UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

Fuzzing and analysis of closed-source binaries with KVM for vulnerability discovery

Autor
David Mateos Romero (alumno)

Director
Gabriel Maciá Fernández (tutor)

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, junio de 2023

Fuzzing and analysis of closed-source binaries with KVM for vulnerability discovery

Autor
David Mateos Romero (alumno)

Director
Gabriel Maciá Fernández (tutor)

Fuzzing y análisis de binarios de código cerrado con KVM para el descubrimiento de vulnerabilidades

David Mateos Romero (alumno)

Palabras clave: *fuzzing*, emulación, rendimiento, vulnerabilidades, cadenas de Markov, análisis de programas

Resumen

El *fuzzing* es un método que tiene como objetivo descubrir errores y vulnerabilidades en software, comprobando su comportamiento frente a entradas generadas automáticamente. En los últimos años, el fuzzing ha sido ampliamente utilizado y estudiado. Su eficacia es proporcional a la velocidad de ejecución del *fuzzer* y del programa objetivo. Sin embargo, esta se reduce cuando no se dispone del código fuente del programa objetivo, ya que se suele hacer uso instrumentación dinámica y emulación, lo que conlleva una pérdida de rendimiento considerable.

Este proyecto tiene el objetivo de diseñar y desarrollar KVM-FUZZ, un *fuzzer* para binarios guiado por cobertura de código que incluye su propio hipervisor y kernel basados en KVM. Hace uso de aceleración por hardware para lograr una velocidad de ejecución casi nativa e incluye un mecanismo de *snapshots* que permite restaurar el estado del programa objetivo miles de veces por segundo.

Además, el proyecto incluye una propuesta de un modelo de Markov que describe el comportamiento de ejecución del programa objetivo, tanto en espacio de *kernel* como de usuario. Dicho modelo se puede utilizar para analizar el programa objetivo, detectar cuellos de botella tanto en el binario como en el *fuzzer*, y realizar un estudio probabilístico del tiempo de ejecución estimado dados cambios en el binario y/o el *kernel*.

Fuzzing and analysis of closed-source binaries with KVM for vulnerability discovery

David Mateos Romero (student)

Keywords: Fuzzing, Emulation, Performance, Vulnerabilities, Markov chains, Program analysis

Abstract

Fuzzing is a method for discovering bugs and vulnerabilities in software, testing its behaviour against automatically generated inputs. Over the past few years, fuzzing has been broadly used and studied. Its effectiveness is proportional to the execution speed of the fuzzer and the target program. However, this is reduced when the source code of the target program is not available, since dynamic instrumentation and emulation are often used, which entails a considerable loss of efficiency.

This project has the goal of designing and developing KVM-FUZZ, a code coverage guided binary fuzzer that includes its own KVM-based hypervisor and kernel. It makes use of hardware acceleration in order to achieve near-native execution speed, and includes a snapshots mechanism that allows the target program to be restored thousands of times per second.

Furthermore, the project includes a proposal of a Markov model describing the execution behaviour of the target program, both in kernel and user space. Such a model can be used to analyse the target program, detect bottlenecks in both the binary and the fuzzer, and perform a probabilistic study of the estimated execution time given some changes to the binary and/or the kernel.

D. **Gabriel Maciá Fernández (tutor)**, Profesor del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Fuzzing and analysis of closed-source binaries with KVM for vulnerability discovery***, ha sido realizado bajo su supervisión por **David Mateos Romero (alumno)**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 26 de junio de 2023 .

El director:

Gabriel Maciá Fernández (tutor)

Agradecimientos

A Gabriel Maciá Fernández, que más que un tutor es un amigo.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation and context of the project

The discovery of vulnerabilities in software is an increasingly relevant critical security problem. Google Project Zero conducted an analysis of actively exploited zero day vulnerabilities (0-days) in 2021 [3]. 58 actively exploited security flaws were detected, more than double the year before, of which 39 were memory corruption vulnerabilities, including classes such as use-after-free, out-of-bounds read or write, buffer overflow and integer overflow. These vulnerabilities were found in known projects, including browsers such as Google Chrome, WebKit (Safari) or Internet Explorer, operating systems such as Windows, iOS/macOS and Android, and applications such as Microsoft Exchange Server. On the other hand, CVE Details lists 421 different memory corruption vulnerabilities in commonly used software disclosed in 2022 [4].

All this suggests that nowadays vulnerabilities are present in all software, and finding them is of great interest to both developers and security researchers. However, manually discovering vulnerabilities is a difficult task, specially in complex pieces of software.

Fuzzing is an automated method that has been studied as an effective way of discovering vulnerabilities. The main idea is to run the target program with automatically generated inputs with the goal of discovering bugs. An example to illustrate its impact is OSS-Fuzz [5], a project from Google which uses different fuzzing tools continuously to automatically find and report vulnerabilities. Since its creation, it has found thousands of security flaws in hundreds of open source projects.

In the process of fuzzing, in order to analyze the behaviour of the target program with each input, it is common to introduce small pieces of code when compiling through a process called instrumentation. However, this is not possible in binary-only fuzzing, i.e., when the source code of the target program is not available, presenting additional challenges. The most common solution is to use introduce runtime instrumentation with QEMU, but this results in a significant performance loss.

1.2 Project goals and achievements

This projects aims at improving the performance of binary-only fuzzing by avoiding the issues involved in emulating and introducing runtime instrumentation with QEMU. We present KVM-FUZZ, a x86-64 binary fuzzer based on code coverage that implements its own hypervisor and kernel and makes use of different hardware acceleration mechanisms. The use of a hypervisor and virtual machines allows to isolate the target program from the host system, as well as make use of snapshots restoration and different high-performance code coverage methods. On the other hand, the use of its own kernel allows system calls emulation, thus achieving a speed higher than native in many cases. These features allows KVM-FUZZ to achieve greater performance and execution speed than similar alternatives.

Furthermore, taking advantage of the kernel and breakpoint-based coverage, we are able to trace the execution of the target program both in kernel and user space. With such traces, we propose a Markov model describing the execution graph of the target program across different runs. Since it is based on real execution traces performed in virtual machines, it is a Control-flow Graph Recovery method much more powerful than other static approaches. In addition to using the model to analyse the target program behaviour, when coupled with traces timestamps, it allows us to perform a probabilistic study of the estimated execution time given some changes to the binary and/or the underlying kernel.

In summary, we make the following contributions:

- We present the design and implementation of KVM-FUZZ, a coverage-guided, binary-only emulation and fuzzing tool that makes use of different hardware acceleration mechanisms to outperform similar state-of-the-art fuzzers.
- We show a Markov model that describes the execution graph of the target program, both in user and kernel space, and helps to analyze its runtime behaviour.

1.3 Structure of the document

The structure of the document is as follows:

1. In Chapter 1, Section 1.4 presents and details the theoretical contents needed for the correct understanding of the project. This includes

processes like fuzzing, technologies like QEMU, Kernel Virtual Machine (KVM) and operating systems, and mathematical concepts like Markov chains. Section 1.5 studies the state of the art in binary fuzzing, focusing on the fuzzers AFL++ and NYX.

2. Chapter 2 presents the planning of the development of the project and an estimate of its costs.
3. Chapter 3 details the requirements of the project in Section 3.1, and analyses some of the design decisions that arose when trying to meet them in Section 3.2.
4. Chapter 4 presents the design of the different components of KVM-FUZZ, including the interface of every class and their associated UML diagrams.
5. Chapter 5 shows the implementation details of the different classes and modules exposed in the design.
6. Chapter 6 details how the correct functioning of KVM-FUZZ is tested in Section 6.1, and evaluates it through a number of experiments in Section 6.2.
7. Finally, Chapter 7 concludes the documents summing up the contributions of the project.

1.4 Theoretical contents for the understanding of the project

This section aims at introducing and explaining the core concepts related to the project. First, Section 1.4.1 introduces the concept of fuzzing and different fuzzing strategies to increase performance. Then, Sections 1.4.2 and 1.4.3 describe QEMU and KVM respectively, which are commonly used for emulating and fuzzing closed-source binaries and systems. Section 1.4.4 introduces different low level concepts of x86-64 operating systems, and finally Section 1.4.5 presents Markov chains and some results about them.

1.4.1 Fuzzing

Fuzzing is one of the most widely-used technique to discover software vulnerabilities. It consists in repeatedly running a program with automatically generated inputs, monitoring its execution and detecting crashes.

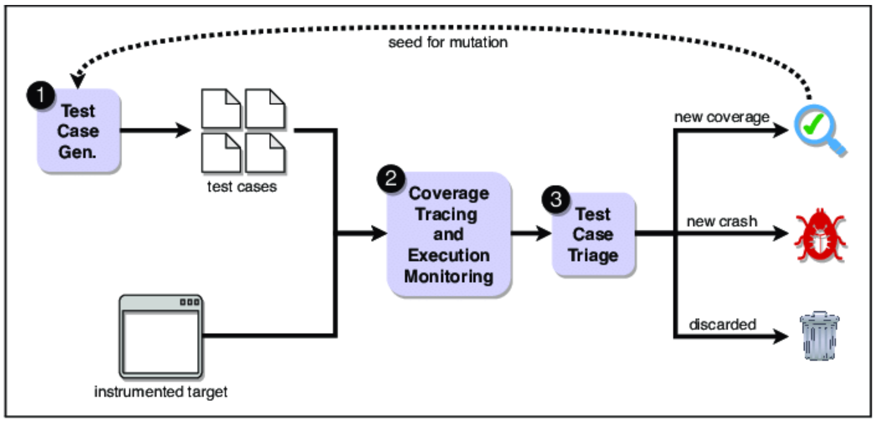


Figure 1.1: Coverage guided fuzzing illustration. Generated test cases are run in the instrumented target, generating coverage information. That information is analysed, and if it supposes new coverage, the test case is saved for further mutating it. Figure taken from [1].

Coverage guided fuzzing

Although even applying fuzzing blindly is enough in many cases to catch shallow bugs, in general it is more effective to take more intelligent approaches. Among them, code coverage guided fuzzing stands out, which consists in analyzing which parts of the code of the program under test have been executed with each input. That way, when an input manages to execute a piece of code that had not been previously reached, said input is saved to continue mutating it and generate new inputs that could be effective for reaching new parts of the code. Therefore, coverage guided fuzzing is essentially a genetic algorithm where the individuals are inputs to the target program, and natural selection is based on achieving unique code coverage.

This simple concept makes it possible for a generic fuzzer that lacks information about a program's input format to learn and generate suitable inputs that reach deep code paths, thus discovering more bugs.

Essentially, code coverage is a metric that indicates how much of the source code of an application is being executed when a fuzz test case is run. More accurate code coverage information can include which parts of the program were executed and in which order. There are different types of code coverage, based on its coverage unit: functions, basic blocks, statements, instructions, edge coverage, etc. The more granular is the coverage unit, the more accurate is the code coverage information, but it also takes more space and time to obtain, store and analyze.

Code coverage is commonly used while testing in order to measure the effectiveness of a test suite, since a program with high code coverage has more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with low test coverage.

An important byproduct of coverage-guided fuzzing is the generated corpus or set of inputs, which typically achieves better code coverage than any manually collected set. This corpus can be used outside of fuzzing, e.g. for a regular continuous testing process. When a bug is found by fuzzing, the faulty input can be added to the corpus to serve as a regression test [6].

In order to get code coverage information in each execution, it is common to make use of instrumentation. Code instrumentation is a technique that consists in modifying the original code of an application in order to debug it, measure its performance, or modify its behaviour. In the case of code coverage, instrumentation inserts small pieces of code in different parts of the original program in order to know which parts of the code have been executed in each iteration. If the source code of the target program is available, this is usually implemented in a compiler plugin that performs the aforementioned process. Otherwise, you can opt for a static approach (modify the original binary) or a dynamic approach (modify the program behaviour at runtime) to insert instrumentation. In particular, the dynamic approach will be further explained in Section 1.5.1.

Snapshot fuzzing

When each execution ends, it is necessary to restore the state of the program before starting the next execution. The easiest way is to launch the program in a new process from scratch every execution, and wait for the process to finish. However, this is very inefficient and does not scale properly (Section 6.2.1). A more effective manner of doing this is with snapshot fuzzing: the state of the process is captured before execution begins, and then it is restored at the end of each execution. With this approach, by having a single process whose state is continuously restored, the cost of creating and terminating processes is greatly reduced.

Taking and restoring snapshots of a process requires access to its registers and memory, among other things. It can be done from user space making use of operating system interfaces such as `ptrace` and `/proc/pid/mem` [7]. However, it is more efficient if it is done directly in kernel space [8], or by making use of virtual machines [9].

The ability to store and restore the state of a process or a system gives room to some different ideas used in fuzzing. For example, incremental snapshots consists in taking temporal snapshots at some more advanced points of execution in order to fuzz different specific parts of the code given an initial state [9]. When the process crashes, the snapshot is saved so the crash can be reproduced and inspected. Another use of snapshot fuzzing is the ability to launch the program under a debugger, stop the execution at some point, take a snapshot, and then load it into the fuzzer to continue fuzzing from that point. [10]

Persistent fuzzing

In case the target program lacks a global state that could be modified in each execution, it is possible make use of a technique called persistent fuzzing. Similar to snapshot fuzzing, it consists in having a single instance of the program running continuously. However, since there is no global state that can cause some executions to affect others, there is no need to restore the state of the process. Therefore, each iteration simply sets a new input, executes the program, and starts over, all without terminating the process. This achieves the best performance, because it is almost as fast as having a loop running the fuzzed code.

In order to modify the behaviour of the original program and make it suitable for persistent fuzzing, it is common to use instrumentation. The idea is to modify the start and end of the program to implement the aforementioned loop logic, in which it communicates with the fuzzer, gets the new input, and runs again, preventing the process from terminating.

Fuzzing harness

Sometimes, the code we want to fuzz is exposed by a small and easy application. The input format are just bytes, which are read from standard input and used by the target code. The program is stateless, that is, there is no state saved between fuzz cases. These applications are very easy to fuzz, and binary parsers are an example of this.

However, that is often not the case. There is usually a gap between how the fuzzer feeds input to the target and how the application actually expects the input. This is solved by a fuzzing harness, which is a program in charge of making the target code suitable for fuzzing. This usually includes reading the input from the fuzzer and delivering it to the target code in an appropriate way. It may also need to reset some state after each execution, inject memory, perform hooks, and change the target code behaviour according to what is needed. Some of these tasks can be performed at the fuzzer level if the fuzzing framework is powerful enough. Then, instead of fuzzing the target code directly, it is the harness that is fuzzed.

For example, when fuzzing libraries, it is needed to create a small harness that reads some input and feeds it to the library through its API. The API might be fixed or change depending on the input. Another example is when fuzzing operating systems. If the input delivered by the fuzzer is just a bunch bytes, a harness is needed to translate those bytes to a suitable format that is accepted by the operating system. One way to do this is by parsing the bytes as a set of system calls and their arguments. For example, the first byte of the input could indicate the number of a system call, and the following bytes could be its arguments. After parsing the input, the harness could execute the parsed system calls, effectively fuzzing the system call interface of the operating system.

In most cases, a harness that is curated to aid the fuzzer in talking to the target program paired with a fuzzer that is smart enough to generate test cases based on the target program will be very effective.

1.4.2 QEMU

QEMU is an emulator that allows the execution of code of a given architecture by dynamically translating said code to the architecture of the host system. The translation process is called Tiny Code Generation (TCG), and it is similar to a Just In Time (JIT) compilation process. First, the code of the emulated architecture is translated into instructions of the intermediate language Tiny Code, which is architecture-independent. Finally, the intermediate language code is translated into instructions of the host architecture, which can be executed under the host machine. The introduction of an intermediate language is a technique widely used by compilers to simplify the translation process.

The translation is done at the basic code block level, which is a sequence of code that has no control flow, i.e. it is executed sequentially from beginning to end. Therefore, it is possible to model the execution flow of a program by using basic blocks, which are linked together by jump instructions. Similarly, QEMU performs the translation process for each basic block, and then chains the resulting pieces of code for greater performance.

QEMU has two main execution modes, user mode and full system mode, which are further explained below.

User mode

QEMU user mode allows running user programs compiled for a different architecture than the host system. First, it emulates its instructions using TCG, the code translation technique described above. However, when running an user application, not only its code is executed, but also the kernel code. User programs perform system calls, transferring execution flow to the kernel. In user mode, QEMU does not emulate the operating system. Instead, it redirects system calls to the host operating system. In order to do this, it features a system call translator, which is in charge of adjusting the endianness, structure and size of the arguments of the system calls to make them suitable for the host architecture. For example, when emulating an Executable and Linking Format (ELF) compiled for MIPS32 big-endian on a x86-64 little-endian host, QEMU changes the endianness of the arguments from big to little, and extends them from 32 to 64 bits.

Apart from the application itself and the kernel, when running an user program there are also libraries involved. Therefore, in order to run a dynamically linked binary of a foreign architecture, QEMU also needs the dynamic loader and libraries it is linked to.

QEMU user mode is largely used in state-of-the-art fuzzing, as described in Section 1.5.1.

Full system mode

On the other hand, full system mode allows emulating a complete system, including the operating system and the peripherals, inside a virtual machine. Instructions are emulated with TCG, same as in user mode. However, now this also includes the operating system, which runs at a higher privilege level. Some of these privileged instructions are Input-Output (IO) operations, which access the underlying hardware. In order to emulate them, QEMU in full system mode implements virtual devices.

When running QEMU in full system emulation, there are a number of options that can be customised, such as the number of Virtual Central Processing Units (vCPUs), the amount of memory, the emulated devices, the disk image used, the hardware accelerator, etc. Some of these features are detailed below.

Device emulation QEMU supports the emulation of a large number of devices, from peripherals such network cards and USB devices to integrated systems on a chip [11]. In order to do so, QEMU must implement the emulation of controllers and the behaviour that the guest operating system expects, which has a great complexity. As an example, just for USB emulation, QEMU implements 4 different controllers [12], which can be backed by a real or virtual USB device.

QEMU also has a feature called device pass through, which consists in giving the virtual device access to the underlying hardware. This has two main usages. First, it can be used to eliminate the emulation overhead, since operations are handled by real hardware instead of emulated by software. Second, it allows things like exposing an USB device on the host system to the guest, or dedicating a video card in a PCI slot to the exclusive use of the guest.

Accelerators When the guest architecture matches the host architecture, QEMU in full system mode can leverage hardware acceleration by using hypervisors. In this way, instead of having to perform the translation process through TCG, the processor enters guest mode and executes the guest instructions directly. As instructions are now executed by the real CPU, this achieves an almost native execution speed.

When using a hypervisor, Virtual Machines (VMs) run on a different context than QEMU itself, which is an user application. Therefore, every time an IO instruction is executed, a VM exit is triggered, and the instruction is emulated by QEMU from the host system.

Some of the supported hypervisors are KVM, Xen, and Windows Hypervisor Platform. KVM, which is the most used on Linux, is further detailed on Section 1.4.3.

1.4.3 KVM

KVM is a hypervisor implemented in the Linux kernel that uses hardware virtualization. It exposes the `/dev/kvm` interface, which an user space program can use to create and manage VMs.

To do this, KVM makes use of Intel VT-x and AMD-V virtualization technologies. These are architecture extensions that implement privilege levels to discern if some code is running in guest mode or in host mode (corresponding to being inside or outside the virtual machine, respectively), as well as a set of instructions to be able to create, manage and execute VMs. The hypervisor can configure the VM so that when the processor is running inside it in guest mode, some privileged instructions cause a VM exit and switch to host mode. It is then up to KVM or the user application to handle said instructions. For example, KVM does not provide device emulation. Instead, IO operations are some of the instructions that generate a VM exit, so it is the userspace program (e.g. QEMU) that must emulate those operations and the devices they can access.

It is important to highlight that Intel VT-x and AMD-V virtualization technologies work at a very low level and provide very little abstraction. KVM is the one that abstracts all the hard work at the kernel level, so it is easier for the userspace part of the hypervisor to develop the intended functionality. For example, with Intel VT-x, VM configuration at the architectural level is done with a huge data structure called Virtual Machine Control Structure (VMCS). Correct handling of the VMCS requires deep knowledge of Intel manuals, and is fortunately done by KVM. It also abstracts the underlying virtualization technology, so the same hypervisor works under different processors.

Since KVM makes use of processor virtualization technologies, it only allows the execution of virtual machines of the same architecture as the host system.

KVM API The KVM interface exposes the file `/dev/kvm`. An user space program can open it and use ioctls to interact with it. Ioctls (from IO control) are a set of commands that can be issued with the syscall `ioctl` to a file descriptor, and are used as an interface to different subsystems of the Linux kernel. KVM exposes different types of ioctls [13]:

- System ioctls: they are issued to the `/dev/kvm` file descriptor, and are used to query and set global attributes which affect the whole

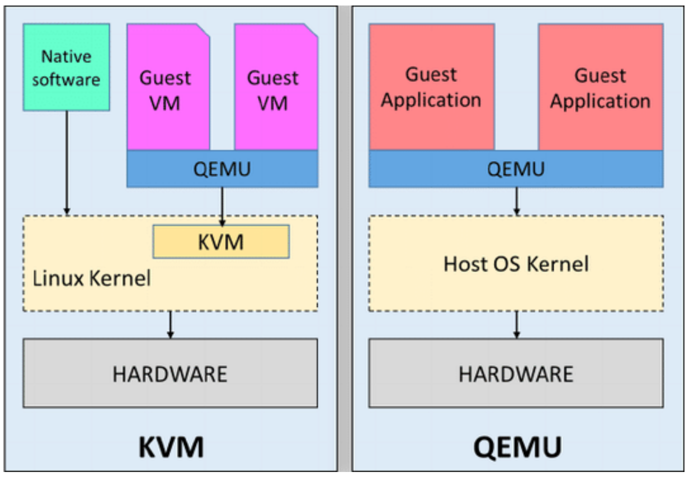


Figure 1.2: On the left, QEMU in full system mode with KVM. Virtual machines have their own operating system. QEMU is in charge of emulating devices, and KVM of providing virtualization. On the right, QEMU in user mode. It emulates applications with dynamic code translation (TCG) and forwards system calls to the host kernel. Figure taken from [2].

KVM subsystem, or to create VMs. When a VM is created, a VM file descriptor is returned.

- VM ioctls: they are issued to VM file descriptors, and are used to query and set attributes that affect the entire virtual machines. Some examples are creating virtual devices, assigning memory to the VM, getting a list of changed (dirty) memory pages, or creating vCPUs. When a vCPU is created, a vCPU file descriptor is returned.
- vCPU ioctls: they are issued to vCPU file descriptors, and are used to query and set attributes of a single virtual CPU. Some examples are setting and getting the state of registers (general purpose registers, but also special registers, such as segment selectors, debug registers, Model Specific Registers (MSRs), extended control registers, the Global Descriptor Table (GDT) and the Interrupt Descriptor Table (IDT)), and also running the vCPU.

Apart from ioctls, KVM also implements another interface through `mmap`. The file descriptor of a vCPU can be mapped to memory, resulting in a `kvm_run` structure accessible by user space. A feature it provides is register mapping, which consists in having registers mapped to memory through this structure. Therefore, the application can read and modify the registers of a vCPU by accessing its `kvm_run` structure in memory, which has a greater performance than using an `ioctl` syscall.

Another feature provided by the mapped `kvm_run` is getting information of each VM exit. After running the vCPU, the application can get the reason of the VM exit from this structure, along with some more data depending on the reason. For example, if the cause was an IO instruction, the application can get the port, the direction (in or out) and the size of the operation. This can be used to implement hypercalls (calls from the guest kernel to the hypervisor): the kernel executes an IO instruction on a certain, predefined port, which triggers a VM exit. The hypervisor then detects the IO operation, handles the hypercall, and resumes the execution inside the VM.

1.4.4 Operating systems

An operating system is a piece of software controlling the operation of a computer system and its resources [14]. It is in charge of loading and executing user programs, providing standardized IO interface and other common services for them. Operating systems distributions usually come with other components, such as a Graphical User Interface, a suite of utilities, or other user applications like web browsers or games.

The core element of an operating system is the kernel. The kernel is in charge of managing resources such as memory and hardware devices, allowing user applications to run, and providing them an interface to use those resources. Depending on the architecture of the kernel, there are two major models: monolithic kernel and microkernel.

Monolithic kernels are designed as a single program that runs in privileged mode. Everything, from device drivers to the different subsystems of the kernel, run inside the same address space. The main advantage to this approach is efficiency, specially on x86 architectures where task switches are a particularly expensive operation. As a downside, due to its single address, a bug in a device driver could bring down the entire system.

Microkernels, on the other hand, try to run most of its services and device drivers such as networking and filesystems in userspace, and have only the essential part running in kernel mode, such as physical memory allocation and messaging (Inter Process Communication). This makes microkernels more modular and stable, because there is less code running in privileges mode, and a crash in a service does not bring down the entire system. However, they need a high amount of messaging between application and services, which makes microkernels conceptually slower than monolithic kernels.

In the following subsections, different aspects and components of a x86-64 monolithic kernel will be described.

System calls

System calls are the main functionality of the kernel with regards to user applications. They are an interface exposed by the kernel to allow user programs to interact with it. The idea is to allow user applications to transfer execution to the kernel, which handles the request in privileged mode, and returns to userspace.

There are different methods to implement system calls. One of them is by software interrupts (Section 1.4.4). However, to have a greater performance, x86-64 implemented two instructions specific for this, called `syscall` and `sysret`. The first one transfers execution to the kernel, while the latter goes back to userspace. System calls have a calling convention specifying how to pass arguments from the user to the kernel. On Linux, registers `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` are used, in that order. This is similar to the System V Application Binary Interface (ABI) [15], which is the calling convention commonly used by user applications on Linux. It only differs in the fourth argument, which System V ABI sets at `rcx`, while system calls require it at `r10`. This is because the `syscall` instruction, besides jumping to the kernel syscall handler and switching to kernel mode, it also saves the user `rflags` into `r11` and the user `rip` into `rcx`. On the other hand, `sysret` restores `rip` from `rcx` and `rflags` from `r11`. The kernel is in charge of preserving their initial values after `syscall` so they can be correctly restored later with `sysret`. As those two registers are clobbered, they can not be used for argument passing.

To configure system calls, the kernel must setup some MSRs, namely:

- STAR: Ring 0 and Ring 3 segment selectors. When `syscall` and `sysret` run, they load the segments specified here from the GDT.
- LSTAR: Address of the kernel syscall handler entry. When `syscall` runs, it jumps to that address, transferring execution to the kernel.
- SFMASK: When `syscall` runs, it masks the `rflags` register with the negation of this value. This means that every bit set in SFMASK will be cleared in `rflags`. That way, the kernel can disable interrupts or change the IO Privige Level, both of which are managed via this register.

System calls offer many functionalities to user applications, including filesystem access, networking, process communication and messaging, memory management, synchronization, and information access.

Model Specific Registers and Performance Counters

MSRs are special control registers available on x86 that are used for debugging, program execution tracing, computer performance monitoring, and toggling certain CPU features [16]. Reading and writing to these registers is handled by the `rdmsr` and `wrmsr` instructions, respectively. As they are privileges instructions, they can be executed only by the operating system.

As shown in Section 1.4.4, system calls are configured by a set of MSRs. Another example are CPU performance counters, which are hardware registers that measure different events occurring in the processor, such as number of instructions executed, cache misses, branch miss-predictions and interrupts. CPUs have a number of performance counters that can be used to count these events. In order to configure them, the kernel must write to one of the performance event select MSRs, which are called `IA32_PERFEVTSELx`. They have several bit fields that allows selecting which event to count, enabling the counter, and choosing whether to count while in user mode, kernel mode or both. After configuring it, the counter itself can be read through the `IA32_PMCx` MSR.

There are three performance events that are counted with another type of registers, called fixed-function performance counter registers, and which are MSRs `IA32_FIXED_CTR0` through `IA32_FIXED_CTR2`. Similarly to the others, configuring and enabling them is done by writing to bit fields, but this time in the MSRs `IA32_FIXED_CTR_CTRL` and `IA32_PERF_GLOBAL_CTRL`. However, unlike before, each of these counters has a fixed event associated. For example, `IA32_FIXED_CTR1` measures the number of cycles executed.

Furthermore, performance counters can be checked against overflow. Namely, they can be configured to generate an exception through the local APIC on overflow by setting one of the bit fields. This allows kernel developers to get a notification when some condition happens.

APIC

The Advanced Programmable Interrupt Controller (APIC) is the chip in charge of managing hardware interrupts in the x86-64 architecture by receiving them from devices and sending them to the CPU. For instance, when a keyboard registers a key hit, it sends a pulse along its interrupt line to the APIC chip, which then translates the interrupt request into a system interrupt, and sends a message to interrupt the CPU. The kernel is then in charge of handling the interrupt (Section 1.4.4). Without a chip like the APIC, the kernel would have to poll all the devices in the system to see if they want to do anything. The APIC also enables more sophisticated interrupt redirection and sending interrupts between processors [17].

In an APIC-based system each CPU has a local APIC, which is responsible for handling CPU-specific interrupt configuration. The local APIC is enabled at boot time, and its registers are memory mapped in a certain physical page, indicated by the MSR `IA32_APIC_BASE`. The APIC is then configured by writing to these registers.

One of the advantages of the APIC is that it can be used as a CPU-specific timer, unlike the older Programmable Interval Timer, which is a separate circuit. Its timer has two different modes:

- Periodic Mode. Kernel sets a value that the APIC uses as initial counter. Each cycle, the APIC decrements the counter until it reaches zero. It then generates a timer interrupt request, which is handled by the kernel. After that, the counter is reset, and it starts decrementing again.
- One-shot Mode. The functioning is similar to periodic mode, but the counter does not reset when reaching zero. Instead, the kernel has to set a new count each time it wants to enable it.

The local APIC is commonly used to implement a generic high precision timer service, which in multi-tasking operating systems can be used to interrupt user tasks after their time slice has been spent.

Global Descriptor Table

The GDT is a data structure that contains entries telling the CPU about memory segments. Segments are contiguous chunks of memory with consistent properties [18]. Each entry in the table, called descriptor, describes a memory segment. They are 8 bytes long and have a complex structure, describing the base address, limit, Descriptor Privilege Level and permissions of the segments, among other flags.

In modern x86-64 kernels, segmentation is not used anymore as the primary memory model since it was replaced by paging, but the structure must be set up nonetheless. The base address and limit fields are ignored. It is common to have 6 entries:

- Null Descriptor: it is mandatory that the first entry is null.
- Kernel Code Descriptor: executable, privilege level 0.
- Kernel Data Descriptor: not executable, privilege level 0.
- User Code Descriptor: executable, privilege level 3.
- User Data Descriptor: not executable, privilege level 3.
- Task State Segment Descriptor: this entry is twice as large, and its base address, instead of being ignored, points to the Task State Segment (TSS) (Section 1.4.4).

Once set up, the kernel tells the CPU the address of the GDT with the privileged instruction `lgdt` (from "load GDT").

Segment descriptors are referenced by segment selectors, which are offsets into the GDT. Segment selectors can be loaded into a segment register, which are CPU registers that refers to a segment for a particular purpose, such as CS (Code Segment), DS (Data Segment) and SS (Stack Segment), or for general use, such as FS and GS. Segment selectors are also used in the IDT (Section 1.4.4) and in the STAR MSR (Section 1.4.4) to indicate which segment should the CPU switch to when an interrupt or system call occurs, respectively.

Task State Segment

The TSS is a data structure which on x86-64 is used to change the stack pointer after an interrupt or permission level change. Its fields are grouped in two types:

- RSP0, RSP1, RSP2: these fields are stack pointers that are loaded when an interrupt triggers and privilege level change occurs from a lower privilege to a higher one. For example, when the timer interrupt kicks in, the CPU may be running in user mode. Therefore, a switch from user mode (ring 3) to kernel mode (ring 0) occurs, and the stack from RSP0 is loaded.
- IST1-7: these fields are the Interrupt Stack Table (IST). They are stack pointers that an entry in the IDT (Section 1.4.4) can reference, so they are loaded when that specific interrupt happens.

Once set up, the kernel tells the CPU the location of the TSS with the privileged instruction `ltr`. Instead of the address of the TSS, this instruction specifies the TSS selector, which refers to the TSS descriptor in the GDT, which finally points to the TSS.

Interrupts

Interrupts are signals from a device, an instruction or the CPU itself that are delivered to the CPU, interrupting its execution. When an interrupt triggers, the CPU looks up an entry for that specific interrupt from a table provided by the kernel, called Interrupt Descriptor Table. This entry contains information about how to handle the interrupt, including a pointer to the Interrupt Service Routine (ISR), which is the code that is run in response to the interrupt and that is in charge of handling it.

There are three main classes of interrupts:

- Exceptions: they are generated internally by the CPU when executing an instruction, and are used to alert the kernel of an event or situation which requires its attention. Some examples are Page Fault, General Protection Fault, Invalid Opcode, Breakpoint and Division Error. Some of them push an error code to the stack before jumping to the interrupt handler, which provides additional information about the error

For example, Page Fault's error code tells whether the memory violation happened on write or read access, on a page-protection violation or on an unmapped page, on user or on kernel mode, etc.

- Interrupt Requests or Hardware Interrupts: they are generated externally by the chipset (Section 1.4.4) on response to a hardware event, like a keyboard press or a timer.
- Software Interrupts: they are signaled by user software running on the CPU to communicate with the kernel. On x86, the instruction `int X` generates the software interrupt `X`, transferring execution to the kernel. For example, on Linux x86-32, `int 0x80` is used as mechanism to implement system calls.

In order to tell the CPU about the location of the interrupt handlers and other information, there is a data structure called the Interrupt Descriptor Table. Each entry in the IDT has a complex structure. Some of its fields are:

- Offset: the address of the ISR. This is where the CPU will jump to when the interrupt occurs.
- Selector: a Segment Selector referencing the code segment in the GDT that will be used when handling the interrupt.
- IST: an index from 1 to 7 into the IST in the TSS. If not zero, the stack pointer in the corresponding field of the IST in the TSS will be loaded when the interrupt kicks in. This is specially useful for the Double Fault exception, an interrupt that happens when an exception is unhandled or when an exception occurs while the CPU is trying to call an exception handler. Under normal conditions, if kernel stack corruption happens and causes a Page Fault exception due to an invalid memory access, the CPU will fault when trying to use the stack to call the Page Fault ISR, triggering a Double Fault. The same problem will happen when trying to invoke the Double Fault ISR, thus generating a triple fault and rebooting the system. In order to avoid this scenario, it is common to use a special stack in the IST for Double Fault. That way, a valid stack will be loaded when calling the Double Fault handler.
- Descriptor Privilege Level: describes the privilege levels which are allowed to access this interrupt via the `int` instruction (ignored for hardware interrupts).

Once set up, the kernel tells the CPU the address of the IDT with the privileged instruction `lidt` (from "load IDT").

On x86-64, when an interrupt happens, the CPU pushes to the stack some information before jumping to the ISR so that it can be restored later:

`rip` (since it is being replaced by the offset field, the address of the ISR), `CS` or Code Segment (since it is being replaced by the selector field), `rflags` (also being clobbered) and `rsp` (since it may be replaced by one of the fields `RSPx` or `ISTx` of the TSS). However, the rest of the registers are not saved, and the ISR is the one in charge of saving them and restoring them after handling the interrupt. At the end of the ISR, the kernel uses the instruction `iret` to return from the interrupt and restore the state that the CPU previously pushed to the stack.

Memory management

Memory management is a critical part of any operating system kernel, both for the kernel itself and for the user programs running under it. Many platforms, including x86, use CPUs that include a Memory Management Unit (MMU), a component that handles memory translation and protection. This allows having different virtual address spaces, isolated from each other, and usually one for each task. x86-64 uses a memory translation system called paging, which consists in dividing the memory into chunks called pages. In order to manage all this complexity, kernels usually implement the following components.

Physical Memory Manager The physical memory is the real memory which is globally visible to the CPU. Physical addresses are directly used to access a real location in RAM. A page of physical memory is often called frame or physical frame.

The Physical Memory Manager (PMM) or Physical Memory Allocator is the component in charge of keeping track of the amount of available physical memory, which parts of it are free and which are currently being used, and also of providing an interface to allocate and free a frame of physical memory. Among the different strategies and data structure that can be used to implement a PMM we can see the use of a bitmap to indicate whether a frame is free or not, or a stack/list of free pages.

Virtual Memory Manager MMUs allow virtual addresses to be used. Virtual address are translated to a physical address by the MMU. It is said that a virtual page is mapped to a physical frame of memory. Therefore, it is possible to create different virtual address space, where the same virtual addresses may correspond to different physical memory. It is common for each task to have its own address space, so memory is isolated between tasks.

The Virtual Memory Manager (VMM) is the component that manages the different virtual address spaces and memory mappings, and provide an interface to allocate and free virtual memory. In order to allocate virtual memory on a given address space, the VMM first allocates physical frames from the PMM, and then maps those frames to the address space.

Paging Paging is the memory translation model used in x86-64. Instead of having an individual virtual-to-physical mapping for each address, which is highly ineffective, the virtual address space is split up in chunks called pages, and virtual addresses are mapped to physical addresses *page-wise*.

In order to perform those mappings, a data structure called page table is used. However, instead of having an entry for each virtual page to indicate the physical frame it is mapped to, a strategy called multi-level paging is used. It is a paging scheme that consists of two or more levels of page tables in a hierarchical manner. The entries of the level 1 page table are pointers to a level 2 page table, whose entries are pointers to a level 3 page table, and so on. The entries of the last level page table are called Page Table Entrys (PTEs), and store the actual physical frame address, page permissions, whether it's accessible from user mode, and other flags.

To translate a virtual address to a physical address, it is split in chunks of a certain size, which are used as indexes into the different page tables. There is a register, usually called Page Table Base Register (PTBR), that holds the address of the top-level page table. The first index is used into this table to get a pointer to a level 2 page table. The second index is then used into this second table to get a pointer to a level 3 page table, and so on. The last index into the last level page table finally provides the PTE, which contains the physical address.

Although this task is largely handled by the MMU by hardware, it would be very costly to perform translation on every memory access. Therefore, there is a data structure inside the CPU called the Translation Lookaside Buffer (TLB), which acts as a cache for recent translations. Before translating a virtual address, the CPU checks if it is present in the TLB. If that is not the case, it is called a TLB miss, and the translation process must be carried out.

In Linux x86-64, normal pages are of size 4096 bytes. Page tables are one page size, and each has 512 entries of 8 bytes (the pointer size). There are 4 levels of page tables, although since recently 5 levels are also supported [19]. Furthermore, the register that holds the address of the top-level page table is `CR3`, and each page table has a different name depending on its level. The translation process is represented in Figure 1.3.

Big picture In order to illustrate the different memory management layers, the example of Linux is described. User programs memory is managed as follows:

1. The PMM allocates memory from the RAM in the form of physical frames.

2. The VMM maps virtual memory regions to those physical frames in the task address space by updating the task page table. This is typically done via `sbrk()` or `mmap()` system calls.

