Fachhochschule Münster

Fachbereich Elektrotechnik und Informatik

Masterarbeit

zur Erlangung

des akademischen Grades

Master of Science (M.Sc.)

im Studiengang Informatik

Design and Implementation of a Hardware Accelerated, General Purpose and Coverage-Guided Operating System Fuzzer

Erstprüfer: Prof. Dr.-Ing. Sebastian Schinzel

Zweitprüfer: Hendrik Schwartke M.Sc.

vorgelegt am 13. Dezember 2016

von Sergej Schumilo

Matrikel-Nr. 700075

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und ist noch nicht veröffentlicht worden. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Steinfurt, 13. Dezember 2016, _____

Unterschrift

Abstract

Memory corruption bugs are common vulnerabilities and can be encountered in user mode applications as well as in kernel code. Past experience has shown, that attackers were more focused on the exploitation of user mode vulnerabilities, since those bugs are much easier and more reliable to exploit. With the advent of more enforcing security mechanism provided by the kernel, the exploitation has become much harder. Since comprehensive kernel self-protecting is hard to achieve, attackers have started to focus more on kernel vulnerabilities, instead.
Unfortunately, kernel bugs are typically much harder to spot, especially in closed source operating systems. This thesis describe the design and implementation of a novel approach to find kernel vulnerabilities in an automated fashion using latest Intel processor features and fuzzing methods, which has proven as highly efficient in the field of fuzzing user mode applications. Moreover, this approach is potentially able to fuzz even closed source operating systems. The first implementation outperforms other kernel fuzzers and has found several security vulnerabilities in the Linux kernel during development.

II

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Several vulnerability-classes such as memory corruptions, race conditional memory accesses, and use-after-free vulnerabilities are known threats for programs running in user mode as well as for the Operating System (OS) core itself. Past experience has shown that attackers are typically focusing on user mode applications. This is because vulnerabilities in user mode programs are notoriously easier and more reliable to exploit. However, with the appearance of different kinds of exploit defense mechanisms, especially in user mode, it has become much harder to exploit known vulnerabilities nowadays. Due to those advanced defense mechanisms in user mode, the kernel has become even more appealing to an attacker, since most kernel defense mechanisms are not mandatory in use. This is due to more complex implementations, which may affect the system performance and therefore are deactivated by default. Furthermore, some of them are not part of the official mainline code base or even require latest CPU extension support (e.g. SMAP / SMEP on x86-64). Nevertheless, with the compromise of the OS, an attacker typically gains full access to the system resources (except for virtualized systems). Kernel-level vulnerabilities are usually used for privilege escalation exploits or kernel-based rootkits to gain persistence.

Since such vulnerabilities are in general difficult to spot, recent fuzzing techniques have proven as promising to discover bugs in an automated fashion. Especially the novel fuzzer AFL (American Fuzzy Lop) has shown an effective way of finding memory corruption type bugs coverage-guided. Instead of applying bruteforce fuzzing on the targeted application, AFL measures feedback from within the application to rate the progress. A simplified genetic algorithm is based on this progress rate and used to synthesize new inputs, which may trigger new path executions. Unfortunately, AFL is limited to apply fuzz testing on userspace applications only and suffers lack of kernel support.

1.1 Contribution

This thesis proposes an approach to fuzz x86-64 based operating systems without any recompilation needed by utilizing latest Intel CPU extensions. This includes the leverage of Intel Processor Trace, a new CPU feature which provides flow information about running code directly by hardware. Furthermore it describes the design and development of a novel kernel fuzzer which is partially inspired by the AFL fuzzing model and uses the above mentioned approach. This fuzzer is able to fuzz any x86-64 operating system, which includes even closed-source code. The only requirement to fuzz an operating system is the ability to load a specific required driver. In comparison to other feedback-driven kernel fuzzers, the proposed approach provides higher efficiency due to hardware-acceleration. In addition, it provides high effectiveness due to the used fuzzing engine inspired by AFL and it does not require any recompilation of targeted supervisor code. Therefore, it is capable of fuzzing closed-source operating systems. To summarize, this thesis contributes with the following:

(I) An approach to extend KVM to become an Intel PT aware Virtual Machine Monitor (VMM). Based on this research an academic prototype called `vmx_pt` was developed. To our best knowledge, this is the first implementation of an Intel PT aware VMM, which allows to trace multiple guest's vCPU without any race conditional side effects.

(II) An appropriate interface for QEMU to use `vmx_pt` capabilities in real world usage. To process the proprietary Intel Process trace format rapidly, an own Intel PT decoder was developed with a focus on efficiency.

(III) A kernel fuzzer, called kAFL (kernel AFL), which is able to fuzz any virtualized OS by utilizing KVM, `vmx_pt`, and QEMU. Due to multiprocessing and Intel Process Trace, kAFL is extremely fast compared to other kernel fuzzers.

It is also intended to publish all of these implementations as GPLv2 licensed software during the next few months.

1.2 Related Work

An unofficial Google project called syzkaller was released by Dmitry Vyukov in October 2015 and is the first publicly available graybox coverage-guided kernel fuzzer

[Vyu]. As of time of writing, up to 211 kernel bugs have been found by using syzkaller*.

Vegard Nossum and Quentin Casanovas demonstrate that most linux file system drivers are vulnerable to feedback-driven fuzzing by using an adapted version of AFL [NC16]. This modified AFL version is based on gluecode to the kernel in form of a driver interface to measure feedback during fuzzing file system drivers of the own kernel and expose this data to the userspace. Nossum and Casanovas presented this modified version of AFL at Vault 2016. Their work was published later this year and is available at Github †. Since their fuzzer runs inside the targeted operating system, an occurred crash then terminates the fuzzing session.

In summer 2016 Jesse Hertz and Tim Newsham released a modified version of AFL called *ProjectTriforce* [HN16b]. Their work is based on a modification of QEMU and utilizes the corresponding emulation backend to measure fuzzing progress by determining the current instruction pointer after execution of a control flow altering instruction. In theory, their fuzzer is able to fuzz any OS emulated in QEMU. In practice, the ProjectTriforce fuzzer is limited to operating systems that are able to boot from read-only file systems, which narrows down the candidates to classic UNIX-like operating systems, such as Linux, FreeBSD, NetBSD, or OpenBSD. Therefore, ProjectTriforce is currently not able to fuzz closed-source operating systems, such as macOS or Windows.

1.3 Structure of the Work

The structure of the thesis is as follows:

(I) The first chapter provides a brief overview of all technologies used during this thesis and is essential in order to comprehend further chapters.

(II) Chapter 2 describes the implementation details and concepts applied to kAFL, the kernel driver for Intel Processor Trace, the decoder engine, and the QEMU extension.

(III) In Chapter 3 an overview of the measured performance, the performance boost in relation to other kernel fuzzers, and other considerable Intel PT related insights is given.

*List of all bugs found by syzkaller (November 2016): https://github.com/google/syzkaller/wiki/Found-Bugs

†https://github.com/oracle/kernel-fuzzing

(IV) Finally, a short conclusion and an overview of future work is given.

