read only, important data structures' state are saved and reloaded just before the injection of the interrupt, modifications of IDTR are not possible thanks to the pinning of that register. Furthermore, also all the involved page tables entries are reloaded before executing the monitoring code. In conclusion, all techniques previously discussed were very useful to prevent the attacker tamper with the system.

In any case, the attacker could still try to not subvert the system but to install a rootkit to regain root access easily. To test whether the proposed system also increased the security level of the running kernel, some of most used rootkit techniques were used, inspired by Diamorphine and Reptile. One of the most used way to attack the kernel is to modify the `sys_call_table`. It is a data structure of function pointers, one for each possible syscall, used by the kernel to execute what

the user has asked. The entire `sys_call_table` should not change over time, and that is why the kernel mark it as read only. However, attackers running code at Ring 0 can easily circumvent this policy. There are at least two ways to do that: walking the page tables and modifying the bit to allow write access or temporarily disable the CR0.WP bit. Both these two kind of attacks were unsuccessful: the `sys_call_table` is placed in a PMC, thus modifying the page tables is useless, since writes to that part of memory are not allowed by the hypervisor. In addition, the CR0.WP bit is one of the pinned bits of CR0, and any attempt to modify it after its pinning is denied and logged.

These were the most relevant examples that needed to be discussed. Many other attacks could be implemented, but the point is that the memory protection and the save & reload approach turned out to be effective against most of them, proving the robustness of the system. However, one of the most difficult attacks to deal with is the modification of the guest page tables. The previous discussion took into consideration the modification of the W bit in the page tables, but not what would happen if the translation was made pointing to another physical page. For instance, the translation of the symbol `sys_call_table` can be modified to point to a page containing a fake table, having some of the entries modified for malicious purposes. To detect this situation, the monitoring code can also check the integrity of the page tables and of the translation of relevant symbols using the Save & Compare approach. Unfortunately, this allows to implement only a monitoring approach. To actually implement an active protection of the guest page tables, hardware support is needed, and Intel is beginning to work on it, as it will be discussed in Chapter 5.

4.2 Hypercall performance

Other relevant aspects to be evaluated are related to the performance of the system. One of the most important performance metrics to measure is the time needed to perform a hypercall, which is a crucial building block used by the monitoring code, allowing guest and host to communicate. To evaluate the time spent to perform a hypercall, three more of them were created just for this purpose. In particular, these are:

- Start timer hypercall: it is used by the FX device driver to start a timer inside the hypervisor.
- Empty hypercall: like its name suggests, it does nothing, since the objective here is to only measure how long it takes to exit the guest, returning to host userspace.
- End timer hypercall: it is used by the FX device driver to stop the timer inside the hypervisor.

The measurement must only take into account the time spent executing the empty hypercall, avoiding measuring the time for starting and stopping the timer. To solve this problem, a simple averaging strategy is used: the device driver starts the timer, then it performs n empty hypercalls before stopping the timer. To compute the time spent for a single empty hypercall, the overall time measured by the timer is divided by the number of empty hypercalls performed. Of course, the error of this measurement decreases more and more as n increases, until it can be con-

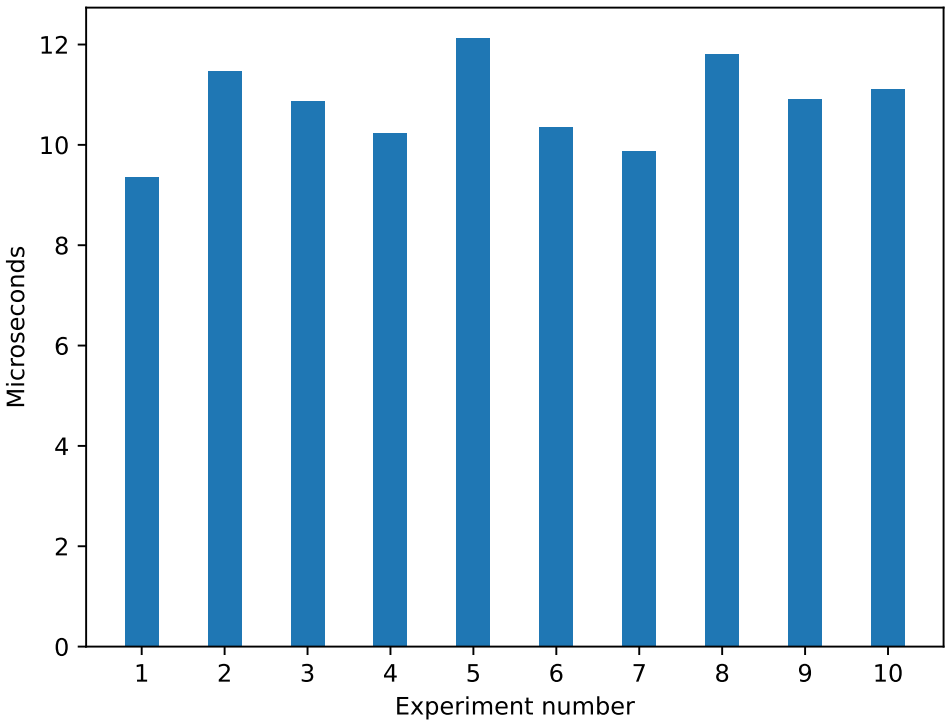

Figure 4.1: Bar chart of hypercall time measurement experiments. The experiments were done with $n = 100000$

sidered negligible. The obtained results are shown in Figure 4.1. The average time spent is 10.912 μs.