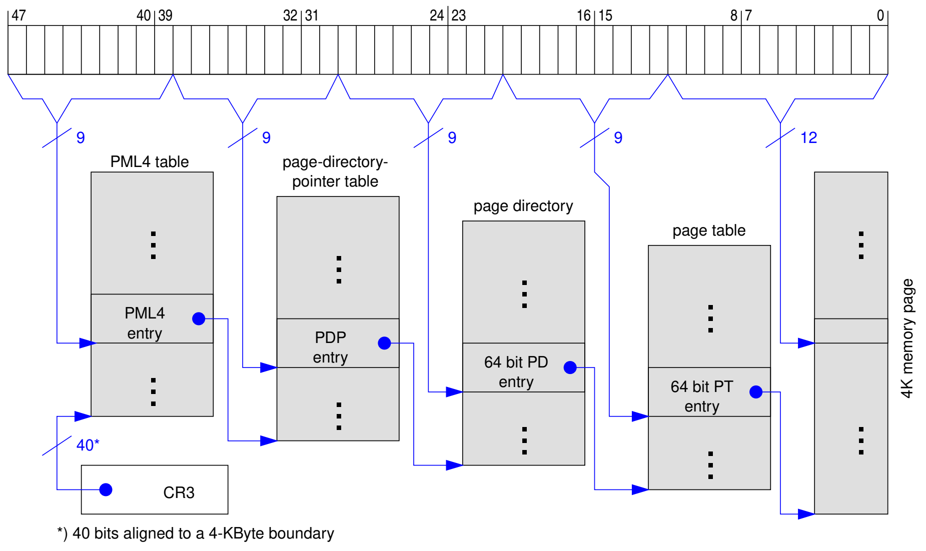


Figure 1.3: Virtual address translation with 4-level paging on Linux x86-64

3. An userspace allocator handles userspace dynamic memory requests from a memory region called the heap, which has previously been created with the aforementioned syscalls. This is typically done via the `malloc()` user library function.

Kernel memory is handled very similarly, but instead of mapping memory in the task address space, it is mapped in kernel address space. Also, the kernel heap is managed by a kernel library, and the function to allocate memory is called `kmalloc()` instead of `malloc()`.

1.4.5 Markov chains

This section presents mathematical concepts and results related to stochastic modelling and Markov chains. It is partially inspired by the resources in the bibliography [20] [21] [22].

Definitions

Definition 1.1. A **stochastic process** is a collection of random variables that is indexed by a mathematical set. This set is called the index set, and is usually a subset of the real line, giving it the interpretation of time. We will usually consider the natural numbers as index set, say time is discrete, and notate the stochastic process as $\{X_n\}_{n\geq 0}$.

In general, $n = 0$ means now, and $n > 0$ are times in the future.

Definition 1.2. The **state space** is the mathematical space where each of the random variables of a stochastic process takes values in. We will notate it as $\mathcal{S}$. Its elements are called states, and we will notate them as $s \in \mathcal{S}$. When countable, we will identify its elements as integers.

Definition 1.3. A **Markov chain** is a discrete time stochastic process $\{X_n\}_{n\geq 0}$ taking values in a space $\mathcal{S}$ that satisfies:

$$\mathbf{P}(X_{n+1} = s_{n+1} \mid X_n = s_n, ..., X_0 = s_0) = \mathbf{P}(X_{n+1} = s_{n+1} \mid X_n = s_n)$$

for all $n \geq 0$.

Thus, given the evolution of the process $\{X_n\}_{n\geq 0}$ up to any current time n, the probabilistic description of its behaviour at time $n+1$ (and, by induction, the probabilistic description of all its subsequent behaviour) depends only on the current state $X_n = s_n$. This is the **Markov property**.

Definition 1.4. A **trajectory** of a Markov chain is a particular set of values for X_0, X_1, X_2, ... That is, if we refer to the trajectory $\{s_0, s_1, s_2, \ldots\}$ such that $s_i \in \mathcal{S}$, we mean that $X_0 = s_0, X_1 = s_1, X_2 = s_2, \ldots$

From now on, we will assume the state space $\mathcal{S}$ is countable, and notate its elements as $i, j \in \mathcal{S}$.

Definition 1.5. A Markov chain is **time homogeneous** if, for all $i, j \in \mathcal{S}$ and $n \geq 1$:

$$\mathbf{P}(X_{n+1} = i \mid X_n = j) = \mathbf{P}(X_n = i \mid X_{n-1} = j)$$

That is, the probability of transitions do not depend on the time n, and there exists p_{ij} such that:

$$\mathbf{P}(X_{n+1} = i \mid X_n = j) = p_{ij}$$

We will call that the probability of making a transition from state i to state j.

Definition 1.6. The **transition matrix** of a time homogeneous Markov chain is:

$$P = \begin{pmatrix} p_{11} & p_{12} & p_{13} & \cdots \\ p_{21} & p_{22} & p_{23} & \cdots \\ p_{31} & p_{32} & p_{33} & \cdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

We will also use a shorter notation for it, $P = (p_{ij})_{i,j\in\mathcal{S}}$.

Note that necessarily:

$$p_{ij} \geq 0 \ \forall s_i, s_j \in \mathcal{S} \qquad \sum_{j\in\mathcal{S}} p_{ij} = 1 \ \forall i \in \mathcal{S}$$

i.e. P is a **stochastic matrix**.

Probability calculations

Notation. Let X be a random variable taking values in $\mathcal{S}$. We will write $X \rightsquigarrow \pi$ to indicate that its probability distribution is given by the row vector $\pi = (\pi_i)_{i\in\mathcal{S}}$, i.e. $\mathbf{P}(X = i) = \pi_i$.

Suppose that $\{X_n\}_{n\geq 0}$ is a time homogeneous Markov chain with state space $\mathcal{S}$ and transition matrix $P = (p_{ij})_{i,j\in\mathcal{S}}$. Further, suppose $X_0 \rightsquigarrow \mu = (\mu_i)_{i\in\mathcal{S}}$. Then, the probabilistic description of the entire process $\{X_n\}_{n\geq 0}$ is determined by μ and P. We may think of μ as representing (the probability of) the initial state of the process, and P as representing its subsequent dynamics. In particular, we have the following result.

Proposition 1.1. *For any sequence of states $i_0, \dots i_n$, we have:*

$$\boldsymbol{P}(X_0 = i_0, \dots, X_n = i_n) = \mu_{i_0} \cdot p_{i_0 i_1} \cdot \dots \cdot p_{i_{n-1} i_n}$$

Proof.

$$\begin{aligned}
&\mathbf{P}(X_0 = i_0, \dots, X_n = i_n) = \\
&\quad = \mathbf{P}(X_0 = i_0)\mathbf{P}(X_1 = i_1 \mid X_0 = i_0) \dots \mathbf{P}(X_n = i_n \mid X_0 = i_0, \dots, X_{n-1} = i_{n-1}) \\
&\quad = \mathbf{P}(X_0 = i_0)\mathbf{P}(X_1 = i_1 \mid X_0 = i_0) \dots \mathbf{P}(X_n = i_n \mid X_{n-1} = i_{n-1}) \\
&\quad = \mu_{i_0} \cdot p_{i_0 i_1} \cdot \dots \cdot p_{i_{n-1} i_n}
\end{aligned}$$

□

We would like to also study the probability distribution of every of the X_n, which are random variables.

Definition 1.7. The **n-step transition probability** is the probability of transitioning from state i to state j in n steps:

$$p_{ij}^{(n)} = \mathbf{P}(X_{m+n} = j \mid X_m = i)$$

We will notate the matrix of n-step transition probabilities as:

$$P^{(n)} = (p_{ij}^{(n)})_{i,j\in\mathcal{S}}$$

Note these probabilities are independent of m by time homogeneity, and $P^{(n)}$ is also a stochastic matrix.

Theorem 1.1 (Chapman-Kolmogorov equation). *For any $n, m > 0$ and $i, j \in \mathcal{S}$:*

$$p_{ij}^{(n+m)} = \sum_{k\in\mathcal{S}} p_{ik}^{(n)} p_{kj}^{(m)}$$

Proof.

$$\begin{aligned} p_{ij}^{(n+m)} &= \mathbf{P}(X_{n+m} = j \mid X_0 = i) \\ &= \sum_{k\in\mathcal{S}} \mathbf{P}(X_n = k, X_{n+m} = j \mid X_0 = i) \\ &= \sum_{k\in\mathcal{S}} \mathbf{P}(X_n = k \mid X_0 = i)\mathbf{P}(X_{n+m} = j \mid X_0 = i, X_n = k) \\ &= \sum_{k\in\mathcal{S}} \mathbf{P}(X_n = k \mid X_0 = i)\mathbf{P}(X_{n+m} = j \mid X_n = k) \\ &= \sum_{k\in\mathcal{S}} p_{ik}^{(n)} p_{kj}^{(m)} \end{aligned} \tag{1.1}$$

□

Corollary 1.1. *For any $n > 0$, $i, j \in \mathcal{S}$ and transition matrix P:*

$$p_{ij}^{n} = (P^n)_{ij}$$

In matrix form:

$$P^n = P^{(n)}$$

Proof. Chapman-Kolmogorov equation in matrix form becomes:

$$P^{(n+m)} = P^{(n)}P^{(m)}$$

For $n-1$ and 1:

$$P^{(n)} = P^{(n-1)}P^{(1)} = P^{(n-1)}P$$

Applying this recursively we get the desired result. □

Proposition 1.2. *Let $\{X_n\}_{n\geq 0}$ be a time homogeneous Markov chain with state space $\mathcal{S}$, transition matrix P, and such that $X_0 \rightsquigarrow \mu$. Then:*

$$X_n \rightsquigarrow \mu P^n$$

Proof. Let $j \in \mathcal{S}$. Then:

$$\begin{aligned} \mathbf{P}(X_n = j) &= \sum_{k\in\mathcal{S}} \mathbf{P}(X_0 = k) \cdot \mathbf{P}(X_n = j \mid X_0 = k) \\ &= \sum_{k\in\mathcal{S}} \mu_k \cdot p_{kj}^{(n)} \\ &= \sum_{k\in\mathcal{S}} \mu_k \cdot (P^n)_{kj} \\ &= (\mu P^n)_j \end{aligned}$$

□

Classification of states

From now on $\{X_n\}_{n\geq 0}$ is a time homogeneous Markov chain with state space $\mathcal{S}$ and transition matrix $P = (p_{ij})_{i,j\in\mathcal{S}}$. We are going to see how the state space of a Markov chain can be partitioned into a set of disjoint **communicating classes**. Two states are in the same communicating class if there is some way of getting from one to another and vice-versa.

Definition 1.8. For any $i, j \in \mathcal{S}$, we say i **communicates** with j and write $i \to j$ if there is some $n \geq 0$ such that $p_{ij}^{(n)} > 0$. If $i \to j$ and $j \to i$, we will say that states i and j **intercommunicate** and write $i \leftrightarrow j$.

Remark. Note that:

1. If i and j intercommunicate, they do not necessarily intercommunicate directly, i.e. in single jumps.

2. $i \to j$ if and only if there is n such that

$$p_{ik_1} \cdot p_{k_1k_2} \cdot \ldots \cdot p_{k_{n-1}j} > 0$$

for some (possibly empty) sequence of intermediate states $\{k_1, \ldots, k_{n-1}\} \subseteq \mathcal{S}$.

Proposition 1.3. *The relation $\leftrightarrow$ is an equivalence relation.*

Proof. 1. Reflexivity: $i \in \mathcal{S} \implies i \leftrightarrow i$ for $n = 0$, as $p_{ii}^{(0)} = 1$.

2. Symmetry: let $i, j \in \mathcal{S}$ such that $i \leftrightarrow j$. That is, $p_{ij}^{(n)} > 0$ and $p_{ji}^{(n)} > 0$ for some $n, m \geq 0$, and thus $j \leftrightarrow i$.

3. Transitivity: let $i, j, k \in \mathcal{S}$ such that $i \leftrightarrow j$ and $j \leftrightarrow k$. Going to the right, that means $p_{ij}^{(n)} > 0$, $p_{jk}^{(m)} > 0$ for some $n, m \geq 0$. Then, applying Chapman-Kolmogorov equation:

$$p_{ik}^{(n+m)} = \sum_{l\in S} p_{il}^{(n)} p_{lk}^{(m)} \geq p_{ij}^{(n)} p_{jk}^{(m)} > 0 \implies i \to k$$

Proving $k \to i$ is analogous.

□

Hence, $\leftrightarrow$ is an equivalence relation, and so partitions the state space into equivalence classes which we will call **communicating classes**. In any class, all the state intercommunicate, but none of them intercommunicates with any state outside the class.

However, it may be possible to have one-way communication between states in different classes.

Definition 1.9. A communicating class C is said to be **closed** if:

$$\forall i \in C,\ j \notin C, \quad i \not\to j$$

Otherwise, the class C is said to be **nonclosed**.

That is, it is not possible to leave a closed class C, since there is no one-way communication from states within C to states outside C.

Definition 1.10. The Markov chain $\{X_n\}_{n\geq 0}$ is said to be **irreducible** if its state space $\mathcal{S}$ is a single, necessarily closed, class, i.e., all states intercommunicate.

Definition 1.11. A state $i \in \mathcal{S}$ is said to be **absorbing** if the set $\{i\}$ is a closed class.

Hitting probabilities

Let A be some subset of the state space $\mathcal{S}$. It may be interesting to calculate the probability of ever reaching any of the states in A starting from the initial state i, and the time it may take.

Definition 1.12. The **hitting time** is the random variable H_A taking values in $\{0, 1, 2, \ldots\} \cup \{\infty\}$ given by:

$$H_A = min\{n \geq 0 : X_n \in A\}$$

When $A = i$ is a single state, we may notate $H_A = H_i$. Furthermore, we use the convention that the minimum of an empty set is ∞, that is, $H_A = \infty$ if $X_n \notin A$ for all n. This random variable represents the time taken before hitting set A for the first time.

Definition 1.13. The **hitting probability** of $A \subset \mathcal{S}$ starting from state $i \in \mathcal{S}$ is:

$$h_{iA} = \mathbf{P}(X_n \in A \text{ for some } n \geq 0 \mid X_0 = i) = \mathbf{P}(H_A < \infty \mid X_0 = i)$$

When A is a closed class, this is called the **absorption probability**. We will call the vector of hitting probabilities $h_A = (h_{iA})_{i\in\mathcal{S}}$.

Definition 1.14. The **expected hitting time** of $A \subset \mathcal{S}$ starting from state $i \in \mathcal{S}$ is given by:

$$\eta_{iA} = \mathbf{E}(H_A \mid X_0 = i)$$

We will call the vector of expected hitting times $\eta_A = (\eta_{iA})_{i\in\mathcal{S}}$.

Remark. We have:

$$\begin{aligned}\eta_{iA} &= \mathbf{E}(H_A \mid X_0 = i) \\ &= \sum_{n=0}^{\infty} n \cdot \mathbf{P}(H_A = n \mid X_0 = i) + \infty \cdot \mathbf{P}(H_A = \infty \mid X_0 = i)\end{aligned}$$

Therefore, η_{iA} can only be finite if $\mathbf{P}(H_A = \infty \mid X_0 = i) = 0$, i.e $h_{iA} = 1$ (the probability to hit A from state i is 1).

These quantities can be calculated explicitly by means of certain linear equations associated with the transition matrix P.

Notation. Henceforth we will notate, for any event E and $i \in \mathcal{S}$:

$$\mathbf{P}_i(E) = \mathbf{P}(E \mid X_0 = i)$$

Theorem 1.2. *The vector of hitting probabilities $h_A = (h_{iA})_{i\in\mathcal{S}}$ is the minimal non-negative solution to the following system of linear equations:*

$$\begin{cases} h_{iA} = 1 & \text{for } i \in A \\ h_{iA} = \sum_{j\in\mathcal{S}} p_{ij} h_{jA} & \text{for } i \notin A \end{cases} \tag{1.2}$$

Remark. h_A being the 'minimal non-negative solution' means that:

1. the values $\{h_{iA}\}$ collectively satisfy the equations above (solution);
2. each value h_{iA} verifies $h_{iA} \geq 0$ (non-negative);
3. given any other non-negative solution to the equations above, say $x = (x_i)_{i\in\mathcal{S}}$, then $h_{iA} \leq x_i\ \forall i \in \mathcal{S}$ (minimal).

Proof. First, we show that h_A satisfies 1.2. If $X_0 = i \in A$, then:

$$\begin{gathered} H_A = min\{n \geq 0 : X_n \in A\} = 0 \\ h_{iA} = \mathbf{P}_i(H_A < \infty) = 1 \end{gathered}$$

If $X_0 = i \in A$, then $H_A \geq 1$. Given another $j \in \mathcal{S}$, then by the Markov property:

$$\mathbf{P}_i(H_A < \infty \mid X_1 = j) = \mathbf{P}_j(H_A < \infty) = h_{jA}$$

Therefore:

$$\begin{aligned} h_{iA} = \mathbf{P}_i(H_A < \infty) &= \sum_{j\in\mathcal{S}} \mathbf{P}_i(H_A < \infty, X_1 = j) \\ &= \sum_{j\in\mathcal{S}} \mathbf{P}_i(H_A < \infty \mid X_1 = j)\mathbf{P}_i(X_1 = j) \\ &= \sum_{j\in\mathcal{S}} p_{ij} h_{jA} \end{aligned}$$

Suppose now that $x = (x_i)_{i\in\mathcal{S}}$ is any solution to 1.2. Then, for $i \in A$, $x_i = 1 = h_{iA}$. Suppose $i \notin A$. Then:

$$x_i = \sum_{j\in\mathcal{S}} p_{ij}x_j = \sum_{j\in\mathcal{A}} p_{ij} + \sum_{j\notin\mathcal{A}} p_{ij}x_j$$

Substitute for x_j to obtain:

$$\begin{aligned} x_i &= \sum_{j\in\mathcal{A}} p_{ij} + \sum_{j\notin\mathcal{S}} p_{ij}\left(\sum_{k\in\mathcal{A}} p_{jk} + \sum_{k\notin\mathcal{A}} p_{jk}x_k\right) \\ &= \mathbf{P}_i(X_1 \in A) + \mathbf{P}_i(X_1 \notin A, X_2 \in A) + \sum_{j\notin A}\sum_{k\notin A} p_{ij}p_{jk}x_k \end{aligned}$$

By repeated substitution for x in the final term, after n steps we obtain:

$$\begin{aligned} x_i = \mathbf{P}_i(X_1 \in A) + \ldots + \mathbf{P}_i(X_1 \notin A, \ldots, X_{n-1} \notin A, X_n \in A) + \\ + \sum_{j_1\notin A} \cdots \sum_{j_n\notin A} p_{ij_1} \cdot p_{j_1j_2} \cdot \ldots \cdot p_{j_{n-1}j_n} \cdot x_{j_n} \end{aligned}$$

Now if x is non-negative, so is the last term on the right, and the remaining terms sum to $\mathbf{P}_i(H_A \le n)$. So $x_i \ge \mathbf{P}_i(H_A \le n)$ for all n and then:

$$x_i \ge \lim_{n\to\infty} \mathbf{P}_i(H_A \le n) = \mathbf{P}_i(H_A < \infty) = h_{iA}$$

□

Theorem 1.3. *The vector of expected hitting times $\eta_A = (\eta_{iA})_{i\in\mathcal{S}}$ is the minimal non-negative solution to the following system of equations:*

$$\begin{cases} \eta_{iA} = 1 & \text{for } i \in A \\ \eta_{iA} = 1 + \sum_{j\notin A} p_{ij}m_{jA} & \text{for } i \notin A \end{cases} \tag{1.3}$$

Proof. As before, we show that η_A satisfies 1.3. Clearly, if $i \in A$ then $\eta_{iA} = 0$ (as the chain hits A immediately). Suppose now that $i \notin A$. Then:

$$\begin{aligned} \eta_{iA} &= \mathbf{E}(H_A \mid X_0 = i) \\ &= 1 + \sum_{j\in\mathcal{S}} \mathbf{E}(\eta_A \mid X_1 = j)\mathbf{P}(X_1 = j \mid X_0 = i) \\ &\text{(conditional expectation: take 1 step to get to} \\ &\text{state } j \text{ at time 1, then find } \mathbf{E}(H_A) \text{ from there)} \\ &= 1 + \sum_{j\in\mathcal{S}} \eta_{jA} \cdot p_{ij} \qquad \text{(by definitions)} \\ &= 1 + \sum_{j\notin\mathcal{A}} p_{ij} \cdot \eta_{jA} \qquad (\text{because } \eta_{jA} = 0 \text{ for } j \in A) \end{aligned}$$

□

Graphs

Markov chains can be represented as weighted directed graphs, and every concept presented up to this point has its analogous in graph theory.

Definition 1.15 ([23]). A **weighted directed graph** is a pair $G = (V, E)$, where:

1. V is a set of nodes (also called vertices or points);
2. E is a set of edges (also called links or arcs), which are triplets composed of an ordered pair of vertices and their associated weight: $E \subseteq V^2 \times \mathbb{R}$.

For each element $(x, y, w) \in E$, we say that nodes x and y are connected with a weight w.

Definition 1.16. The **graph associated** to $\{X_n\}_{n\geq 0}$, a time homogeneous Markov chain with state space $\mathcal{S}$ and transition matrix $P = (p_{ij})_{i,j\in\mathcal{S}}$, is given by:

1. $V = \mathcal{S}$, that is, every state is represented as a node.
2. $E = \{(i, j, p_{ij}) \mid i, j \in \mathcal{S},\ p_{ij} > 0\}$, that is, there is an edge between those states for which a transition exist, and the weight of the edge is the probability of the transition.

Remark. Note the transition matrix P of the Markov chain fully identifies the graph. In fact, we can consider P the **adjacency matrix** of the graph, where p_{ij} is the weight of the edge from i to j, which exists if it's non zero.

Example 1.1. The Markov chain given by the transition matrix:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

has the associated graph of Figure 1.4:

Remark. Note that:

1. Given two states $i, j \in \mathcal{S}$, $i \to j$ if and only if there is a way to get from i to j in the associated graph.
2. The communicating classes of the Markov chain are the **strongly connected components** of its associated graph [24].
3. When the Markov chain is irreducible, its associated graph is strongly connected.

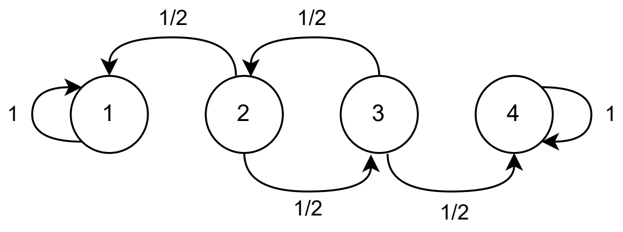


Figure 1.4: Graph associated to Markov chain in Example 1.1.

1.5 State of the art

This section intends to illustrate the state of the art of the techniques used for fuzzing binaries, and how some fuzzers implement them.

In binary fuzzing, code coverage is achieved by using one of the following strategies:

- **Hardware-assisted tracing.** Most modern processors include mechanisms that allow obtaining code coverage by hardware with very little overhead, as is the case of Intel Processor Trace (Intel PT). However, produced traces require a post-processing phase of significant cost. This is the method employed by fuzzers like Nyx [9] or PTfuzz [25]. A different strategy is the use of breakpoints to obtain the traces. A breakpoint is placed at the beginning of each basic block of code, and it is removed when it is reached. In this way, the overhead tends to zero as breakpoints disappear. This is the method employed by Mesos [26].

- **Dynamic translation.** It consists in introducing instrumentation on the fly making use of the process of dynamic translation of tools such as QEMU (Section 1.4.2), DynamoRIO [27] or PIN [28]. It allows for a great level of introspection, but also entails a significant loss of performance, making it unfeasible in many cases.

- **Binary rewriting.** This idea is to modify the original binary and insert the instrumentation prior to its execution. This achieves a similar result to having the source code available and inserting instrumentation via a compiler. Some projects that make use of this mechanism for fuzzing are ZAFL [29], RetroWrite [30] and AFL-Dyninst [31]. However, binary rewriting is complex, and in many cases they restrict to binaries with symbols or compiled in a certain way, or they end up altering the behaviour of the original program.

In the following sections, we will detail the functioning of two coverage-guided binary fuzzers that stand out for their wide use and relevance to the proposal: AFL++, which uses dynamic translation, and Nyx, which uses hardware-assisted tracing.

1.5.1 AFL++

AFL++ [32] is the maintained version of AFL. It is a fuzzer than includes numerous contributions from the scientific community, and it is often used as basis for prototyping idea, many of which end up being integrated into the project.

In order to fuzz programs whose source code is not available, AFL++ uses a QEMU fork called `qemuafl` in user mode. Some of the reasons for using an emulator such as QEMU for fuzzing are:

- Ability to run and fuzz binaries from other architectures, even without source code.
- Dynamic instrumentation. By taking advantage of dynamic code translation via TCG, `qemuafl` introduces instrumentation. When it translates a new basic block, the code it emits includes instrumentation to measure code coverage. Similarly, as it has the ability to modify the code emitted by each instruction in the target program, `qemuafl` also instruments memory accesses and comparison instructions. This allows the fuzzer to learn the values it compares against, and use them to generate future inputs that satisfy those comparisons, reaching deeper code paths.
- Memory errors detection. By instrumenting memory accesses and hooking heap-related functions, `qemuafl` implements QASan [33], a memory sanitizer similar to Address Sanitizer (ASan) but for binaries executed under QEMU, which does not require source code. QASan is able to detect vulnerabilities such as out-of-bounds accesses to heap, stack and globals, use-after-free, and double-free, which may go silent otherwise.

Fork server

In order to run a program in a new process in Linux, you must make use of the system calls `fork()` (to duplicate the current process) and `exec()` (to replace the children process with the target process). However, this is very expensive to do on each run. To solve this, AFL++ makes use of a technique called fork server, which allows to use only the `fork()` system call on each execution. The idea is to run the target program under `qemuafl` in a new process, which is called the fork server. When execution reaches the entry point of the target program, said process stops. Now, on each execution, it is only necessary for this fork server to use the `fork()` system call, and it will be the children process that performs the execution. It is not necessary to use `exec()`, because the fork server is already running the target program.

Additionally, another technique AFL++ uses is to stop the execution at a later point than the entry point, for example, after the initialisation routines. This allows such routines to not be executed on each iteration, effectively placing the start of each execution after them. This technique is known as deferred initialization, and it achieves a significant performance increase by reducing the amount of code that has to be executed on each iteration.

The use of a fork server reduces the amount of system calls required on each execution to only one: `fork()`. However, this system call is handled by the host operating system. When you have multiple instances of the fuzzer running in parallel, the heavy use of fork and its associated synchronization in the kernel acts as a bottleneck, as only one instance of the kernel is available to satisfy all the calls to `fork()` made by each of the processes. The result is worse scaling as the number of available cores increases, contrary to what is desired.

Persistent fuzzing and snapshots

AFL++ also implements persistent fuzzing and snapshots. By using dynamic instrumentation, `qemuafl` can modify the execution flow of the program to loop between two given execution points, without needing to create new processes. This results in high performance and better scaling, since it gets rid of `fork()` and reduces the system calls required per execution to simply those involved in inter-process communication between the fuzzer and `qemuafl`.

However, most programs with a minimum of complexity cannot make use of persistent fuzzing on its own, as it is required to restore the memory to its initial state before each execution. AFL++ solves this by using snapshots. Similarly as described above, it can save the state of the program before entering the execution loop, and restore it after each iteration. Nevertheless, this is a complex task, as it is done from user space. There is no proper mechanism for keeping track of modified memory pages, so you must restore the content of all pages that have write permissions, even if they have not been modified. This means that AFL++ snapshot fuzzing can be slower than the fork server, but as it avoids the use of the `fork()` system call it can achieve better scaling, resulting in better performance when there are a large number of cores.

The AFL++ community developed a Linux Kernel Module [8] based on [34] that implemented snapshots and partially solved the issue of having to restore all memory pages, but the project was discontinued.

1.5.2 Nyx

NYX [9] is a code coverage guided fuzzer that stands out for its ability to fuzz complex targets: hypervisors, browsers, kernels, etc.

NYX executes the target program inside a virtual machine. To do so, it employs KVM-PT and QEMU-PT, modified versions of KVM and QEMU respectively. In this case, and unlike AFL++, it uses QEMU in its full system emulation mode, making use of virtual machines. This allows NYX not only to fuzz Linux user applications as AFL++ does, but also applications of other operating systems like Windows, or even code that runs in privileged mode (ring 0), such as drivers or the operating system kernel itself.

In order to do this, NYX has a program running inside the VM called the agent, which is in charge of executing the target code and communicating with the fuzzer, and that depends on the type of component fuzzed. For the agent to communicate from inside the VM to the fuzzer outside the VM, hypercalls (calls to the hypervisor) are used. They consist in the agent setting up the arguments and executing a specific privileged instruction that triggers a VM exit, transferring the execution to the hypervisor. The hypervisor then processes the arguments, handles the call, and resumes execution inside the VM. This type of communication is used for a variety of things, such as getting a new test case for the target program or notifying the end of a run.

NYX extends KVM and QEMU with the ability to perform snapshots that allow very fast saving and restoring of the state of virtual machines, including emulated hardware. To do this, they make use of a KVM feature called Page Modification Logging (PML), which leverages hardware acceleration to efficiently identify which pages of the virtual machine have been dirtied and must be restored. Together with a similar mechanism for the state of emulated hardware in QEMU, it allows them to restore the state of the virtual machine in a differential way: only the memory that has been modified is restored. Being able to take and restore full-system snapshots allows NYX to fuzz programs that due to their great complexity it would be impossible to do otherwise, such as the inter-process communication mechanisms of browsers like Firefox.

NYX goes one step further and incorporates incremental snapshots. Instead of taking a single snapshot at the start of the execution and restore the program state to that point each time, the idea is to perform temporary snapshots at later execution points. For example, in a target that consumes network packets, such as a server, it is possible to take a snapshot after having received N packets. Therefore, subsequent executions will temporarily start from the point where these packets have already been consumed, allowing to fuzz the remaining packets in a very efficient way.

For code coverage, one of the options proposed by NYX is Intel PT. Intel PT is a technology built into modern Intel processors, which allows obtaining information about a program execution flow, producing highly compressed traces. As it is a feature implemented in hardware, it entails very little overhead in terms of performance. This allows code coverage to be obtained in a generic way, either in binaries that have not been instrumented or in kernel code running on ring 0. However, by default KVM does not allow the use of Intel PT to trace code running inside virtual machines. That is the purpose of KVM-PT: to extend KVM with the ability to enable Intel PT when the processor is running in guest mode inside a virtual machine.

Intel PT produces traces in a highly compressed packet format, which must be parsed and decoded afterwards. QEMU-PT extends QEMU with the ability to communicate with KVM-PT to enable Intel PT in VMs and obtain the resulting traces. These are later decoded to get the control flow and code coverage information that is used in the fuzzing process. In order to decode the traces, QEMU-PT uses `libxdc` [35], a library developed by the same authors that achieves the goal with high performance.

Chapter 2

Planning and costs

2.1 Phases

This section aims at describing the different phases carried out to develop the project and providing an estimate of the hours they took. Four different phases have been identified:

1. State of the art survey phase
2. Analysis and design phase
3. Development phase
4. Documentation phase

Each phase is detailed in the following sections.

2.1.1 State of the art survey phase

This phase consisted in reading and studying the literature of current state of the art fuzzers and technologies. Namely, it had the following subphases:

1. General study of coverage-guided binary fuzzing.

2. Study of AFL++. This included reading the AFL++ paper [32], its documentation, part of its code, and using it to fuzz different test binaries. Some related projects were also studied, such as the Linux Kernel Module for snapshots [8], or QEMU and its process of dynamic translation. This provided a general understanding of AFL++'s architecture and some of its problems.

3. Study of Nyx. This included reading the Nyx paper [9], learning about virtual machines, hypervisors, KVM and snapshots, and discussing some ideas with the authors.

4. Study of KVM. This included reading the documentation of its API, part of the code of some hypervisors such as QEMU, and also part of

Intel manuals. This provided the knowledge to build a hypervisor that made use of KVM.

5. Study of operating systems development. This included extensively reading of the OSDev wiki [14], and provided knowledge about different kernel components, including memory management, system calls, interrupts, some x86-64 structures like the GDT and IDT, etc. This allowed me to develop a kernel from scratch.

6. Study of Markov chains. This included reading different resources about stochastic modelling and proving some of the results that did not include proof.

2.1.2 Analysis and design phase

Once modern binary fuzzers and related technologies were studied, it took place a phase to analyse the problem and propose and design a solution. It had the following subphases:

1. Specification of requirements. This included deciding what the project was going to be about and its functionalities. It was decided that the project would be about a fast binary-only fuzzer that would improve modern fuzzers execution speed.

2. Analysis of the requirements. This included proposing technologies and ideas about how the requirements were going to be met.

3. General design. This consisted in designing the fuzzer and its higher level components (the kernel, the hypervisor and the model) and how they were going to interact.

2.1.3 Development phase

After the general idea of the project and its different components was decided, the development phase began. It was the most extensive part of the project because of the complexity of the developed system. Its subphases have been split in three different categories, one for each component.

Hypervisor

1. Creation of a basic hypervisor that could run x86-64 code.

2. Implementation of ELF parsing and loading. This included debug information parsing and stacktraces.

3. Implementation of emulation features, such as breakpoints, hooks, hypercalls, and the snapshot mechanism.

4. Implementation of code coverage.
5. Implementation of tracing.

Kernel

1. Creation of a basic kernel that would write something to the terminal by using a hypercall.
2. Implementation of basic x86-64 components, such as the GDT and the IDT.
3. Implementation of memory management, including components such as paging, a PMM and a VMM.
4. Implementation of user execution and system calls. This included implementing virtual files managing and multitasking.

Model

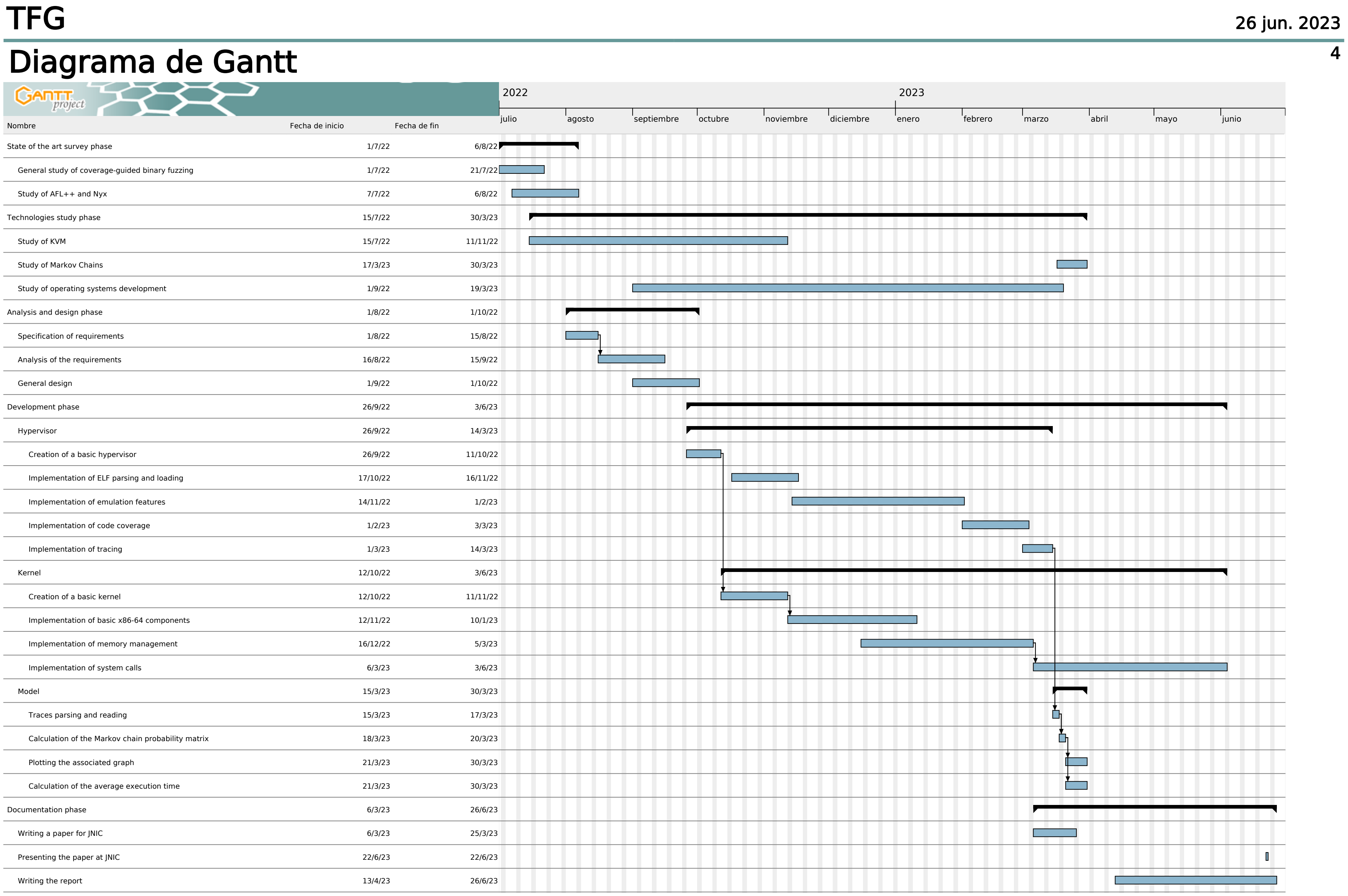
1. Traces parsing and reading.
2. Calculation of the Markov chain probability matrix.
3. Plotting the graph associated to the Markov chain.
4. Solving system of equations to calculate the average execution time.

2.1.4 Documentation phase

This phase consisted in documenting the work that was done for this project. It was divided in the following subphases:

1. Writing a paper and submitting it to the Jornadas Nacionales de Investigación en Ciberseguridad (JNIC).
2. Writing the report. This included drawing diagrams about the general architecture of KVM-FUZZ and UML class diagrams.
3. Presenting the project at the JNIC.

The different phases have been represented in time in a Gantt diagram which can be seen in Figure 2.1.


TFG

26 jun. 2023

Diagrama de Gantt

4

Nombre	Fecha de inicio	Fecha de fin
State of the art survey phase	1/7/22	6/8/22
General study of coverage-guided binary fuzzing	1/7/22	21/7/22
Study of AFL++ and Nyx	7/7/22	6/8/22
Technologies study phase	15/7/22	30/3/23
Study of KVM	15/7/22	11/11/22
Study of Markov Chains	17/3/23	30/3/23
Study of operating systems development	1/9/22	19/3/23
Analysis and design phase	1/8/22	1/10/22
Specification of requirements	1/8/22	15/8/22
Analysis of the requirements	16/8/22	15/9/22
General design	1/9/22	1/10/22
Development phase	26/9/22	3/6/23
Hypervisor	26/9/22	14/3/23
Creation of a basic hypervisor	26/9/22	11/10/22
Implementation of ELF parsing and loading	17/10/22	16/11/22
Implementation of emulation features	14/11/22	1/2/23
Implementation of code coverage	1/2/23	3/3/23
Implementation of tracing	1/3/23	14/3/23
Kernel	12/10/22	3/6/23
Creation of a basic kernel	12/10/22	11/11/22
Implementation of basic x86-64 components	12/11/22	10/1/23
Implementation of memory management	16/12/22	5/3/23
Implementation of system calls	6/3/23	3/6/23
Model	15/3/23	30/3/23
Traces parsing and reading	15/3/23	17/3/23
Calculation of the Markov chain probability matrix	18/3/23	20/3/23
Plotting the associated graph	21/3/23	30/3/23
Calculation of the average execution time	21/3/23	30/3/23
Documentation phase	6/3/23	26/6/23
Writing a paper for JNIC	6/3/23	25/3/23
Presenting the paper at JNIC	22/6/23	22/6/23
Writing the report	13/4/23	26/6/23

Figure 2.1: Gantt diagram of the development of the project

2.2 Budget

This section intends to enumerate the different resources used to carry out the project and give an estimate of their cost. They are listed in Table 2.1. Note we have assumed an amortization period of 3 years for the computer. Its original cost has been estimated to be 900€, so it would be 300€ per year. As the project duration has been of approximately one year, its final cost would be 300€. On the other hand, the experiments computer has been assumed to be rented for 5 hours at 10€ per hour.