Chapter 2

Background

In this chapter the reader is provided with a brief overview of all used concepts and technologies. This background is essential in order to comprehend further chapters of this thesis. Chapter 2.1 describes the basic idea and concepts of AFL, which are also adapted by kAFL. Furthermore, chapter 2.2 introduces briefly the Intel IA-32 virtualization extension to provide virtualization by the hardware itself. In chapter, 2.3 the process of vCPU creation and KVM / userspace communication are explained. Finally, the new Intel Processor Trace feature and the way it might be used in VMX operations is described in chapter 2.4.

2.1 American Fuzzy Lop

This thesis is based on the idea behind the American Fuzzy Lop (AFL) fuzzer. Therefore, this chapter provides insights into technical aspects and implementation details of AFL.

2.1.1 General

Fuzzing is a technique for software regression tests. Originally, the term *fuzzing* was introduced by Barton Miller at the University of Wisconsin in 1988 [MFS90]. Fuzzers are usually used to generate specific semi-valid inputs, which may trigger unforeseen behavior in the targeted application. Such misbehavior can be categorized as a potential bug or even a critical software vulnerability. Therefore, fuzzing has become a valuable approach in the vulnerability and bug finding process of security researchers. In general, fuzzing approaches can be categorized in the following three types:

Blackbox-Fuzzers:

Fuzzers which did not examining the source code or the disassembled binary are called blackbox-fuzzers. Blackbox-fuzzers are not able to make any assumptions of the possible program behavior. Typically, blackbox-fuzzers are developed to fuzz specific interfaces, protocols or parser types.

Whitebox-Fuzzers:

Fuzzers utilizing several information based on the source code of the application are called *whitebox fuzzers*. A whitebox-fuzzer is able to make assumptions of the resulting control-flow. Thus, the synthetization process of new fuzzing inputs can be assisted by source code analysis and control-flow assumptions. This approach was proven to be effective and was a major topic of recent academic research [GLM08; CDE08].

Greybox-Fuzzers:

In contrast to whitebox-fuzzers, greybox-fuzzers do not analyze the associated source code, but the targeted applications disassembly or dynamic runtime information. As such, they fill the gap between white- and blackbox-fuzzers. To gain access to dynamic runtime information, greybox-fuzzers usually require a recompilation of a program using an adapted compiler to insert compile-time instrumentations or must rely on other tracing approaches such as PIN [Luk+05], emulation via QEMU [HN16b] / Bochs [Jur16] or Intel BTS [Swi].

AFL is a modern greybox fuzzer targeting applications userspace. This fuzzer was originally developed by Michael Zalewski. In contrast to other greybox-fuzzers, AFL utilizes compile-time instrumentations in the targeted application to generate coverage metrics and uses an "exceedingly simple but rock-solid instrumentation-guided genetic algorithm"[Zal]. The basic idea is based on multiple mutation-techniques and brute-force. This approach has proven as highly effective and is able to find even more bugs than most whitebox fuzzers.

2.1.2 Algorithm

Although AFL was proven to be highly effective, its theory is quite simple. For illustration purposes, a simplified algorithm representation is given in the following listing (see algorithm 1).

Algorithm 1: Simplified AFL fuzzing loop algorithm

```
1 input_queue ← gen_queue(initial_inputs)
2 while true do
3 |   current_payload ← get_next(input_queue)
4 |   foreach strategies do
5 |   |   new_input, new_path ← MUTATE_STRATEGY(current_payload)
6 |   |   if PATH_IS_NEW(new_path) then
7 |   |   |   input_queue ← input_queue + new_input
```

Initially, AFL loads user-provided inputs into the `input_queue` (Line 1). This input is termed as *seed*. It then starts the fuzzing loop based on the seed (Line 2). At the beginning of the loop, ALF chooses the next *interesting* input based on several information from the *queue*. Afterwards, AFL applies several fuzzing *techniques* on the chosen input, mutates it, and injects it into the targeted application hereafter. Every applied and injected mutation is measured via compile-time instrumentation feedback. If the newly generated input hits a new path or rather a new state transition, the associated input will be appended to the queue for later use.

2.1.3 Fuzzing Techniques

AFL uses six fuzzing techniques to generate new mutation-based inputs, which may result in new state transitions. Any generated input is discarded and not used for the next iteration during the same phase, except for the havoc and splicing phase. Those techniques are also called *strategies* and are as follows:

Bit-Flips / Byte-Flips:

During the first *deterministic* fuzzing strategy stage, AFL applies multiple bit- and byte-flips on the current input data. Therefore, the fuzzer successively bit-flips (bitwise NOT operation) every bit, applies the newly input to the target application and rewinds the modification. This is done for every single bit as well as for all 2- and 4-bit sequences. Later, the same approach is applied for bytes and byte-sequences (2 and 4 bytes).

Arithmetic:

As the second deterministic fuzzing strategy, AFL implements the incrementation and decrementation of 1, 2 and 4 byte values. By default, ALF increments and

decrements every value up to 35 times. To ensure that every mutation is only applied once, AFL uses helper functions to distinguish already applied duplicates, which may have already been generated during this or a prior stage.

Interesting Integers:
As the last deterministic fuzzing stage, AFL replaces bytes and byte-sequences with known values. For instance, interesting values are `MAXINT` values for different sizes and signedness, NULL-bytes and several 2^N variations. To filter already applied mutations, AFL verifies the uniqueness of newly generated input compared to mutation-based inputs of the prior two stages.

Havoc:
In order to include a certain amount of randomness, the Havoc stage is used. During the Havoc stages, a varying amount of different fuzzing approaches are applied on the initial input and reused for several iterations. As the name suggests, this stage injects more variety of unexpected randomness to the current input.

Splicing:
To extend the Havoc stages, AFL eventually switches thereafter to the Splicing stage, if no new path was found during previous stages. The Splicing stage is a combinatorial approach and splices two queued inputs as well as found crash-inputs together at a random pivot. Afterwards, the Havoc approach is applied on the combined input for several iterations.

Dictionary:
To deal with less binary-oriented inputs, a more sophisticated approach is needed. Therefore, AFL uses the additional Dictionary stage. During this stage, AFL injects imported strings in the current input. This stage has proven to be highly effective for certain types of parsers*.

2.1.4 AFL-Bitmap

To determine state transitions, AFL uses a shared memory buffer which is 64 KB in size by default and is mapped into the targeted application as well as into the fuzzer's virtual memory. This buffer, also termed AFL-Bitmap, stores all occurring state transitions as a hashed tuple of two edge identifiers. Those identifiers are inserted by the AFL gcc

*https://lcamtuf.blogspot.de/2015/01/afl-fuzz-making-up-grammar-with.html

or clang compiler wrapper `afl-gcc` or `afl-clang` after any x86-64 conditional and unconditional `jmp` instruction during compile-time and they identify the edge of any basic block (see listing 2.1 and 2.2).