Chapter 5

Conclusions

The isolation guarantees offered by virtualization technologies were exploited in many different ways to add a layer of security to a running guest operating system. This work proposed a system developed using a *hybrid* approach, implementing a part of it both in the hypervisor and guest kernel. This allowed to introduce a hypervisor-protected guest agent inside the running virtual machine, capable to retrieve data from the guest kernel and to make them being available to the hypervisor. All information retrieved inside the guest are passed to the hypervisor by means of a secured paravirtualized channel, used to implement a hypercall-based API between the guest and the hypervisor. Some of the most important hypercall that were implemented were used to protect the guest agent or to enforce security measures on important kernel data structures. In particular, these hypercalls allowed to implement the Memory Protection Hypercall, the Save & Compare and the Save & Reload approach. Then, it was shown how the guest agent, which is a LKM inside the guest, can retrieve the address of any of the symbols of the running

kernel using the double kprobe technique. This was useful to let the guest agent indicate to the hypervisor where important objects reside in memory and it is the means through which the semantic gap problem was solved. It was also shown, through the implementation of an attack scenario, how the system tries to detect malicious activity inside the guest. In particular, this was possible by implementing a monitoring agent capable to retrieve the list of processes, their open files, and their network connections.

Future work will try to build on top of the proposed system. In particular, the proposed system can be used as a framework to collect new data from the running guest kernel and to build another software tool that can examine them in real-time. This new part of the system could potentially run inside the hypervisor or even in another virtual machine, to enhance its security even more. Then, the next step is to actually block the attack once it is detected. This is achievable by killing guest processes when malicious activities are detected. To actually improve the detection part, the system can implement a rule engine: data will be checked against a set of rules, that are made up of a condition and an action. Whenever the condition is met, the corresponding action will be performed inside the guest by the guest agent. In the near future, Intel will also release the Hypervisor-Managed Linear Address Translation, HLAT for short, documented in Chapter 6 of the Instruction Set Extensions and Future Features Programming Reference [5]. HLAT will allow the hypervisor to enforce guest virtual address translations. The objective of this new hardware feature is twofold:

1. It is used to ensure page table entries integrity

2. It is used to ensure integrity of code and data of the running kernel, without using the EPT bits

The hypervisor will take the ownership of a part of the guest page tables. It will be able to use their bits directly to enforce specific kind of memory accesses. By taking the ownership, the guest will no longer be able to modify them and that is why this hardware feature allows to implement both point 1 and 2. The proposed system could try to use this new Intel extension to further improve itself in the future.

List of Figures

Listings

Bibliography

[1] *Advanced Linux kernel Rootkit Techniques*. URL: https://vx-underground.org/papers/h2hc/.

[2] *CVE-2015-3456*. 2015. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-3456.

[3] *Diamorphine Rootkit*. URL: https://github.com/m0nad/Diamorphine.

[4] *Generation of Debian rootfs for multiple architectures*. URL: https://github.com/jubinson/debian-rootfs.

[5] *Instruction Set Extensions and Future Features Programming Reference*. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf.

[6] *Introducing the TDP MMU*. URL: https://lwn.net/Articles/832835/.

[7] *KVM-specific MSRs*. URL: https://www.kernel.org/doc/html/latest/virt/kvm/msr.html.

[8] *LibVMI*. URL: https://libvmi.com/.

[9] *Linux Kernel Runtime Guard*. URL: https://www.openwall.com/lkrg/.

[10] *Linux kernel source tree*. URL: https://github.com/torvalds/linux.

[11] Peter M. Mell and Timothy Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA, 2011.

[12] *QEMU Memory API documentation*. URL: https://qemu.readthedocs.io/en/latest/devel/memory.html.

[13] *QEMU official mirror*. URL: https://github.com/qemu/qemu.

[14] Avi Qumranet et al. "KVM: The Linux virtual machine monitor". In: *Proceedings Linux Symposium* 15 (Jan. 2007).

[15] *Randomizing structure layout*. URL: https://lwn.net/Articles/722293/.

[16] *Reptile Rootkit*. URL: https://github.com/f0rb1dd3n/Reptile.

[17] *Security Things in Linux kernel v5.3*. URL: https://outflux.net/blog/archives/2019/11/14/security-things-in-linux-v5-3/.

Acknowledgments

Lorem Ipsum sit dolor amet