Concept	**Amount**	**Unit cost (€)**
State of the art survey phase (hours)	30	30
Technologies study phase (hours)	100	30
Analysis and design phase (hours)	30	30
Development phase (hours)	300	30
Documentation phase (hours)	50	30
Professor mentoring (hours)	25	50
Development computer, CPU: Intel Core i7-6700K @ 4.00GHz, RAM: 32GB.	1	300
Experiments computer: Intel Xeon Silver 4314, 64 threads (rent hours)	5	10
Visual Studio Code	1	0
Other software licenses	1	0
Total		**16900**

Table 2.1: Budget table

Therefore, the total estimated cost of the project is 16900€.

Chapter 3

Analysis of the problem

3.1 Requirements specification

This project aims to design and develop KVM-FUZZ, a fuzzing tool that can compete with other state-of-the-art fuzzers in terms of performance and usability. As a coverage-guided binary fuzzer, it must be able to fuzz programs whose source code is not available, get coverage on each execution, mutate interesting inputs and detect and store crashes. Furthermore, the proposed Markov model must be able to read execution traces, produce a graph detailing the control flow of the program, and accurately calculate the average execution time.

3.1.1 Functional requirements

Functional requirements are divided in 4 groups, according to the following functionalities:

- Inputs management: requirements related to the target program inputs.
- ELFs management: features required to handle the target binary and its dependencies.
- Virtual machines: requirements related to running the target program inside a virtual machine in order to fuzz it.
- Model: requirements of the Markov model that describes the program control flow.

Inputs management

FR1 Read initial input files and store them in memory.

FR2 Mutate inputs applying different random mutation strategies, in order to generate new inputs that may trigger new parts of the code.

FR3 Keep track of the total code coverage achieved, in order to be able to determine if a given input is interesting or not.

FR4 Keep inputs that trigger new coverage and crashes and store them to disk.

FR5 Corpus minimization mode: attempt to minimize a set of inputs, both in terms of number and size, while keeping the same code coverage among all.

FR6 Crash minimization mode: attempt to minimize a set of crashing inputs, in terms of size, while keeping the same crash each of them.

ELFs management

FR7 Parse ELF headers in order to get information about the symbols, sections, segments and relocations.

FR8 Load ELF into the memory of a VM.

FR9 Virtual address resolution to symbol and source code location, if there is debug information available.

FR10 Extract stacktrace across the set of ELFs loaded in a VM at a given point.

FR11 including libraries and interpreter or dynamic loader.

Virtual Machines

FR12 Interface for accessing memory and registers, both for reading and writing. For memory, this requires implementing address translation from virtual to physical memory addresses.

FR13 Set breakpoints where execution will stop, which can be used to stop the run at a given point, or to implement code coverage, hooks and tracing.

FR14 Virtual files, which will be loaded from the host filesystem into the VM memory and can be accessed by the target program.

FR15 Create VMs and prepare them for execution on 64-bit mode. This includes setting up the initial state of the registers, memory, GDT and page table.

FR16 Execute a VM, handling VM exits and calls to the hypervisor (hypercalls).

FR17 Detect crashes of the target program or the kernel running inside the VM.

FR18 Get code coverage of the execution of the target program with each input.

FR19 Reset the state of a VM to a previous state. This includes resetting registers, memory and current trace.

FR20 Set hooks that will be executed when the target program gets to a given execution point, which can be used to inspect its state or alter its behaviour.

FR21 Gather statistics of each run, such as execution time and number of instructions executed both in user and kernel space.

FR22 Get execution traces of each run, which will be used to create the Markov model. Measurements could be in instruction or in cycles, and in kernel space (syscalls) or in user space (basic blocks).

Model

FR23 Parse traces and check they are suitable for creating the Markov model.

FR24 Create a Markov chain that represents the execution behaviour of the target program according to given traces.

FR25 Use the Markov chain to calculate average time to run the program. Check it matches the average time of the traces.

FR26 Plot graph associated to Markov chain.

3.1.2 Non functional requirements

Non functional requirements have a great relevance in this project, since they are part of what differentiates it from other similar fuzzing tools.

NFR1 Multithreading. With a simple command line option, it must be possible to activate multithreading and create multiple virtual machines that will be fuzzed in parallel.

NFR2 Linear scaling. If there are N available cores, the speed measured in executions per seconds must be close to N times the speed of a single core. And this relation must not decrease for any N.

NFR3 Low memory usage by VMs. As the goal is to have multiple VMs running in parallel, it is necessary that they have a low memory footprint.

NFR4 Execution performance. Execution speed of the target program must be close or superior to native, which is its speed when executing on the host system under normal circumstances.

3.2 Analysis

In this section we will analyze different problems that have arisen when developing KVM-FUZZ and attempting to meet the aforementioned requirements.

3.2.1 Execution performance

The requirement that inspired this project, and probably the most difficult one, is **NFR4**: execution performance. As explained in Section 1.5.1, in order to fuzz binaries whose source code is not available, some fuzzers like AFL++ use QEMU in user mode, which carries out dynamic translation through TCG. However, this is known to reduce performance greatly.

On the other hand, when target architecture matches host architecture, QEMU in full system mode has a hardware acceleration mode that makes use of KVM. The aim is for KVM-FUZZ to emulate binaries taking advantage of said hardware acceleration, as QEMU does when in full system mode. In this way, it avoids dynamic translation and achieves a significant performance improvement in the fuzzing process. To do this, a user program that makes use of KVM is implemented, which from now on will be referred to as hypervisor.

Then problem with KVM is that it requires the use of full system virtual machines. That is, in addition to the target user program, they need a kernel. NYX (Section 1.5.2) opts to use traditional virtual machines with QEMU, with their respective full operating system (Linux, Windows, etc) and devices emulation. However, the goal of KVM-FUZZ is to fuzz only an user application. Therefore, we do not need the entire operating system, but just a simpler mechanism that allows us to emulate the system calls that the target program performs.

The initial idea is that the system calls are emulated from the hypervisor outside the virtual machine. In this way, the kernel of the virtual machine is simply a piece of code that is executed when the target application makes a system call, and which is in charge of performing a hypercall, that is, it executes a specific privileged instruction that causes an exit from the VM to the hypervisor (VM exit). The hypervisor, upon detecting the use of said instruction, emulates the desired system call and continues execution. This architecture is illustrated in Figure 3.1.

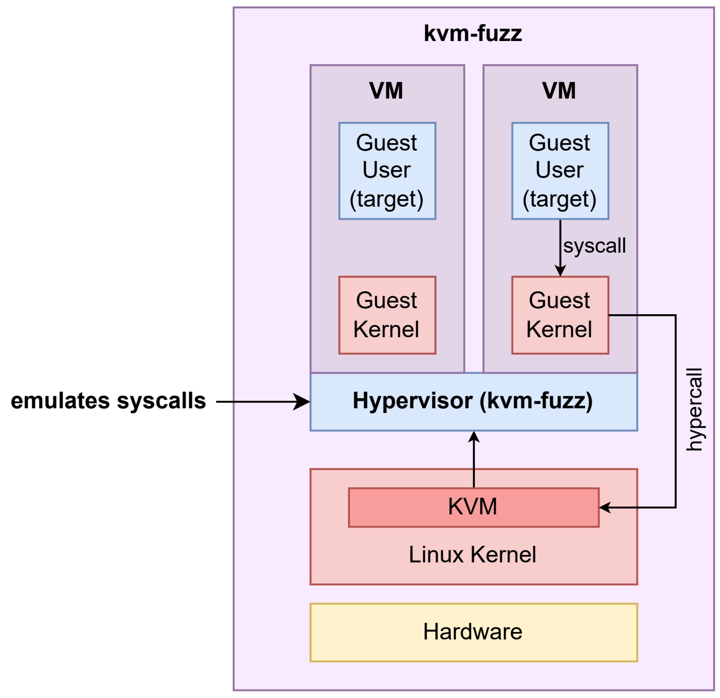


Figure 3.1: First proposal of KVM-FUZZ architecture

This form of emulation has the advantage that the instructions of the target application are executed inside the virtual machine with KVM, and therefore at native speed, as it avoids dynamic translation and its consequent loss of performance. The drawback of this method is the high cost of system calls. Since it is the hypervisor that is in charge of emulating them, every time the application makes a system call, many context switches occur:

guest user (application) → guest kernel (simply performs VM exit) → host kernel (KVM) → host user (our hypervisor, emulates system call) → ... (and back)

To solve this, the proposal incorporates its own kernel, more complex than the one suggested above but still much simpler than common kernels, which is in charge of emulating the system calls instead of forwarding them to the hypervisor. Therefore, the context switches required for each system call are the same as they would be in a real system:

guest user (application) → guest kernel (emulates the system call) → ... (and back)

Since the emulation occurs now entirely inside the virtual machine, the costs of VM exits and communications with the hypervisor is almost completely removed, achieving **NFR4**. The final architecture is illustrated in Figure 3.2.

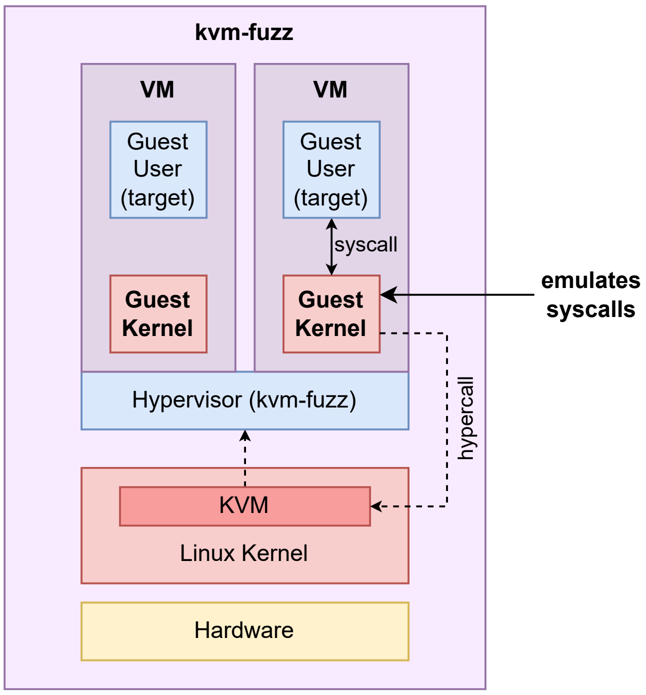


Figure 3.2: Final proposal of KVM-FUZZ architecture

As an example, using `readelf` as the target program, the initial design that emulates system calls from the hypervisor outside the VM averages 2800 executions per second, performing a total of 76 VM exits (one per system call) per execution. Most of the execution time is silently lost in the context switches described above. In comparison, the proposed design that emulates system calls from the kernel inside the VM averages 15000 executions per second, with only one VM exit to finish each run.

Additionally, since the developed kernel is intended to run on our own hypervisor, it can make use of hypercalls for operations that would otherwise be difficult to perform from kernel space, such as reading files from disk or printing to the screen. This simplifies the kernel and eases its development burden, as it does not require drivers and other components that are supplied by the hypervisor.

3.2.2 Programming language for the kernel

While developing the kernel component in C++, some issues arose that made it worth switching to Zig [36], a programming language that intends to be a modern, improved version of C. It focuses on writing and maintaining robust and optimal software, while keeping the simplicity of C. Below are detailed some of the issues presented by C++, and how Zig solved them.

Toolchain

When trying to do operating system development with C or C++, it is required to have a cross-compiler. This is because by default `gcc` is set at compile-time to compile for the host target, in this case `x86_64-linux-gnu`. This means it is compiling user programs for Linux under the GNU libc (C standard library), and it uses system libraries, headers and the Linux `libgcc`. However, when compiling a kernel, we can not make use of Linux headers and libc, since they are meant for user applications to interact with the Linux kernel. Therefore, it is needed to compile `binutils` and `gcc` with a target suitable for a freestanding environment, where there is no C standard library.

This means that whenever some user wanted to use the project or build the kernel, he needed to build the whole `binutils` and `gcc` with some custom flags in order to compile, which was a hassle. Furthermore, the build process was complicated because it needed two compilers: the specific cross-compiler for the kernel, and the usual system compiler for the hypervisor, which is an user application.

On the other hand, the Zig compiler has cross-compilation enabled by default to any target with just a command line option, and can compile both Zig and C/C++. So by switching the build system and toolchain to Zig, we are now depending solely on the Zig compiler, which is included in a 42MB archive, instead of on a custom build of `gcc` and the system `gcc`. The change would be worth it even if the Zig programming language was not used at all, just because of the cross-compilation capabilities of the Zig compiler for C/C++ code.

Standard library

When building C/C++ code in a freestanding environment with a cross-compiler, there is no standard library available, because it is not included. Even if included manually, it cannot be used as it relies on being a user application for Linux. This means kernel developers have to write their own libc, including memory and string utilities, memory allocation routines, etc. In addition, in C++ the usual Standard Template Library (STL) is also not available, so the required data structures must be implemented by hand too.

Furthermore, since the goal of the proposed kernel is to emulate Linux syscalls, it needs to access many Linux headers that define structures used in system calls. When coding in C++ with a cross-compiler, system headers are not available, so they must be manually included and shipped with the project.

On the other hand, most of the Zig standard library is platform agnostic, which means it can be used even in a freestanding environment. This is achieved by separating the core logic from the platform-dependent one. As an example, when data structures need to allocate memory, they require an *allocator* parameter that defines how to manage memory. Therefore, kernel developers only need to define the memory allocation routines to be able to use the well-tested data structures and routines of the Zig standard library. Another example is `printf()`, a complex function for printing data in different formats. Since the standard library is not available, hobby C/C++ kernels often need to include an external implementation of this routine. However, in Zig, freestanding developers only need to provide a function to print a sequence of characters to the screen, and then the Zig standard library implementation of format printing can be used.

In addition, the Zig standard library support for Linux can be used even in a freestanding environment, so it is not needed to ship Linux headers in order to access its structures.

All these Zig standard library features follow the idea of code reuse, allowing the developer to focus on the right problems and saving many hours of debugging buggy data structures implementations.

Error handling

As the kernel goal is to emulate syscalls from the target program, it must be able to handle errors correctly and recover from them, and it must not crash no matter the behaviour of the application. For example, since the kernel runs in a restricted environment with a very limited amount of memory[1], it must be able to handle all Out Of Memory (OOM) errors successfully.

In C, this is usually done using special return values to indicate errors. The caller of a function must always remember to check if the return value is an error, which is rather cumbersome and error-prone. However, in C++ there is no way to do this. One could try to follow the C style, but constructors cannot return errors, and the STL data structures usually do not follow that approach. Instead, C++ encourages to use exceptions. C++ exceptions are complex, require the C++ runtime and the unwind library to be ported (which requires `pthreads` and other POSIX abstractions to be implemented in the kernel), and are usually discouraged to use in a freest-

[1]The amount of memory needed by the VM depends on the target binary size. For example, in order to fuzz a statically compiled version of `readelf`, the VM needs 8MB of memory.

anding environment. To sum up, there is no easy way in C++ to handle errors in general, and the language encourages to “just crash” in case of errors like OOM.

The approach of Zig is exactly the opposite. It implements the concept of errors inside the language as error unions, which are types that describe either a value or an error. If a function can fail, that is, its return type is an error union, you can not use its return value without checking first if it’s an error. Zig also implements syntactic sugars for common actions, such as forwarding the error to the caller, indicating that a function call can not fail, or handling each possible error. Since every piece of code has to take errors into consideration, situations like being OOM are not a problem anymore: every attempt to allocate memory will return an error, and the syscall that triggered it will return `ENOMEM` (the Linux error for OOM). This approach makes it very easy to create robust software that can run in restricted environments, as the kernel intends to be.

This was the main reason for switching to Zig. Crashing the kernel every time it ran out of memory was not an option.

Safety

C/C++ are known as memory unsafe languages. The total absence of safety becomes more obvious in kernel land, where there is no operating system behind covering your back. Specially in kernel development, which is greatly low-level and close to hardware, memory corruption bugs may happen but go silent, or only crash the kernel much later, making them very difficult to debug.

Zig does not achieve complete memory safety as other system programming languages like Rust do, but it introduces safety checks on unsafe operations, which can avoid many pitfalls. Some of these checked operations are: index out of bounds, integer overflows, unsafe integer casting, division by 0, accessing an inactive member of an union, etc. Furthermore, Zig includes a memory allocator that can also check for heap vulnerabilities, such as use-after-free, double-free, and even leaked memory.

In case any of these happens, the program (in this case, the kernel) prints an error and a stack trace with source code annotations pointing where the failed check is and how it got there, and aborts. This helped the development of the kernel greatly, detecting bugs quickly and granting more confident on the correctness and quality of the code written. Also, once the code seems correct, safety checks can be disabled in release builds in order to get a small performance boost.

Personal preference

I personally enjoy writing Zig a lot. It is a very simple language, so I could begin working on the kernel very soon after having learnt a bit about the language. This would have taken months on more complex languages like Rust. Despite its simplicity shared with C, one can feel Zig is a modern language, as it avoids many of the pitfalls and issues of other traditional programming languages. Knowing you are writing safe, fast and robust code is a nice feeling. To sum up, I think switching from C++ to Zig for the kernel was a great decision.

Chapter 4

Design

This chapter intends to illustrate the general design of the project. KVM-FUZZ consists of three main components: the hypervisor, the kernel, and the model. The general overview is as follows. The hypervisor manages VMs and implements the fuzzing logic. The kernel runs inside the VMs, emulating system calls performed by the target program. The hypervisor and the kernel communicate via hypercalls and shared pointers. Finally, the hypervisor produces traces that are later consumed by the model. Each component is further detailed in the following sections.

4.1 Hypervisor

The hypervisor implements both the virtual machine and fuzzing logic. It includes functionalities for managing VMs, loading ELFs into them, running them with different emulation features such as breakpoints and hooks, getting code coverage, detecting crashes and generating new inputs.

Its design is represented in Figure 4.1 with an UML class diagram. The classes are further detailed below.

4.1.1 Vm

The Vm class represents a virtual machine, and it is the most important class of the hypervisor. It implements the following features:

- Creating a VM, assigning it a given amount of memory, and loading the kernel, the target ELF and its dependencies, with the constructor. It also sets up the VM for executing in long mode (64 bits mode of x86).
- Access to registers and memory, with methods `regs()` and `mmu()`, and some higher level methods like `stack_push()`, `stack_pop()` and `read_msr()`.

- Access to the loaded target ELF, with method `elf()`.
- Snapshots: VM forking with the copy constructor, and resetting with the `reset()` method.
- Running, with methods `run()` and `run_until()`. This runs the VM while handling some VM exits, such as hypercalls and breakpoints.
- Code coverage, with methods `setup_coverage()` for initialization, `coverage()` for accessing the current coverage information, and `reset_coverage()` for resetting it. The underlying data structure is different depending on the coverage type (Section 4.1.9).
- Breakpoints and hooks, with methods `set_breakpoint()`, `remove_breakpoint()`, `set_hook()`, and `remove_hook()`, which write to the target memory address to place an breakpoint so execution is stopped when it gets to that instruction.
- Virtual files, with methods `set_file()`, `set_shared_file` and `read_and_set_shared_file()` (Section 4.1.10).
- Timeouts, with methods `set_timeout()` and `reset_timer()`.
- Access to the `Tracing` object that generates traces for the model, with method `tracing()` (Section 4.1.11).
- VM introspection and debugging, with methods `print_stacktrace()`, `dump_regs()` and `dump_memory()`.
- Hypercall handling, with private method `handle_hypercall()`, which calls the appropriate individual handler. Hypercalls are further explained in Section 4.2.1.

4.1.2 Mmu

The Mmu class represents a MMU. It handles the memory of a VM, both physical and virtual. It implements the following features:

- Memory creation and assignment to a VM, with the constructors. The memory and its length are held in private fields `m_memory` and `m_length` respectively.
- Definition of memory layout, which includes defining the addresses of the initial page table, the kernel stack, or the load addresses of the target ELF and its interpreter.
- Creation of the physmap, which is a virtual mapping of all the physical memory, so the kernel can access physical memory directly.

- Dirty memory tracking and resets, with method `reset()`.
- Physical memory allocation, with methods `alloc_frame()` and `alloc()`. The latter also performs virtual memory mapping. This is done with a pointer `m_next_page_alloc` that is incremented and returned every time a frame is allocated. Physical memory management and ownership is transferred to the kernel after its initialisation with the hypercall GetMemInfo. At that point, these methods are disabled by `disable_allocations()`.
- Virtual to physical memory translation, with method `virt_to_phys()`.
- Read and write access to memory, with methods `read_mem()`, `write_mem()` and `set_mem()`. There are higher level versions of these, such as the template versions to access an arbitrary type from memory `read<T>()` and `write<T>()`, or `read_string` to read a null-terminated string from memory. Furthermore, `readp<T>()` and `writep<T>()` are similar but operate on physical memory instead of virtual.
- ELF loading, with method `load_elf()`.

In order to perform dirty memory tracking to be able to later reset the memory to a previous state, the Mmu does two things. First, every write access that happens inside the VM is tracked using KVM capabilities. However, when the hypervisor writes to memory, that is not tracked by KVM. Therefore, additionally every write access from the hypervisor to memory is accounted, and the physical address of the modified page is saved in `m_dirty_extra`.

4.1.3 PageWalker

The PageWalker class is an auxiliary class used by the Mmu to abstract paging. The Mmu class can only access memory through physical addresses. PageWalker is the one in charge of performing virtual to physical address translation, iterating the page table, and providing access to its entries. It provides the following interface:

- Creation of a PageWalker for a range of virtual memory, with the constructors.
- Advancing to the next page of the range, with method `next()`.
- Mapping current page to a physical address with given flags, with method `map()`.
- Allocating a frame of physical memory from the Mmu and map current page to it, with method `alloc_frame()`.

- Getting information about the current page: the physical address of the PTE with `pte()`, its value with `pte_val()`, its flags with `flags()`, the page virtual address with `vaddr()`, whether the page is mapped or not with `is_mapped()`, and in case it is, its physical address with `paddr()`.

The existence of this class eases the burden of the Mmu, and isolates all the logic that deals with page tables in the hypervisor.

4.1.4 ElfParser

The ElfParser class is in charge of parsing ELF files and providing information about them. It implements the following features:

- ELF parsing from a disk file or from memory, with constructors. If parsed from a disk file, it is loaded into memory and the ElfParser object owns the memory. If parsed from an extern memory location, the ElfParser object does not own the memory.

- Providing information about the ELF: its entry point with `entry()`, its load address with `load_addr()`, whether its Position Independent Executable (PIE) with `pie()`, its path and the path of its interpreter if it exists with `path()` and `interpreter()`, and other information with methods such as `symbols()`, `segments()` and `sections()`.

- Getting the library dependencies of the ELF with method `get_dependencies()`.

- Virtual address resolution to symbol and source code location, with methods `addr_to_symbol()`, `addr_to_source()` and similar.

- Getting stacktraces: given the state of the registers and access to memory, method `get_stacktrace()` performs stack unwinding to return the current call stack, that is, the list of the addresses of the functions that were executed to get to that point. There is a static version of the method which takes a list of ELFs, so the stacktrace can be across different ELFs.

Often, the release ELF or libraries are stripped, meaning they have no symbols or debug info. Instead, they indicate the path of a different ELF, which lacks actual code but contains all that additional information. In those cases, the debug binary is also loaded into an ElfParser, and pointed to by the original ElfParser with its private field `m_debug_elf`. On the other hand, debug information is handled by the field `m_debug`.

4.1.5 ElfDebug

The ElfDebug class manages the debug information of an ELF. It makes use of the `libdwarf` library to provide the following features:

- Stack unwinding, with method `next_frame()`. This method receives a set of register and access to memory through an Mmu, and accesses the debug information to update the state of the registers to how they were in the previous frame. This includes updating the instruction pointer to the previous function in the call stack.
- Mapping a virtual address to a location in source code, which includes source code file and line.

These capabilities are very important for a great user experience. Implementing our own stack unwinding across different ELFs allows the hypervisor to accurately get and display the current execution state at any moment, with both raw virtual address and source code information. The user can just stop the VM at any point and see exactly where that point is in code and how it got there. This is specially useful when a crash is detected, both in user space (a crash found by fuzzing) or in kernel space (a bug in the kernel).

4.1.6 Elfs

The Elfs class is composed of all the ELFs that are loaded into a virtual machine. This includes the kernel, the target elf, and its interpreter and libraries in case it is dynamically linked. In order to add a library, the method `add_library()` receives a reference to the contents of the ELF file, and creates an ElfParser from that. This is because libraries are loaded from disk when added as shared virtual files (Section 4.1.10). After that, a reference to that memory that lives inside the `s_shared_files` field of the Vm is used to construct the ElfParser. This design allows library ELFs to not be duplicated in memory, which would happen if ElfParsers were created with the constructor that only receives the elf path, as it loads the file into memory again.

Apart from providing access to the different ELFs and allow creating ElfParsers for libraries, the Elfs class also exposes the method `set_library_load_address()`, which allows modifying the load address of a library when it is loaded into the user memory. The load addresses of libraries are sent to the hypervisor by the kernel via the hypercall LoadLibrary.

4.1.7 Corpus

The Corpus is in charge of loading initial inputs from disk and keeping all interesting inputs and crashes. It provides the following features:

- The constructor loads initial inputs from disk. It also receives a parameter called `nthreads`, which is the number of threads that are simultaneously using the Corpus.
- Setting mode, which can be normal, corpus minimization, or crashes minimization. Minimization modes attempt to reduce input sizes while keeping the same coverage or crashes, respectively.
- Getting a new input, which will be a mutation from one or more random inputs of the corpus, with method `get_new_input()`.
- Reporting the code coverage obtained after running an input, with method `report_coverage()`. The Corpus then decides if the input is interesting, i.e. triggers new code paths, and in that case adds the input to the corpus and saves it to disk. In order to know if an input provides new coverage, the Corpus has the attribute `m_recorded_coverage`, the set of all recorded coverage.
- Reporting a crash obtained after running an input, with method `report_crash()`. If it is unique, its information is displayed and the input is stored to disk. In order to know if a crash is unique, the Corpus has the attribute `m_crashes`, the set of every unique crash.

Some methods, such as `get_new_input()`, `report_crash()` and `report_coverage()`, receive an `id` as argument. This is the id of the calling thread, which is used as index into `m_mutated_inputs`. For example, `get_new_input()` creates the mutated input in that position, and returns a reference to it. That way, access to the mutated inputs can be done simultaneously by different threads without locks, because each one will access a different index.

4.1.8 Mutator

The Mutator class is used by the Corpus to mutate inputs. Its main method is `mutate_input`, which receives a reference to an input and mutates it in place. In order to do that, it applies a random number of the different binary mutation strategies it implements, such as switching a bit, incrementing or negating a byte, swapping two parts of the input, inserting or replacing some magic constants that are more likely to trigger bugs, or splicing two different inputs. If the `minimize` argument is set to `true`, it will perform `mut_shrink()` at least once, so the resulting mutated input is smaller than the original one.

4.1.9 Coverage

There are a number of classes representing code coverage, depending on the mode (Intel PT or breakpoints) and whether it's the coverage of a single run, which is passed to the Corpus in `report_coverage()`, or it's the shared total coverage of all the inputs together, which is kept by the Corpus.

Having different classes depending on whether it is the coverage of a single run or it is the shared total coverage allows the implementation to change the underlying data structure. This is motivated by the fact that it is often needed to check if any element is present on a given coverage, but it is not present on the shared coverage. This would mean the input that triggered said given coverage is interesting. Therefore, normal coverage would need to be optimised for iterating, while shared coverage would need to be optimised for checking if an element belongs or not. On the other hand, having different classes depending on the code coverage mode leaves some level of freedom to implementation details.

In any case, every Coverage class must be able to:

- Keep track of code coverage, usually represented by virtual addresses.
- Be reset, with the method `reset()`.

In addition, shared Coverage classes must be able to be joined with another coverage object, checking if there were any new code coverage, with the method `add()`.

The result is classes CoverageIntelPT and CoverageBreakpoints, that represent coverage of a single run depending on the mode, and SharedCoverageIntelPT and SharedCoverageBreakpoints, that keep the total coverage. As it is chosen between Intel PT and breakpoints based coverage at compile time, this is a compile time interface.

4.1.10 Virtual files

Virtual files are an important feature of the hypervisor. It allows the target program to access memory-loaded files through filesystem-related system calls.

In order to load a file into the virtual machine the kernel submits two pointers to the hypervisor: a pointer to the buffer, and a pointer to the file length. The hypervisor then writes the file contents and its length to those pointers.

From the hypervisor point of view, there are two types of files. Normal files are held in a FileRefsByPath class. It contains a map from path to GuestFiles, which consist of a FileRef (a pointer and a length) and a GuestPtrs (kernel pointers to the buffer and length inside the virtual machine). This class does not own the memory of the files, it only has a reference to them. The actual file contents are owned by other objects, like the Corpus or the SharedFiles. The FileRefsByPath is held by the Vm in the private field `m_files`.

The other type of files are shared files, which are common to every virtual machine, and are held in a SharedFiles class that inherits from FileRefsByPath. However, SharedFiles owns the memory of the files in its field `m_file_contents`, indexed by filename. It is held by the Vms in the static field `s_shared_files`.

When the Vm calls `set_file()` or `set_shared_file()`, it first calls `set_file()` on the appropriate object (`m_files` or `m_shared_files`). Then, if the kernel already submitted the file pointers, it copies the new content of the file and its length into the kernel, successfully updating the file. Otherwise, the contents will be copied when the kernel submits the pointers with the hypercall SubmitFilePointers.

As an example, files from the host filesystem that want to be made accessible by the target program are created as shared files, so they are shared by every VM and are loaded in memory only once. This includes libraries, which are set as virtual files so the dynamic linker running inside the VM can access them and map them to user memory. On the other hand, the file that the target program uses as input is created as normal file. That way, it is specific to each VM, and its memory is actually held by the Corpus.

4.1.11 Tracing

The Tracing class is in charge of producing traces that are later consumed by the Markov model. Traces are a list of elements that are composed of a name and a value. As specified by the functional requirements, the value of the elements can be in two different units: cycles and instructions. These are counted by the kernel by using performance counters, and are read by the `get_tracing_measure()`, which reads the appropriate MSR of the VM depending on the unit (Section 1.4.4).

Furthermore, measurements can be performed in user space on each basic block, or in kernel space on each syscall. For syscalls, the kernel uses the hypercall NotifySyscallStart at the beginning of the syscall, which the Vm handles by calling the `prepare()` method on the Tracing object. The Tracing object then takes a measurement, and saves the value and syscall name on the `m_measure` field. After that, when the syscall ends, the kernel uses the hypercall NotifySyscallEnd, which executes the `trace()` method of the Tracing object, taking a second measurement and adding the difference between both measurements to the trace. That way, it measures how many cycles or instructions the syscall took.

On the other hand, when tracing user space basic blocks, the `trace()` method is called at the beginning of every basic block, and `m_measure` holds the measurement taken by `trace()`. That way, each element of the trace is the difference between every two subsequent measurements.