Listing 2.1 Disassembly of an unmodified basic block

```
/* ... */
je <LOCATION>









/* ... */
```

Listing 2.2 Disassembly of a modified basic block

```
/* ... */
je <LOCATION>

/* AFL INSTRUMENTATION */
leaq -(128+24)(%rsp), %rsp
movq %rdx, 0(%rsp)
movq %rcx, 8(%rsp)
movq %rax, 16(%rsp)
movq <COMPILE_TIME_RANDOM>, %rcx
call __afl_maybe_log
movq 16(%rsp), %rax
movq 16(%rsp), %rcx
movq 16(%rsp), %rdx
leaq (128+24)(%rsp), %rsp

/* ... */
```

The following hash function code (listing 2.3) is part of the `__afl_may_log()` function and is executed after each instrumented edge is hit. The hashing is applied on the shared bitmap buffer:

Listing 2.3 AFL Instrumentation

```
1:  cur_location = <COMPILE_TIME_RANDOM>;
2:  shared_mem[cur_location ^ prev_location]++;
3:  prev_location = cur_location >> 1;
```

Due to the selected hash function, AFL is able to distinguish the state transition direction. If, for instance, the `current_location` and `prev_location` is swapped, the hash function will generate a different value, due to the shift operation in line 3. Moreover, AFL is also able to detect loops and other frequently executed code areas owning to the increment operation in line 2. Therefore, any additional loop iteration will thus be detected as a new state transition. To avoid potential path explosions, AFL uses a *buck-*

eting approach. Only if the number of iterations matches any 2^N *bucket* value (except for $N < 0 \vee N > 8$), the state transitions will be considered as new path. The selected 2^N handling is based on a fast implementation of a bit-masking approach for byte comparisons of AFL bitmap values, whereas only one bit in the bit-mask is set at the same time. The result of this approach is that only 8 possible state transitions per loop are recognized compared to $2^8 - 1$ possible paths.

Due to the limited size of the bitmap buffer, hash collisions become possible. This is especially true for larger programs. In such case, the bitmap buffer should be enlarged by the user to deal with more complex applications.

2.1.5 Fork-Server

By default, AFL calls `execve()` per fuzzing iteration to launch the target application. This produces high CPU load on the system, since the OS and the dynamic loader have to load the application, dynamically link all shared libraries and repeatedly execute the same initial code path of the application.

However, AFL deals with this issue and avoids unnecessary load by using the fork syscall instead. Most POSIX-conform OS implement the fork syscall Copy-on-Write (CoW)-aware. This means, if `fork()` was executed, the OS just copies the parental page table and reuses all pages as long as no write-attempt is occurred. In case of a write-attempt, the OS will copy only the requested page instead of all related pages at once. AFL exploits this functionality by copy and eventually launch the targeted application via `fork()`. This improves significantly the fuzzing performance opposed to the `execve()` approach.

However, the user has to include an initial call to the AFL-forkserver code in the targeted application. Since the position of the forkserver represents the new entrypoint, the less frequent code has to be executed from within the new entrypoint, the more performance will be improved.

2.1.6 Runtime Complexity

In most cases, the runtime complexity of AFL is much lower compared to other fuzzers. This is because fuzzing without any feedback coverage or control-flow analysis heavily depends on brute-force methods. For instance, the condition in listing 2.4 is hard to

satisfy for blackbox-fuzzers, since the comparison of three different bytes at the same time would yield over 2^{24} combinations. In contrast, AFL will find the crashing input in $3 \cdot 2^8$ guesses at worst. This is due to the runtime-feedback, that breaks the exponential blackbox problem into a linear problem.

Listing 2.4 Linear AFL Problem

```
1: if (input[0] == 'A' && input[1] == 'F' && input[2] == 'L')
2:         crash();
```

Nevertheless, the AFL approach has some scope-depended disadvantages. For example, there are several known condition patterns (see listing 2.5) that are hard to satisfy for feedback-driven fuzzers, but simple to solve by for concolic execution [Ste+16].

Listing 2.5 Difficult AFL Problem

```
1: if (input == 0xaabbccdd0badc0de)
2:         crash();
```

2.1.7 Challenges of Applying AFL on Kernelspace

There are several challenges which must be solved in order to apply the AFL fuzzing engine on supervisor mode code. For instance, it is much harder to gather dynamic branch information during runtime. Several approaches have been proposed, but most of them require the ability to recompile the targeted OS [**kcov**] or would result in poor runtime performance*†. Moreover, a different crash detection and handling mechanism is required, since the AFL user mode approach would not work on kernel level. Since kernel misbehavior results in a terminated state, the fuzzer must either run in another independent domain than the target OS and reset the initial kernel state or ensure that the fuzzer state is saved even if the OS crashes. The first case also requires the ability to detect kernel misbehavior. Finally, the fuzzer has to deal with non-deterministic runtime information such as the occurrence of an interrupt or trap handler during the fuzzing process. Otherwise, the the results in the AFL bitmap would be clobbered by "noise".

*According to Jesse Hertz and Tim Newsham the QEMU TCG approach would result in 1–10 exec/sec single threaded performance running on an unspecified quadcore CPU [HN16a].

†Mateusz Jurczyk describes a slowdown of 20-50x time by using Bochs to gather branch information [Jur16].

2.2 Intel VT-x

Virtualization has become an indispensable technology in many computing domains during the last ten years. The kernel fuzzing approach introduced in this thesis relies on modern x86-64 hardware virtualization technology. Thus, this section will provide a brief overview about the Intel proposed hardware virtualization technology called Intel VT-x.

2.2.1 Terminology

Virtualization is the process of providing virtual resources separated from physical resources to run arbitrary software within a virtualized context.

The virtualization role model is divided into two components: the VMM and the Virtual Machine (VM). The VMM, also termed hypervisor or host, is a privileged software that has full control over the physical CPU and provides restricted access to physical resources to virtualized guests. The VM, also termed guest, is a software that is transparently executed within the virtualized context provided by the VMM.

The term "virtualization" is often used to refer to "full virtualization". Full virtualization means that software, which was originally designed to run on physical hardware, is able to run within the virtualization context without any modifications needed. Popek and Goldberg specify this term even further [PG74]:

1. In contrast to full system emulation, a virtual machine has to execute as much as possible directly on the CPU without any intervention (except for a subset of *hard-to-virtualize* instructions).

2. Software that runs within the virtualized context has no control or access to host physical resources, except for assigned virtual resources.

3. Software has to behave exactly as it would on real hardware.

2.2.2 General

Intel VT-x is an extension of IA-32, which provides support for hardware assisted virtualization. Prior to hardware supported virtualization, [RI00] described the x86 architecture to be partially or even fully non-virtualizable. They stated that "seventeen [x86]

instructions did not meet virtualization requirements because they were sensitive and unprivileged".

This has changed with the advent of new hypervisors using a technology termed *Binary Translation (BT)*. Instead of modifying the OS to fit the x86 virtualization requirements before execution (known as paravirtualization), BT modifies code during runtime on the fly. This does not affect the execution of user mode code, but it does impact the execution of the guest kernel mode. In order to deal with non-virtualizable instructions, which in general are privileged instructions running in supervisor mode, BT replaces those during runtime with other proxy-instructions to achieve practically the same effect without breaking out of the virtualized context. BT comes in return of efficiency (at least in ring 0 execution) and requires a certain amount of implementation complexity. To provide a further layout of isolation, BT-based hypervisors typically executes virtualized ring 0 code in between of ring 0 (kernel mode) and ring 3 (user mode).

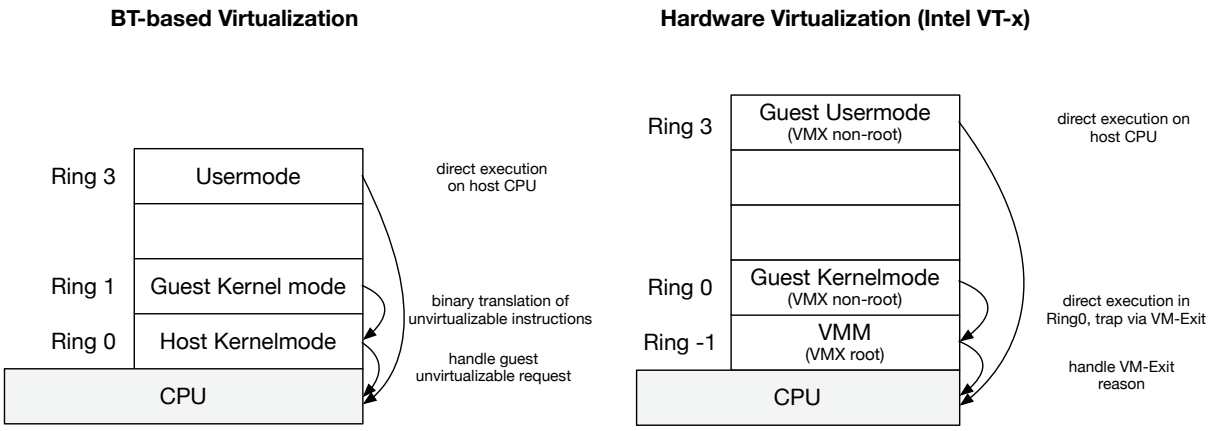

Figure 2.1: Binary Translation and Intel VT-x, figure based on [VMw08]

To minimize such efforts, Intel and AMD have released specific IA-32 extensions to circumvent the required runtime observation of the kernel control-flow. Instead, if an non-virtualizable instruction is executed in a virtualized context, the CPU will independently trap into the VMM. Those events are termed *VM-Exit* events and delegated to the VMM, which handles the request by emulating the required instruction for the virtualized guest.

In comparison, hardware virtualization is making the implementation of full virtualization less complex and is achieving higher virtualization performance compared to virtualization based on Binary Translation*.

Nevertheless, hardware assisted virtualization has become the state-of-the-art virtualization technique, among others, because Long Mode virtualization on Intel CPUs is only efficiently possible by using VT-x due to technical limitations [VMw09]. Furthermore, hardware assisted virtualization is often used in combination with paravirtualization device- and driver-components (e.g. virtio [Rus08]) to achieve nearly native performance in common use-cases (e.g. networking).

2.2.3 VMX Operation

To provide full hardware assisted virtualization support, Intel VT-x adds two additional execution modes upon to the known Current Privilege Level (CPL) model [Inta]. These modes are termed Virtual Machine Extension (VMX) root and VMX non-root operations and must explicitly be enabled by software. If VMX operations are disabled, the CPU is in the VMX Off-state and behaves as usual. Otherwise, the CPU is in the VMX On-state and either in VMX root or VMX non-root operation. Typically, the VMM code is executed within VMX root operation and guest code within VMX non-root operation. Since VMX operations run upon the CPL model, guest software can run in the indented CPL domain without any modifications.

2.2.4 VMX Transitions

To transit from VMX Off-state to VMX root operation, software executes `VMXON` instruction and conversely executes `VMXOFF` instruction to switch to VMX Off. *VM Entry* events are transitions from VMX root to VMX non-root operation. A VM Entry event is initiated by executing `VMLAUNCH` instruction in VMX root operation. To resume an already launched VM, the VMM has to execute `VMRESUME` instruction instead.

*Except for first generation implementations of Intel VT-x and AMD SVM (Pacifica) [VMw08].

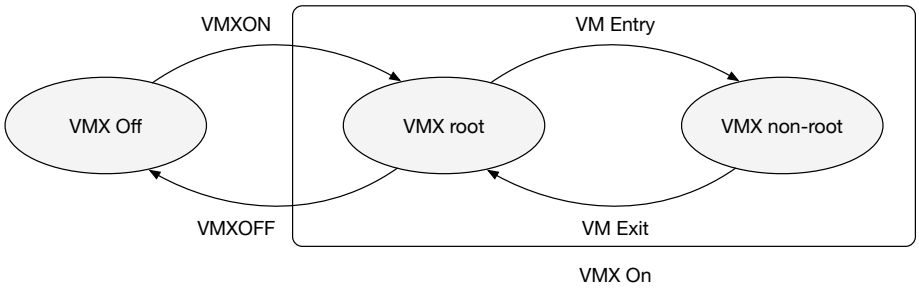

Figure 2.2: VMX Transitions, figure taken from [Cha15]

Since VMX non-root operation is a restricted context, a VM Exit event occurs in response to several reasons and initiates a transition to VMX root operation. VM Exit events are the result of the attempting execution of certain restricted instructions, an expired VMX-preemption timer or attempting access to restricted or emulated resources such as virtual device memory.

2.2.5 Virtual Machine Control Structures

To create, launch and execute a VM, the VMM has to prepare a Virtual Machine Control Structure (VMCS) [Intb] beforehand. A VMCS contains all essential information about the VM and the current state. The VMCS is especially used to facilitate the VMX Entry and the VMX Exit events and specify the guest execution behavior in VMX non-root operation. Only one VMCS can be used within the same logical CPU at the same time. The VMCS data consists of six logical groups and is structured as follows:

Guest-state area:

The guest state is stored in this area after the CPU transits from VMX non-root to VMX root operation (VM Exit event). Accordingly, the state is loaded if a VM Entry transition is initiated. This area stores, among others, all mutable control and debug register values (such as `CR0`, `CR3`, `CR4` and `DR7`), certain registers (`RSP`, `RIP` and `RFLAGS`) and segment registers (`SS`, `CS`, `DS`, `ES`, `FS` and `GS`). Other General Purpose Register (GPR) has to be saved immediately after the VM Exit transition by the VMM in VMX root operation.

Host-state area:

This area represents the counterpart of the Guest-state area and stores the host state after the CPU switches from VMX root to VMX non-root operation (VM entry). Accordingly, the state is loaded if a VM exit transition occurs. Obviously, this area contains all host values that might get clobbered by VMX non-root execution.

VM-execution control fields:

The control fields for VM execution specify the VMX non-root behavior. This includes the option to enable the VMX-preemption timer, the option to raise a VM Exit event on the execution of certain instructions (e.g. `hlt`) or on modifications of the `CR3` register*. Moreover, by modifying this area, software is also able to activate VT-x features such as Extended Page Tables (EPT).

VM-exit control fields:

The VM Exit Control fields area specifies the behavior of the CPU after a VM Exit event occurs. This covers configurable options such as the predefined automatic load of certain Model Specific Register (MSR) values directly after a VM Exit transition.

VM-entry control fields:

The VM Entry Control fields area specifies the behavior of the CPU after a VM Entry event occurs. This area is the counter part of VM Exit control fields area and it also offers an equivalent option-set.

VM-exit information fields:

This field stores the VM exit transition information and reasons. Generally, after a VM Exit event occurs, the VMM has to consult this area to get the VM Exit reason to react appropriately.