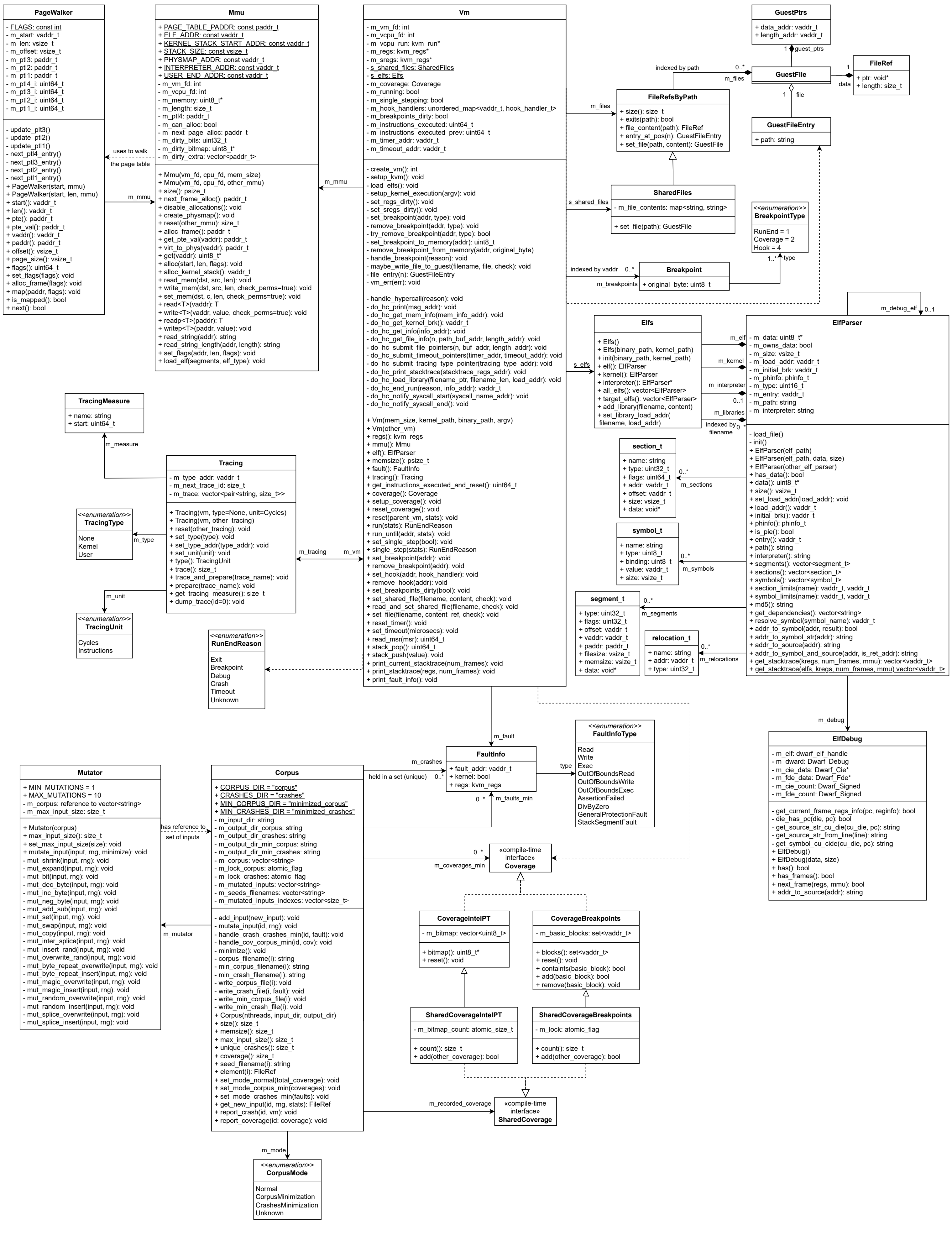

PageWalker
- FLAGS: const int
- m_start: vaddr_t
- m_len: vsize_t
- m_offset: vsize_t
- m_ptl3: paddr_t
- m_ptl2: paddr_t
- m_ptl1: paddr_t
- m_ptl4_i: uint64_t
- m_ptl3_i: uint64_t
- m_ptl2_i: uint64_t
- m_ptl1_i: uint64_t
- update_ptl3()
- update_ptl2()
- update_ptl1()
- next_ptl4_entry()
- next_ptl3_entry()
- next_ptl2_entry()
- next_ptl1_entry()
+ PageWalker(start, mmu)
+ PageWalker(start, len, mmu)
+ start(): vaddr_t
+ len(): vaddr_t
+ pte(): paddr_t
+ pte_val(): paddr_t
+ vaddr(): vaddr_t
+ paddr(): paddr_t
+ offset(): vsize_t
+ page_size(): vsize_t
+ flags(): uint64_t
+ set_flags(flags): void
+ alloc_frame(flags): void
+ map(paddr, flags): void
+ is_mapped(): bool
+ next(): bool
uses to walk the page table
m_mmu
Mmu
+ PAGE_TABLE_PADDR: const paddr_t
+ ELF_ADDR: const vaddr_t
+ KERNEL_STACK_START_ADDR: const vaddr_t
+ STACK_SIZE: const vsize_t
+ PHYSMAP_ADDR: const vaddr_t
+ INTERPRETER_ADDR: const vaddr_t
+ USER_END_ADDR: const vaddr_t
- m_vm_fd: int
- m_vcpu_fd: int
- m_memory: uint8_t*
- m_length: size_t
- m_ptl4: paddr_t
- m_can_alloc: bool
- m_next_page_alloc: paddr_t
- m_dirty_bits: uint32_t
- m_dirty_bitmap: uint8_t*
- m_dirty_extra: vector<paddr_t>
+ Mmu(vm_fd, cpu_fd, mem_size)
+ Mmu(vm_fd, cpu_fd, other_mmu)
+ size(): psize_t
+ next_frame_alloc(): paddr_t
+ disable_allocations(): void
+ create_physmap(): void
+ reset(other_mmu): size_t
+ alloc_frame(): paddr_t
+ get_pte_val(vaddr): paddr_t
+ virt_to_phys(vaddr): paddr_t
+ get(vaddr): uint8_t*
+ alloc(start, len, flags): void
+ alloc_kernel_stack(): vaddr_t
+ read_mem(dst, src, len): void
+ write_mem(dst, src, len, check_perms=true): void
+ set_mem(dst, c, len, check_perms=true): void
+ read<T>(vaddr): T
+ write<T>(vaddr, value, check_perms=true): void
+ readp<T>(paddr): T
+ writep<T>(paddr, value): void
+ read_string(addr): string
+ read_string_length(addr, length): string
+ set_flags(addr, len, flags): void
+ load_elf(segments, elf_type): void
m_mmu
Vm
- m_vm_fd: int
- m_vcpu_fd: int
- m_vcpu_run: kvm_run*
- m_regs: kvm_regs*
- m_sregs: kvm_regs*
- s_shared_files: SharedFiles
- s_elfs: Elfs
- m_coverage: Coverage
- m_running: bool
- m_single_stepping: bool
- m_hook_handlers: unordered_map<vaddr_t, hook_handler_t>
- m_breakpoints_dirty: bool
- m_instructions_executed: uint64_t
- m_instructions_executed_prev: uint64_t
- m_timer_addr: vaddr_t
- m_timeout_addr: vaddr_t
- create_vm(): int
- setup_kvm(): void
- load_elfs(): void
- setup_kernel_execution(argv): void
- set_regs_dirty(): void
- set_sregs_dirty(): void
- set_breakpoint(addr, type): void
- remove_breakpoint(addr, type): void
- try_remove_breakpoint(addr, type): bool
- set_breakpoint_to_memory(addr): uint8_t
- remove_breakpoint_from_memory(addr, original_byte)
- handle_breakpoint(reason): void
- maybe_write_file_to_guest(filename, file, check): void
- file_entry(n): GuestFileEntry
- vm_err(err): void
- handle_hypercall(reason): void
- do_hc_print(msg_addr): void
- do_hc_get_mem_info(mem_info_addr): void
- do_hc_get_kernel_brk(): vaddr_t
- do_hc_get_info(info_addr): void
- do_hc_get_file_info(n, path_buf_addr, length_addr): void
- do_hc_submit_file_pointers(n, buf_addr, length_addr): void
- do_hc_submit_timeout_pointers(timer_addr, timeout_addr): void
- do_hc_submit_tracing_type_pointer(tracing_type_addr): void
- do_hc_print_stacktrace(stacktrace_regs_addr): void
- do_hc_load_library(filename_ptr, filename_len, load_addr): void
- do_hc_end_run(reason, info_addr): void
- do_hc_notify_syscall_start(syscall_name_addr): void
- do_hc_notify_syscall_end(): void
+ Vm(mem_size, kernel_path, binary_path, argv)
+ Vm(other_vm)
+ regs(): kvm_regs
+ mmu(): Mmu
+ elf(): ElfParser
+ memsize(): psize_t
+ fault(): FaultInfo
+ tracing(): Tracing
+ get_instructions_executed_and_reset(): uint64_t
+ coverage(): Coverage
+ setup_coverage(): void
+ reset_coverage(): void
+ reset(parent_vm, stats): void
+ run(stats): RunEndReason
+ run_until(addr, stats): void
+ set_single_step(bool): void
+ single_step(stats): RunEndReason
+ set_breakpoint(addr): void
+ remove_breakpoint(addr): void
+ set_hook(addr, hook_handler): void
+ remove_hook(addr): void
+ set_breakpoints_dirty(bool): void
+ set_shared_file(filename, content, check): void
+ read_and_set_shared_file(filename, check): void
+ set_file(filename, content_ref, check): void
+ reset_timer(): void
+ set_timeout(microsecs): void
+ read_msr(msr): uint64_t
+ stack_pop(): uint64_t
+ stack_push(value): void
+ print_current_stacktrace(num_frames): void
+ print_stacktrace(regs, num_frames): void
+ print_fault_info(): void
GuestPtrs
+ data_addr: vaddr_t
+ length_addr: vaddr_t
1 guest_ptrs
indexed by path
0..*
m_files
GuestFile
1
data
FileRef
+ ptr: void*
+ length: size_t
1 file
m_files
FileRefsByPath
+ size(): size_t
+ exits(path): bool
+ file_content(path): FileRef
+ entry_at_pos(n): GuestFileEntry
+ set_file(path, content): GuestFile
GuestFileEntry
+ path: string
s_shared_files
SharedFiles
- m_file_contents: map<string, string>
+ set_file(path): GuestFile
<<enumeration>>
BreakpointType
RunEnd = 1
Coverage = 2
Hook = 4
1..* type
indexed by vaddr 0..*
m_breakpoints
Breakpoint
+ original_byte: uint8_t
m_debug_elf 0..1
s_elfs
Elfs
+ Elfs()
+ Elfs(binary_path, kernel_path)
+ init(binary_path, kernel_path)
+ elf(): ElfParser
+ kernel(): ElfParser
+ interpreter(): ElfParser*
+ all_elfs(): vector<ElfParser>
+ target_elfs(): vector<ElfParser>
+ add_library(filename, content)
+ set_library_load_addr(filename, load_addr)
m_elf
m_kernel
m_interpreter
0..1
m_libraries
indexed by filename
0..*
ElfParser
- m_data: uint8_t*
- m_owns_data: bool
- m_size: vsize_t
- m_load_addr: vaddr_t
- m_initial_brk: vaddr_t
- m_phinfo: phinfo_t
- m_type: uint16_t
- m_entry: vaddr_t
- m_path: string
- m_interpreter: string
- load_file()
- init()
+ ElfParser(elf_path)
+ ElfParser(elf_path, data, size)
+ ElfParser(other_elf_parser)
+ has_data(): bool
+ data(): uint8_t*
+ size(): vsize_t
+ set_load_addr(load_addr): void
+ load_addr(): vaddr_t
+ initial_brk(): vaddr_t
+ phinfo(): phinfo_t
+ is_pie(): bool
+ entry(): vaddr_t
+ path(): string
+ interpreter(): string
+ segments(): vector<segment_t>
+ sections(): vector<section_t>
+ symbols(): vector<symbol_t>
+ section_limits(name): vaddr_t, vaddr_t
+ symbol_limits(name): vaddr_t, vaddr_t
+ md5(): string
+ get_dependencies(): vector<string>
+ resolve_symbol(symbol_name): vaddr_t
+ addr_to_symbol(addr, result): bool
+ addr_to_symbol_str(addr): string
+ addr_to_source(addr): string
+ addr_to_symbol_and_source(addr, is_ret_addr): string
+ get_stacktrace(kregs, num_frames, mmu): vector<vaddr_t>
+ get_stacktrace(elfs, kregs, num_frames, mmu) vector<vaddr_t>
section_t
+ name: string
+ type: uint32_t
+ flags: uint64_t
+ addr: vaddr_t
+ offset: vaddr_t
+ size: vsize_t
+ data: void*
0..* m_sections
symbol_t
+ name: string
+ type: uint8_t
+ binding: uint8_t
+ value: vaddr_t
+ size: vsize_t
0..* m_symbols
segment_t
+ type: uint32_t
+ flags: uint32_t
+ offset: vaddr_t
+ vaddr: vaddr_t
+ paddr: paddr_t
+ filesize: vsize_t
+ memsize: vsize_t
+ data: void*
0..* m_segments
relocation_t
+ name: string
+ addr: vaddr_t
+ type: uint32_t
0..* m_relocations
m_debug
ElfDebug
- m_elf: dwarf_elf_handle
- m_dward: Dwarf_Debug
- m_cie_data: Dwarf_Cie*
- m_fde_data: Dwarf_Fde*
- m_cie_count: Dwarf_Signed
- m_fde_count: Dwarf_Signed
- get_current_frame_regs_info(pc, reginfo): bool
- die_has_pc(die, pc): bool
- get_source_str_cu_die(cu_die, pc): string
- get_source_str_from_line(line): string
- get_symbol_cu_cide(cu_die, pc): string
+ ElfDebug()
+ ElfDebug(data, size)
+ has(): bool
+ has_frames(): bool
+ next_frame(regs, mmu): bool
+ addr_to_source(addr): string
TracingMeasure
+ name: string
+ start: uint64_t
m_measure
Tracing
- m_type_addr: vaddr_t
- m_next_trace_id: size_t
- m_trace: vector<pair<string, size_t>>
+ Tracing(vm, type=None, unit=Cycles)
+ Tracing(vm, other_tracing)
+ reset(other_tracing): void
+ set_type(type): void
+ set_type_addr(type_addr): void
+ set_unit(unit): void
+ type(): TracingUnit
+ trace(): size_t
+ trace_and_prepare(trace_name): void
+ prepare(trace_name): void
+ get_tracing_measure(): size_t
+ dump_trace(id=0): void
<<enumeration>>
TracingType
None
Kernel
User
m_type
m_unit
<<enumeration>>
TracingUnit
Cycles
Instructions
m_tracing
m_vm
<<enumeration>>
RunEndReason
Exit
Breakpoint
Debug
Crash
Timeout
Unknown
m_fault
FaultInfo
+ fault_addr: vaddr_t
+ kernel: bool
+ regs: kvm_regs
type
<<enumeration>>
FaultInfoType
Read
Write
Exec
OutOfBoundsRead
OutOfBoundsWrite
OutOfBoundsExec
AssertionFailed
DivByZero
GeneralProtectionFault
StackSegmentFault
m_crashes
held in a set (unique) 0..*
0..* m_faults_min
Mutator
+ MIN_MUTATIONS = 1
+ MAX_MUTATIONS = 10
- m_corpus: reference to vector<string>
- m_max_input_size: size_t
+ Mutator(corpus)
+ max_input_size(): size_t
+ set_max_input_size(size): void
+ mutate_input(input, rng, minimize): void
- mut_shrink(input, rng): void
- mut_expand(input, rng): void
- mut_bit(input, rng): void
- mut_dec_byte(input, rng): void
- mut_inc_byte(input, rng): void
- mut_neg_byte(input, rng): void
- mut_add_sub(input, rng): void
- mut_set(input, rng): void
- mut_swap(input, rng): void
- mut_copy(input, rng): void
- mut_inter_splice(input, rng): void
- mut_insert_rand(input, rng): void
- mut_overwrite_rand(input, rng): void
- mut_byte_repeat_overwrite(input, rng): void
- mut_byte_repeat_insert(input, rng): void
- mut_magic_overwrite(input, rng): void
- mut_magic_insert(input, rng): void
- mut_random_overwrite(input, rng): void
- mut_random_insert(input, rng): void
- mut_splice_overwrite(input, rng): void
- mut_splice_insert(input, rng): void
has reference to set of inputs
m_mutator
Corpus
+ CORPUS_DIR = "corpus"
+ CRASHES_DIR = "crashes"
+ MIN_CORPUS_DIR = "minimized_corpus"
+ MIN_CRASHES_DIR = "minimized_crashes"
- m_input_dir: string
- m_output_dir_corpus: string
- m_output_dir_crashes: string
- m_output_dir_min_corpus: string
- m_output_dir_min_crashes: string
- m_corpus: vector<string>
- m_lock_corpus: atomic_flag
- m_lock_crashes: atomic_flag
- m_mutated_inputs: vector<string>
- m_seeds_filenames: vector<string>
- m_mutated_inputs_indexes: vector<size_t>
- add_input(new_input): void
- mutate_input(id, rng): void
- handle_crash_crashes_min(id, fault): void
- handle_cov_corpus_min(id, cov): void
- minimize(): void
- corpus_filename(i): string
- min_corpus_filename(i): string
- min_crash_filename(i): string
- write_corpus_file(i): void
- write_crash_file(i, fault): void
- write_min_corpus_file(i): void
- write_min_crash_file(i): void
+ Corpus(nthreads, input_dir, output_dir)
+ size(): size_t
+ memsize(): size_t
+ max_input_size(): size_t
+ unique_crashes(): size_t
+ coverage(): size_t
+ seed_filename(i): string
+ element(i): FileRef
+ set_mode_normal(total_coverage): void
+ set_mode_corpus_min(coverages): void
+ set_mode_crashes_min(faults): void
+ get_new_input(id, rng, stats): FileRef
+ report_crash(id, vm): void
+ report_coverage(id: coverage): void
0..*
m_coverages_min
«compile-time interface»
Coverage
CoverageIntelPT
- m_bitmap: vector<uint8_t>
+ bitmap(): uint8_t*
+ reset(): void
CoverageBreakpoints
- m_basic_blocks: set<vaddr_t>
+ blocks(): set<vaddr_t>
+ reset(): void
+ contains(basic_block): bool
+ add(basic_block): bool
+ remove(basic_block): void
SharedCoverageIntelPT
- m_bitmap_count: atomic_size_t
+ count(): size_t
+ add(other_coverage): bool
SharedCoverageBreakpoints
- m_lock: atomic_flag
+ count(): size_t
+ add(other_coverage): bool
m_recorded_coverage
«compile-time interface»
SharedCoverage
m_mode
<<enumeration>>
CorpusMode
Normal
CorpusMinimization
CrashesMinimization
Unknown

Figure 4.1: UML hypervisor

4.1.12 Fuzzing loop

The fuzzing loop is the continuous process of running inputs under the target program in order to detect crashes. It starts after the VM has been created and set up, and the kernel has initialised its different components and started to run the target program. At that point, the VM is stopped, and the hypervisor creates a number of threads. Each thread forks the VM, and uses the resulting VM to run the main fuzzing loop in parallel, which is as follows:

1. Get a new input. The corpus and mutator create a new input from the existing ones by mutations, as explained in Sections 4.1.7 and 4.1.8.

2. Write the input to the virtual machine memory. If the target program reads the input from a file, then the input will be a virtual file, and it will be written to the buffer in kernel memory, as described in Section 4.1.10. On the other hand, if the target program reads the input from a buffer, it will be written directly into its memory.

3. Run the virtual machine until the kernel notifies the end of the run. This can be because of three main reasons: the target program used the system call `exit()`, or it crashed, or it time-outed.

4. Get the code coverage reached in the run. If the input has executed new interesting code (for example, code that has not been run previously with any other input), then it is saved in the corpus to generate new inputs from it.

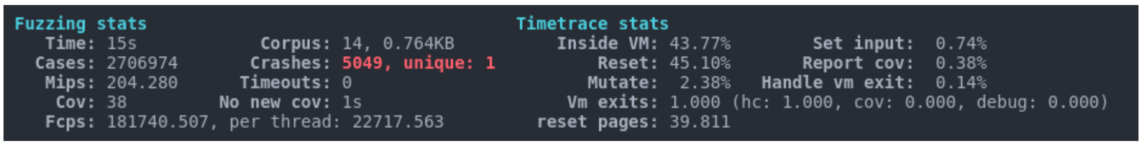
5. Restore the virtual machine to its original state when it was forked.

This loop will run indefinitely until the process is interrupted.

4.1.13 Stats

While the fuzzer is running, there is another thread that prints useful stats to the screen, as can be seen in Figure 4.2. On the left side, there are fuzzing statistics. This includes the time the fuzzer has been running for, how many fuzz cases it has executed, how many of them were crashes or timeouts, its execution speed in MIPS (millions of instructions per second) and in FCPS (fuzz cases per second), how much coverage it has found and the number of elements in the corpus and their total size.

On the other side we can find timetrace stats. These include information about where the fuzzer is spending most of the time. As fuzzing is a very performance-related activity, this information is important enough to be displayed at the front. The main parts where the fuzzer can be spending time are inside the VM running the target program, resetting the VM, mutating inputs, copying them to the VM memory, reporting and analysing coverage, and handling VM exits. The best situation would be spending as much time as possible inside the VM running the target program, as that would be mean that the fuzzing overhead is minimum. Apart from that, these stats also include the average number of VM exits and pages reset per run, which we want to minimize.



```
Fuzzing stats                                       Timetrace stats
   Time: 15s              Corpus: 14, 0.764KB           Inside VM: 43.77%        Set input:   0.74%
  Cases: 2706974         Crashes: 5049, unique: 1           Reset: 45.10%       Report cov:   0.38%
   Mips: 204.280        Timeouts: 0                        Mutate:  2.38%   Handle vm exit:   0.14%
    Cov: 38          No new cov: 1s                      Vm exits: 1.000 (hc: 1.000, cov: 0.000, debug: 0.000)
   Fcps: 181740.507, per thread: 22717.563            reset pages: 39.811
```

Figure 4.2: Stats displayed on the terminal by KVM-FUZZ

Furthermore, when a crash is found, the hypervisor prints the fault information. This includes the type of crash, the state of the registers at the moment of the crash, and a stacktrace. The stacktrace displays not only raw addresses, but also the symbols and location in source code they correspond to. If source code information is not available for one of the frames, the binary it belongs to is printed instead. Listing 4.1 shows an example of the information displayed when KVM-FUZZ finds a real vulnerability on an outdated version of `readelf`, a program of the binutils package.

```
[CRASH: AssertionFailed] RIP: 0x7ffaaab42a7c, address: 0x0
rip: 0x00007ffaaab42a7c
rax: 0x00000000000000ea  rbx: 0x00007ffaaaaab600  rcx: 0x0000000000000000  rdx: 0x0000000000000006
rsi: 0x00000000000004d2  rdi: 0x00000000000004d2  rsp: 0x00007fffffff7b0  rbp: 0x00000000000004d2
r8:  0x00007ffffffff880  r9:  0x000a2e64656c6961  r10: 0x0000000000000008  r11: 0x0000000000000000
r12: 0x0000000000000006  r13: 0x0000000000000016  r14: 0x000000000046ec70  r15: 0x0000000000000001
rflags: 0x0000000000000246

#0 0x00007ffaaab42a7c pthread_kill + 0x12b from /lib/x86_64-linux-gnu/libc.so.6
#1 0x00007ffaaaaee476 gsignal + 0x15 from /lib/x86_64-linux-gnu/libc.so.6
#2 0x00007ffaaaad47f3 abort + 0xd2 from /lib/x86_64-linux-gnu/libc.so.6
#3 0x00007ffaaaad471b _dl_audit_preinit@plt + 0x4a from /lib/x86_64-linux-gnu/libc.so.6
#4 0x00007ffaaaae5e96 __assert_fail + 0x45 from /lib/x86_64-linux-gnu/libc.so.6
#5 0x000000000041405a find_section + 0x119 at ./binutils-2.30/binutils/readelf.c:658
#6 0x0000000000433b0d process_object + 0xd1c at ./binutils-2.30/binutils/readelf.c:15538
#7 0x0000000000407ead main + 0x61c at ./binutils-2.30/binutils/readelf.c:19084
#8 0x00007ffaaaad5d90 __libc_init_first + 0x8f from /lib/x86_64-linux-gnu/libc.so.6
#9 0x00007ffaaaad5e40 __libc_start_main + 0x7f from /lib/x86_64-linux-gnu/libc.so.6
#10 0x000000000040817e _start + 0x2d from ./test_bins/binutils-2.30-build/binutils/readelf
```

Listing 4.1: Fault information displayed when a crash is found. Based on a real vulnerability in readelf, from binutils 2.30.

4.2 Kernel

The kernel runs inside the virtual machine, and its main goal is to emulate the target program syscalls. In order to do so, it must also implement memory management, paging, interrupts, virtual files, multi-tasking, and other components. It is a monolithic kernel for the x86-64 architecture. It is non-preemptive, which means it can not be interrupted, so interrupts can only happen in user space. Also, it runs on a single CPU, so there is no real parallelism inside the kernel. Instead, the parallelism in KVM-FUZZ is achieved by the hypervisor running multiple VMs on different CPUs.

UML class diagrams described in this section follow the Zig convention for types: `?type` is an optional `type`, which can be `null` or a value of the given type. Similarly, `!type` can be an error or a value of the given type. Furthermore, classes whose names are underlined are modules rather than classes, that is, they are not instantiated and all its members are static.

The architecture and design of the kernel is divided in different packages, which are detailed in the following sections.

4.2.1 Hypercalls

Hypercalls are calls from the kernel to the hypervisor. They work by executing a privileged instruction that triggers a VM exit, transferring execution to the hypervisor, which handles the request. They are used by different components of the kernel, mostly on initialisation. The different hypercalls are listed below:

- Print: prints a string to stdout. Since the kernel does not have access to standard output of the hypervisor, a hypercall is needed to print output. It is based on the private function `printChar()`, which is buffered so not every character written to stdout triggers a hypercall.
- GetMemInfo: gets information about the state of the physical memory, including where the free region starts (how much the hypervisor has allocated), its length, and the address where the physmap is mapped in memory. This also transfers control of physical memory from the hypervisor to the kernel, which means the hypervisor can no longer allocate physical memory. It is called from the PMM on initialisation (Section 4.2.2).
- GetKernelBrk: returns the address of the kernel break, which is the next page after the kernel in memory. This is commonly where the heap is located in userspace programs, and the kernel heap mimics that behaviour. It is called by the VMM on initialisation (Section 4.2.2).
- GetInfo: returns general information about the VM, such as the name of the loaded ELF, its break address, number of files, user entry point, range of addresses of the interpreter if any, etc. It is called from the kernel entry point, and used to initialise multiple components.
- GetFileInfo: gets the filename and size of every file. It is called from the file manager (Section 4.2.4) to allocate a buffer for the file contents.

- SubmitFilePointers: submits pointers to the file buffer and length, so the hypervisor can fill them with the appropriate file contents and length (Section 4.1.10). It is called from the file manager on initialisation.

- SubmitTimeoutPointers: submits pointers to the timer and timeout values (Section 4.2.3). This allows the hypervisor to reset the timer and change the timeout. It is called from the perf module.

- SubmitTracingTypePointer: submits pointer to the tracing type. This allows the hypervisor to enable and disable tracing, and set its mode to kernel or user (Section 4.1.11). It is called from the hypercalls module on initialisation.

- PrintStackTrace: requests the hypervisor to print a stacktrace starting from some registers (Section 4.1.5). For example, these registers can be the user registers, which would print the stacktrace of the current execution point in the user program. If no registers provided, the registers at the point of the hypercall will be used, printing the stacktrace of the kernel. It is only called for debugging and virtual machine introspection.

- LoadLibrary: tells the hypervisor that the user program has loaded a library at a certain address. That way, the hypervisor can set that address as the base address for the library (on its corresponding ElfParser object) to correctly resolve symbols and stacktraces. It is called from the `mmap()` syscall handler when the syscall was invoked from the interpreter binary, which is in charge of loading user libraries.

- EndRun: tells the hypervisor that the run ended for a specific reason, which can be because the user program finished, crashed, or timeouted. If it crashed, it also includes information about the fault that caused it. Depending on the reason, it is called from the `exit()` syscall handler, the different exception handlers, or the perf module.

- NotifySyscallStart and NotifySyscallEnd: tells the hypervisor that a system call started and ended, for tracing purposes (Section 4.1.11). It is called from the syscall handler.

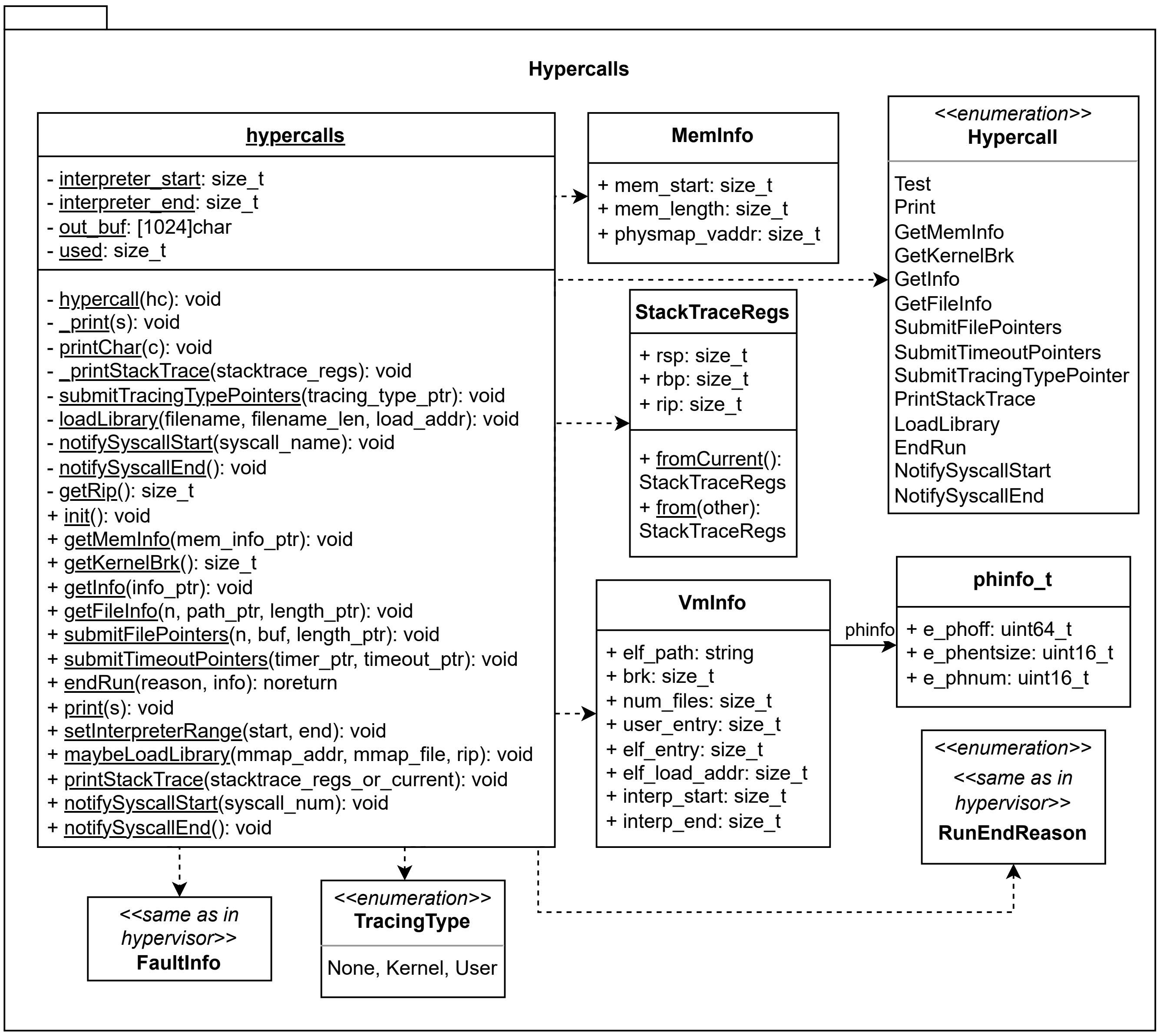


Figure 4.3: UML kernel hypercalls

4.2.2 Memory management

As explained in Section 1.4.4, memory management is one of the core functionalities of kernels, as many others depend on it. Its design in KVM-FUZZ is described in the UML class diagram in Figure 4.4. It is further divided in the following parts.

Paging

In order to manage paging, the following data structures and classes exist. First, a PageTableEntry represents a PTE in memory. It is an 8-bytes-long packed structure whose layout is specified by the x86-64 architecture. It contains `phys`, which indicates the physical frame that page is mapped to, and many flags indicating its permissions and other options. Note there are some bits that the architecture marks as unused, but are used by the kernel for other specific purposes. They are `shared`, which indicates whether an user page should be copied when the task calls `fork()`, and `ref_count`, which serves as reference counter for the frame the PTE is mapped to. PageTableEntry has some helper methods to access and modify its fields more easily, such as `setFrameBase()` or `flags()`.

The class PageTable represents a top-level (level 4) page table. Its only field `ptl4` points to 512 PageTableEntry's consecutive in memory. This class implements the logic for dealing with page tables, including mapping a given virtual address to a physical one with given options, changing permissions, and cloning memory. In order to map memory, the PageTable sometimes needs to create intermediary (levels 3, 2 and 1) page tables. This requires allocating physical memory, which is why it depends on the PhysicalMemoryManager. There is a PageTable for each address space, that is, for each task running in userspace, and an additional one for the kernel.

The kernel page table is represented by the KernelPageTable class that inherits from PageTable. This class makes sure that every map and unmap operation is performed to kernel space. It also sets the shared bit of the last entry of the page table level 4, which represents the virtual memory region where the kernel lives. That way, the kernel is mapped to every address space, and it is not duplicated when this address space is cloned. Finally, the KernelPageTable also implements the reference counting logic for frames.

Physical Memory Manager

The physical memory manager exposes an interface for allocating and freeing physical memory in page-size chunks called frames. Its functioning is as follows. It has a vector of free frames called `free_frames`. In `allocFrame()`, it returns a frame from this list. Then, in `freeFrame()` it adds the freed frame to the list. Methods `allocFrames()` and `freeFrames()`

are the same, but operate with multiple frames. Note that `allocFrames()` does not need to return contiguous frames, as they can be later mapped so that they are virtually contiguous memory.

When initialised, the PMM performs the hypercall `GetMemInfo`. Upon that point, management of physical memory is transferred from the hypervisor to the kernel, that is, the hypervisor can no longer allocate physical memory. With that information, the PMM calculates how much free memory is left, allocates a small slice of that memory for the vector of free frames `free_frames`, and adds the rest of the frames to the vector. It is important to do it that way, because there is no VMM or heap to provide access to dynamic memory.

It was also considered to implement the PMM as a bitmap, where each bit indicated if a frame is free or not. However, that would make `allocFrame()` O(N), where N is the size of the bitmap, as it would require iterating it to find a free frame. In constrast, the current design has `allocFrame()` and `freeFrame()` both O(1).

The PMM also provides an interface to the physmap. The physmap or physical map is a virtual mapping of all the physical memory. It is needed for the kernel to be able to write to physical memory, for example when dealing with page tables. The PMM implements `physToVirt()`, which given a physical address, it returns its corresponding virtual address inside the physmap, and `virtToPhys()`, which performs the opposite operation.

Virtual Memory Manager

The VMM is the module in charge of managing kernel virtual memory. It owns the kernel page table, and implements methods for allocating and freeing pages in the kernel address space. Note that the concept of "allocating memory in an address space" implies first allocating physical memory, and then mapping it into the address space via its page table.

The VMM has a bitmap that indicates which pages are free (not mapped) and which are allocated (mapped). Therefore, `allocPages(n)` first looks for a range of `n` contiguous pages that are free by using the bitmap. Then, for each of them, it allocates a frame of physical memory from the PMM, maps the page to that frame with given options, and marks the page as allocated in the bitmap. Similarly, `freePages()` iterates every page, marking it as free and unmapping it.

The bitmap is a static bitmap, that is, has a fixed, static size determined at compile time. It can not be dynamic, because dynamic memory is provided by the VMM, and the VMM can not depend on itself. This means the maximum memory used by the virtual machine is bounded at compile time.

Heap

The heap package is the abstraction for dynamic memory allocation that the rest of the kernel uses. The Allocator interface defines a way of allocating and freeing blocks of memory, similar to how `malloc()` and `free()` work in C. Every object that implements said interface is called an allocator. There is no global, implicit allocator, so every function or object that needs to allocate memory needs to receive or save an allocator.

There are three allocator implementations. The first one is PageAllocator, which forwards every allocation to the VMM. In order to allocate a chunk of a given length, it calculates the number of pages it takes and allocates them with write permissions from the VMM. Similarly, `free()` simply frees the underlying pages. The PageAllocator is thought to be used as a backing allocator for other, more complex implementations.

The second is the HeapAllocator. It has an allocation area from which memory allocations are served, which starts at `base` and has a size `size`. It also keeps an offset `used`, which is the offset of the next allocation inside that memory region. Therefore, every allocation simply returns `base + used`, and increases `used` by the size of the allocation, in a stack-like fashion. This means every two allocations are contiguous in memory, which helps cache locality. Also, allocating is a O(1) operation. However, freeing is a no-operation, which means there is no memory reuse. When there is no more free space, the memory region is enlarged via the VMM.

Finally, the third one is BlockAllocator. It keeps 9 linked lists of free blocks of different sizes (powers of 2 from 8 to 2048). For each allocation request, its size is rounded to the closest higher power of 2 and served from that free list. If there are no free blocks of that size, a page is allocated with the PageAllocator and split in chunks of that size, which are added to the free list. Similarly, freeing a chunk simply adds it to its corresponding free list. This means both `alloc()` and `free()` are O(1) operations. On the other hand, memory allocation requests of size larger than the largest free list (2048) are forwarded to the PageAllocator.

AddressSpace

The AdressSpace class represents the address space of an user process. It contains a PageTable and a RegionManager, and provides methods to satisfy user memory allocation requests. The method `mapRange()` allocates physical memory and maps it to a given range of memory in the address space with given permissions. The method `mapRangeAnywhere()` performs the same task, but it does not receive the start address of the region. Instead, it will create the mapping wherever there is space. Similarly, `unmapRange()` and `setRangePerms()` respectively unmap and modify the permissions of a mapped range. These methods are thought to be called from `mmap()`, `munmap()` and `mprotect()` system calls.

In order to keep track of the memory regions allocated inside an user process, AddressSpace has an instance of RegionManager. This class has a list of mapped Regions, which represent mapped memory regions limited by a start and an end address. When created, the object will receive a `total` region, and will verify that none of the pages inside that range are mapped by checking the page table. Then, it provides the methods `setMapped()` and `setNotMapped()` to set a region of memory as mapped or not mapped. It is in charge of creating, expanding, shrinking and merging elements in its list of regions, according to what is needed. Finally, it provides `findNotMapped()`, which finds the start address of a region of not mapped memory of a given length inside the `total` region. That is the whole purpose of this class: providing a quick way for `mapRangeAnywhere()` to find space for a memory request in the address space, instead of having to iterate the page table.

MemSafe

The MemSafe package includes some utilities to represent and access user memory safely. When the user provides a pointer as argument to a system call, the kernel must verify that it is valid. There are three different reasons an user pointer could be invalid:

1. It points to kernel memory.
2. It points to unmapped memory.
3. It points to user mapped memory with insufficient permissions.

Issue 1 could lead to an user overwriting kernel memory, allowing him to escalate privileges, while issues 2 and 3 could crash the kernel (denial of service). Therefore, user pointers should be treated cautiously. With that goal, UserPtr and UserSlice are classes that wrap an user pointer or slice (pointer and length) so they are not mixed with regular pointers and slices. They provide creation methods from flat, integer values passed as arguments for syscalls, and check they belong to user range by using functions like `isAddressInUserRange()`, solving issue 1.

Regarding issues 2 and 3, there are different strategies to access a potentially invalid user pointer. The first is to verify the validity of the pointer ahead of time by checking that the pointed memory region is mapped in the current address space, and that it has correct permissions. That would involve accessing the page table, which is a costly operation to do for every access to user memory. Furthermore, on a multi-core or preemptive kernel, this could lead to a Time-Of-Check to Time-Of-Use (TOCTOU) vulnerability, as the user could modify the mapping in the small window between the check and the actual access to memory.

A better solution is to just perform the memory access, and if it fails, catch the generated page fault exception and return an error. The details are as follows. The private function `copyBase()` copies bytes from `src` to `dest`. If any of those pointers are invalid, accessing them will generate a page fault exception, and the CPU will call the page fault handler (Section 1.4.4). The page fault handler will check if the fault occurred in kernel space, and in that case it will call `handleSafeAccessFault()` inside the MemSafe package. This function will check if the invalid memory access occurred inside the `copyBase()` function. If that is the case, it will modify the instruction pointer of the interrupt frame, so that the interrupt returns to `copyBase()`, which will return an error. That way, when pointers are valid, `copyBase()` performs the copy with zero performance penalty, and when they are invalid, it returns an error. Finally, the SafeMem package exposes the functions `copyToUser()`, `copyFromUser()` and similar, which accept UserPtr and UserSlice as arguments and just call `copyBase()` on their underlying pointers.

This last approach is the one used by Linux nowadays to access user memory from syscall handlers, and they call it exception tables [37].

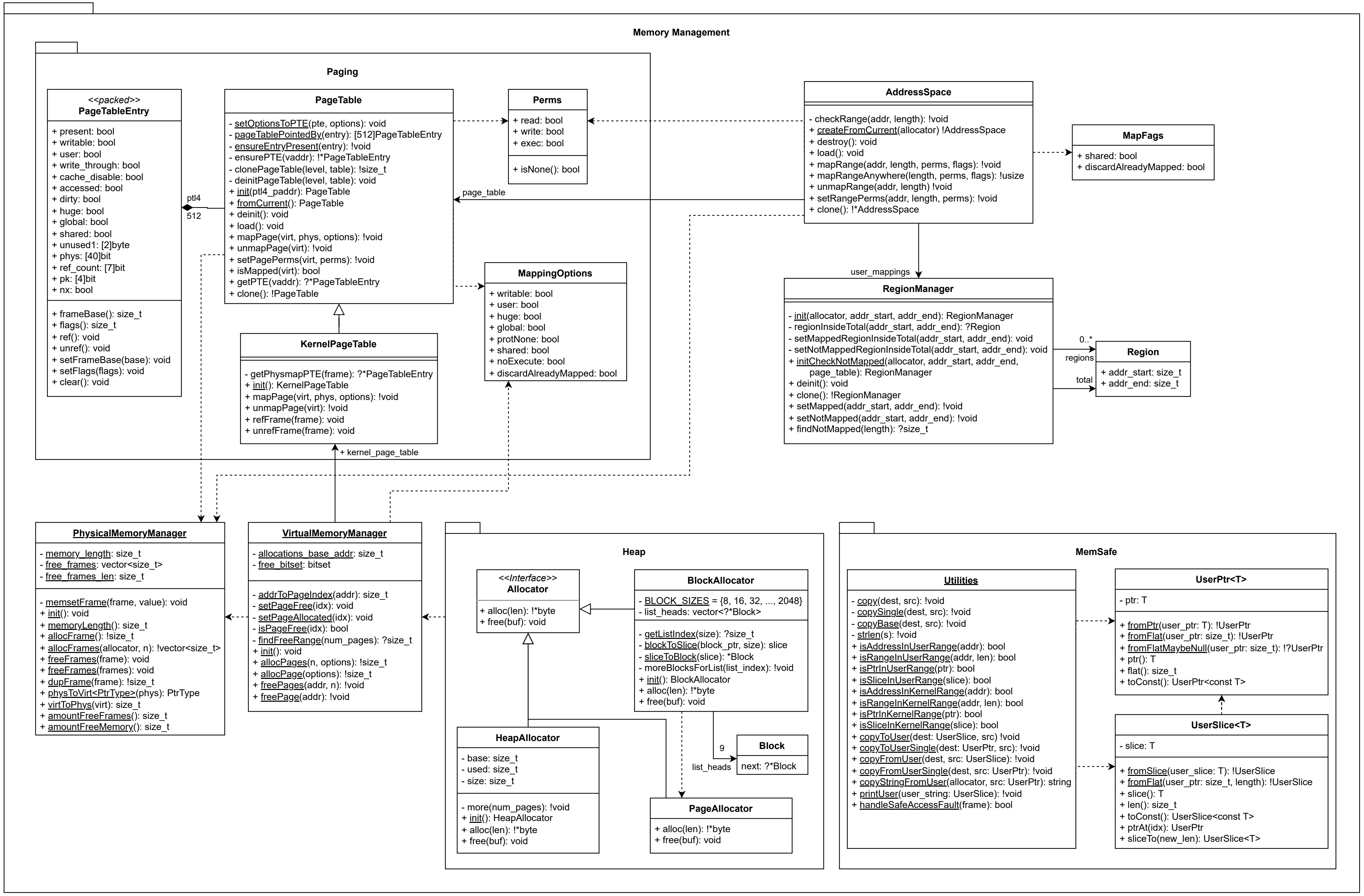

Memory Management
Paging
<<packed>>
PageTableEntry
+ present: bool
+ writable: bool
+ user: bool
+ write_through: bool
+ cache_disable: bool
+ accessed: bool
+ dirty: bool
+ huge: bool
+ global: bool
+ shared: bool
+ unused1: [2]byte
+ phys: [40]bit
+ ref_count: [7]bit
+ pk: [4]bit
+ nx: bool
+ frameBase(): size_t
+ flags(): size_t
+ ref(): void
+ unref(): void
+ setFrameBase(base): void
+ setFlags(flags): void
+ clear(): void
ptl4
512
PageTable
- setOptionsToPTE(pte, options): void
- pageTablePointedBy(entry): [512]PageTableEntry
- ensureEntryPresent(entry): !void
- ensurePTE(vaddr): !*PageTableEntry
- clonePageTable(level, table): !size_t
- deinitPageTable(level, table): void
+ init(ptl4_paddr): PageTable
+ fromCurrent(): PageTable
+ deinit(): void
+ load(): void
+ mapPage(virt, phys, options): !void
+ unmapPage(virt): !void
+ setPagePerms(virt, perms): !void
+ isMapped(virt): bool
+ getPTE(vaddr): ?*PageTableEntry
+ clone(): !PageTable
Perms
+ read: bool
+ write: bool
+ exec: bool
+ isNone(): bool
MappingOptions
+ writable: bool
+ user: bool
+ huge: bool
+ global: bool
+ protNone: bool
+ shared: bool
+ noExecute: bool
+ discardAlreadyMapped: bool
KernelPageTable
- getPhysmapPTE(frame): ?*PageTableEntry
+ init(): KernelPageTable
+ mapPage(virt, phys, options): !void
+ unmapPage(virt): !void
+ refFrame(frame): void
+ unrefFrame(frame): void
+ kernel_page_table
page_table
AddressSpace
- checkRange(addr, length): !void
+ createFromCurrent(allocator) !AddressSpace
+ destroy(): void
+ load(): void
+ mapRange(addr, length, perms, flags): !void
+ mapRangeAnywhere(length, perms, flags): !usize
+ unmapRange(addr, length) !void
+ setRangePerms(addr, length, perms): !void
+ clone(): !*AddressSpace
MapFags
+ shared: bool
+ discardAlreadyMapped: bool
user_mappings
RegionManager
- init(allocator, addr_start, addr_end): RegionManager
- regionInsideTotal(addr_start, addr_end): ?Region
- setMappedRegionInsideTotal(addr_start, addr_end): void
- setNotMappedRegionInsideTotal(addr_start, addr_end): void
+ initCheckNotMapped(allocator, addr_start, addr_end, page_table): RegionManager
+ deinit(): void
+ clone(): !RegionManager
+ setMapped(addr_start, addr_end): !void
+ setNotMapped(addr_start, addr_end): !void
+ findNotMapped(length): ?size_t
0..*
regions
total
Region
+ addr_start: size_t
+ addr_end: size_t
PhysicalMemoryManager
- memory_length: size_t
- free_frames: vector<size_t>
- free_frames_len: size_t
- memsetFrame(frame, value): void
+ init(): void
+ memoryLength(): size_t
+ allocFrame(): !size_t
+ allocFrames(allocator, n): !vector<size_t>
+ freeFrames(frame): void
+ freeFrames(frames): void
+ dupFrame(frame): !size_t
+ physToVirt<PtrType>(phys): PtrType
+ virtToPhys(virt): size_t
+ amountFreeFrames(): size_t
+ amountFreeMemory(): size_t
VirtualMemoryManager
- allocations_base_addr: size_t
- free_bitset: bitset
- addrToPageIndex(addr): size_t
- setPageFree(idx): void
- setPageAllocated(idx): void
- isPageFree(idx): bool
- findFreeRange(num_pages): ?size_t
+ init(): void
+ allocPages(n, options): !size_t
+ allocPage(options): !size_t
+ freePages(addr, n): !void
+ freePage(addr): !void
Heap
<<Interface>>
Allocator
+ alloc(len): !*byte
+ free(buf): void
BlockAllocator
- BLOCK_SIZES = {8, 16, 32, ..., 2048}
- list_heads: vector<?*Block>
- getListIndex(size): ?size_t
- blockToSlice(block_ptr, size): slice
- sliceToBlock(slice): *Block
- moreBlocksForList(list_index): !void
+ init(): BlockAllocator
+ alloc(len): !*byte
+ free(buf): void
9
list_heads
Block
next: ?*Block
HeapAllocator
- base: size_t
- used: size_t
- size: size_t
- more(num_pages): !void
+ init(): HeapAllocator
+ alloc(len): !*byte
+ free(buf): void
PageAllocator
+ alloc(len): !*byte
+ free(buf): void
MemSafe
Utilities
- copy(dest, src): !void
- copySingle(dest, src): !void
- copyBase(dest, src): !void
- strlen(s): !void
+ isAddressInUserRange(addr): bool
+ isRangeInUserRange(addr, len): bool
+ isPtrInUserRange(ptr): bool
+ isSliceInUserRange(slice): bool
+ isAddressInKernelRange(addr): bool
+ isRangeInKernelRange(addr, len): bool
+ isPtrInKernelRange(ptr): bool
+ isSliceInKernelRange(slice): bool
+ copyToUser(dest: UserSlice, src) !void
+ copyToUserSingle(dest: UserPtr, src): !void
+ copyFromUser(dest, src: UserSlice): !void
+ copyFromUserSingle(dest, src: UserPtr): !void
+ copyStringFromUser(allocator, src: UserPtr): string
+ printUser(user_string: UserSlice): !void
+ handleSafeAccessFault(frame): bool
UserPtr<T>
- ptr: T
+ fromPtr(user_ptr: T): !UserPtr
+ fromFlat(user_ptr: size_t): !UserPtr
+ fromFlatMaybeNull(user_ptr: size_t): !?UserPtr
+ ptr(): T
+ flat(): size_t
+ toConst(): UserPtr<const T>
UserSlice<T>
- slice: T
+ fromSlice(user_slice: T): !UserSlice
+ fromFlat(user_ptr: size_t, length): !UserSlice
+ slice(): T
+ len(): size_t
+ toConst(): UserSlice<const T>
+ ptrAt(idx): UserPtr
+ sliceTo(new_len): UserSlice<T>

Figure 4.4: UML kernel mem

4.2.3 x86

This package includes a bunch of things that are very low level and specific to the x86 architecture. It is further divided into the following packages.