To configure and set a VMCS, certain new instructions have been introduced by Intel to serve this purpose. To load an already prepared VMCS, VMX provides the `VMPTRLD` instruction which sets the targeted VMCS as current. The VMM is also able to read and write certain values of the current VMCS by executing the `VMREAD` instruction or the `VMWRITE` instruction.

*On x86-64 systems, the CR3 register stores the value of the Page Map Level 4 Table (PML4T) in Long Mode.

2.3 KVM / QEMU

Kernel-based Virtual Machine (KVM) is a Linux kernel subsystem for virtualization. In contrast to VMware, KVM only supports full virtualization based on hardware-assisted virtualization extension such as Intel VT-x (see 2.2) and AMD SVM. This section describes the fundamental concepts of KVM, Quick Emulator (QEMU) and provides a brief overview of the interaction between both components.

2.3.1 General

KVM is a Linux kernel subsystem which exposes hardware virtualization features to the userspace. It is shipped in form of multiple kernel modules (`kvm.ko` and `kvm_intel.ko` or `kvm_amd.ko` on x86 systems) and available since Linux 2.6.20. Since hardware-assisted virtualization extensions require supervisor privileges to operate, the kernel has to initiate VM preparations and eventually launch the VM. KVM handles the process of VMX transitions and exposes virtualization abilities via an standardized interface to the user mode. Over time, KVM has become the state-of-the-art hypervisor for Linux systems. Due to the kernel integration, KVM does not only provide good virtualization performance, but also enables the implementation of more complex features within the kernelspace like KSM [AEW09] and virto paravirtualized devices.

QEMU is a GPLv3 licensed CPU and hardware emulator for almost all common used microarchitectures. The emulation backend is based on dynamic binary translation and provides consequently high performance. In addition to CPU-emulation, QEMU is able to emulate various system components of the desired emulation environment, such as network interfaces, CD-ROM drives, hard-drives, various PCI-devices and graphic adapters. Furthermore, QEMU provides multiple high level capabilities such as TUN network tunneling and CoW-aware virtual hard-disk images.

In combination with KVM, QEMU becomes a hosted-hypervisor which is able to execute code via KVM at nearly native performance, whereas QEMU deals with the essential emulation of hardware components.

2.3.2 KVM Architecture

KVM exposes all features via a common Unix device node (typically mapped to `/dev/kvm`). This device node is accessible via an extensive `ioctl`-interface*. In addition to the common kernel- and user-mode, KVM adds another mode termed *guest mode* [Kiv+07]. In general, the KVM guest mode corresponds to the VMX non-root operation. This mode is maintained and entered by KVM and thus only accessible via `ioctl`-commands from within the user mode. Any VM has a configurable number of virtual CPUs (vCPUs) and its own (virtual) physical memory that is shared with several pages of the virtual memory space associated with the instructing user mode process. KVM does not schedule guest code by its own. Instead, the corresponding user mode process is accountable to repeatedly acquire KVM for guest mode execution. On VM Exit events, KVM either handles the exit reason in kernel mode or requests the user mode process to handle it †.

2.3.3 KVM API

To interact with KVM from the user mode, an application must obtain a file descriptor of the device node `/dev/kvm` and communicate with this file descriptor via `ioctl`-commands. Via the `KVM_CREATE_VM` `ioctl`-request, KVM creates a new VM and provides a file descriptor referenced to this VM. By another `ioctl`-command, the user mode process can acquire a vCPU to the associated VM. For this purpose, the `KVM_CREATE_VCPU` `ioctl`-command is used. Afterwards, KVM provides another specific file descriptor referenced to the newly created vCPU. To enter guest mode on the vCPU, the user mode process has to invoke the `KVM_RUN` `ioctl`-command. Listing 2.6 illustrates the common process of VM creation, vCPU acquiring and VM launching.

*For further details about the KVM `ioctl`-interface please see: https://kernel.org/doc/Documentation/virtual/kvm/api.txt

†Usually seen within the QEMU device emulation on I/O requests.

Listing 2.6 KVM API interaction

```
1: kvm = open("/dev/kvm", O_RDWR | O_CLOEXEC);
2: vmfd = ioctl(kvm, KVM_CREATE_VM, NULL);
3: /* allocate memory region... */
4: vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, NULL);
5: /* prepare vCPU registers... */
6: while (1) {
7:         exit_reason = ioctl(vcpufd, KVM_RUN, NULL);
8:         /* ... handle delegated exit reason */
9: }
```

2.4 Intel Processor Trace

With the fifth generation of Intel Core processors (Broadwell architecture), Intel has introduced a new processor feature, *Intel Processor Trace (Intel PT)*, to provide execution and branch tracing information directly by the processor. Unlike other branch tracing technologies such as *Intel Last Branch Record (LBR)*, the size of the output buffer is no longer strictly limited by hardware. In theory, Intel PT is capable of long-term tracing. In practice, this is only limited by the size of the buffer of the output target (e.g. the main memory or an external PT-aware device). If the output target is *emptied* repeatedly and timely, buffer overruns will be circumvented and the trace session duration becomes unlimited. The processor's output format is packet-oriented and separated into two different types: general execution information and control-flow information packets. To reconstruct the control-flow from within the trace data, an Intel PT software decoder as well as the software that was executed at the time of the tracing is needed.

2.4.1 Execution Information Packets

Intel PT produces a set of miscellaneous packets describing various events and states of the CPU during trace sessions. This includes the following packet types:

Packet Stream Boundary (PSB):
PSB packets indicate the boundary of an Intel PT trace sample and help an Intel PT software decoder to find the beginning of the actual trace data. PSBs are included automatically by the CPU in the Intel PT data stream and no option is provided to deactivate the generation of PSBs.

Time-Stamp Counter (TSC):
TSC packets store the value of the software accessible time-stamp counter of the associated CPU at the moment during record. The generation of this packet type can be deactivated by modifying the `IA32_RTIT_CTL.TSCEn` MSR field. This packet type was not used during this thesis.

Core Bus Ratio (CBR):
CBR packet contains the value of the bus clock ratio. This packet type is not relevant in the field of flow reconstruction and was not used during this thesis. Unfortunately, this packet type is not configurable and it's generation cannot be deactivated. Each occurrence of this packet in the trace data can safely be ignored.

CoFI Type	Instructions
Conditional Branch	JA, JAE, JB, JBE, JC, JCXZ, JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ
Unconditional Direct Branch	JMP (E9 xx, EB xx), CALL (E8 xx)
Indirect Branch	JMP (FF /4), CALL (FF /2)
Near Ret	RET (C3, C2 xx)
Far Transfers	INT3, INTn, INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

Table 2.1: CoFI Types ([Intc], Table 36-1, CoFI Type for Branch Instructions)

Overflow (OVF):

As already mentioned, it is possible that the configured output target might be overrun during runtime. In such case, an OVF packet is generated by the CPU to indicate to the Intel PT software decoder a potential loss of packets.

Paging Information Packet (PIP):

PIPs indicate the modification of the CR3 register during runtime. As a result, the Intel PT software decoder is able to comprehend a paging transition post-mortem. Typically, a paging transition occurs if another user mode process obtains CPU time by the scheduler and thus the virtual memory address space is switched.