Asm

The Asm package includes several assembly routines that execute privileged instructions. Some of them are `wrmsr` and `rdmsr` to write and read from MSRs, `lgdt`, `ltr`, `lidt` to load the GDT, TSS and IDT respectively, `inb` and `outb` to perform IO operations, etc.

GDT and TSS

GDT entries, called descriptors, are represented by the packed structure GlobalDescriptor. Its constructor `init()` accepts the type of descriptor (data or code) and its privilege level (kernel or user), and sets the rest of the bit fields in a suitable way, configuring the descriptor for x86-64 (Section 1.4.4).

The TSS (Section 1.4.4) is represented by the packed structure TaskStateSegment. When initialised, it sets `rsp[0]` and `ist[1]` to point to `stack_rsp0` and `stack_ist1`, stacks that are used when an interrupt causes a switch to ring 0 and when a double fault occur, respectively. The TSS is referenced by the TaskStateSegmentDescriptor, a special descriptor, twice the size of a regular descriptor, which has a full 64 bits `base` field (split in 4 different bit fields) that points to the TSS.

The GDT itself is a vector of 7 descriptors: the null descriptor, descriptors for kernel code and data, user code and data, and finally the TaskStateSegmentDescriptor which takes up two spaces. When the module is initialised, it loads the GDT with the `lgdt` instruction and the TSS with the `ltr` instruction.

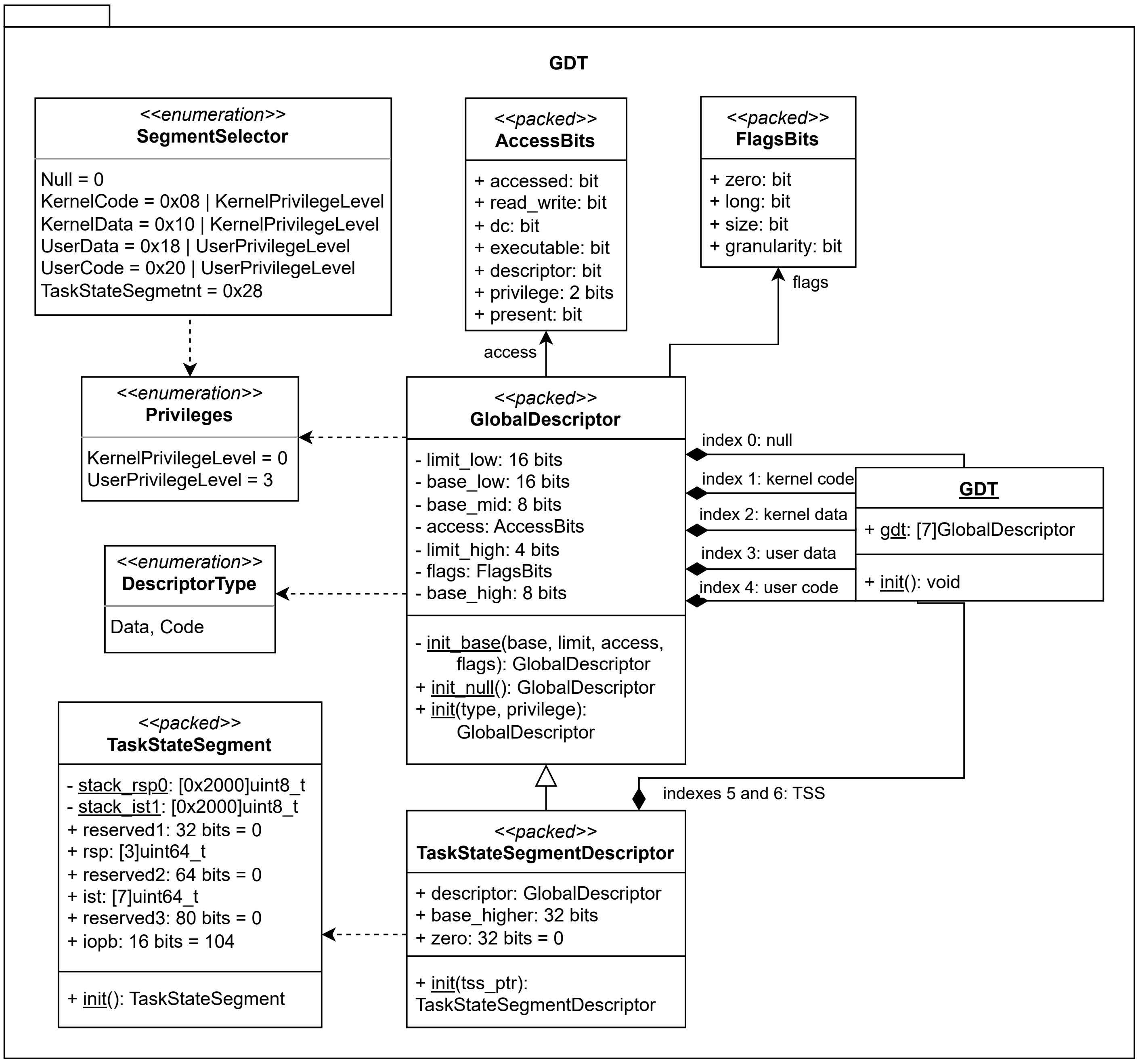


Figure 4.5: UML kernel GDT

IDT

Similarly to the GDT, IDT entries are represented by the packed structure InterruptDescriptor. Its constructor accepts a pointer to the interrupt handler that will be executed when the corresponding interrupt fires, and an optional index into the IST, and fills the rest of the bit fields accordingly.

The IDT itself is a vector of 256 descriptors. Entries from 0 to 31 are exceptions, and entry 32 is the interrupt request number for the APIC timer interrupt. Once initialised, it is loaded with the `lidt` instruction.

Interrupts

The Interrupts package provides handlers for the different interrupts. For each interrupt, it creates a low level function of type InterruptHandlerEntryPoint. This is where execution will jump to whenever the interrupt happens, and is written in assembly. For some exceptions, the CPU pushes an error code into the stack, and for others it doesn't. This entry point fixes that by pushing a 0 into the stack for those interrupts that do not have an error code. Then, it pushes its interrupt number and jumps to the function `interruptHandlerCommon()`, which is also written in assembly and is common to every interrupt handler. This one is in charge of pushing the registers to the stack to create an InterruptFrame structure and calling `interruptHandler()`. This function will perform the final dispatch according to the interrupt number and call the higher level InterruptHandler functions, which take this frame as parameter. The reasoning behind this design is to have an unified interrupt frame in the stack (with an error code even for those exceptions for which the CPU does not push one), and a single function that saves and restores registers. Interrupt handlers entry points are exposed by the function `getInterruptHandlerEntryPoint()`, which is called by the IDT module.

The interrupts that have a special handler implemented are page fault, general protection fault, division by zero, stack segment fault, breakpoint, and APIC timer. The first four represent a crash (except page faults on some cases, as explained in Section 4.2.2), and they end the current run with reason Crash. The breakpoint handler simply panics, because breakpoints are handled from the hypervisor and not from the kernel. That is, the hypervisor configures breakpoint interrupts to be delivered to it instead of to the kernel. The kernel breakpoint handler just makes sure breakpoints are not being delivered to the kernel. Finally, the APIC timer interrupt is used for scheduling tasks and for measuring execution time and issuing a time-out in case it surpasses a certain value (Section 4.2.3).

Perf

The perf module has two main purposes. First, it enables two hardware performance counters (Section 1.4.4) to count number of cycles and instructions executed. These counters are read by the hypervisor to show statistics and to perform tracing.

Second, it keeps track of the duration of the current run in a variable called `current_timer`, which is a counter that is incremented every time the APIC interrrupt happens. If the timer surpasses the value of `timer_timeout`, the run ends with reason time-out. On initialisation, the addresses of `current_timer` and `timer_timeout` are sent to the hypervisor via the hypercall SetTimeoutPointers, so it can reset the timer and modify the timeout value. This time-out feature prevents the fuzzer from getting stuck when the target program hits an infinite loop.

APIC

The APIC module configures the APIC (Section 1.4.4) to trigger the timer interrupt every `TIMER_MICROSECS` microseconds, which is by default set to 1000 (1 milliseconds). However, the APIC timer does not measure in microseconds. Instead, it has a counter that decreases at a speed that depends on the processor frequency, and that is principle unknown. Therefore, on initialisation, this module must make use of the Programmable Interval Timer (PIT), another device which runs at a known frequency, to calculate the frequency of the APIC and be able to approximately measure `TIMER_MICROSECS` microseconds. The APIC is configured to raise a timer interrupt on a specific interrupt request number (32). When the interrupt handler finishes, it calls `resetTimer()` to signal an end of interrupt and reset the timer.

Syscalls

The syscalls module is in charge of configuring system calls by setting up the handler entry point (Section 1.4.4). The syscall handler entry point is written in assembly and it is where execution jumps to whenever the user program executes the instruction `syscall`. It saves the user stack, sets the kernel stack, saves user registers and then calls the higher level syscall handler. After that, it restores user registers and stack and finally returns to user mode.

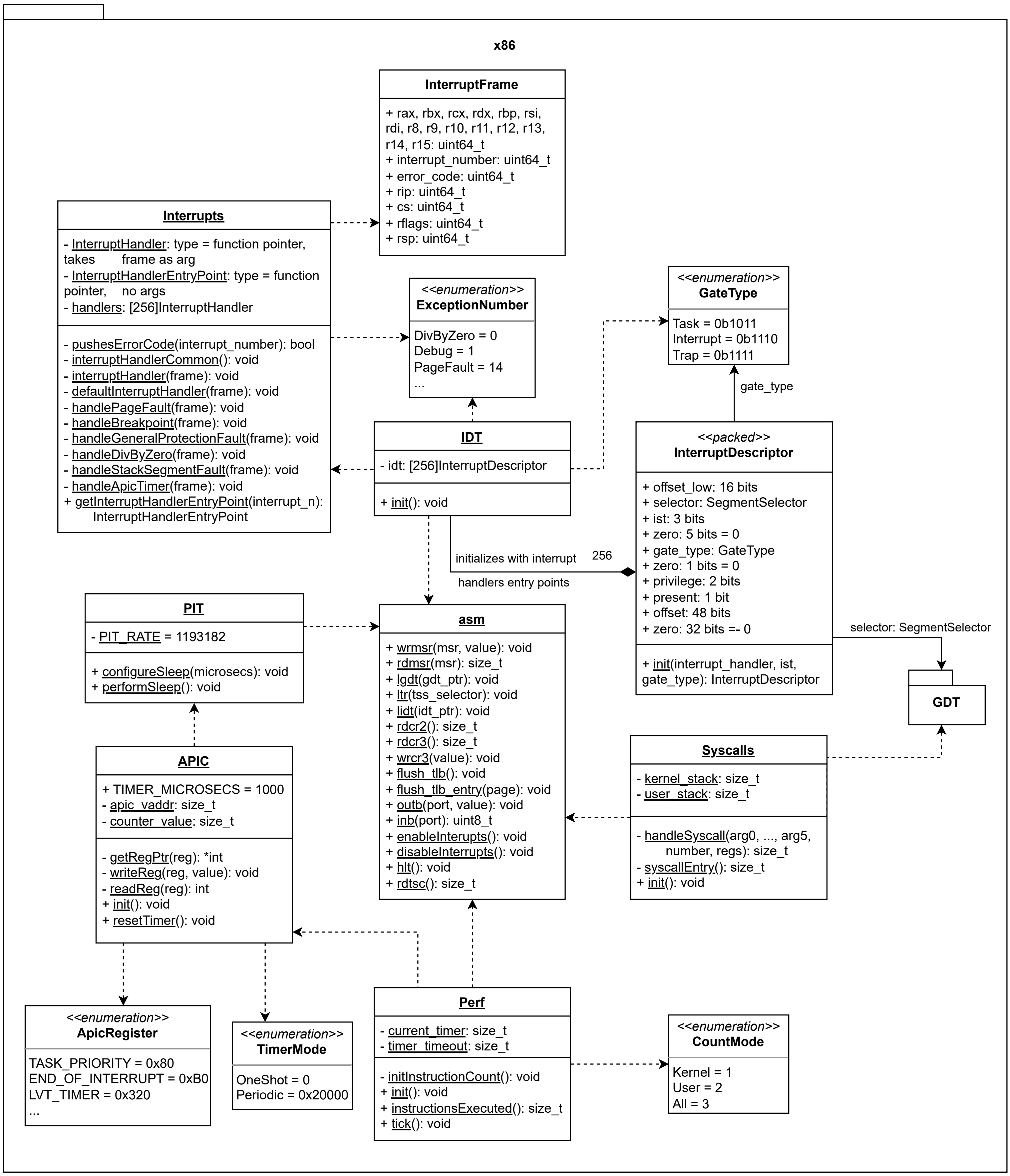

x86
InterruptFrame
+ rax, rbx, rcx, rdx, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15: uint64_t
+ interrupt_number: uint64_t
+ error_code: uint64_t
+ rip: uint64_t
+ cs: uint64_t
+ rflags: uint64_t
+ rsp: uint64_t
Interrupts
- InterruptHandler: type = function pointer, takes frame as arg
- InterruptHandlerEntryPoint: type = function pointer, no args
- handlers: [256]InterruptHandler
- pushesErrorCode(interrupt_number): bool
- interruptHandlerCommon(): void
- interruptHandler(frame): void
- defaultInterruptHandler(frame): void
- handlePageFault(frame): void
- handleBreakpoint(frame): void
- handleGeneralProtectionFault(frame): void
- handleDivByZero(frame): void
- handleStackSegmentFault(frame): void
- handleApicTimer(frame): void
+ getInterruptHandlerEntryPoint(interrupt_n): InterruptHandlerEntryPoint
<<enumeration>>
ExceptionNumber
DivByZero = 0
Debug = 1
PageFault = 14
...
<<enumeration>>
GateType
Task = 0b1011
Interrupt = 0b1110
Trap = 0b1111
gate_type
IDT
- idt: [256]InterruptDescriptor
+ init(): void
initializes with interrupt handlers entry points
256
<<packed>>
InterruptDescriptor
+ offset_low: 16 bits
+ selector: SegmentSelector
+ ist: 3 bits
+ zero: 5 bits = 0
+ gate_type: GateType
+ zero: 1 bits = 0
+ privilege: 2 bits
+ present: 1 bit
+ offset: 48 bits
+ zero: 32 bits =- 0
+ init(interrupt_handler, ist, gate_type): InterruptDescriptor
selector: SegmentSelector
GDT
PIT
- PIT_RATE = 1193182
+ configureSleep(microsecs): void
+ performSleep(): void
asm
+ wrmsr(msr, value): void
+ rdmsr(msr): size_t
+ lgdt(gdt_ptr): void
+ ltr(tss_selector): void
+ lidt(idt_ptr): void
+ rdcr2(): size_t
+ rdcr3(): size_t
+ wrcr3(value): void
+ flush_tlb(): void
+ flush_tlb_entry(page): void
+ outb(port, value): void
+ inb(port): uint8_t
+ enableInterrupts(): void
+ disableInterrupts(): void
+ hlt(): void
+ rdtsc(): size_t
APIC
+ TIMER_MICROSECS = 1000
- apic_vaddr: size_t
- counter_value: size_t
- getRegPtr(reg): *int
- writeReg(reg, value): void
- readReg(reg): int
+ init(): void
+ resetTimer(): void
Syscalls
- kernel_stack: size_t
- user_stack: size_t
- handleSyscall(arg0, ..., arg5, number, regs): size_t
- syscallEntry(): size_t
+ init(): void
<<enumeration>>
ApicRegister
TASK_PRIORITY = 0x80
END_OF_INTERRUPT = 0xB0
LVT_TIMER = 0x320
...
<<enumeration>>
TimerMode
OneShot = 0
Periodic = 0x20000
Perf
- current_timer: size_t
- timer_timeout: size_t
- initInstructionCount(): void
+ init(): void
+ instructionsExecuted(): size_t
+ tick(): void
<<enumeration>>
CountMode
Kernel = 1
User = 2
All = 3

Figure 4.6: UML kernel x86

4.2.4 File System

The File System package, and in particular the FileManager module, is the kernel component in charge of handling virtual files (Section 4.1.10). Its UML class diagram can be seen in Figure 4.7. On file manager initialisation, for each file it obtains its size and name with the hypercall GetFileInfo, allocates a buffer for its contents, and gets its contents with the hypercall SubmitFilePointers. That way, it populates its `file_contents` field, which is a map from filename to file file contents. This is the kernel representation of files.

The file manager also provides an interface for opening regular files given a filename and open flags with the `open()` method, and for opening other kind of files with methods `openStdin()`, `openStdout()`, `openStderr()` and `openSocket()`. All these return a pointer to a FileDescription.

A FileDescription is an abstract class representing an open file. It contains the fields `buf`, which is the file content; `flags`, which is the flags the file was opened with; and `offset`, which is the position of the cursor inside the file, that is, where the file will be read or written to in the next operation. It implements methods to easily access its flags (`isReadable()`, `isOffset`) and to access and move its offset (`isOffsetPastEnd()`, `moveOffset()`). Classes that inherit from FileDescription must implement the functions `stat()`, `read()` and `write()`, which receive user pointers and perform the same operation as the Linux system calls with their same name.

There are five different types of file descriptions:

- FileDescriptionRegular represents a regular file whose contents can be read. Writing to them is not supported. The `stat` function, similar to Linux, returns information about the file, including its size in bytes, its size in blocks, and an unique inode to each file.

- FileDescriptionStdin is the file description for stdin, the standard input. When reading from it, it will operate as the regular file with name “input”, which is the input file used when fuzzing. When writing to it, it will operate as stdout.

- FileDescriptionStdout is the file description for stdout, the standard output. When writing to it, it calls the static function `printUserMaybe()`, which depending on a compilation option it will perform the hypercall Print to print to the hypervisor stdout or do nothing. Reading from it is not supported.

- FileDescriptionStderr is the same as FileDescriptionStdout.

- FileDescriptionSocket is the file description that represents an open socket. It has some fields describing its state (whether it’s bound, listening or connected) and the additional methods to change its state, analogous to Linux system calls `bind()` and `listen()`.

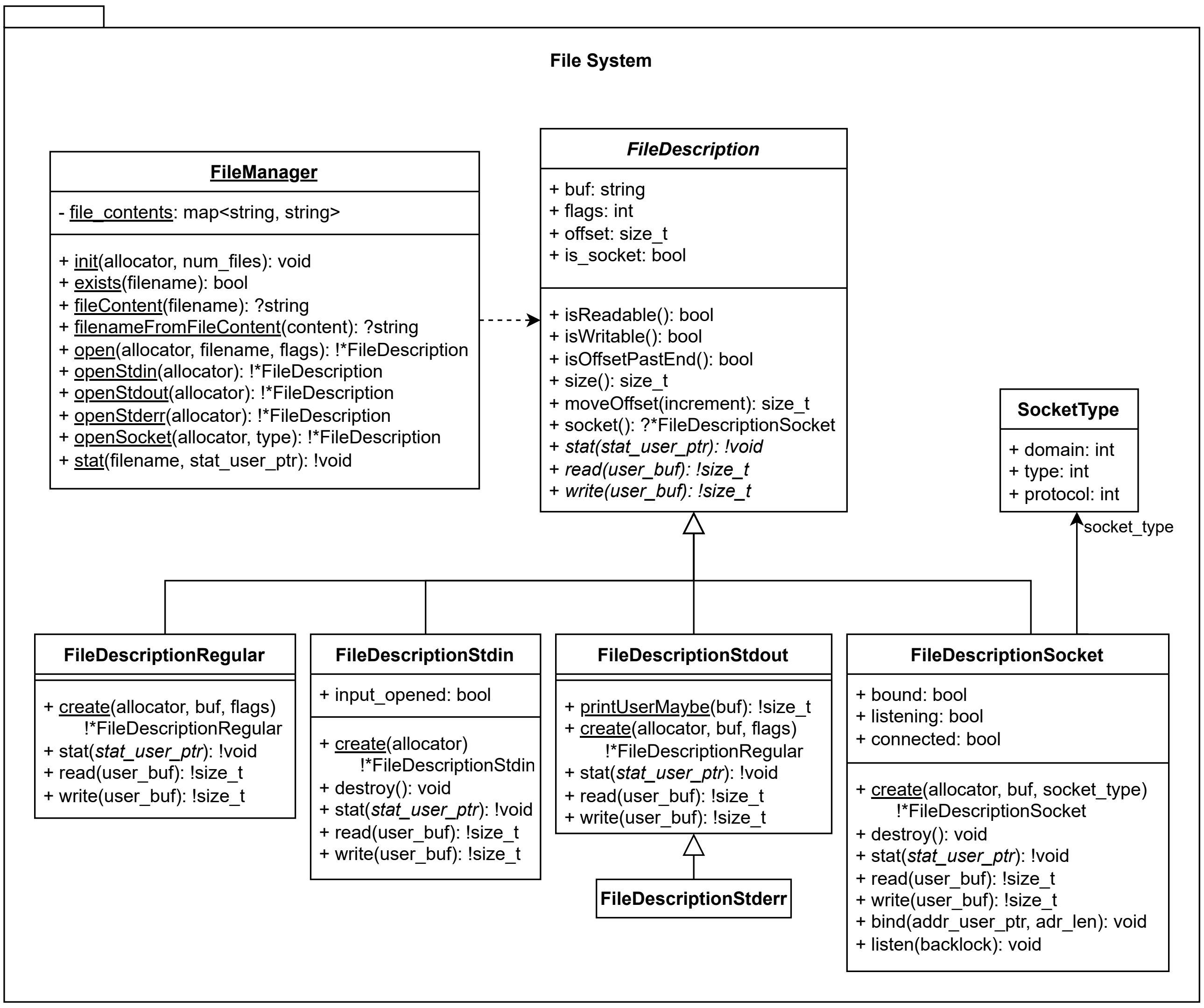

File System
FileManager
- file_contents: map<string, string>
+ init(allocator, num_files): void
+ exists(filename): bool
+ fileContent(filename): ?string
+ filenameFromFileContent(content): ?string
+ open(allocator, filename, flags): !*FileDescription
+ openStdin(allocator): !*FileDescription
+ openStdout(allocator): !*FileDescription
+ openStderr(allocator): !*FileDescription
+ openSocket(allocator, type): !*FileDescription
+ stat(filename, stat_user_ptr): !void
FileDescription
+ buf: string
+ flags: int
+ offset: size_t
+ is_socket: bool
+ isReadable(): bool
+ isWritable(): bool
+ isOffsetPastEnd(): bool
+ size(): size_t
+ moveOffset(increment): size_t
+ socket(): ?*FileDescriptionSocket
+ stat(stat_user_ptr): !void
+ read(user_buf): !size_t
+ write(user_buf): !size_t
SocketType
+ domain: int
+ type: int
+ protocol: int
socket_type
FileDescriptionRegular
+ create(allocator, buf, flags) !*FileDescriptionRegular
+ stat(stat_user_ptr): !void
+ read(user_buf): !size_t
+ write(user_buf): !size_t
FileDescriptionStdin
+ input_opened: bool
+ create(allocator) !*FileDescriptionStdin
+ destroy(): void
+ stat(stat_user_ptr): !void
+ read(user_buf): !size_t
+ write(user_buf): !size_t
FileDescriptionStdout
+ printUserMaybe(buf): !size_t
+ create(allocator, buf, flags) !*FileDescriptionRegular
+ stat(stat_user_ptr): !void
+ read(user_buf): !size_t
+ write(user_buf): !size_t
FileDescriptionStderr
FileDescriptionSocket
+ bound: bool
+ listening: bool
+ connected: bool
+ create(allocator, buf, socket_type) !*FileDescriptionSocket
+ destroy(): void
+ stat(stat_user_ptr): !void
+ read(user_buf): !size_t
+ write(user_buf): !size_t
+ bind(addr_user_ptr, adr_len): void
+ listen(backlock): void

Figure 4.7: UML kernel filesystem

4.2.5 Processes

Process

User processes are represented by the class Process. There are two ways of creating an instance of a Process. First, the initial process is created by the static method `initial()`, which fills its different fields with default values and with information obtained by the hypercall GetVmInfo. Second, subsequent processes are created by the `clone()` system call handler, inheriting most of its attributes from its parent process.

The main function of a Process is to handle system calls according to the current state of the process. To do so, processes have many different fields:

- Fields `pid`, `tgid`, `pgid` and `ptgid` refer to different identifiers. `pid` is the process id, unique for each instance of Process, and it is returned by the `gettid()` syscall. `tgid` is the thread group id, unique for each group of processes. In a group of threads, they all are processes that share the same tgid. Additionally, the main thread has its tgid equal to its pid. It is returned by the `getpid()` system call. `pgid` stands for parent group id, and is returned by the `getpgid()` system call, while `ptgid` stands for parent thread group id. The nomenclature is slightly confusing, but it was chosen to be that way to mimic Linux.

- Field `space` is a pointer to an AddressSpace, in the memory package (Section 4.2.2). It represents the address space of the process, and is accessed by system calls like `mmap()`, `munmap()` and `brk()`. Note that different processes may have the same address space.

- Field `state` indicates the current state of the process. It could be active (running or ready to run), waiting for a child process to exit, blocked on a futex, or it may have already exited and be in a zombie state waiting for its parent process to wait on him, similar to what happens on Linux. It is accessed by the scheduler, and by the `wait()` and `futex()` syscall handlers.

- Field `files` is a pointer to the file descriptor table, which is accessed by filesystem related syscalls, such as `open()`, `read()`, `dup()`, etc.

- Field `elf_path` is the path of the ELF in the host filesystem, and is returned by the `readlink()` syscall when performed on the file `/proc/self/exe`.

- Fields `brk` and `min_brk` are the current break and minimum break of the process. They are used by the `brk()` system call.

- Field `user_regs` holds the saved registers of the process when it is not the process currently running.

- The rest of the fields are related to intricate details of Linux and are needed to correctly emulate some system calls.

The Process class implements the method `handleSyscall()`, which is called by the syscall handler entry point (Section 4.2.3) and dispatches to the appropriate private method `handle_sys_[syscall]` (for example, `handle_sys_open`). This method is in charge of casting raw arguments from integer values to their correct, higher level types. Notably, this includes creating UserPtrs and UserSlices for wrapping user pointers (Section 4.2.2). After that, it calls the method `sys_[syscall]` (for example, `sys_open`), which receives the cast arguments and finally handles the system call.

FileDescriptorTable

The FileDescriptorTable class represents the file descriptor table of a process. It is a map from file descriptors (integers that user programs use to refer to open files) to pointers of FileDescriptions (representation of open files inside the kernel). When created, it opens files stdin, stdout and stderr, which are available by default to every process as file descriptors 0, 1 and 2 respectively. Note that two file descriptors may refer to the same underlying FileDescription (as a result of the `dup()` system call, for example), and that two processes may have the same file descriptor table (as a result of a `clone()` system call with flag `CLONE_FILES`, for example).

Scheduler

Finally, processes are managed by the Scheduler module. The scheduler has a list of processes, and is in charge of deciding which process should run. This is called scheduling. When a timer interrupt fires, it calls the scheduler method `schedule()`, which may decide to swap the current active process for another one. This is implemented by the `switchToProcessIdx()` method, which performs two main actions. First, it switches to the next process address space. Second, it saves the registers of the current interrupt frame to the `user_regs` field of the current process, and stores the `user_regs` of the next process in the interrupt frame. That way, when the interrupt finishes, the registers of the next process will be restored, and its execution will continue.

Apart from scheduling and switching processes, the scheduler is also in charge of implementing most of the `wait()` syscall, which waits for a given process to finish (and may block and switch the current process) and the `futex()` syscall, which is an userspace construct used for blocking and synchronization.

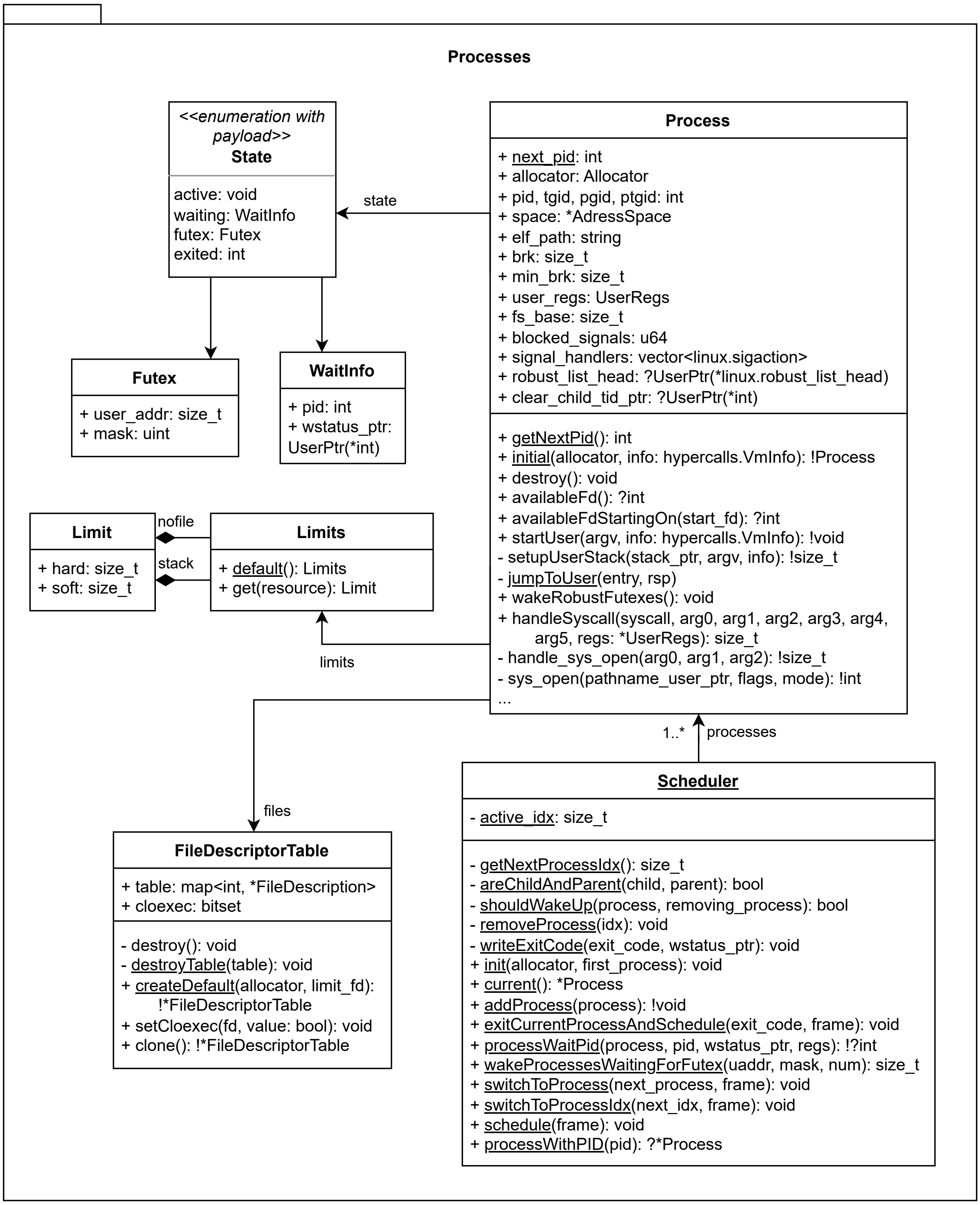

Processes
<<enumeration with payload>>
State
active: void
waiting: WaitInfo
futex: Futex
exited: int
state
Process
+ next_pid: int
+ allocator: Allocator
+ pid, tgid, pgid, ptgid: int
+ space: *AdressSpace
+ elf_path: string
+ brk: size_t
+ min_brk: size_t
+ user_regs: UserRegs
+ fs_base: size_t
+ blocked_signals: u64
+ signal_handlers: vector<linux.sigaction>
+ robust_list_head: ?UserPtr(*linux.robust_list_head)
+ clear_child_tid_ptr: ?UserPtr(*int)
+ getNextPid(): int
+ initial(allocator, info: hypercalls.VmInfo): !Process
+ destroy(): void
+ availableFd(): ?int
+ availableFdStartingOn(start_fd): ?int
+ startUser(argv, info: hypercalls.VmInfo): !void
- setupUserStack(stack_ptr, argv, info): !size_t
- jumpToUser(entry, rsp)
+ wakeRobustFutexes(): void
+ handleSyscall(syscall, arg0, arg1, arg2, arg3, arg4, arg5, regs: *UserRegs): size_t
- handle_sys_open(arg0, arg1, arg2): !size_t
- sys_open(pathname_user_ptr, flags, mode): !int
...
Futex
+ user_addr: size_t
+ mask: uint
WaitInfo
+ pid: int
+ wstatus_ptr: UserPtr(*int)
Limit
+ hard: size_t
+ soft: size_t
nofile
stack
Limits
+ default(): Limits
+ get(resource): Limit
limits
1..*
processes
files
FileDescriptorTable
+ table: map<int, *FileDescription>
+ cloexec: bitset
- destroy(): void
- destroyTable(table): void
+ createDefault(allocator, limit_fd): !*FileDescriptorTable
+ setCloexec(fd, value: bool): void
+ clone(): !*FileDescriptorTable
Scheduler
- active_idx: size_t
- getNextProcessIdx(): size_t
- areChildAndParent(child, parent): bool
- shouldWakeUp(process, removing_process): bool
- removeProcess(idx): void
- writeExitCode(exit_code, wstatus_ptr): void
+ init(allocator, first_process): void
+ current(): *Process
+ addProcess(process): !void
+ exitCurrentProcessAndSchedule(exit_code, frame): void
+ processWaitPid(process, pid, wstatus_ptr, regs): !?int
+ wakeProcessesWaitingForFutex(uaddr, mask, num): size_t
+ switchToProcess(next_process, frame): void
+ switchToProcessIdx(next_idx, frame): void
+ schedule(frame): void
+ processWithPID(pid): ?*Process

Figure 4.8: UML kernel processes

4.3 Markov model

4.3.1 Modelling program execution with Markov chains

While running, the hypervisor can trace the execution of the target program, both in userspace tracing basic blocks, and in kernel tracing system calls (Section 4.1.11). The trace files it produces are a list of these elements.

We can model the execution behaviour of the program as a Markov chain $\{X_n\}_{n \geq 0}$, where each of the X_i are random variables describing the state of the program at a given point. When tracing basic blocks, we define the state space $\mathcal{S}$ (Definition 1.2) as:

$$\mathcal{S}_{basicblocks} = \{addr \in \mathbb{N} \mid addr \text{ is a virtual address in the target program}\},$$

i.e., the set of all virtual addresses in the target program. On the other hand, when tracing system calls, the state space is the set of syscalls:

$$\mathcal{S}_{syscalls} = \{n \in \mathbb{N} \mid n \text{ is a valid Linux system call}\}$$

In both cases, the state space is finite.

As a trace is a succession of elements of the state space, we can interpret it as a trajectory (Definition 1.4). That is, every run of the target program yields a particular trajectory in the Markov chain $\{X_n\}_{n \geq 0}$, describing the different execution points that were reached during that run and in which order.

However, traces are finite, and they can end at different states. We want to represent that in our model. In order to do so, we add a special state s_{null}, which represents the end of the execution, and which is reached after the end of every trace.

Furthermore, every execution trace starts at the same state, which is the initial state. For simplicity, we can assume this to be state 0. Therefore, if $X_0 \rightsquigarrow \mu = (\mu_i)_{i \in \mathcal{S}}$, that is, μ is the probability distribution of X_0, then it is given by:

$$\mu = (1, 0, \ldots, 0)$$

4.3.2 Transition matrix

Given a set of traces, we would like to obtain the transition matrix (Definition 1.6) of the Markov chain modelling the behaviour of the program. Therefore, for every state $i \in \mathcal{S}$, we want to calculate the probability p_{ij} of transitioning to every other state j. This can done by analysing every trace, getting a list of successors for each state, and calculating the probability of each successor according to its frequency in the list. Let's formalise this.

For a given Markov chain $\{X_n\}_{n \geq 0}$, say we have a set of trajectories:

$$\mathcal{T} = \{T = (t_0, t_1, \ldots) \mid T \text{ is a trajectory, that is, } t_k \in \mathcal{S}\}$$

We define the successor function as follows. The successor of the state t_k in a trace T is t_{k+1} (the next state) if t_k is not the last state of the trace, and s_{null} otherwise.

$$succ(T,k) = \begin{cases} t_{k+1} & \text{if } k \neq |T|-1 \\ s_{null} & \text{if } k = |T|-1 \end{cases}$$

Then, let S_i be the set of all successors of the state $i \in \mathcal{S}$ across the trajectories in $\mathcal{T}$. Note S_i is actually a multiset: it can contain repeated elements, but order does not matter. It can be calculated as follows:

$$S_i = \bigcup_{T=(t_0,t_1,\ldots)\in\mathcal{T}} \{succ(T,k) \mid t_k = i\}$$

Finally, given any other state $j \in \mathcal{S}$, the probability of transitioning from i to j can be calculated as follows:

$$p_{ij} = \frac{n_{ij}}{|S_i|}, \text{where } n_{ij} \text{ is the number of occurrences of } j \text{ in } S_i$$

With that, we already have a model of the execution of the program.

4.3.3 Hitting times

We have modelled the program execution as a Markov chain $\{X_n\}_{n\geq 0}$ with state space $\mathcal{S}$ and transition matrix P. We can now apply the theory of hitting times and probabilities (Section 1.4.5). For example, it is now possible to calculate the hitting probability of a part of the target program which is buggy and crashes. As the model is based on execution traces, this matches the percentage of those traces that actually hit that state.

4.3.4 Adding weights

Apart from a list of the execution points, each trace also includes information about how much time was spent in that point. The unit used for measuring that is cycles or instructions, as indicated in Section 4.1.11. Therefore, we can consider traces consist of pairs of states and weights. We will call them weighted trajectories of the Markov chain $\{X_n\}_{n\geq 0}$.

$$\mathcal{T} = \{((t_1,w_1),(t_2,w_2),\ldots) \mid t_k \in \mathcal{S}, w_k \in \mathbb{R}\}$$

Therefore, given a set of weighted trajectories, apart from building the transition matrix of the Markov chain, we can assign each state a weight, which is the average weight of the state across all the traces. Given a state

$i \in \mathcal{S}$, we will call its associated weight W_i, defined as follows:

$$M_i = \bigcup_{T=((t_0,w_0),(t_1,w_1),\ldots)\in\mathcal{T}} \{w_k \mid (t_k, w_k) \in T, t_k = i\}$$

$$W_i = \frac{1}{|M_i|} \sum_{w \in M_i} w$$

Now, the Markov chain not only models the execution behaviour of the program, but also its execution time, measured in cycles or in instructions.

4.3.5 Average weight

Given that we now have a Markov chain $\{X_n\}_{n\geq 0}$ with state space $\mathcal{S}$, transition matrix P, and such that every state $i \in \mathcal{S}$ has an associated weight W_i, we want to calculate the average weight of the Markov chain. This represents the average time (measured in instructions or cycles) that the target program took to run across the different executions.

In order to calculate the average time of the Markov chain, we define a_i as the average time it takes to run from a state $i \in \mathcal{S}$. Then, the average time of a Markov chain is a_0, as every execution starts from the same initial state.

The vector of average times from states $a = (a_i)_{i\in\mathcal{S}}$ is the unique solution to the following system of linear equations, which is similar to the one for calculating hitting probabilities described in Theorem 1.2:

$$a_i = W_i + \sum_{j \in \mathcal{S}} p_{ij} a_j$$

These equations can be interpreted as: the time it takes to run from a state is the time the state takes to run plus the time it takes to run from its successors.

Note that, by definition, the time of the Markov chain a_0 must match the average time of the traces in $\mathcal{T}$:

$$\frac{1}{|\mathcal{T}|} \sum_{T=((t_0,w_0),(t_1,w_1),\ldots)\in\mathcal{T}} \sum_{i=0}^{|T|} w_i$$

4.3.6 Associated graph

Given the execution traces of a program, we have calculated and modelled its execution behaviour with a Markov chain $\{X_n\}_{n\geq 0}$. Now, we can represent it through its associated graph (1.16). Nodes represent execution points of the program, and every transition is represented as an edge with the probability of the transition as weight. Furthermore, every node also has a weight, which is the average time it takes to run.

The associated graph can be plotted. An example of this can be seen in Figure 4.9, which represents the syscall execution behaviour of the program `readelf`. Node weights are indicated by their color: the closer to red, the more time it takes to run.

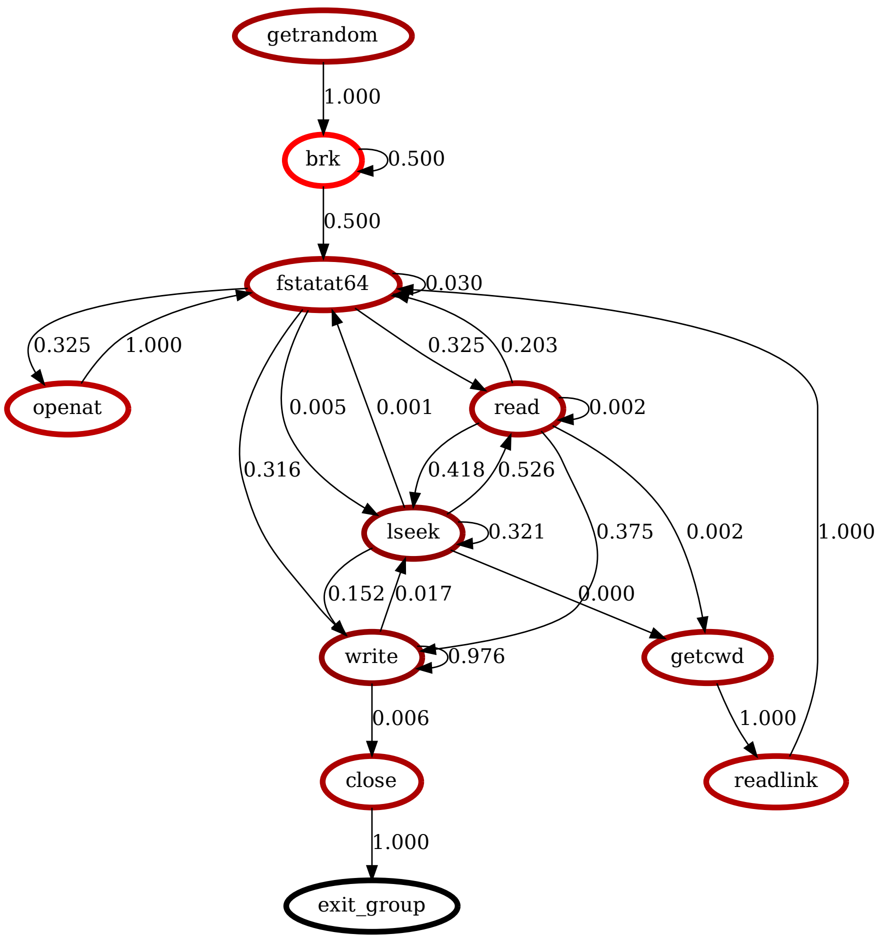


Figure 4.9: Graph associated to a Markov chain modelling syscall execution behaviour of `readelf`

Chapter 5

Implementation

5.1 Overview

5.1.1 Repository structure

The implementation of KVM-FUZZ is available at the GitHub repository `https://github.com/klecko/kvm-fuzz/`. It has the following files and directories:

1. hypervisor: folder containing the implementation of the hypervisor, as described in Section 5.2.
2. kernel: folder containing the implementation of the kernel, as described in Section 5.3.
3. scripts: folder containing auxiliary scripts that run tests, generate a list of basic blocks for a binary, or plot coverage.
4. tests: folder containing the implementation of tests, as described in Section 6.1.
5. build.zig: build script in charge of compiling the hypervisor, the kernel, and the different tests and experiments.
6. markov.py: script implementing the Markov model, as described in Section 5.4.
7. README.md: text file giving a general overview of the project, how to build it, and its different options.

Each item will be further described in the following sections. Code snippets may be simplified.

5.1.2 Dependencies and build system

The hypervisor is developed in C++, while the kernel is implemented in Zig. The rationale behind this is addressed in Section 3.2.2. The Zig compiler is able to compile both Zig and C/C++, and includes its own build system that uses the `build.zig` file as build script. Therefore, the only dependencies are:

1. The Zig compiler, version 0.10.1. It can be downloaded from the official webpage.
2. System libraries libdwarf and libelf, which are used for parsing and providing ELF debug information, and libssl, which is used for calculating MD5 hashes. They can be installed from the system package manager:
   ```
   sudo apt install libdwarf-dev libelf-dev libssl-dev
   ```
3. Optionally, python with the angr package, which can be used for automatically calculating basic blocks for breakpoints based coverage. It can be installed from the system package manager:
   ```
   sudo apt install python3
   python3 -m pip install angr
   ```

In order to build KVM-FUZZ, the command `zig build` is used, which invokes the Zig compiler. It has the following build options:

1. `-Drelease-safe`, `-Drelease-fast`, `-Drelease-small`. These options select the build mode. The first one enables optimizations while keeping safety checks. The other two disable safety checks and enable optimizations focusing on speed and reducing binary size respectively. Generally, `-Drelease-safe` should be used, as the performance overhead of safety checks is small. If none of these options are provided, debug build mode is selected, which does not enable optimizations and includes debug information.
2. `-Dinstruction-count=[value]`. It enables instruction counting, which is used for reporting statistics and tracing. The value can be kernel, user, all or none, which chooses the context where instructions will be counted. By default, user is used.
3. `-Denable-guest-output`. It enables guest output, so every time the target program prints to standard output it will be printed. Default is disabled.
4. `-Denable-mutations`. It enabled input mutating. If disabled, inputs will be chosen from the initial corpus and they will not be mutated. Default is enabled.

5. `-Dcoverage=[value]`. It selects the method of code coverage used, which can be breakpoints or intelpt. Default is breakpoints.

By default, `zig build` command only builds the hypervisor and the kernel. Additional arguments `syscalls_tests`, `hypervisor_tests` or `experiments` need to be provided in order to build the rest of the components. Furthermore, the following scripts can be used to correctly build syscalls tests and run them under Linux and KVM-FUZZ, as detailed in the file `README.md`:

```
./scripts/run_tests_on_linux.sh
./scripts/run_tests_on_kvm-fuzz.sh
```

5.2 Hypervisor

The hypervisor implementation is in the folder `hypervisor`. As usual with C++ projects, it is divided in two folders `include` and `src`, which contain the header files and the implementation files respectively. Besides those, there is also the `experiments` folder, which contains the source code for some of the experiments detailed in Section 6.2.

In the following sections, implementation details of some of the classes presented in the design (Section 4.1) are discussed.

5.2.1 Fuzzing loop

The fuzzing loop is implemented in the file `src/main.cpp` by the `worker()` function, which is run on each thread. The code related to statistics has been simplified. It works as follows. Before starting, the `base` VM is cloned into a `runner` VM. Then, the loop starts. First, an input is mutated and obtained from the corpus. Then, it is set as contents for the file “input” and copied into kernel memory. After that, the VM is run, and the end reason is checked. Finally, the coverage is reported to the corpus and the state of the VM is reset to the state of `base`, which is the VM it was forked from.

```cpp
void worker(int id, const Vm& base, Corpus& corpus, Stats& stats){
    Vm runner(base);
    Rng rng;
    Vm::RunEndReason reason;
    while (true) {
        FileRef input = corpus.get_new_input(id, rng, stats);
        vm.set_file("input", input, Vm::CheckCopied::Yes);
        reason = runner.run(local_stats);
        switch (reason) {
            case Vm::RunEndReason::Breakpoint:
            case Vm::RunEndReason::Exit:
                break;
            case Vm::RunEndReason::Timeout:
                stats.timeouts++;
                break;
            case Vm::RunEndReason::Crash:
                stats.crashes++;
                corpus.report_crash(id, runner);
                break;
            default:
                die("unexpected RunEndReason");
        }

        corpus.report_coverage(id, runner.coverage());
        runner.reset_coverage();
        runner.tracing().dump_trace(id);
        runner.reset(base, stats);
    }
}
```

5.2.2 Vm

The main part of the Vm class is implemented in the files `src/vm.cpp` and `include/vm.h`. There is also the file `src/hypercalls.cpp`, which includes the different hypercall handlers.

Virtual machines are created with the KVM API, as described in Section 1.4.3. The following listing illustrates the code in charge of opening the `/dev/kvm` device. It is set as constructor, so it is run on initialisation. After opening device, it checks the KVM version of the kernel matches the version the hypervisor was compiled for. Finally, if Intel PT was chosen as coverage option at compile time, it checks it is supported by the host kernel.

```
int g_kvm_fd = -1;

_attribute_((constructor))
void init_kvm() {
    g_kvm_fd = open("/dev/kvm", O_RDWR);
    ERROR_ON(g_kvm_fd == -1, "open /dev/kvm");

    int api_ver = ioctl(g_kvm_fd, KVM_GET_API_VERSION, 0);
    ASSERT(api_ver == KVM_API_VERSION);

#ifdef ENABLE_COVERAGE_INTEL_PT
    int vmx_pt = ioctl(g_kvm_fd, KVM_VMX_PT_SUPPORTED);
    ASSERT(vmx_pt != -1, "vmx_pt is not loaded");
    ASSERT(vmx_pt != -2, "Intel PT is not supported on this CPU");
#endif
}
```

Then, the following method is called from the Vm constructor to actually create the VM. It uses the `KVM_CREATE_VM` ioctl to create it and get its file descriptor, and then assigns it some devices. After that, it creates a virtual cpu and maps it, getting access to memory-mapped registers.

```
int Vm::create_vm() {
    m_vm_fd = ioctl_chk(g_kvm_fd, KVM_CREATE_VM, 0);

    struct kvm_pit_config pit = {
        .flags = KVM_PIT_SPEAKER_DUMMY,
    };
    ioctl_chk(m_vm_fd, KVM_CREATE_IRQCHIP, 0);
    ioctl_chk(m_vm_fd, KVM_CREATE_PIT2, &pit);

```

```cpp
    m_vcpu_fd = ioctl_chk(m_vm_fd, KVM_CREATE_VCPU, 0);

    size_t vcpu_run_size = ioctl_chk(g_kvm_fd, KVM_GET_VCPU_MMAP_SIZE, 0);
    m_vcpu_run = (kvm_run*)mmap(nullptr, vcpu_run_size, PROT_READ|
        ↪ PROT_WRITE, MAP_SHARED, m_vcpu_fd, 0);
    ERROR_ON(m_vcpu_run == MAP_FAILED, "mmap vcpu_run");

    m_regs  = &m_vcpu_run->s.regs.regs;
    m_sregs = &m_vcpu_run->s.regs.sregs;
    return m_vm_fd;
}
```

The method `setup_kvm()` is in charge of setting up some VM aspects so the VM can directly start running in long mode (64 bits mode of x86). Its implementation is not detailed here since it is not very relevant. Then, after the different ELFs have been loaded into memory by the MMU, the method `setup_kernel_execution()` finishes setting the VM up, by writing program arguments to the stack and setting register initial values.

```cpp
void Vm::setup_kernel_execution(const vector<string>& argv) {
    m_regs->rsp = m_mmu.alloc_kernel_stack();
    m_regs->rflags = 2;
    m_regs->rip = s_elfs.kernel().entry();

    // Write argv strings saving pointers to each arg
    vector<vaddr_t> argv_addrs;
    for (const string& arg : argv) {
        m_regs->rsp -= arg.size() + 1;
        m_mmu.write_mem(m_regs->rsp, arg.c_str(), arg.size()+1);
        argv_addrs.push_back(m_regs->rsp);
    }
    argv_addrs.push_back(0); // nullptr ptr, end of argv

    // Align rsp
    m_regs->rsp &= ~0x7;

    // Set up argv
    for (auto it = argv_addrs.rbegin(); it != argv_addrs.rend(); ++it) {
        m_regs->rsp -= 8;
        m_mmu.write<vaddr_t>(m_regs->rsp, *it);
    }

    // Setup args for kmain
    m_regs->rdi = argv.size(); // argc
    m_regs->rsi = m_regs->rsp; // argv
    set_regs_dirty();
}
```

Then, the method `run()` is the one that actually runs the VM. It calls the KVM_RUN ioctl in a loop while handling VM exits until the execution stops.

```cpp
Vm::RunEndReason Vm::run(Stats& stats) {
    RunEndReason reason = RunEndReason::Unknown;
    m_running = true;

    while (m_running) {
        ioctl_chk(m_vcpu_fd, KVM_RUN, 0);
        switch (m_vcpu_run->exit_reason) {
            case KVM_EXIT_IO:
                if (m_vcpu_run->io.direction == KVM_EXIT_IO_OUT &&
                    m_vcpu_run->io.port == 16)
                {
                    handle_hypercall(reason);
                } else {
                    vm_err("IO");
                }
                break;

            case KVM_EXIT_DEBUG: {
                int exception = m_vcpu_run->debug.arch.exception;
                switch (exception) {
                    case Exception::Debug:
                        reason = RunEndReason::Debug;
                        m_running = false;
                        break;
                    case Exception::Breakpoint:
                        handle_breakpoint(reason);
                        break;
                }
                break;
            }

#ifdef ENABLE_COVERAGE_INTEL_PT
            case KVM_EXIT_VMX_PT_TOPA_MAIN_FULL:
                stats.vm_exits_cov++;
                update_coverage(stats);
                break;
#endif
        }
    }

    return reason;
}
```

As can be seen, when the CPU hits a breakpoint, it is handled by the `handle_breakpoint()` method. Breakpoints are held in `m_breakpoints`, a map from virtual address to breakpoint information. The method gets the breakpoint and performs a different action depending on its type.

```
void Vm::handle_breakpoint(RunEndReason& reason) {
    vaddr_t addr = m_regs->rip;
    Breakpoint& bp = m_breakpoints[addr];

    // If it's of type RunEnd, stop running and stop handling the
       ↪ breakpoint
    if (bp.type & Breakpoint::RunEnd) {
        reason = RunEndReason::Breakpoint;
        m_running = false;
        return;
    }

    // If it's a hook handle it, then remove breakpoint, single
    // step and set breakpoint again
    if (bp.type & Breakpoint::Type::Hook) {
        hook_handler_t hook_handler = m_hook_handlers[addr];
        hook_handler(*this);
        remove_breakpoint(addr, Breakpoint::Hook);
        single_step();
        set_breakpoint(addr, Breakpoint::Hook);
    }

#ifdef ENABLE_COVERAGE_BREAKPOINTS
    // If it's a coverage breakpoint, add address to basic block
    // hits, and remove it only if we are not tracing user. If we
    // are tracing user, we want to keep it to have full coverage.
    if (bp.type & Breakpoint::Type::Coverage) {
        if (m_tracing.type() == Tracing::Type::User) {
            tracing_add_addr(addr);
            remove_breakpoint(addr, Breakpoint::Coverage);
            single_step(dummy);
            set_breakpoint(addr, Breakpoint::Type::Coverage);
        } else {
            remove_breakpoint(addr, Breakpoint::Coverage);
        }

        m_coverage.add(addr);
    }
#endif
}
```

Breakpoints are set to memory by the following function. It writes a 0xCC byte to memory, which corresponds to the `INT3` x86 instruction. When executed, it raises a debug exception which is forwarded to the hypervisor by KVM. In order to write that byte to memory, it uses two different interfaces, depending on whether we want the operation to dirty memory (and therefore it will be restored afterwards) or not.

```
uint8_t Vm::set_breakpoint_to_memory(vaddr_t addr) {
    uint8_t val;
    if (m_breakpoints_dirty) {
        val = m_mmu.read<uint8_t>(addr);
        m_mmu.write<uint8_t>(addr, 0xCC, CheckPerms::No);
    } else {
        uint8_t* p = m_mmu.get(addr);
        val = *p;
        *p = 0xCC;
    }
    ASSERT(val != 0xCC, "setting breakpoint twice at 0x%lx",addr);
    return val;
}
```

The method `reset()`, shown below, is in charge of resetting the state of the VM to that of a parent VM it was forked from. It delegates most of the work to the Mmu, that resets the memory. Apart from that, it resets the state of the CPU and the tracing object.

```
void Vm::reset(const Vm& other, Stats& stats) {
    stats.reset_pages += m_mmu.reset(other.m_mmu);
    memcpy(m_regs, other.m_regs, sizeof(*m_regs));
    memcpy(m_sregs, other.m_sregs, sizeof(*m_sregs));
    m_tracing.reset(other.m_tracing);
}
```

Hypercalls are implemented in the file `src/hypercalls.cpp` as methods of the Vm class. The method `handle_hypercall()` dispatches to specific handlers according to the `rax` register, which indicates the hypercall number. Then, the return value is assigned to the `rax` register, which is then read from the kernel inside the VM.

```
void Vm::handle_hypercall(RunEndReason& reason) {
    uint64_t ret = 0;
    switch (m_regs->rax) {
        case Hypercall::GetFileInfo:
            do_hc_get_file_info(m_regs->rdi, m_regs->rsi, m_regs->rdx);
            break;
        case Hypercall::SubmitFilePointers:
            do_hc_submit_file_pointers(m_regs->rdi, m_regs->rsi, m_regs->
        ↪ rdx);
            break;
        [...]
    }
    m_regs->rax = ret;
    set_regs_dirty();
}
```

As an example, the following code illustrates the handler for the GetFileInfo hypercall. It gets the entry associated to the file number `n`, which may be a shared file or a normal file, and writes the file name and length to memory.

```
GuestFileEntry Vm::file_entry(size_t n) {
    if (n < s_shared_files.size())
        return s_shared_files.entry_at_pos(n);
    else
        return m_files.entry_at_pos(n - s_shared_files.size());
}

void Vm::do_hc_get_file_info(size_t n, vaddr_t path_buf_addr, vaddr_t
        ↪ length_addr) {
    GuestFileEntry entry = file_entry(n);
    m_mmu.write_mem(path_buf_addr, entry.path.c_str(), entry.path.length()
        ↪  + 1);
    m_mmu.write<vsize_t>(length_addr, entry.file.data.length);
}
```

5.2.3 Mmu

The Mmu class is implemented in the file `./src/mmu.cpp`. On creation, it allocates a region of memory and assigns it to a Vm. It also initialises other fields.

```
Mmu::Mmu(int vm_fd, int vcpu_fd, size_t mem_size)
    : m_vm_fd(vm_fd)
    , m_vcpu_fd(vcpu_fd)
    , m_memory((uint8_t*)mmap(nullptr, mem_size, PROT_READ|PROT_WRITE,
        ↪ MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE, -1, 0))
    , m_length(mem_size)
    , m_ptl4(PAGE_TABLE_PADDR)
    , m_can_alloc(true)
    , m_next_page_alloc(PAGE_TABLE_PADDR + 0x1000)
    , m_dirty_bits(m_length/PAGE_SIZE)
    , m_dirty_bitmap(new uint8_t[m_dirty_bits/8])
{
    memset(m_dirty_bitmap, 0, m_dirty_bits/8);
    madvise(m_memory, m_length, MADV_MERGEABLE);
    struct kvm_userspace_memory_region memreg = {
        .slot = 0,
        .flags = KVM_MEM_LOG_DIRTY_PAGES,
        .guest_phys_addr = 0,
        .memory_size = mem_size,
        .userspace_addr = (unsigned long)m_memory
    };
    ioctl_chk(m_vm_fd, KVM_SET_USER_MEMORY_REGION, &memreg);
}
```

Mmu provides methods for reading and writing to memory. The code for writing to a range of memory is listed below. As we can see, it makes use of the PageWalker class to iterate the pages of the given range, translate to physical addresses, and perform the memory copy page-wise. Furthermore, the physical address of every page is added to `m_dirty_extra`, so it can be reset by the `reset()` method.

```
void Mmu::write_mem(vaddr_t dst, const void* src, vsize_t len, CheckPerms
       ↪ check) {
    PageWalker pages(dst, len, *this);
    do {
        ASSERT(!(check == CheckPerms::Yes) || (pages.flags() & PDE64_RW),
               "writing to not writable page %lx", pages.vaddr());
        memcpy(
            m_memory + pages.paddr(),
            (uint8_t*)src + pages.offset(),
            pages.page_size()
        );
        m_dirty_extra.push_back(pages.paddr() & PTL1_MASK);
    } while (pages.next());
}
```

The `load_elf()` method receives a list of ELF segments and loads them to memory. For each segment, its memory range is mapped and copied from the ELF data.

```
void Mmu::load_elf(const vector<segment_t>& segments, ElfType elf_type) {
    uint64_t flags;
    for (const segment_t& segm : segments) {
        if (segm.type != PT_LOAD)
            continue;

        flags = parse_perms(segm.flags);
        flags |= (elf_type ==ElfType::Kernel ? PDE64_SHARED : PDE64_USER);
        alloc(segm.vaddr, segm.memsize, flags);
        write_mem(segm.vaddr, segm.data, segm.filesize, CheckPerms::No);
        set_mem(segm.vaddr + segm.filesize, 0, segm.memsize - segm.
       ↪ filesize, CheckPerms::No);
    }
}
```

The `reset()` method resets the state of the memory of the Vm. To do so, it makes use of the `KVM_GET_DIRTY_LOG`, which fills a bitmap with the pages that have been dirtied and must be restored. Then, the bitmap is iterated, and each dirty page is restored from the memory of the other VM. Finally, the same process is done for the pages in the `m_dirty_extra` list.

```
size_t Mmu::reset(const Mmu& other) {
    size_t count = 0;

    // Get dirty pages bitmap
    kvm_dirty_log dirty = {
        .slot = 0,
        .dirty_bitmap = m_dirty_bitmap
    };
    ioctl_chk(m_vm_fd, KVM_GET_DIRTY_LOG, &dirty);

    // Iterate the dirty bitmap, restoring pages associated with set bits
    const uint8_t* dirty_bitmap = m_dirty_bitmap;
    size_t dirty_bytes = m_dirty_bits/8;
    for (size_t i = 0; i < dirty_bytes; i += sizeof(size_t)) {
        size_t dirty_word = (size_t)(dirty_bitmap + i);
        while (dirty_word != 0) {
            // Get set bit inside the word
            size_t r = __builtin_ctzl(dirty_word);

            // Restore page
            paddr_t paddr = (i*8 + r)*PAGE_SIZE;
            memcpy(m_memory + paddr, other.m_memory + paddr, PAGE_SIZE);
            count++;

            // Clear bit
            size_t t = dirty_word & -dirty_word;
            dirty_word ^= t;
        }
    }

    // Reset the bitmap
    memset(dirty.dirty_bitmap, 0, m_dirty_bits/8);

    // Reset extra pages and clear vector
    for (paddr_t paddr : m_dirty_extra) {
        memcpy(m_memory + paddr, other.m_memory + paddr, PAGE_SIZE);
    }
    count += m_dirty_extra.size();
    m_dirty_extra.clear();

    // Reset state
    m_next_page_alloc = other.m_next_page_alloc;
    return count;
}
```

5.2.4 PageWalker

The PageWalker class is implemented in the file `./src/page_walker.cpp`. It is in charge of walking and modifying the page table of the VM. The main mechanism is virtual to physical address translation through multi-level paging with 4 levels, explained in Section 1.4.4. On creation, the PageWalker gets the indexes inside the different page tables from the virtual address with the `PLTn_INDEX()` macros. Then, it calls `update_pltn()` methods, where $n \in \{1, 2, 3\}$. Each of these methods is in charge of updating the field `m_ptln`, which is the physical address of the page table level n used to translate the given virtual address. First, they calculate the physical address of the entry in page table level $n - 1$, which points to the page table level n we are looking for. Then, the entry is read. If it is empty, then a frame is allocated and set as page table level n. Otherwise, its value is returned. Note there is no `update_ptl4()`, because the PTL4 is the first page table, whose physical address is fixed and it is held by the Mmu.

```cpp
Mmu::PageWalker::PageWalker(vaddr_t vaddr, Mmu& mmu)
    : m_start(vaddr)
    , m_len(numeric_limits<vaddr_t>::max())
    , m_mmu(mmu)
    , m_offset(0)
    , m_ptl4_i(PTL4_INDEX(vaddr))
    , m_ptl3_i(PTL3_INDEX(vaddr))
    , m_ptl2_i(PTL2_INDEX(vaddr))
    , m_ptl1_i(PTL1_INDEX(vaddr))
{
    update_ptl3();
    update_ptl2();
    update_ptl1();
}

void Mmu::PageWalker::update_ptl1() {
    paddr_t p_ptl1 = m_ptl2 + m_ptl2_i * sizeof(paddr_t);
    if (!m_mmu.readp<paddr_t>(p_ptl1)) {
        m_mmu.writep(p_ptl1, m_mmu.alloc_frame() | FLAGS);
    }
    m_ptl1 = m_mmu.readp<paddr_t>(p_ptl1) & PHYS_MASK;
}
```

That way, when a PageWalker object is created with a not mapped virtual address, every page table is created. However, the page corresponding to the virtual address may not be mapped yet. This is done by the `map()` method, which sets the given physical address as entry in the current PTE.

```cpp
paddr_t Mmu::PageWalker::pte() {
    return m_ptl1 + m_ptl1_i*sizeof(paddr_t);
}

void Mmu::PageWalker::map(paddr_t paddr, uint64_t flags) {
    m_mmu.writep(pte(), paddr | flags);
}

void Mmu::PageWalker::alloc_frame(uint64_t flags) {
    map(m_mmu.alloc_frame(), flags);
}
```

Apart from methods for accessing the current PTE like the ones described above, the PageWalker can also iterate the page table with the method `next()`, which advances to the next page, updating indexes recursively.

```cpp
bool Mmu::PageWalker::next() {
    // Advance offset
    m_offset += PAGE_SIZE - PAGE_OFFSET(vaddr());

    // Check if we are at the end of the range
    if (m_offset >= m_len)
        return false;

    // Advance one PTE
    next_ptl1_entry();
    return true;
}

void Mmu::PageWalker::next_ptl1_entry() {
    m_ptl1_i++;
    if (m_ptl1_i == PTRS_PER_PTL1) {
        m_ptl1_i = 0;
        next_ptl2_entry();
    }
}

void Mmu::PageWalker::next_ptl2_entry() {
    m_ptl2_i++;
    if (m_ptl2_i == PTRS_PER_PTL2) {
        m_ptl2_i = 0;
        next_ptl3_entry();
    }
    update_ptl1();
}

...
```

5.2.5 ElfParser

The ElfParser class is in charge of parsing ELF files. It is implemented in the file `./src/elf_parser.cpp`. After the ELF file is mapped to memory, it is parsed and every of the fields are initialised. Data is parsed from the ELF data structures to friendlier C++ structures. An example of getting the list of symbols can be seen below. It involves accessing the `SHT_SYMTAB` section and its associated string section.

```
    for (const section_t& section : m_sections) {
        if (section.type != SHT_SYMTAB && section.type != SHT_DYNSYM)
            continue;

        Elf_Sym* syms = (Elf_Sym*)section.data;
        size_t n_syms = section.size / sizeof(Elf_Sym);

        const char* sec_strtab = (char*)m_sections[section.link].data;
        for (i = 0; i < n_syms; i++) {
            symbol_t symbol = {
                .name       = string(sec_strtab + syms[i].st_name),
                .type       = (uint8_t)ELF_ST_TYPE(syms[i].st_info),
                .binding    = (uint8_t)ELF_ST_BIND(syms[i].st_info),
                .visibility = (uint8_t)ELF_ST_VISIBILITY(syms[i].st_other),
                .shndx      = syms[i].st_shndx,
                .value      = syms[i].st_value,
                .size       = syms[i].st_size
            };
            m_symbols.push_back(symbol);
        }
    }
```

PIE binaries can have their base address changed when their are loaded to memory. This is done by the following function, which calculates the difference between the previous and new base addresses and updates the rest of the addresses accordingly.

```
void ElfParser::set_load_addr(vaddr_t load_addr) {
    vaddr_t diff = load_addr - m_load_addr;
    m_load_addr = load_addr;
    m_entry       += diff;
    m_initial_brk += diff;
    for (segment_t& segment : m_segments) {
        segment.vaddr += diff;
        segment.paddr += diff;
    }
    for (section_t& section : m_sections) {
        section.addr += diff;
    }
    for (symbol_t& symbol : m_symbols) {
        symbol.value += diff;
    }
    for (relocation_t& relocation : m_relocations) {
        relocation.addr += diff;
    }
    if (m_debug_elf)
        m_debug_elf->set_load_addr(load_addr);
}
```

Once the ELF has been parsed and all the ElfParser fields have been initialised, transforming from address to symbol is as simple as:

```
vaddr_t ElfParser::resolve_symbol(const string& symbol_name) const {
    for (const symbol_t& symbol : m_symbols)
        if (symbol.name == symbol_name)
            return symbol.value;
    return 0;
}

bool ElfParser::addr_to_symbol(vaddr_t addr, symbol_t& result) const {
    // Symbols are sorted by address from higher to lower
    for (const symbol_t& symbol : symbols()) {
        if (addr >= symbol.value) {
            result = symbol;
            return true;
        }
    }
    return false;
}

string ElfParser::addr_to_symbol_str(vaddr_t pc) const {
    string result;
    symbol_t symbol;
    if (addr_to_symbol(pc, symbol)) {
        size_t offset = pc - symbol.value;
        result = symbol.name + " + 0x" + utils::to_hex(offset);
    }
    return result;
}
```

Stacktraces across several ELFs are obtained by the following function, which makes use of the `m_debug` field, an instance of ElfDebug that handles debug information.

```
vector<pair<vaddr_t, const ElfParser*>> ElfParser::get_stacktrace(
    const vector<const ElfParser*>& elfs,
    const kvm_regs& kregs, size_t num_frames, Mmu& mmu
) {
    vector<pair<vaddr_t, const ElfParser*>> stacktrace;

    // Transform to dwarf format
    vsize_t regs[DwarfReg::MAX];
    kvm_to_dwarf_regs(kregs, regs);

    // For each frame, get the elf it belongs to and use its DWARF info to
    // get the next frame.
    size_t i = 0;
    const ElfParser* elf = nullptr;
    do {
        elf = elf_with_addr_in_text(elfs, regs[DwarfReg::ReturnAddress]);
        if (!elf)
            break;
        stacktrace.push_back({regs[DwarfReg::ReturnAddress], elf});
        if (elf->is_pie())
            regs[DwarfReg::ReturnAddress] -= elf->load_addr();
    } while (
        ++i < num_frames &&
        elf->m_debug.next_frame(regs, mmu)
    );

    return stacktrace;
}
```

5.2.6 Corpus

The Corpus class is implemented in the file `src/corpus.cpp`. When the Corpus is created, its `m_mutated_inputs` field is initialised as a vector of size `nthreads`, and `m_corpus` is initialised with strings corresponding to the contents of every file in the input directory. Then, the method `get_new_input()` is in charge of providing a new input to the fuzzer. It receives `id`, which is an integer from 0 to `nthreads-1` unique to each thread, and which represents a position in `m_mutated_inputs`. The method selects a random item from the corpus, copies it to that position, and mutates it in place.

```
FileRef Corpus::get_new_input(int id, Rng& rng, Stats& stats) {
    while (m_lock_corpus.test_and_set());
    size_t i = rng.rnd(0, m_corpus.size() - 1);
    m_mutated_inputs[id] = m_corpus[i];
    m_lock_corpus.clear();

    mutate_input(id, rng);
    return FileRef::from_string(m_mutated_inputs[id]);
}
```

After the input is run, the worker thread will report coverage through `report_coverage()`, which in normal mode will add the coverage to its recorded coverage, and save the input in case it triggered new paths.

```
void Corpus::report_coverage(int id, const Coverage& cov) {
    switch (m_mode) {
        case Mode::Normal:
            if (m_recorded_coverage.add(cov)) {
                add_input(m_mutated_inputs[id]);
            }
            break;
        [...]
    }
}
```

5.2.7 Mutator

The class Mutator is implemented in the file `src/mutator.cpp`. The main method is `mutate_input()`, which mutates an input in place by applying a random number of mutations. If the argument `minimize` is set, then the mutation `mut_shrink()` will be called at least once, so the result is smaller than the original input.

```
void Mutator::mutate_input(string& input, Rng& rng, bool minimize){
    size_t n_muts = rng.rnd(MIN_MUTATIONS, MAX_MUTATIONS);
    mutation_strat_t mut_strat;
    if (!minimize) {
        for (size_t i = 0; i < n_muts; i++){
            mut_strat = mut_strats[rng.rnd(0, mut_strats.size()-1)];
            (this->*mut_strat)(input, rng);
        }
    } else {
        size_t j, i_mut_shrink = rng.rnd(0, n_muts - 1);
        for (size_t i = 0; i < n_muts; i++) {
            if (i == i_mut_shrink) {
                mut_shrink(input, rng);
            } else {
                j = rng.rnd(0, mut_strats_reduce.size()-1);
                mut_strat = mut_strats_reduce[j];
                (this->*mut_strat)(input, rng);
            }
        }
    }
}
```

Mutations are chosen from the list `mut_strats`, which is a list of pointers to methods that apply a specific mutation strategy. As an example, two of them are listed below. The first one chooses a random offset in the given input and flips a random bit on that position. The second chooses a random position and copies to it a random slice of the input. There are a total of 21 different mutation strategies implemented.

```
void Mutator::mut_bit(string& input, Rng& rng){
    if (input.empty())
        return;
    size_t offset   = rand_offset(input, rng);
    uint8_t bit     = rng.rnd(0, 7);
    input[offset] ^= (1 << bit);
}

void Mutator::mut_copy(string& input, Rng& rng){
        if (input.empty())
        return;
    size_t src = rand_offset(input, rng);
    size_t dst = rand_offset(input, rng);
    size_t src_remaining = input.size() - src;
    size_t dst_remaining = input.size() - dst;
    size_t len = rng.rnd_exp(1, min(src_remaining, dst_remaining));
    input.replace(dst, len, input, src, len);
}
```

5.2.8 Coverage

Coverage classes are implemented in the files include/coverage_breakpoints.h, include/coverage_intel_pt.h and include/coverage_none.h. File include/coverage.h includes one of those files depending on a compile option.

Breakpoints

File include/coverage_breakpoints.h defines classes CoverageBreakpoints and SharedCoverageBreakpoints. CoverageBreakpoints has a template parameter SetContainer which defines the type of the container that will hold the virtual addresses of hit breakpoints, and which by default is a std::set. SharedCoverageBreakpoints inherits from CoverageBreakpoints, and uses a std::unordered_set instead. This is because CoverageBreakpoints prioritizes iteration performance, while the shared version prioritizes performance when checking if a breakpoint has already been hit or not.