2.4.2 Flow Information Packets

Intel PT may also produce various types of control flow related packet types by the processor during runtime. A software decoder is able to reconstruct the exact control flow by decoding those packets and by putting them into correlation with the associated program. Intel describes five types of control-flow affecting instructions called *Change of Flow Instructions (CoFI)* (see Table 2.1). The execution of different CoFI types result in different sequences of Intel PT flow information packets. Fifth generation Intel Core CPUs provide the following set of flow information related packet types:

Taken-Not-Taken (TNT):

If the processor executes any type of conditional jump, the former decision whether

this jump was taken or not will produce a TNT packet containing the value of the taken direction. Those packets are essential for the reconstruction of the exact control flow.

Target IP (TIP):

If the processor executes a jump or transfer instruction, which depends on a value in memory or register, the decoder will not be able to recover the control flow. Therefore, the processor produces a TIP packet on the execution of an indirect branch, near ret and far transfer type instructions. Those TIP packets store the corresponding target IP that was executed by the processor after the transfer / jump occurred. If a special CPU feature called *RET compression* is deactivated, the CPU will include TIP packets even if `ret` instructions are executed. Otherwise, the software decoder is forced to track the state of the current call stack. To minimize complexity, this feature is explicitly deactivated in kAFL. In addition to common TIP packets, there are two further and more specific TIP packet types. These include TIP.PGE packets (*Packet Generation Enabled*) to indicate the first TIP after tracing was enabled and TIP.PGD packets (*Packet Generation Disabled*) to indicate the last traced packet before tracing was eventually disabled.

Flow Update Packets (FUP)

Another case where the processor must produce a hint packet for the software decoder are asynchronous events such as any type of interrupts or traps. Those events are recorded as FUPs and usually followed by a TIP to indicate the following instruction.

MODE:

If a processor mode switch occurs (e.g. from Protected Mode to Long Mode), the processor also notifies such an event in form of a MODE packet.

2.4.3 Intel PT Software Decoder

In order to reconstruct a control flow from Intel PT's output data, the flow information data must first be decoded. Intel PT does not provide a complete list of executed instruction pointers. Instead, Intel PT generates as little information as necessary to reduce the amount of data produced by the processor. Consequently, the Intel PT software decoder does not only require the control flow information data to reconstruct the control flow, but also needs the program that was executed during tracing. If the program is modified during runtime, as often done by Just-in-Time (JIT) compilers in user-

and kernel mode, the decoder will not be able to exactly restore the runtime control flow. To bypass this limitation, decoders will need information on all applied modifications of the program instead of an ordinary memory dump or the executable file.

Listings 2.8 and 2.7 illustrate an example of an Intel PT sample and the associated program*. Listing 2.8 is arranged according to the causing instruction, whereas the value within the brackets indicates the chronological order†.

Listing 2.7 Disassembly of a x86 program running in Real Mode

```
0x1000 mov  dx, 0x3f8
0x1003 add  al, bl
0x1005 add  al, 0x30
0x1007 nop
0x1008 mov  al, 0xa
0x100a nop
0x100b jmp  loc_1011 // (EA XX)
0x1010 hlt
0x1011 mov  al, 0xa
0x1013 jae  loc_1016
0x1015 nop
0x1016 nop
0x1017 jmp  loc_1010 // (EA XX)
```

Listing 2.8 Decoded Intel PT Sample (Flow Information Packets only)

```
(1) TIP.PGE 0x1000




(2) TIP 0x1011
(5) TIP.PGD

(3) TNT (Taken)


(4) TIP 0x1010
```

2.4.4 Trace Filtering

To limit the amount of generated trace data, Intel PT provides multiple options for runtime filtering:

IP-Filtering:

Depending on the processor, it might be possible to configure multiple instruction pointer filter ranges. In general, those filter ranges only affect virtual addresses if paging is enabled. To enable IP filter ranges, software has to modify `IA32_RTIT_ADDRn_A` and `IA32_RTIT_ADDRn_B` MSR, whereas `n` is a placeholder and represents the identifier of the given IP filter‡. Software can configure those IP

*This program was originally taken from https://lwn.net/Articles/658511/. However, this program was slightly adapted to generate more flow information packets.

†This is a simplified trace example. Unfortunately, TNT packets might be cached by the CPU during tracing and thus the resulting order does not necessarily represents the chronological order.

‡According to the Intel documentation, 4 was the maximal number of configurable IP filter ranges at the time of writing.

filter ranges as opt-out ranges as well as opt-in ranges.

CPL-Filtering:
In accordance to the CPL-filtering, it is possible to opt-out the entire activity of the user mode ($CPL > 0$) or kernel mode ($CPL = 0$) from each other. kAFL utilizes this filter option to limit tracing explicitly to kernel mode execution.

CR3-Filtering:
In most cases the focus of tracing is not the whole operating system within all user mode processes. To limit trace generation to one specific virtual memory address space, software can use the CR3 filter. Thus, Intel PT will only produce trace data if the CR3 value matches the configured value in `IA32_RTIT_CR3_MATCH` MSR.

2.4.5 Table of Physical Addresses

Intel PT supports various configurable target domains for output data. According to the Intel documentation, the following options are given:

1. Single Range Output
2. Platform-specific Trace Transport System
3. Table of Physical Addresses (ToPA)

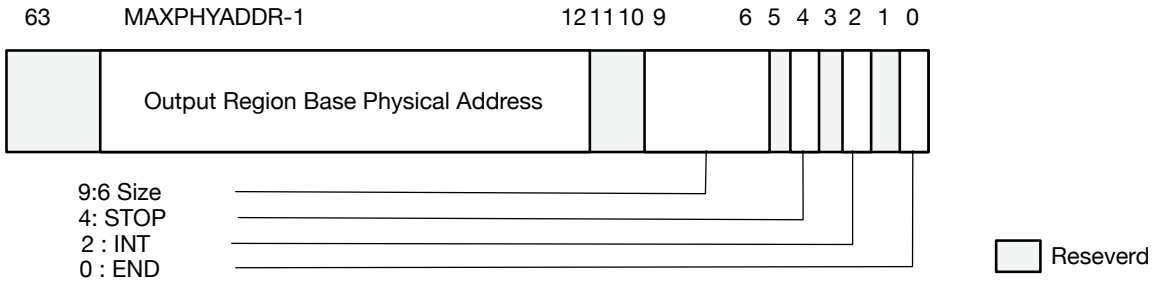

Figure 2.3: ToPA Entry Structure, figure taken from [Intc]

This thesis focuses on the ToPA mechanism only. Other methods are not addressed further. The Table of Physical Addresses is a mechanism to concatenate multiple distributed memory chunks together to an unified contiguous output region. This mechanism is based on a linked list of ToPA tables. Every *ToPA Table* contains multiple *ToPA*

Entries that contain the physical address of the associated memory chunk, termed *ToPA Region*, and the table itself is stored in physical memory. Each ToPA entry is specifically encoded (see Figure 2.3) and contains a physical address, a size specifier for the referred memory chunk in physical memory and multiple type bits. Those type bits specify the behavior on access of the ToPA Entry. This includes the following options:

STOP Entry:
If the linked output region is filled, tracing will be disabled by hardware. The last ToPA entry must configure this option.

INT Entry:
If the associated ToPA output region is filled, a notifying interrupt will be raised and the trace generation will be continued in the next ToPA output region.

END Entry:
This entry is necessary for the ToPA structure to indicate the last entry within this table. Therefore, this ToPA entry points to the next ToPA table instead of a ToPA output region.

The following figure illustrates a sample ToPA configuration with to distributed ToPA tables (*Table A* and *Table B*) and multiple ToPA output regions:

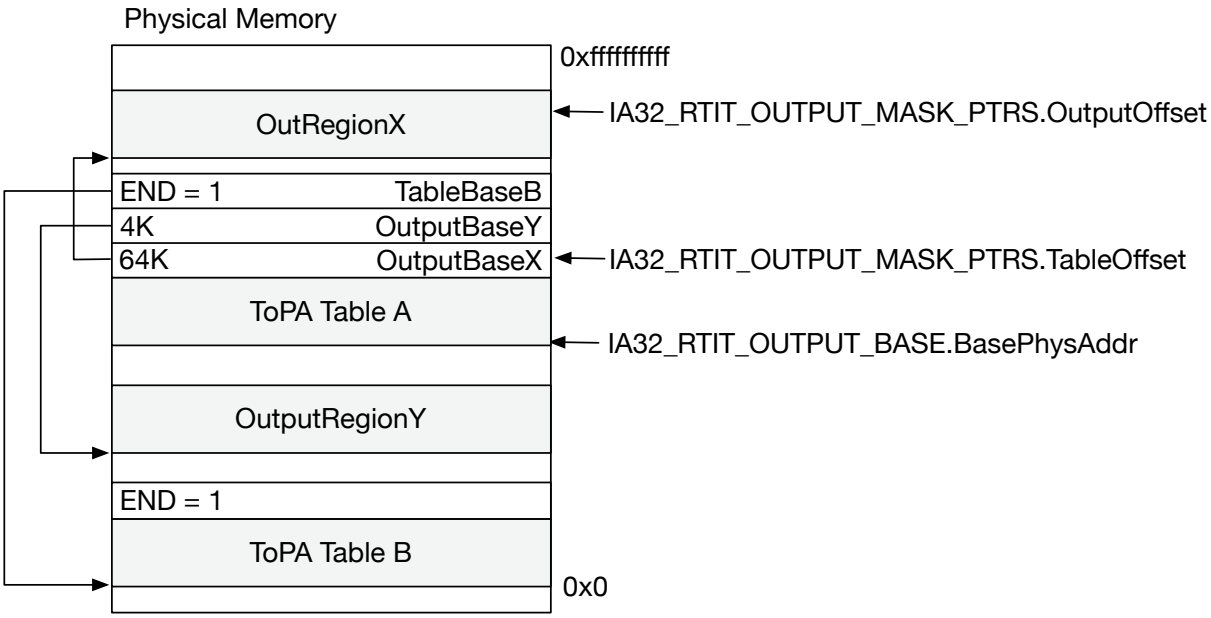

Figure 2.4: Table of Physical Addresses, figure taken from [Intc]

As shown in figure 2.4, `IA32_RTIT_OUTPUT_BASE` stores the physical base address of the current ToPA table. The `IA32_RTIT_OUTPUT_MASK` MSR stores the offset value of the table (`TableOffset`) as well as the value for the current ToPA output region offset (`OutputOffset`). According to the Intel documentation, these values are only visible if tracing is disabled.

If the *INT* bit is set in a ToPA entry, the processor will raise a Local Vector Table Performance Monitor Interrupt (LVT PMI) if the corresponding ToPA output region is filled and the interrupt was enabled by software. This mechanism is useful to notify software about an occurred or upcoming buffer overrun. Since the LVT PMI is not "precise" according to the Intel documentation, software must be aware of this peculiarity. This means, software has to ensure that a backup output region following after the *INT* ToPA entry is prepared and large enough to store upcoming trace data before the belated PMI is raised.

2.4.6 VMX Tracing

Initially, Intel PT was unable to trace VMX non-root operations and thus was not suitable for VM tracing. Approximately since the sixth generation of Intel Core processors, this limitation is gone and the latest processor models support this capability. To ensure that the processor is capable of VMX non-root tracing, the driver must consult the value of `IA32_VMX_MISC` MSR during runtime and check if the 14th bit is set. Consequently, if this bit is not set, the processor does not support VMX non-root tracing. Unfortunately, Intel does not provide any further information about which processor does support VMX non-root tracing.

Chapter 3

Design and Implementation

kernel AFL (kAFL) is a prototype for hardware-accelerated kernel fuzzing developed during this thesis. This chapter describes kAFL's design and implementation aspects and is structured as follows:

In chapter 3.1 the fuzzer engine is described and insights about the first instrumentation-based approach of kAFL and the inter-VM communication of virtualized target OSs are given. Chapter 3.2 covers the developed KVM extension driver called `vmx_pt`. The `vmx_pt` KVM extension integrates support for Intel PT tracing for vCPUs. In combination with a developed QEMU extension, which is introduced in chapter **??**, this feature is also accessible via user mode and can be used within common virtualization use-cases. Finally, in chapter 3.4, a novel Intel PT decoder engine is described, which is directly integrated in QEMU to deal with `vmx_pt` provided trace data in an automated and highly efficient fashion.

By combining all components, kAFL becomes a hardware-accelerated x86-64 supervisor mode fuzzer that leverages modern virtualization techniques and provides much better performance compared to other proposed coverage-guided kernel fuzzers. Besides that, kAFL does not require any recompilation of the target OS. Instead, the current version of kAFL requires only the usage of an OS specific kernel driver to interact with kAFL via an inter-VM bridge.

3.1 kAFL Fuzzer

Initially, the objective of this project was to extend the already developed USB fuzzing framework vUSBf [SSS14]. This includes support for feedback-capabilities inspired by AFL and the ability to fuzz USB device drivers of various OS coverage-guided. Eventually, this objective was discarded later during development and instead a general-purpose feedback-driven kernel fuzzer was developed. kAFL currently does not support fuzzing of USB drivers but this feature might be implemented in later versions. Nevertheless, most of the KVM and QEMU specific code of vUSBf was reused. kAFL is a project mostly written in Python. Besides that, all fuzzing techniques of AFL were ported to kAFL.