Methods implementations are trivial, as they just access the underlying container. The implementation of the method add() is shown below. It joins a CoverageBreakpoints to a SharedCoverageBreakpoints, and is called every time the fuzzer reports coverage to the Corpus. As the instance of SharedCoverageBreakpoints is shared by all threads, access must be controlled with a lock.

```
template <class T> inline
bool SharedCoverageBreakpoints::add(const CoverageBreakpoints<T>& other) {
    while (m_lock.test_and_set());
    size_t prev_count = count();
    blocks().insert(other.blocks().begin(), other.blocks().end());
    bool new_cov = count() != prev_count;
    m_lock.clear();
    return new_cov;
}
```

Intel PT

File include/coverage_intel_pt.h defines classes CoverageIntelPT and SharedCoverageIntelPT. CoverageIntelPT has a bitmap and simply provides access to it, which is filled by the Intel PT module. SharedCoverageIntelPT inherits from it, adding the method add() and the field m_bitmap_count, which is an atomic indicating the number of bits set in the bitmap.

The method add() iterates the given bitmap and adds its set bits to the shared bitmap. As the bitmap is shared, this must be performed atomically. The given bitmap is iterated using the type word_t, which depends on the capabilities of the CPU where the hypervisor is built, and can be an integer ranging from 64 to 256 bits.

```
// The type we'll use to iterate the bitmap
#if defined(__AVX2__)
typedef __m256i_u word_t;
#elif defined(__SSE2__)
typedef __m128i_u word_t;
#elif defined(__SIZEOF_INT128__)
typedef __uint128_t word_t;
#else
typedef size_t word_t;
#endif
```

In order to compare if two instances of `word_t` are equal, the appropriate SIMD intrinsic is used, as can be seen in the following listing. If they are not, the bits in the word are then iterated by `add_bits_in_word_at_byte()`. Finally, in order to access the shared bitmap atomically, the function `lock_bts()` implements inline assembly wich uses the `lock bts` instruction.

```
inline bool SharedCoverageIntelPT::add(const CoverageIntelPT& other) {
    uint8_t *bitmap = this->bitmap();
    const uint8_t* other_bitmap = other.bitmap();
    size_t new_cov = 0;

    // Go over both bitmaps using word_t. When there is new coverage in
        ↪ one
    // of those words, go over its bits to see which is the new one.
    for (size_t i = 0; i < COVERAGE_BITMAP_SIZE; i += sizeof(word_t)) {
        word_t cov_v       = *(word_t*)(bitmap + i);
        word_t other_cov_v = *(word_t*)(other_bitmap + i);
#if defined(__AVX2__)
        word_t comp = (cov_v | other_cov_v) == cov_v;
        bool equals = _mm256_movemask_epi8(comp) == 0xffffffff;
#elif defined(__SSE2__)
        word_t comp = (cov_v | other_cov_v) == cov_v;
        bool equals = _mm_movemask_epi8(comp) == 0xffff;
#else
        bool equals = (cov_v | other_cov_v) == cov_v;
#endif
        if (!equals) {
            // There is new coverage. Test each bit in the word.
            new_cov += add_bits_in_word_at_byte(i, bitmap, other_bitmap);
        }
    }

    m_bitmap_count += new_cov;
    return new_cov > 0;
}

static size_t add_bits_in_word_at_byte(size_t i, uint8_t* bitmap, const
        ↪ uint8_t* other_bitmap) {
    // Check each of the bytes of the word starting at byte i.
    size_t new_cov = 0;
    for (size_t j = i*8; j < (i + sizeof(word_t))*8; j++) {
        size_t j_q = j / 8;
        size_t j_r = j % 8;
        if (other_bitmap[j_q] & (1 << j_r)) {
            // Set bit in our bitmap. If it was 0, then that's a new bit
            new_cov += !lock_bts(j_r, &bitmap[j_q]);
        }
    }
    return new_cov;
}
```

```

// Bit test and set: set a bit and return its previous value atomically
__attribute__((always_inline)) inline
bool lock_bts(int i, uint8_t* p) {
    bool test = false;
    asm(
        "lock bts %[i], %[val];"
        "setc %[test];"
        : [val] "+m" (*p),
          [test] "+r" (test)
        : [i] "r" (i)
        : "cc"
    );
    return test;
}
```

5.2.9 Virtual files

Classes FileRefsByPath and SharedFiles implement virtual files on the hypervisor, and are quite simple. They are implemented in the file `./src/files.cpp`. As an example, below is the implementation of the method `set_file` for both classes. The implementation for FileRefsByPath receives a FileRef pointing to memory it doesn't own, and simply sets it as content for the given path. On the other hand, the implementation for SharedFiles receives a string, which is copied into its field `m_file_contents`. Then, a reference to that string, which is now owned by SharedFiles, is set as content for the file through a call to `FileRefsByPath::set_file()`.

```
GuestFile FileRefsByPath::set_file(const string& path, FileRef content) {
    GuestFile& file = m_files[path];
    file.data = content;
    return file;
}

GuestFile SharedFiles::set_file(const string& path, string content) {
    string& content_ref = m_file_contents[path];
    content_ref = move(content);
    FileRef ref = FileRef::from_string(content_ref);
    return FileRefsByPath::set_file(path, ref);
}

GuestFile SharedFiles::set_file(const string& path) {
    return set_file(path, utils::read_file(path));
}
```

5.2.10 Tracing

The Tracing class holds the current execution trace, and is in charge of performing the measurements in cycles or instructions and dumping the traces to disk. Measurements are taken with the following method, which reads the MSR that has the appropriate counter (Section 1.4.4).

```
size_t Tracing::get_tracing_measure() {
    switch (m_unit) {
        case Unit::Instructions:
            return m_vm.read_msr(MSR_FIXED_CTR0);
        case Unit::Cycles:
            return m_vm.read_msr(MSR_FIXED_CTR1);
    };
}
```

As described in Section 4.1.11, measurements can be performed in user space on each basic block, or in kernel space on each syscall. When tracing in kernel space, the method `prepare()` is called at the beginning of each syscall, and `trace()` is called at the end. Both function perform a measurement, and the second one adds the difference to the trace.

```
void Tracing::prepare(string name) {
    // Special case for exit_group, because it's the last syscall and
       ↪ therefore
    // there won't be a call to trace().
    if (name == "exit_group") {
        m_trace.push_back({name, 0});
        return;
    }

    m_measure = Measure{
        .name = name,
        .start = get_tracing_measure(),
    };
}

size_t Tracing::trace() {
    size_t current_measure = get_tracing_measure();
    if (!m_measure.name.empty()) {
        size_t measure = current_measure - m_measure.start;
        m_trace.push_back({m_measure.name, measure});
    }
    m_measure = {};
    return current_measure;
}
```

When tracing in user space, method `trace_and_prepare()` is called on each breakpoint. It is implemented this way because when tracing basic blocks, a measurement is taken on each breakpoint, and the difference with the previous measurement is added to the trace.

```
void Tracing::trace_and_prepare(string name) {
    size_t current_measure = trace();
    m_measure = Measure{
        .name = name,
        .start = current_measure,
    };
}
```

Finally, traces are dumped to disk with the method `dump_trace()`.

```
void Tracing::dump_trace(size_t id) {
    if (m_type == Type::None)
        return;

    size_t trace_id = m_next_trace_id++;
    string filename = "traces/" + to_string(id) + "_" + to_string(trace_id
       ↪ );
    ofstream out(filename);
    for (pair<string, size_t> element : m_trace) {
        out << element.first << " " << to_string(element.second) << endl;
    }
    m_trace.clear();
}
```

5.3 Kernel

The kernel implementation is in the folder `kernel`. It further contains the folder `src`, which consists of the source code files, and the file `linker.ld`, which is linker script that tells the linker how to link the kernel binary.

In the following sections, implementation details of some of the classes and modules presented in the design (Section 4.2) are discussed.

5.3.1 Hypercalls

Hypercalls are implemented in the `src/hypercalls.zig` file. As already detailed, hypercalls work by executing a privileged instruction that triggers a VM exit, transferring execution to the hypervisor, which handles the request. This is implemented using inline assembly. There is a basic function `hypercall` that performs the privileged instruction, and then one function for each hypercall, which sets the `rax` register to the corresponding hypercall identifier and jumps to `hypercall`.

```
comptime {
    asm (
        \\hypercall:
        \\  outb %al, $16;
        \\  ret;
        \\
        \\_print:
        \\  mov $1, %rax
        \\  jmp hypercall
        \\
        \\getMemInfo:
        \\  mov $2, %rax
        \\  jmp hypercall
        \\
        \\getKernelBrk:
        \\  mov $3, %rax
        \\  jmp hypercall
        \\
        ...
    );
}
```

Then, some hypercalls such as `getFileInfo` are just exported like that, while others provide a wrapper. In this case, the hypercall `loadLibrary` is not exported directly. Instead, it can be accessed through the function `maybeLoadLibrary`, which is called by the `mmap` syscall handler, and checks if it was called from the interpreter binary. In that case, it gets the name of the file being mapped, and finally calls the hypercall.

```
pub extern fn getFileInfo(n: usize, path_buf: [*]u8, length_ptr: *usize)
        ↪ void;
extern fn loadLibrary(filename: [*]const u8, filename_len: usize,
        ↪ load_addr: usize) void;
pub fn maybeLoadLibrary(mmap_addr: usize, mmap_file: []const u8, rip:
        ↪ usize) void {
    // Check if mmap syscall was called from the interpreter
    if (!(interpreter_range.start <= rip and rip < interpreter_range.end))
        return;

    // Get the filename of the library and tell the hypervisor to load it
    const filename = fs.file_manager.filenameFromFileContent(mmap_file)
        ↪ orelse return;
    loadLibrary(filename.ptr, filename.len, mmap_addr);
}
```

5.3.2 Memory management

Memory management package is implemented in the `./src/mem` folder. Its main modules and classes are exposed by the `./src/mem/mem.zig` file.

Physical Memory Manager

The PMM is implemented in the `pmm3.zig` file. It manages physical memory as follows:

```
var free_frames: []usize = undefined;
var free_frames_len: usize = 0;

pub fn allocFrame() Error!usize {
    if (free_frames_len == 0)
        return Error.OutOfMemory;
    free_frames_len -= 1;
    const frame = free_frames[free_frames_len];
    memsetFrame(frame, 0);
    return frame;
}

pub fn freeFrame(frame: usize) void {
    assert(mem.isPageAligned(frame));
    memsetFrame(frame, undefined);
    free_frames[free_frames_len] = frame;
    free_frames_len += 1;
}

pub fn dupFrame(frame: usize) Error!usize {
    const new_frame = try allocFrame();
    const new_frame_virt = physToVirt(*[std.mem.page_size]u8, new_frame);
    const frame_virt = physToVirt(*[std.mem.page_size]u8, frame);
    std.mem.copy(u8, new_frame_virt, frame_virt);
    return new_frame;
}

fn memsetFrame(frame: usize, value: u8) void {
    std.mem.set(u8, physToVirt(*[std.mem.page_size]u8, frame), value);
}
```

Frame allocation and freeing is done from the free list `free_frames`. Note that when allocating, frame content is always set to 0. Then, when freeing, it is set to the value `undefined`, which means it is set to value `0xAA` in debug mode, but the `memset` will not happen in release modes. This helps catch Use After Free bugs in debug modes.

In order to write to the physical frame, it is needed to access physical memory. This is what the physmap, a virtual mapping of all physical memory, is for. The PMM provides the following functions for accessing it, which convert a physical memory address to its virtual one in the physmap and viceversa.

```
pub fn physToVirt(comptime ptr_type: type, phys: usize) ptr_type {
    assert(phys <= memory_length);
    const ret = mem.layout.physmap + phys;
    if (@typeInfo(ptr_type) == .Pointer) {
        return @intToPtr(ptr_type, ret);
    } else if (@typeInfo(ptr_type) == .Int) {
        return ret;
    } else {
        @compileError("ptr_type must be a pointer or an integer");
    }
}

pub fn virtToPhys(virt: anytype) usize {
    assert(@typeInfo(@TypeOf(virt)) == .Pointer);
    const virt_flat = @ptrToInt(virt);
    const physmap = mem.layout.physmap;
    assert(physmap <= virt_flat and virt_flat < physmap + memoryLength());
    return virt_flat - physmap;
}
```

Virtual Memory Manager

The VMM is implemented in the `vmm.zig` file. It has the following fields:

```
/// The kernel page table.
pub var kernel_page_table: x86.paging.KernelPageTable = undefined;

/// Start of the allocations regions. Set to kernel brk at init().
var allocations_base_addr: usize = 0;

/// Bitset indicating if a page is free or not.
var free_bitset = std.StaticBitSet(bitset_size).initFull();

/// Each page of bitset allows for 4096*8 pages of virtual memory, which
    ↪ is 128MB.
const bitset_size_bytes = std.mem.page_size * 4;
const bitset_size = bitset_size_bytes * std.mem.byte_size_in_bits;
const max_memory = bitset_size * std.mem.page_size;
```

The bitset is accessed with the following functions, which provide a way for setting pages as free or allocated and for finding a range of free pages.

```
fn addrToPageIndex(addr: usize) usize {
    assert(mem.isPageAligned(addr));
    return (addr - allocations_base_addr) / std.mem.page_size;
}

fn setPageFree(i: usize) void {
    assert(i < free_bitset.capacity());
    assert(!free_bitset.isSet(i));
    free_bitset.set(i);
}

fn setPageAllocated(i: usize) void {
    assert(i < free_bitset.capacity());
    assert(free_bitset.isSet(i));
    free_bitset.unset(i);
}

fn isPageFree(i: usize) bool {
    assert(i <= free_bitset.capacity());
    return free_bitset.isSet(i);
}

fn findFreeRange(num_pages: usize) ?usize {
    // Initialize the current range with the first ocurrence of a free
        ↪ page,
    // if there's any.
    var iter = free_bitset.iterator(.{});
    var current_range_i: usize = iter.next() orelse return null;
    var current_range_size: usize = 1;
    var prev_i: usize = current_range_i;

    // If we're looking for just one page, we already have it.
    if (num_pages == 1)
        return current_range_i;

    // Iterate the bitmap until we find a large enough range of contiguous
    // free pages.
    while (iter.next()) |i| {
        if (i == prev_i + 1) {
            current_range_size += 1;
        } else {
            current_range_i = i;
            current_range_size = 1;
        }
        if (current_range_size == num_pages)
            return current_range_i;
        prev_i = i;
    }
    return null;
}
```

Finally, it exposes the following functions, which allocate and free pages into the kernel address space by accessing the bitset and the kernel page table.

```
/// Allocate a number of kernel pages, mapping them with given options.
pub fn allocPages(n: usize, options: MappingOptions) !usize {
    // Check if we have enough memory available. Even if we have now, we
    // may fail later, but this check should almost always avoid that.
    if (mem.pmm.amountFreeFrames() < n)
        return AllocPageError.OutOfMemory;

    // Find range of not allocated pages
    const i_range_start = findFreeRange(n) orelse return AllocPageError.
        ↪ OutOfMemory;
    const i_range_end = i_range_start + n;
    const range_start = allocations_base_addr + i_range_start * std.mem.
        ↪ page_size;

    // Iterate the range, allocating a frame for every page and mapping it
    var i: usize = i_range_start;
    var page_base = range_start;
    while (i < i_range_end) : ({
        i += 1;
        page_base += std.mem.page_size;
    }) {
        const frame = try mem.pmm.allocFrame();
        setPageAllocated(i);
        kernel_page_table.mapPage(page_base, frame, options) catch |err|
        ↪ switch (err) {
            error.OutOfMemory => return AllocPageError.OutOfMemory,
            error.AlreadyMapped => unreachable,
        };
    }

    return range_start;
}

/// Free pages returned by allocPages(), unmapping them and freeing the
/// underlying memory.
pub fn freePages(addr: usize, n: usize) FreePageError!void {
    // Iterate the range, unmapping each page, and thus freeing each frame
    const i_range_start = addrToPageIndex(addr);
    const i_range_end = i_range_start + n;
    var i: usize = i_range_start;
    var page_base = allocations_base_addr + i * std.mem.page_size;
    while (i < i_range_end) : ({
        i += 1;
        page_base += std.mem.page_size;
    }) {
        setPageFree(i);
        try kernel_page_table.unmapPage(page_base);
    }
}
```

Heap

The heap module in `heap.zig` provides three different allocators. First, the PageAllocator class, which simply forwards requests to the VMM.

```
const PageAllocator = struct {
    fn alloc(len: usize) Allocator.Error![]u8 {
        // The pages we're going to allocate
        const num_pages = @divExact(mem.alignPageForward(len), std.mem.
        ↪ page_size);

        // Alocate the pages
        const range_start = try mem.vmm.allocPages(num_pages, .{
            .writable = true,
            .global = true,
            .noExecute = true,
        });

        // Construct the slice and return it
        const range_start_ptr = @intToPtr([*]u8, range_start);
        const slice = range_start_ptr[0..len];
        return slice;
    }

    fn free(buf: []u8) void {
        // Free pages associated to `buf`.
        const buf_len_aligned = mem.alignPageForward(buf.len);
        const num_pages = @divExact(buf_len_aligned, std.mem.page_size);
        mem.vmm.freePages(@ptrToInt(buf.ptr), num_pages) catch |err|
       ↪ switch (err) {
            error.NotMapped => unreachable,
        };
    }
}
```

Then, the HeapAllocator, which satisfies requests from a contiguous memory region, and does not perform free.

```
pub const HeapAllocator = struct {
    base: usize,
    used: usize,
    size: usize,

    pub fn init() HeapAllocator {
        return HeapAllocator{
            .base = 0,
            .used = 0,
            .size = 0,
        };
    }

    fn more(self: *HeapAllocator, num_pages: usize) Allocator.Error!void {
        const ret = try mem.vmm.allocPages(num_pages, .{
            .writable = true,
            .global = true,
            .noExecute = true,
        });
        assert(ret == self.base + self.size);
        self.size += num_pages * std.mem.page_size;
    }

    fn alloc(self: *HeapAllocator, len: usize) Allocator.Error![]u8 {
        const ret_offset = std.mem.alignForward(self.used, ptr_align);
        const free_bytes = self.size - ret_offset;
        if (len > free_bytes) {
            const needed_memory = mem.alignPageForward(len - free_bytes);
            const num_pages = @divExact(needed_memory, std.mem.page_size);
            try self.more(num_pages);
        }

        self.used = ret_offset + len;
        assert(self.used <= self.size);
        const ret = self.base + ret_offset;
        return @intToPtr([*]u8, ret)[0..len];
    }

    fn free(self: *HeapAllocator, buf: []u8) void { }
};
```

And finally, the BlockAllocator, which has linked lists of free blocks of different sizes. When a free linked list is empty, it allocates a page from the page allocator and splits its in blocks, which are added to the linked list.

```
pub const BlockAllocator = struct {
    /// Heads of the linked lists of free blocks.
    list_heads: [BLOCK_SIZES.len]?*Block,
```

```zig

    /// The header of each free block, which contains a pointer to the
    /// next free block of the same size.
    const Block = struct {
        next: ?*Block,
    };

    /// The block sizes. There will be a linked list of blocks of each
    /// size. These sizes must be ordered.
    const BLOCK_SIZES = [_]usize{ 8, 16, 32, 64, 128, 256, 512, 1024, 2048
        ↪  };

    pub fn init() BlockAllocator {
        return BlockAllocator{
            .list_heads = [_]?*Block{null} ** BLOCK_SIZES.len,
        };
    }

    fn getListIndex(size: usize) ?usize {
        for (BLOCK_SIZES) |block_size, i| {
            if (block_size >= size) return i;
        }
        return null;
    }

    fn blockToSlice(block_ptr: *Block, size: usize) []u8 {
        return @ptrCast([*]u8, block_ptr)[0..size];
    }

    fn sliceToBlock(slice: []u8) *Block {
        return @ptrCast(*Block, @alignCast(@sizeOf(Block), slice));
    }

    fn moreBlocksForList(self: *BlockAllocator, list_index: usize) !void {
        const page = try page_allocator.alloc(u8, std.mem.page_size);
        const block_len = BLOCK_SIZES[list_index];
        var block_addr = @ptrToInt(page.ptr);
        var page_end = block_addr + std.mem.page_size;
        while (block_addr < page_end - block_len) : (block_addr +=
        ↪ block_len) {
            var block_ptr = @intToPtr(*Block, block_addr);
            const next_ptr = @intToPtr(*Block, block_addr + block_len);
            block_ptr.next = next_ptr;
        }
        var last_block = @intToPtr(*Block, block_addr);
        last_block.next = null;
        self.list_heads[list_index] = sliceToBlock(page);
    }

    fn alloc(self: *BlockAllocator, len: usize) Allocator.Error![]u8 {
        // Get list index corresponding to `len`, or fallback to page
        // allocator if there isn't any
        const list_index = getListIndex(len) orelse {
            return page_allocator.alloc(page_allocator.ptr, len);
        };

        // Allocate blocks for the list if there isn't any
        if (self.list_heads[list_index] == null)
            try self.moreBlocksForList(list_index);

        // Get the head, and set the next block as the new head
        const block_ptr = self.list_heads[list_index].?;
        const next = block_ptr.next;
        self.list_heads[list_index] = next;

        // Return the slice with aligned length
        const block_len = BLOCK_SIZES[list_index];
        const ret = blockToSlice(block_ptr, len);
        return ret;
    }

    fn free(self: *BlockAllocator, buf: []u8) void {
        const list_index = getListIndex(buf.len) orelse {
            return page_allocator.free(page_allocator.ptr, buf);
        };
        const freed_block = sliceToBlock(buf);
        freed_block.next = self.list_heads[list_index];
        self.list_heads[list_index] = freed_block;
    }
};
```

AddressSpace

The AddressSpace class, which represents the address space of an user process, is implemented in `address_space.zig`. It holds a PageTable, a RegionManager, and a reference counter.

```
pub const AddressSpace = struct {
    page_table: PageTable,
    user_mappings: mem.RegionManager,
    ref: RefCounter,
}
```

The method in charge of allocating and mapping memory receives the following two arguments as options, which indicate the permissions and other flags.

```
pub const Perms = packed struct {
    read: bool = false,
    write: bool = false,
    exec: bool = false,

    pub fn isNone(perms: Perms) bool {
        return !perms.read and !perms.write and !perms.exec;
    }
};

pub const MapFlags = packed struct {
    discardAlreadyMapped: bool = false,
    shared: bool = false,
};
```

However, when interacting with the page table, these permissions and flags must be mapped to an object of type MappingOptions, which is the one actually delivered to the page table. The `mapRange` method allocates memory for the range, marks it as mapped in its RegionManager, and maps every page. The `mapRangeAnywhere` method does the same, but uses the RegionManager to get the address of a free range of memory.

```
pub fn mapRange(
    self: *AddressSpace,
    addr: usize,
    length: usize,
    perms: Perms,
    flags: MapFlags,
) MappingError!void {
    try checkRange(addr, length);

    // Mark range as mapped
    try self.user_mappings.setMapped(addr, addr + length);

    // Attempt to allocate physical memory for the range
    const num_frames = @divExact(length, std.mem.page_size);
    var frames = try mem.pmm.allocFrames(self.ref.allocator, num_frames);
    defer self.ref.allocator.free(frames);

    // Get the mapping options acording to memory permissions and flags
    const mapping_options = x86.paging.PageTable.MappingOptions{
        .writable = perms.write,
        .user = true,
        .protNone = perms.isNone(),
        .shared = flags.shared,
        .noExecute = !perms.exec,
        .discardAlreadyMapped = flags.discardAlreadyMapped,
    };

    // Map every page
    var i: usize = 0;
    var page_base: usize = addr;
    while (i < num_frames) : ({
        i += 1;
        page_base += std.mem.page_size;
    }) {
        try self.page_table.mapPage(page_base, frames[i],mapping_options);
    }
}

pub fn mapRangeAnywhere(
    self: *AddressSpace,
    length: usize,
    perms: Perms,
    flags: MapFlags,
) error{OutOfMemory}!usize {
    const range_base_addr = self.user_mappings.findNotMapped(length)
        ↪ orelse return error.OutOfMemory;
    try self.mapRange(range_base_addr, length, perms, flags);
    return range_base_addr;
}
```

Finally, the unmapRange method sets the range as not mapped and unmaps it from the page table.

```
pub fn unmapRange(
    self: *AddressSpace,
    addr: usize,
    length: usize,
) UnmappingError!void {
    try checkRange(addr, length);

    // Mark range as not mapped
    try self.user_mappings.setNotMapped(addr, addr + length);

    // Unmap every page
    var not_mapped: bool = false;
    const addr_end = addr + length;
    var page_base: usize = addr;
    while (page_base < addr_end) : (page_base += std.mem.page_size) {
        try self.page_table.unmapPage(page_base);
    }
}
```

MemSafe

The MemSafe package implements safe access to user memory in the file `safe.zig`. The low level implementation consists of the function `copyBase`, which performs a copy between two pointers via inline assembly. The `rep movsb` instructions copies `rcx` bytes from `rdi` to `rsi`. The inline assembly expression returns the value of `rcx` after the copy, which under normal circumstances should be 0. However, if the memory access faults, it will trigger a page fault. The page fault handler will then call `handleSafeAccessFault`, which will set the return address to `safe_copy_ins_faulted` inside the assembly of `copyBase`. Execution will continue there, but `rcx` will not be 0 because the copy was not performed completely, so it will return an error. That way, `copyBase` performs a memory copy without crashing the kernel in case any of the pointers are invalid.

```
pub fn handleSafeAccessFault(frame: *interrupts.InterruptFrame) bool {
    if (frame.rip == @ptrToInt(&safe_copy_ins_may_fault)) {
        frame.rip = @ptrToInt(&safe_copy_ins_faulted);
    } else if (frame.rip == @ptrToInt(&safe_strlen_ins_may_fault)) {
        frame.rip = @ptrToInt(&safe_strlen_ins_faulted);
    } else return false;
    return true;
}

fn copyBase(dest: []u8, src: []const u8) Error!void {
    const bytes_left = asm volatile (
        \\safe_copy_ins_may_fault:
        \\rep movsb
        \\safe_copy_ins_faulted:
        : [ret] "={rcx}" (-> usize),
        : [dest] "{rdi}" (dest.ptr),
          [src] "{rsi}" (src.ptr),
          [len] "{rcx}" (src.len),
    );
    if (bytes_left != 0)
        return Error.Fault;
}
```

Then, there are other higher level functions that accept UserPtr and UserSlice as arguments and are easier to use. There is a great amount of assertions to verify a correct behaviour, since invalid memory copies are usually a source of memory corruption and bugs.

```
fn copy(comptime T: type, dest: []T, src: []const T) Error!void {
    assert(dest.len == src.len);
    try copyBase(std.mem.sliceAsBytes(dest), std.mem.sliceAsBytes(src));
}

pub fn copyFromUser(comptime T: type, dest: []T, src: UserSlice([]const T)
    ↪ ) Error!void {
    // Make sure we're copying from user to kernel.
    assert(isSliceInKernelRange(T, dest));
    assert(isSliceInUserRange(T, src.slice()));

    // Try to perform copy.
    try copy(T, dest, src.slice());
}

pub fn copyFromUserSingle(comptime T: type, dest: *T, src: UserPtr(*const
    ↪ T)) Error!void {
    // Make sure we're copying from user to kernel.
    assert(isPtrInKernelRange(T, dest));
    assert(isPtrInUserRange(T, src.ptr()));

    // Try to perform copy.
    try copySingle(T, dest, src.ptr());
}
```

UserPtr and UserSlice are two classes that are simple, thin wrappers over normal pointers and slices, whose goal is to not access user pointers and slices directly, but instead use the functions detailed above. Therefore, their implementation is not worth detailing.

5.3.3 x86

Memory management package is implemented in the `./src/x86` folder. Its main modules and classes are exposed by the `./src/x86/x86.zig` file.

Asm

The Asm package implements inline assembly functions. Some examples can be seen below.

```
pub fn rdmsr(msr: MSR) usize {
    var high: u32 = undefined;
    var low: u32 = undefined;
    asm volatile (
        \\rdmsr
        : [high] "={edx}" (high),
          [low] "={eax}" (low),
        : [msr] "{rcx}" (msr),
        : "memory"
    );
    return (@intCast(u64, high) << 32) | low;
}

pub fn flush_tlb_entry(page_vaddr: usize) void {
    asm volatile (
        \\invlpg (%[page])
        :
        : [page] "r" (page_vaddr),
        : "memory"
    );
}
```

GDT and TSS

The GDT and TSS are implemented in the file `gdt.zig`. The Global-Descriptor definition is a packed structure, which implements methods for initialising it easily.

```
const AccessBits = packed struct {
    accessed: u1,
    read_write: u1,
    dc: u1,
    executable: u1,
    descriptor: u1,
    privilege: u2,
    present: u1,
};

const FlagsBits = packed struct {
    zero: u1 = 0,
    long: u1,
    size: u1,
    granularity: u1,
};

const GlobalDescriptor = packed struct {
    limit_low: u16,
    base_low: u16,
    base_mid: u8,
    access: AccessBits,
    limit_high: u4,
    flags: FlagsBits,
    base_high: u8,

    const Type = enum {
        data,
        code,
    };

    pub fn init_null() GlobalDescriptor {
        return std.mem.zeroes(GlobalDescriptor);
    }

    pub fn init(comptime type_: Type, privilege: u2) GlobalDescriptor {
        ...
    }
};
```

It is similar for the TaskStateSegmentDescriptor, whose constructor accepts a pointer to the TSS:

```
const TaskStateSegmentDescriptor = packed struct {
    descriptor: GlobalDescriptor,
    base_higher: u32,
    zero: u32 = 0,

    pub fn init(tss_ptr: *const TaskStateSegment)
        ↪ TaskStateSegmentDescriptor {
        ...
    }
```

Then, the TSS is implemented as follows:

```zig
/// Stack used when an interrupt causes a change to ring zero. Set in the
    ↪ TSS.
var stack_rsp0: [0x2000]u8 align(std.mem.page_size) = undefined;

/// Stack used when handling interrupts like double faults, where the
    ↪ kernel
/// stack may be corrupted. Set in the TSS, and referenced by the `ist`
    ↪ field
/// in an Interrupt Descriptor.
var stack_ist1: [0x2000]u8 align(std.mem.page_size) = undefined;

/// TSS
const TaskStateSegment = packed struct {
    reserved1: u32 = 0,
    rsp0, rsp1, rsp2: u64,
    reserved2: u64 = 0,
    ist1, ist2, ist3, ist4, ist5, ist6, ist7: u64,
    reserved3: u64 = 0,
    reserved4: u16 = 0,
    iopb: u16 = 104,,

    pub fn init() TaskStateSegment {
        var ret = std.mem.zeroes(TaskStateSegment);
        ret.rsp0 = @ptrToInt(&stack_rsp0) + stack_rsp0.len;
        ret.ist1 = @ptrToInt(&stack_ist1) + stack_ist1.len;
        return ret;
    }
};
```

Finally, the GDT and TSS themselves are initialised by the function `init()` of the module:

```zig
const N_GDT_ENTRIES = 7;
const KernelPrivilegeLevel = 0;
const UserPrivigeLevel = 3;

var gdt_tmp: [N_GDT_ENTRIES]GlobalDescriptor = undefined;
var tss: TaskStateSegment = undefined;

pub fn init() void {
    tss = TaskStateSegment.init();
    gdt[0] = GlobalDescriptor.init_null();
    gdt[1] = GlobalDescriptor.init(.code, KernelPrivilegeLevel);
    gdt[2] = GlobalDescriptor.init(.data, KernelPrivilegeLevel);
    gdt[3] = GlobalDescriptor.init(.data, UserPrivigeLevel);
    gdt[4] = GlobalDescriptor.init(.code, UserPrivigeLevel);
    const tss_descriptor = TaskStateSegmentDescriptor.init(&tss);
    std.mem.copy(GlobalDescriptor, gdt[5..], tss_descriptor);

    x86.lgdt(&gdt);
    x86.ltr(.TaskStateSegment);
}
```

IDT

The IDT is implemented in the file `idt.zig`. Similarly to the GDT, IDT entries are defined by the packed structure InterruptDescriptor, whose definition is not included here for brevity. The IDT itself is initialised as follows:

```
/// The IDT itself.
var idt: [N_IDT_ENTRIES]InterruptDescriptor = undefined;

pub fn init() void {
    // Initialize IDT
    for (idt) |*entry, i| {
        const gate_type: InterruptDescriptor.GateType = if (i < 32) .Trap
        ↪ else .Interrupt;
        const interrupt_handler = interrupt_handlers_entry_points[i];
        const interrupt_stack = if (i == ExceptionNumber.DoubleFault) @as(
        ↪ u3, 1) else 0;
        entry.* = InterruptDescriptor.init(interrupt_handler,
        ↪ interrupt_stack, gate_type);
    }

    // Load IDT
    x86.lidt(&idt);

    log.debug("IDT initialized\n", .{});
}
```

Note the interrupt handler of the Double Fault exception uses a special stack, pointed to by the `ist1` field of the TSS, as described in Section 4.2.3.

Interrupts

The interrupt handlers are implemented in the `interrupt.zig` file. The entry point function is generated by `getInterruptHandlerEntryPoint` for each interrupt number with inline assembly.

```
pub fn getInterruptHandlerEntryPoint(comptime interrupt_number: usize)
        ↪ InterruptHandlerEntryPoint {
    return fn handler() callconv(.Naked) void {
        if (comptime !pushesErrorCode(interrupt_number)) {
            asm volatile ("push $0");
        }

        asm volatile (
            \\push %[interrupt_number]
            \\jmp interruptHandlerCommon
            :
            : [interrupt_number] "im" (interrupt_number),
        );
    };
}
```

Then, `interruptHandlerCommon()`, also implemented in assembly, pushes the registers to the stack, calls `interruptHandler()`, and then restores registers and returns from the interrupt.

```
export fn interruptHandlerCommon() callconv(.Naked) void {
    asm volatile (
        // Push registers in InterruptFrame in reverse order
        \\push %%r15
        ...
        \\push %%rax

        // Call interruptHandler, passing a pointer to the InterruptFrame
        ↪ we
        // just built in the stack as first argument
        \\mov %%rsp, %%rdi
        \\call interruptHandler

        // Restore the registers
        \\pop %%rax
        ...
        \\pop %%r15

        // Skip the error code and the interrupt number
        \\add $16, %%rsp

        // Return from the interrupt
        \\iretq
    );
}
```

The higher-level interrupt handlers expect a pointer to a structure `InterruptFrame`, which is built by the previous lower-level functions. The function `interruptHandler` is the one in charge of dispatching to the corresponding handler.

```
/// The data we'll have in the stack inside every interrupt handler.
pub const InterruptFrame = struct {
    // Registers. Pushed by us, except rsp which is pushed by the CPU and
    // it's below.
    rax, rbx, rcx, rdx, rbp, rsi, rdi: u64,
    r8, r9, r10, r11, r12, r13, r14, r15: u64,

    // Interrupt number, pushed by us.
    interrupt_number: u64,

    // Pushed by the CPU for those interrupts that have error code, and
      ↪ set to
    // zero by us for the rest of them.
    error_code: u64,

    // Pushed by the CPU.
    rip: u64,
    cs: u64,
    rflags: u64,
    rsp: u64,
}

export fn interruptHandler(frame: *InterruptFrame) void {
    handlers[frame.interrupt_number](frame);
}
```

Finally, below we can see an example of interrupt handler, in this case for the page fault exception. We can see it first parses the error code pushed by the CPU to get information about the fault. Then, if the fault occurred in kernel, it calls `handleSafeAccessFault`, from the MemSafe package. Otherwise, it ends up using the hypercall `endRun` to notify the crash to the hypervisor.

```
fn handlePageFault(frame: *InterruptFrame) void {
    const present = (frame.error_code & (1 << 0)) != 0;
    const write = (frame.error_code & (1 << 1)) != 0;
    const user = (frame.error_code & (1 << 2)) != 0;
    const execute = (frame.error_code & (1 << 4)) != 0;
    const fault_addr = x86.rdcr2();

    if (!user and mem.safe.handleSafeAccessFault(frame))
        return;

    // Determine the fault type
    var fault_type: hypercalls.FaultInfo.Type = undefined;
    if (present) {
        if (execute) {
            fault_type = .Exec;
        } else if (write) {
            fault_type = .Write;
        } else fault_type = .Read;
    } else {
        if (execute) {
            fault_type = .OutOfBoundsExec;
        } else if (write) {
            fault_type = .OutOfBoundsWrite;
        } else fault_type = .OutOfBoundsRead;
    }

    // Create the fault and send it to the hypervisor
    const fault = hypercalls.FaultInfo{
        .fault_type = fault_type,
        .fault_addr = fault_addr,
        .kernel = !user,
        .regs = x86.Regs.initFrom(frame.*),
    };

    // This won't return
    hypercalls.endRun(.Crash, &fault);
}
```

Perf

The Perf module is implemented in the `perf.zig` file. It is worth showing the implementation of hardware performance counters (Section 1.4.4). The function `initInstructionCount()` enables two of them by writing to the adequate MSRs. The count mode is chosen according to the build option `instruction_count`.

```
const IA32_FIXED_CTR0_ENABLE = 1 << 32;
const IA32_FIXED_CTR1_ENABLE = 1 << 33;
const IA32_FIXED_CTR2_ENABLE = 1 << 34;

fn initInstructionCount() void {
    const count_mode = switch (build_options.instruction_count) {
        .kernel => CountMode.Kernel,
        .user => CountMode.User,
        .all => CountMode.All,
        .none => return,
    };

    // Set performance counter CTR0 (number of instructions) and CTR1
    // (number of cycles) to count when in given mode
    x86.wrmsr(.FIXED_CTR_CTRL, count_mode | (count_mode << 4));

    // Enable CTR0 and CTR1
    x86.wrmsr(.PERF_GLOBAL_CTRL, IA32_FIXED_CTR0_ENABLE |
       ↪ IA32_FIXED_CTR1_ENABLE);
}
```

Syscalls

The syscalls module is implemented in the file `syscalls.zig`. Similarly to interrupts, syscall handler entry points saves registers and jumps to `handleSyscall`, which ends up calling the syscall handler of the current process.

```zig
export fn handleSyscall(
    arg0: usize,
    ...,
    arg5: usize,
    number: usize,
    regs: *Process.UserRegs,
) usize {
    const sys = std.meta.intToEnum(linux.SYS, number) catch return linux.
        ↪ errno(linux.E.NOSYS);
    const ret = scheduler.current().handleSyscall(sys, arg0, ..., arg5,
        ↪ regs);
    return ret;
}
```

5.3.4 File System

File manager

Files are managed by the file manager module. On initialisation, it performs hypercalls to get the virtual files from the hypervisor. For each file, first it gets the filename and the file size with the `getFileInfo()` hypercall. Then, it allocates a buffer of adequate size, and creates an entry in its `file_contents` field, which is a map from filename to file contents. Finally, it submits the pointers of the length and the buffer to the hypervisor, so it can write to them.

```zig
var file_contents: std.StringHashMap([]u8) = undefined;

pub fn init(allocator: Allocator, num_files: usize) void {
    file_contents = std.StringHashMap([]u8).init(allocator);

    // Temporary buffer for the filename
    var filename_buf: [linux.PATH_MAX]u8 = undefined;

    var i: usize = 0;
    while (i < num_files) : (i += 1) {
        // Get the filename and the file length
        var size: usize = undefined;
        hypercalls.getFileInfo(i, &filename_buf, &size);

        // Calculate filename length and allocate it into a buffer
        const filename_len = std.mem.indexOfScalar(u8, &filename_buf, 0)
        ↪ .?;
        const filename = allocator.dupe(u8, filename_buf[0..filename_len])
        ↪  catch unreachable;

        // Allocate a buffer for the file, insert it into file_contents
        // and submit buf and length pointers to the hypervisor, which
       ↪ will
        // fill the buffer with the file content.
        const buf = allocator.alloc(u8, size) catch unreachable;
        file_contents.put(filename, buf) catch unreachable;
        const length_ptr = &file_contents.getPtr(filename).?.*.len;
        hypercalls.submitFilePointers(i, buf.ptr, length_ptr);
    }
}
```

Apart from that, the file manager module provides other functions for accessing the files. For example, for opening a regular file:

```
pub fn fileContent(filename: []const u8) ?[]u8 {
    return file_contents.get(filename);
}

pub fn open(
    allocator: Allocator,
    filename: []const u8,
    flags: i32,
) OpenError!*fs.FileDescription {
    const file_content = fileContent(filename) orelse {
        log.warn("attempt to open unknown file '{s}'\n", .{filename});
        return OpenError.FileNotFound;
    };
    const file = try fs.FileDescriptionRegular.create(allocator,
       ↪ file_content, flags);
    return &file.desc;
}
```

FileDescription

Open files are represented by the `FileDescription` class, whose definition can be seen below. It is implemented as a virtual class, so subclasses are in charge of implementing methods `stat()`, `read()` and `write()`.

```
pub const FileDescription = struct {
    /// Pointer to file content
    buf: []const u8,

    /// Flags specified when calling open (O_RDONLY, O_RDWR...)
    flags: i32,

    /// Cursor offset
    offset: usize = 0,

    // File operations
    stat: *const fn (self: *FileDescription, stat_ptr: UserPtr(*linux.Stat
       ↪ )) mem.safe.Error!void,
    read: *const fn (self: *FileDescription, buf: UserSlice([]u8))
       ↪ ReadError!usize,
    write: *const fn (self: *FileDescription, buf: UserSlice([]const u8))
       ↪ mem.safe.Error!usize,

    /// Reference counter
    ref: RefCounter,

    is_socket: bool = false,

    const ReadError = mem.safe.Error || error{NotConnected};
    const RefCounter = utils.RefCounter(u16, FileDescription);
    const O_ACCMODE = 3;

    pub fn isReadable(self: *const FileDescription) bool {
        const access_mode = self.flags & O_ACCMODE;
        return (access_mode == linux.O.RDONLY) or (access_mode == linux.O.
        ↪ RDWR);
    }

    pub fn isWritable(self: *const FileDescription) bool {
        const access_mode = self.flags & O_ACCMODE;
        return (access_mode == linux.O.WRONLY) or (access_mode == linux.O.
        ↪ RDWR);
    }

    pub fn isOffsetPastEnd(self: *const FileDescription) bool {
        return self.offset >= self.buf.len;
    }

    pub fn size(self: *const FileDescription) usize {
        return self.buf.len;
    }

    pub fn moveOffset(self: *FileDescription, increment: usize) usize {
        // Check if offset is currently past end
        if (self.isOffsetPastEnd())
            return 0;

        // Reduce increment if there is not enough space available
        const ret = if (self.offset + increment < self.buf.len)
            increment
        else
            self.buf.len - self.offset;

        // Update offset
        self.offset += ret;
        return ret;
    }

    pub fn socket(self: *FileDescription) ?*FileDescriptionSocket {
        return if (self.is_socket)
            @fieldParentPtr(FileDescriptionSocket, "desc", self)
        else
            null;
    }
};
```

As an example, the implementation of FileDescriptionRegular can be seen below. It provides the `create()` method, which initialises the underlying FileDescription properly. It is interesting to stand out how the pointer to the buffer of the file is used as inode. That way, to file descriptions that refer to the same file in the file system have the same inode.

```
pub const FileDescriptionRegular = struct {
    desc: FileDescription,

    pub fn create(allocator: Allocator, buf: []const u8, flags: i32)
        ↪ Allocator.Error!*FileDescriptionRegular {
        const ret = try allocator.create(FileDescriptionRegular);
        ret.* = FileDescriptionRegular{
            .desc = FileDescription{
                .buf = buf,
                .flags = flags,
                .stat = stat,
                .read = read,
                .write = write,
                .ref = FileDescription.RefCounter.init(allocator, null),
            },
        };
        return ret;
    }

    fn stat(desc: *FileDescription, stat_ptr: UserPtr(*linux.Stat))!void {
        // Use the pointer to the buffer as inode, as that's unique for
        ↪ each file.
        return statRegular(stat_ptr, desc.buf.len, @ptrToInt(desc.buf.ptr)
        ↪ );
    }

    fn read(desc: *FileDescription, buf: UserSlice([]u8)) !usize {
        assert(desc.isReadable());
        if (desc.isOffsetPastEnd())
            return 0;

        const prev_offset = desc.offset;
        const length_moved = desc.moveOffset(buf.len());
        const src_slice = desc.buf[prev_offset .. prev_offset +
        ↪ length_moved];
        try mem.safe.copyToUser(u8, buf.sliceTo(src_slice.len), src_slice)
        ↪ ;
        return length_moved;
    }

    fn write(desc: *FileDescription, buf: UserSlice([]const u8)) !usize {
        // not allowed
    }
};
```

5.3.5 Processes

Processes implementation is in the folder `./src/process`. It includes the Process class in `Process.zig`, the FileDescriptorTable in `FileDescriptorTable.zig`, syscall handlers in the `syscall` folder, and utilities to set up user execution in `user.zig`.

The different fields of the Process class have already been explained in Section 4.2.5, so it is not worth detailing. Instead, in order to illustrate interactions between processes and the file manager, the syscall handlers of `open()` and `read()` are shown below. First, the `open()` syscall handler copies the `pathname` from user memory by using the `copyStringFromUser` function from the MemSafe module. Then, it requests the file manager to open said fail, which returns a FileDescription object that is inserted into the file descriptor table of the process. Finally, the file descriptor is returned to the user.

```
fn sys_open(
    self: *Process,
    pathname_ptr: UserCString,
    flags: i32,
    mode: linux.mode_t,
) !linux.fd_t {
    // Get the pathname
    const pathname = try mem.safe.copyStringFromUser(self.allocator,
        ↪ pathname_ptr);
    defer self.allocator.free(pathname);

    // Open file
    const file = try fs.file_manager.open(self.allocator, pathname, flags)
        ↪ ;
    errdefer file.ref.unref();

    // Insert it in our file descriptor table
    const fd = self.availableFd() orelse return error.NoFdAvailable;
    try self.files.table.put(fd, file);

    // Set file descriptor flags
    if (flags & linux.O.CLOEXEC != 0)
        self.files.setCloexec(fd);

    return fd;
}
```

Then, the `read()` syscall handler simply gets the file description associated with the given file descriptor, and calls its function pointer `read()`. This will call the appropiate handler depending on the exact type of the file description (FileDescriptionRegular, FileDescriptionStdin, etc).

```
fn sys_read(self: *Process, fd: linux.fd_t, buf: UserSlice([]u8)) !usize {
    const file = self.files.table.get(fd) orelse return error.BadFD;
    if (!file.isReadable())
        return error.BadFD;
    return file.read(file, buf);
}
```

5.4 Markov model

The Markov model is implemented in the file `markov.py` in the root of the repository. It is in charge of reading execution traces, calculating the Markov chain describing those traces, solving the system of equations described in the design (Section 4.3.5), and plotting the associated graph.

5.4.1 Reading traces

Traces have the following format. Each line corresponds to a state, which can be a basic block or a syscall. It is divided in two fields, the first one being the name and the second one the time it took to execute, measured in cycles or instructions. The function `read_traces()` shown below is in charge of reading and parsing the traces. It automatically detects the tracing type by checking if there is a '+' character in the name, which when tracing user indicates the offset inside a function. It also warns when traces have different last state, which may be due to the run crashing, time-outing, or being incomplete (the program was interrupted while writing it to disk). Finally, it requires every trace to start at the same state. The result is a list of traces. Each trace is a list of 2-tuples, where the first component is the name and second one the time it took to execute.

```python
def read_traces(dir_path):
    dir = Path(dir_path)
    tracing_type = None
    first_state = None
    first_state_filename = None
    last_state = None
    last_state_filename = None
    traces = []
    for filename in dir.iterdir():
        with open(filename) as f:
            trace = f.readlines()
            if not trace:
                continue

            if tracing_type == None:
                if "+" in trace[0].split()[1]:
                    tracing_type = TracingType.User
                else:
                    tracing_type = TracingType.Kernel

            trace = [(" ".join(line.split()[:-1]), int(line.split()[-1]))
        ↪ for line in trace]

            if not last_state:
                last_state = trace[-1][0]
                last_state_filename = filename
            if trace[-1][0] != last_state:
                print(f"Warning: trace {filename} has {trace[-1][0]}" +
                    f"as last state, while trace {last_state_filename}" +
                    f" ends with {last_state}.")

            if not first_state:
                first_state = trace[0][0]
                first_state_filename = filename
            if trace[0][0] != first_state:
                print(f"Not every trace starts with the same state:" +
                    f"file {first_state_filename} starts with " +
                    f"{first_state}, file '{filename}' starts with " +
                    f"{trace[0][0]}. Aborting.")
                exit()

            traces.append(trace)
    return traces
```

5.4.2 States weight

After reading the traces, the program calculates `states_avg_instructions`, which is a map from state name to average time it took to execute said state. That is, it contains the weight W_i associated to each state $i \in \mathcal{S}$, as described in Section 4.3.4.

```python
print("Reading traces")
traces = read_traces("./traces")
print(f"Read {len(traces)} traces")

states_instructions = defaultdict(list)
for trace in traces:
    for line in trace:
        states_instructions[line[0]].append(line[1])
states_avg_instructions = {
    state: sum(instructions)/len(instructions) for state, instructions in
      ↪ states_instructions.items()
}
```

5.4.3 Markov chain

Then, the transition matrix of the Markov chain is calculated as described in Section 4.3.2.

```python
successors = {state:[] for state in states_total_instructions}
for trace in traces:
    trace = [line[0] for line in trace] # get only names
    for i in range(1, len(trace)):
        successors[trace[i-1]].append(trace[i])
    successors[trace[-1]].append(None)

successors_probs = {state: dict() for state in successors.keys()}
for state, succs in successors.items():
    for succ in set(succs):
        successors_probs[state][succ] = succs.count(succ)/len(succs)
```

5.4.4 Plotting graph

That information is enough to plot the graph associated to the Markov chain. To do so, the program uses the `networkx` module. Every node is added with a color indicating its associated weight, which represents the time it takes to execute in average. This color can be calculated using linear or logarithmic scale, as shown below. Furthermore, edges are added with a label indicating the probability of the transition between the states. Finally, the graph is exported as a PNG image.

```python
import networkx as nx

g = nx.DiGraph()

max_avg_instructions = max(states_avg_instructions.values())
for state, avg_instructions in states_avg_instructions.items():
    # option 1: linear
    red_intensity = avg_instructions/max_avg_instructions
    # option 2: logarithmic
    # if avg_instructions == 0: avg_instructions = 1
    # red_intensity = math.log(avg_instructions)/math.log(
      ↪ max_avg_instructions)
    red_intensity = int(red_intensity*0xff)
    g.add_node(state, color=f"#{red_intensity:02x}0000", penwidth=4)

for state, succs_probs in successors_probs.items():
    for succ, prob in succs_probs.items():
        if succ:
            g.add_edge(state, succ, label=f"{prob:.8f}")

a = nx.nx_agraph.to_agraph(g)
a.layout("dot")
a.draw("output.png")
```

An example of a plotted graph can be seen in Figure 4.9.

5.4.5 Average execution time

The program also solves the system of equations described in Section 4.3.5 to calculate the time the program takes to run in average according to the traces. To do so, it uses the package `z3`, which is a satisfiability solver. The script defines the variable `avg_instr_from_states`, which is the vector of unknowns $a = (a_i)_{i\in\mathcal{S}}$ defined in Section 4.3.5. Then, it adds the equations to the `z3` solver:

$$a_i = W_i + \sum_{j\in\mathcal{S}} p_{ij}a_j$$

where a_i is the time it takes to run from the state i, W_i is the average execution time of the state i, and p_{ij} is the probability to transition from state i to j. Finally, the system of equations is solved, and the value of the unknown a_0 is returned, which is the average time to run the program.

```
def avg_instructions():
    import z3
    avg_instr_from_states = {state: z3.Real(f"avg_instr_from_{state}") for
    ↪  state in states_instructions.keys()}
    s = z3.Solver()

    for state, succs_probs in successors_probs.items():
        val = sum([avg_instr_from_states[succ]*prob for succ, prob in
        ↪ succs_probs.items() if succ])
        eq = avg_instr_from_states[state] == val + states_avg_instructions
        ↪ [state]
        s.add(eq)

    assert s.check() == z3.sat
    m = s.model()
    first_state = traces[0][0][0]
    result = m[avg_instr_from_states[first_state]]
    return result.numerator().as_long() / result.denominator().as_long()
```

Finally, the program calculates the average time the program takes to run in two ways. First, by calculating the average time each trace takes to run, in `result_real`. Second, by using the Markov model with the method `avg_instructions()` described above, in `result_calculated`. Then, it prints the results and checks both results match, validating the correctness of the model.

```
instr_count = [sum([line[1] for line in trace]) for trace in traces]
result_real = sum(instr_count)/len(instr_count)
result_calculated = avg_instructions()
```

```
print("calculated:", result_calculated)
print("real:", result_real)
assert result_calculated == result_real
```

Chapter 6

Evaluation and tests

6.1 Testing

This section aims to show how the emulation capabilities of KVM-FUZZ were tested in order to guarantee its correct functioning. Below are detailed the different types of tests.

6.1.1 Hypervisor tests

These tests make use of Catch2 [38], a C++ test framework for unit tests. They use the hypervisor to create virtual machines and run test programs in order to check the correctness of the different emulation features that KVM-FUZZ provides. There are two test programs, which are explained below.

Files

The first one is a C application that simply opens a file and reads its content into a buffer. Then, it passes said buffer to a function called `test_me()`, which in a real fuzzing scenario would typically parse the buffer, but in this case does nothing. Its source code can be seen in Listing 6.1.

```c
#include <fcntl.h>
#include <unistd.h>

void test_me(char* buf) {}

int main() {
    int fd = open("./tests/input_hello_world", O_RDONLY);

    char buf[6];
    read(fd, buf, sizeof(buf)-1);
    buf[sizeof(buf)-1] = 0;

    test_me(buf);

    close(fd);
}
```

Listing 6.1: Test program `files`, which is run inside the hypervisor

The purpose of the test is to verify the basic functioning of the hypervisor, which includes ELF loading and execution, symbol resolution, breakpoints, access to the VM memory and registers, and virtual files. Its source code can be seen in Listing 6.2.

```
TEST_CASE("files") {
    Vm vm(
        8*1024*1024,                // amount of memory
        "zig-out/bin/kernel",       // kernel path
        "zig-out/bin/test_files",   // target program path
        {}
    );
    vm.read_and_set_shared_file("./tests/input_hello_world");

    vaddr_t addr = vm.elf().resolve_symbol("test_me");
    REQUIRE(addr != 0);

    vm.run_until(addr, stats);

    char buf[6];
    vm.mmu().read_mem(buf, vm.regs().rdi, sizeof(buf));
    REQUIRE(strcmp(buf, "hello") == 0);

    std::string s = vm.mmu().read_string(vm.regs().rdi);
    REQUIRE(s == "hello");
}
```

Listing 6.2: Hypervisor test `files`

First, the test creates a VM, loading the kernel and the test program. Next, it loads the file `"./tests/input_hello_world"` as a virtual file for the VM, so it can be accessed by the test program. After that, it resolves the address of the function `test_me()`, checks the operation succeeded, and runs the virtual machine until that point, which internally employs a temporary breakpoint. At that execution point, the test program has already read the file contents into the buffer, whose address is in register `rdi` (first argument for function `test_me()`). The test then reads the buffer from the VM memory into a local buffer, and makes sure its contents are those of the virtual file. This checks the correct functioning of register and memory reading and paging. Finally, it performs the same operation using a higher level API, which returns a `std::string`.

Hooks

The second test program is much simpler. It is an assembly program that just sets a register to 0 and then increments it twice. It can be seen in Listing 6.3.

```
0000000000201120 <_start>:
  201120:       48 31 c0                xor    rax,rax
  201123:       48 ff c0                inc    rax
  201126:       48 ff c0                inc    rax
  201129:       90                      nop
  20112a:       90                      nop
  20112b:       90                      nop
  20112c:       90                      nop
```

Listing 6.3: Test program `hooks`, which is run inside the hypervisor

The purpose of the tests that use this test program is to verify features such as hooks, breakpoints and single steps, which are hard to get right when mixed together. One example of test can be seen in Listing 6.4.

```
TEST_CASE("run_until + hook") {
    Vm vm(
        8*1024*1024,              // amount of memory
        "zig-out/bin/kernel",     // kernel path
        "zig-out/bin/test_hooks", // target program path
        {}
    );
    vm.set_hook(0x201129,  [](Vm& vm) {
        vm.regs().rax = 1234;
    });

    vm.run_until(0x201129, stats);
    REQUIRE(vm.regs().rax == 2);

    vm.run_until(0x20112c, stats);
    REQUIRE(vm.regs().rax == 1234);
}
```

Listing 6.4: Hypervisor test `hooks`

After creating the VM, it sets a hook at address `0x201124`. The `Vm::set_hook()` function receives two parameters: the address of the hook, and the callback that will be executed when VM executes that address. In this case, a lambda function is used as callback, which sets the `rax` register of the VM to the value 1234. Next, the VM is run until address `0x201124`. Since that is after the two `inc rax` instructions but just before the hook, the value of `rax` is tested to be 2. Finally, the VM is run until the end, and the value is now checked to be 1234, verifying that the hook callback has indeed been executed.

The other tests are similar to the previous one, but with little tricky variations such as placing two hooks, using single step, or changing the program counter to alter the execution flow, and are not worth detailing. In total, hypervisor tests consist of 30 assertions in 10 test cases.

6.1.2 Syscalls tests

Syscalls tests are the main ones used to test the kernel component. As hypervisor tests, syscalls tests also make use of Catch2. However, while the former are run on the host machine to check the correct functioning of virtual machines and emulation altogether, the later are run inside the VM to test the kernel. For each syscall there are one or more tests that check its correctness. Importantly, these tests also pass when run under Linux, thus making sure the behaviour of the developed kernel matches Linux. Next, some tests are detailed.

```
TEST_CASE("dup") {
    int fd1 = open("./tests/input_hello_world", O_RDONLY);
    REQUIRE(fd1 > 0);

    int fd2 = dup(fd1);
    REQUIRE(fd2 > 0);
    REQUIRE(fd2 != fd1);

    REQUIRE(read_and_check_first_five_bytes(fd1) == 0);
    REQUIRE(read_and_check_next_seven_bytes(fd2) == 0);

    REQUIRE(lseek(fd1, 0, SEEK_SET) == 0);
    REQUIRE(read_and_check_first_five_bytes(fd2) == 0);

    REQUIRE(close(fd2) == 0);

    REQUIRE(read_and_check_next_seven_bytes(fd1) == 0);

    REQUIRE(close(fd1) == 0);
}
```

Listing 6.5: Syscall test `dup`

Listing 6.5 describes a test for the `dup()` syscall, which duplicates a given file descriptor. First, it opens a file and duplicates it, checking return values are correct and obtaining two file descriptors `fd1` and `fd2`. Since both file descriptors correspond to the same underlying file description, operations on the first one should also affect the second one. To check this, the test reads 5 bytes from `fd1`, and checks they match the known content of the file. After that, it does the same for the next 7 bytes but this time reading from `fd2`, thus verifying it was also affected by the first read. Then, it resets the position of `fd1` to the beginning, and checks it also affected `fd2` by reading the first 5 bytes of the file from it. Finally, it closes `fd2` and tests it did not affect `fd1`.

This test is an example of checking the correct behaviour of a syscall, in this case `dup()`: file descriptors are a handle to a file description, and duplicating them does not duplicate the underlying file description. This test also relies on other syscalls such as `open()`, `read()` and `lseek()` to be implemented correctly, which is why it is necessary for each syscall to have its own tests.

Another example of syscall test can be seen in Listing 6.6, which tests `mmap()`. Since the `mmap()` syscall is complex and has several different options, it has many different tests. This one checks the correct behaviour of file mapping. It opens a file, maps it with `mmap()`, and checks the contents are correct. Finally, it unmaps the file from memory and closes it.

```
TEST_CASE("mmap file") {
    int fd = open("./tests/input_hello_world", O_RDONLY);
    REQUIRE(fd != -1);
    char* p = (char*)mmap(nullptr, PAGE_SIZE, PROT_READ, MAP_PRIVATE, fd,
        ↪ 0);
    REQUIRE(p != MAP_FAILED);
    REQUIRE(strcmp(p, "hello world1\nhello world2") == 0);
    REQUIRE(munmap(p, PAGE_SIZE) == 0);
    REQUIRE(close(fd) == 0);
}
```

Listing 6.6: Syscall test `mmap file`

The idea of having tests that called and checked syscalls was inspired by the Linux Test Project [39]. The idea was extended so tests were run both under Linux and under KVM-FUZZ, thus making sure KVM-FUZZ correctly emulated Linux syscalls. In total, when run under KVM-FUZZ, syscalls tests consist of 2416 assertions in 53 test cases.

6.2 Experiments

This section intends to show different experiments that have been carried out in order to evaluate KVM-FUZZ and compare them to other state of the art fuzzers, such as AFL++ and Nyx (Section 1.5). The experiments of Section 6.2 and Section 6.2.2 have been performed on a machine with an Intel(R) Core(TM) i7-6700K processor with 8 logic cores and 32GB of RAM. On the other hand, the experiments of Section 6.2.1 have been carried out on a machine with an Intel(R) Xeon(R) Silver 4314 processor with 32 physical and 64 logical cores.

Number of instructions executed

This experiment intends to illustrate the performance advantage of having your own kernel with the ability to emulate system calls. Two different programs have been considered: `readelf`, which is a program belonging to the `binutils` collection of tools, and `tiff2rgba`, a program that uses the library `libtiff` for image processing. Both of them are easy to fuzz file parsers that are used across a widely range of applications. The goal is to measure how significant is the performance improvement when they are run under the kernel of KVM-FUZZ, in contrast to Linux. Different options and input files have being used in order to see if there is a difference between long runs (`readelf -a`, and `tiff2rgba` with a big input file) and shorter runs (`readelf -l` and `tiff2rgba` with a small input file).

In each execution we have measured the number of instructions executed from `main()` to `exit()`, both in user and kernel space. In order to perform the measurements on KVM-FUZZ we have made use of performance counters, which are hardware registers that allow counting CPU events (Section 1.4.4). On Linux, we have made use of `perf`, a tool that allows accessing those same performance counters.

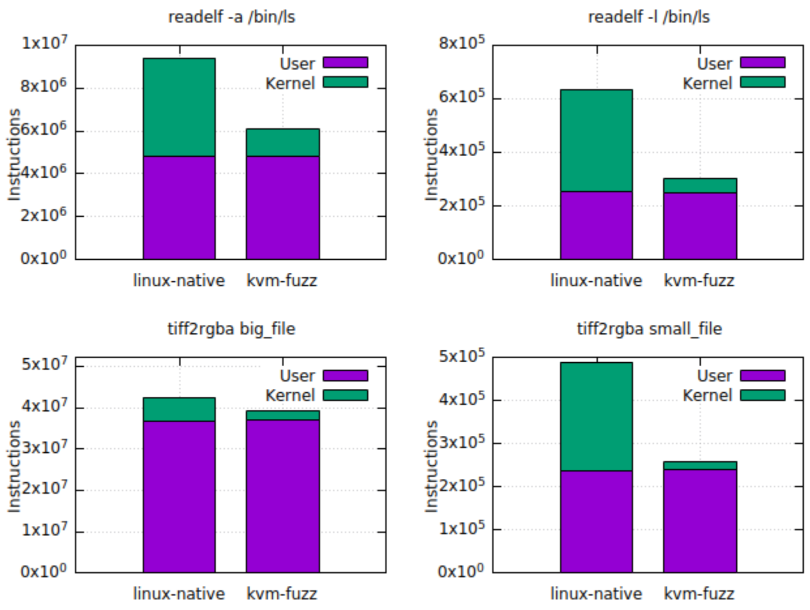


Figure 6.1: Comparison of the number of instructions executed in different programs when run under Linux and under the kernel developed for KVM-FUZZ.

Results can be seen in Figure 6.1. As expected, the number of instructions executed by the application in userspace is independent of the kernel, assuming the kernel behaves correctly. However, we can see a significant reduction in the number of instructions executed in kernel space when the application is run under the developed kernel compared to when it is run under Linux.

On average, in each execution KVM-FUZZ executed 78% fewer instructions in kernel space compared to Linux, and 37% fewer total instructions. The difference in total instructions is less significant in long runs that spend most of the time in user space, as is the case with `readelf -a` or `tiff2rgba` with a large input file. However, in shorter executions that may spend a greater portion of the time in the kernel, such a reduction in the number of instructions in the kernel is reflected in a greater than 50% reduction in the number of total instructions, as occurs in the runs of `readelf -l` and `tiff2rgba` with a small file.

We conclude that the kernel developed for KVM-FUZZ, being simpler, executes fewer instructions than Linux to handle the system calls of an user application, causing a significant performance improvement.

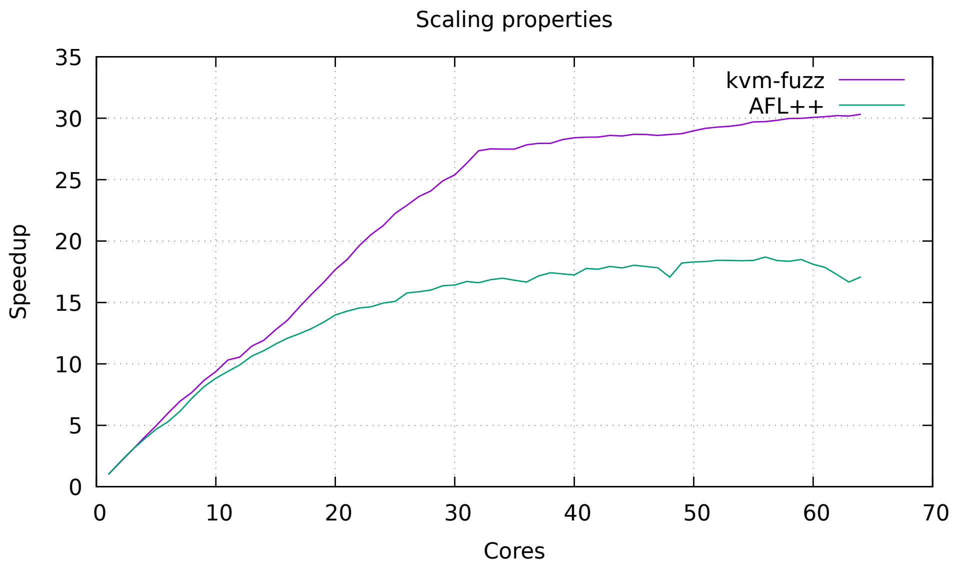


Figure 6.2: Comparison of speedup in execution speed depending on the number of cores with KVM-FUZZ and AFL++.

6.2.1 Scaling

We wish to measure the scaling properties of a fuzzer like KVM-FUZZ, in which each instance of the fuzzer uses an isolated virtual machine, versus a traditional fuzzer like AFL++, in which each instance makes use of `fork()` to run the target program under the host kernel. To do this, we have measured the speedup (defined as parallel execution speed divided by sequential execution speed) of both fuzzers with a test program as a function of the number of cores.

For multi-core execution, KVM-FUZZ has a command line option to enable multi-threading with a certain number of threads, pinning each thread to a core. On the other hand, AFL++ does not have such option or any other automatic method for this, so it requires launching multiple instances of the fuzzer separately and using another program to obtain the statistics.

Results can be seen in Figure 6.2. We can see that KVM-FUZZ shows a linear scaling up to 32 cores. Starting at 32 cores, the speedup gain decreases, because it starts using logical cores with Hyper-Threading instead of physical ones. However, it keeps increasing. This linear scaling allows for maximum utilisation of the CPU resources, as there is almost no contention between cores that could limit performance.

On the other hand, AFL++ performance does not scale linearly. We can see that as the number of cores increases, the speedup increase decreases, even before reaching the threshold where Hyper-Threading starts being used

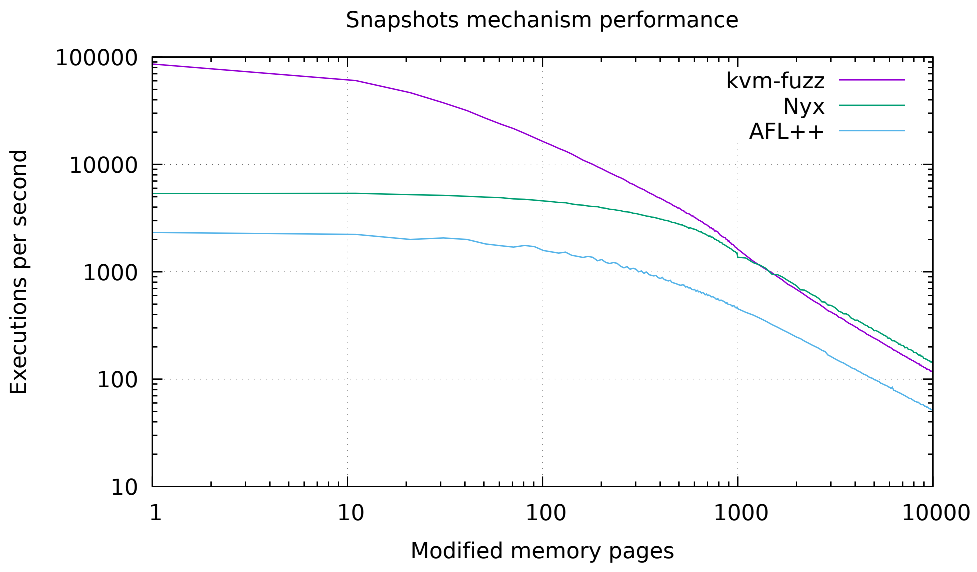


Figure 6.3: Comparison of the execution speed of a program that modifies N memory pages with KVM-FUZZ, NYX and AFL++.

[40]. This entails a significant waste of processor resources when there are a considerable number of cores available, as it is the case in the experiment or in advanced fuzzing campaigns.

6.2.2 Snapshot restoration

In order to measure the performance of the snapshot restoration mechanism, we have used a small test program that allocates a certain number of memory pages and writes to them. The goal is to measure the speed in which a snapshot can be restored depending on the amount of memory that the target program modifies. We have evaluated KVM-FUZZ, NYX and AFL++.

The setup is as follows. Both KVM-FUZZ and NYX take a snapshot after the program has allocated the memory, and run just until after writing to it. AFL++, on the other hand, makes use of the deferred fork server (Section 1.5.1) at the same start point. Although AFL++'s fork server does not strictly use a snapshot mechanism, Linux does make use of copy on write in `fork()`. Therefore, memory pages of the child process are only duplicated when modified, resembling the differential memory restoration of snapshots. It was decided to use the fork server instead of the snapshot mechanism that AFL++ incorporates because its implementation restores all memory pages, regardless of whether they have been modified or not (Section 1.5.1).

The results are illustrated in Figure 6.3. As can be seen, KVM-FUZZ has a performance improvement of up to an order of magnitude for programs that modify few memory pages. The difference from Nyx is due to kernel memory usage. According to the authors of Nyx, "we observe that when the target only dirties ten pages, we reload almost a 100 pages in the kernel" [9]. Besides, Nyx also restores the state of the devices emulated by QEMU. On the other hand, the kvm-fuzz kernel, being simpler, uses less memory, leading to these differences when the target program modifies few pages.

However, this difference in kernel memory usage becomes less significant when the target program modifies a larger number of memory pages. In those cases, performance is limited by the memory hardware speed, since a large number of pages have to be copied to restore the state of the target program. Therefore, under these conditions the snapshots mechanism of kvm-fuzz achieves a performance similar to that of Nyx.

For its part, AFL++ follows a similar curve to Nyx, but with significantly worse performance. This is mainly because the fork server must wait for the child process to terminate on each execution. Meanwhile, the other two fuzzers end execution just after the program has modified the indicated pages, avoiding the performance loss that supposes waiting for the target program to run exit routines.

Chapter 7

Conclusions

To sum up, this project has made the following contributions:

1. Detailed and extensive study of modern binary fuzzers and the techniques they use. This included reading and synthesizing different papers and documentations.

2. Analysis of the performance problems that has the current approach to fuzzing. This included carrying out a number of experiments and employing profiling techniques to detect bottlenecks and fundamental design issues.

3. Design and implementation of KVM-FUZZ, a coverage-guided, binary-only emulation and fuzzing tool that makes use of lightweight virtual machines and different hardware acceleration mechanisms to overcome said issues. This involved developing our own hypervisor and kernel from scratch to emulate the target program with high speed and scalability. This tool can be used to find bugs and vulnerabilities in user applications.

4. Design and implementation of a Markov model that describes the execution behaviour of the target program, both in user and kernel space, and helps to analyze its runtime behaviour. By also modelling the execution time, it can serve as a profiling tool for both the kernel and the target program.

Bibliography

[1] S. Nagy, A. Nguyen-tuong, J. Hiser, J. Davidson, and M. Hicks, "Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing," 09 2022.

[2] Y. Song, H. Wang, and T. Soyata, *Hardware and Software Aspects of VM-Based Mobile-Cloud Offloading*, 08 2015, pp. 247–271.

[3] M. Stone, "The more you know, the more you know you don't know: A year in review of 0-days used in-the-wild in 2021," https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html, Accessed: March 25, 2023.

[4] CVE Details, "Vulnerabilidades de corrupción de memoria publicadas en 2022," https://www.cvedetails.com/vulnerability-list/year-2022/opmemc-1/memory-corruption.html, Accessed: March 25, 2023.

[5] Google, "OSS-Fuzz: continuous fuzzing for open source software," https://github.com/google/oss-fuzz, Accessed: March 25, 2023.

[6] R. S. Kostya Serebryany, Souheil Moghnie and A. Sudhakar, "Focus on Fuzzing: A Closer Look at Coverage-Guided Fuzzing," https://safecode.org/blog/focus-on-fuzzing-a-closer-look-at-coverage-guided-fuzzing/, Accessed: March 25, 2023.

[7] h0mbre, "Fuzzing like a caveman 4: Snapshot/code coverage fuzzer!" https://h0mbre.github.io/Fuzzing-Like-A-Caveman-4/, Accessed: March 25, 2023.

[8] A. Fioraldi and M. Heuse, "AFL++ Snapshot LKM," https://github.com/AFLplusplus/AFL-Snapshot-LKM, Accessed: March 25, 2023.

[9] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security*

21), 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[10] A. '0vercl0k' Souchet, "what the fuzz," https://github.com/0vercl0k/wtf, Accessed: March 25, 2023.

[11] "QEMU device emulation," https://www.qemu.org/docs/master/system/device-emulation.html, Accessed: March 25, 2023.

[12] "QEMU USB emulation," https://www.qemu.org/docs/master/system/devices/usb.html, Accessed: March 25, 2023.

[13] "KVM API Documentation," https://www.kernel.org/doc/html/latest/virt/kvm/api.html, Accessed: March 25, 2023.

[14] "OSDev Introduction," https://wiki.osdev.org/Introduction, Accessed: March 25, 2023.

[15] "System V ABI," https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf, Accessed: March 25, 2023.

[16] "Model Specific Register - Wikipedia," https://en.wikipedia.org/wiki/Model-specific_register, Accessed: March 25, 2023.

[17] "OSDev APIC," https://wiki.osdev.org/Apic, Accessed: March 25, 2023.

[18] "OSDev Global Descriptor Table," https://wiki.osdev.org/Global_Descriptor_Table, Accessed: March 25, 2023.

[19] "LWN Linux Five-level Page Tables," https://lwn.net/Articles/717293/, Accessed: March 25, 2023.

[20] "Markov Chains - University of Auckland," https://www.stat.auckland.ac.nz/~fewster/325/notes/ch8.pdf, Accessed: March 25, 2023.

[21] M. Aldridge, "Introduction to Markov Processes - University of Leeds," https://mpaldridge.github.io/math2750/S08-hitting-times.html, Accessed: March 25, 2023.

[22] J. Norris, "Markov Chains - University of Cambridge," https://www.statslab.cam.ac.uk/~james/Markov/s13.pdf, Accessed: March 25, 2023.

[23] "Graph (discrete mathematics) - Wikipedia," https://en.wikipedia.org/wiki/Graph_(discrete_mathematics), Accessed: March 25, 2023.

[24] "Strongly connected components - Wikipedia," https://en.wikipedia.org/wiki/Strongly_connected_component, Accessed: March 25, 2023.

[25] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. PP, pp. 1–1, 06 2018.

[26] B. Falk, "Mesos," https://github.com/gamozolabs/mesos, Accessed: March 25, 2023.

[27] "DynamoRIO," https://github.com/DynamoRIO/dynamorio, Accessed: March 25, 2023.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[29] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21).* USENIX Association, Aug. 2021, pp. 1683–1700. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

[30] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1497–1511.

[31] M. Heuse, "AFL-Dyninst," https://github.com/vanhauser-thc/afl-dyninst, Accessed: March 25, 2023.

[32] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020.

[33] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *2020 IEEE Secure Development Conference (SecDev)*, 2020, pp. 23–30.

[34] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2313–2328. [Online]. Available: https://doi.org/10.1145/3133956.3134046

[35] S. Schumilo and C. Aschermann, "libxdc," https://github.com/AFLplusplus/AFL-Snapshot-LKM, Accessed: March 25, 2023.

[36] Zig Software Foundation, "Zig," https://ziglang.org/, Accessed: March 25, 2023.

[37] J. Pommnitz, "Linux kernel exception tables," https://www.kernel.org/doc/html/next/x86/exception-tables.html, Accessed: March 25, 2023.

[38] "Catch2," https://github.com/catchorg/Catch2, Accessed: March 25, 2023.

[39] "Linux test project," https://github.com/linux-test-project/ltp, Accessed: March 25, 2023.

[40] B. Falk, "Tweet sobre el rendimiento de fork," https://twitter.com/gamozolabs/status/1265292822292230144, Accessed: March 25, 2023.

Acronyms

ABI	Application Binary Interface.
APIC	Advanced Programmable Interrupt Controller.
ASan	Address Sanitizer.
ELF	Executable and Linking Format.
GDT	Global Descriptor Table.
IDT	Interrupt Descriptor Table.
Intel PT	Intel Processor Trace.
IO	Input-Output.
ISR	Interrupt Service Routine.
IST	Interrupt Stack Table.
JIT	Just In Time.
JNIC	Jornadas Nacionales de Investigación en Ciberseguridad.
KVM	Kernel Virtual Machine.
MMU	Memory Management Unit.
MSR	Model Specific Register.
OOM	Out Of Memory.
PIE	Position Independent Executable.
PIT	Programmable Interval Timer.
PML	Page Modification Logging.

PMM Physical Memory Manager.

PTBR Page Table Base Register.

PTE Page Table Entry.

STL Standard Template Library.

TCG Tiny Code Generation.

TLB Translation Lookaside Buffer.

TOCTOU Time-Of-Check to Time-Of-Use.

TSS Task State Segment.

vCPU Virtual Central Processing Unit.

VM Virtual Machine.

VMCS Virtual Machine Control Structure.

VMM Virtual Memory Manager.