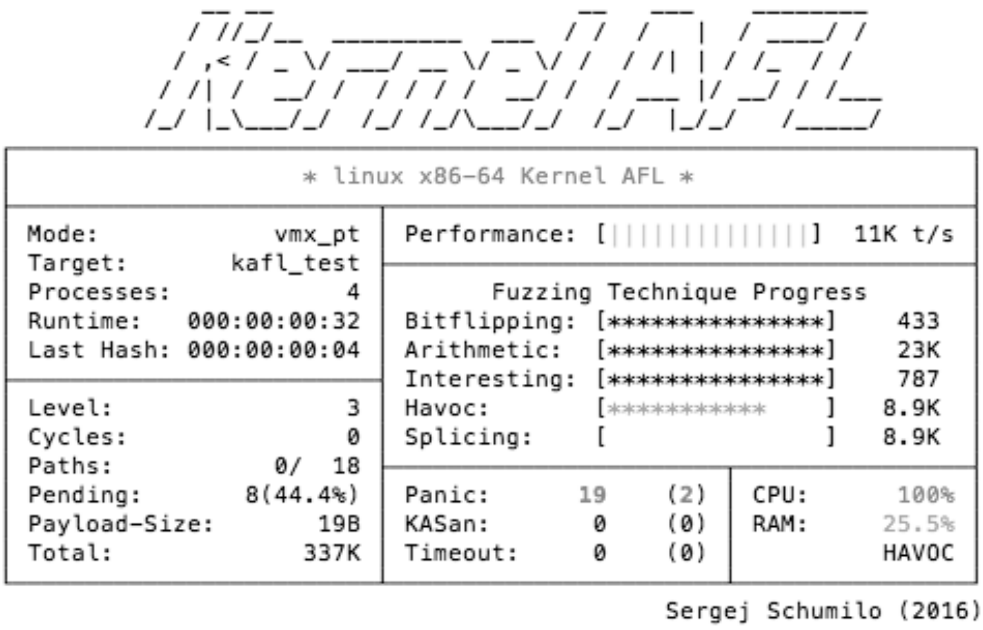

Figure 3.1: kAFL Screenshot

During this project two different approaches to gain coverage-feedback were implemented. This includes an instrumentation-based approach as well as a novel Intel PT based approach. This section covers only the instrumentation-based approach. The more sophisticated Intel PT approach is discussed later in section 3.2, section 3.3 and section 3.4. Because kAFL initially does not utilize the Intel PT provided capabilities, it was only capable to fuzz OSs which can be recompiled to integrate compile-time instrumentations as already demonstrated by AFL. A proof-of-concept implementation for this approach was developed for Linux x86-64.

3.1.1 Architecture

The kAFL fuzzer consist of multiple components, whereby each offers a specific functionality. Figure 3.2 illustrates the interaction between all components. Note that the number of *Slave Processes* is fully scalable:

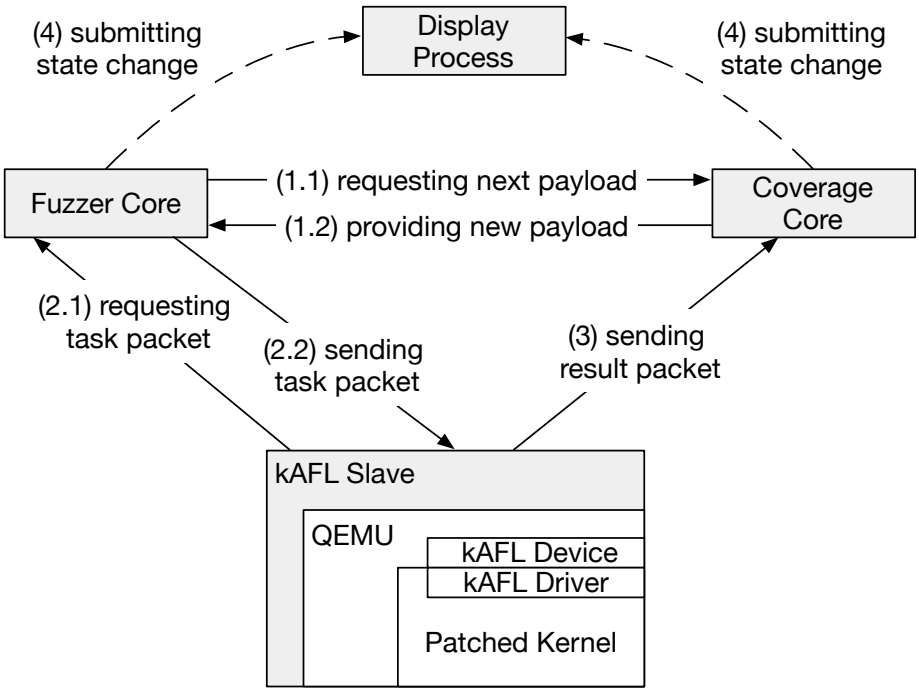

Figure 3.2: kAFL Architecture

Fuzzer Core:

The *Fuzzer Core* of kAFL represents the fuzzer itself. This component generates new input in reply of request from *Slave Processes*. To generate new mutations, kAFL utilizes the same fuzzing techniques as AFL (see chapter 2.1). The strategies are reimplemented in Python, but code in frequent executed sections is based on native calls to avoid performance-losses. If the Fuzzer Core has applied all fuzzing techniques on the current input data, the Fuzzer Core will request the input data for the next iteration from the *Coverage Core* process. The *Display Process* is notified periodically about the internal state of the Fuzzer Core. Therefore, the Fuzzer Core sends the state data to the Display Process. This includes the current fuzzing stage, the current progress and the total number of applied mutations.

Coverage Core:

The Coverage Core receives AFL bitmaps from the Slave Processes after a mutated input is applied to the target. Based on those information, the Coverage Core is able to measure progress during the fuzzing process and determines the next preferred input mutation. The Coverage Core also provides the next determined input if the Fuzzer Core consults it. On internal state changes the Display Process is notified as well. This includes the discovery of a new detected path, an occurred crash or an expired timeout.

Slave Process:

The target domain of this fuzzer is a virtualized OS running inside QEMU / KVM. Therefore, kAFL has to utilize QEMU to start and manage those VMs. Due to the usage of KVM, the fuzzer provides a nearly native performance. The communication between the Slave Process and the OS itself is achieved via an inter-VM communication bridge, which is based on the *kAFL guest device* and the *kAFL guest driver* (see chapter 3.1.3) of the OS. The synchronization and interaction via the inter-VM bridge is managed by the Slave Process. A Slave Process interacts with the Fuzzer Core to request and receive new input data. Furthermore, the Slave Process sends resulting AFL bitmap to the Coverage Core.

Display Process:

Due to the distributed architecture, the information is distributed between the Fuzzer Core and the Coverage Core. This results in an loss of information between each component, which does not affect the effectivity of the fuzzer. Unfortunately, the user interface highly depends on this set of information. Therefore, both components submit state changes to the *Display Process*. Using the provided state changes, the Display Process is able to output this information in regular intervals depending on the configuration of kAFL.

3.1.2 Compiler Wrapper

The first implementation is fundamentally based on compile-time instrumentation to track branch transitions. To serve this purpose, an own GNU `gcc` wrapper was developed. The kAFL-compiler wrapper is inspired by and works mostly the same way as the `afl-gcc` compiler. The compiler wrapper intercepts the compile process of GNU `gcc` and modifies the intermediate assembly produced by GNU `gcc`. The insertion of x86-64 instrumentation code is performed before GNU `gcc` eventually calls GNU `as` to translate the intermediate x86-64 assembly to binary.

The wrapper includes the following instrumentation after each basic block: The instrumentation code saves all GPRs that might get clobbered during execution. This also includes the `EFLAGS` register. The `EFLAGS` register stores status, control and systems bits. Especially the status category is relevant, since this category includes the *Carry Flag*, *Parity Flag*, *Adjust Flag*, *Zero Flag*, *Sign Flag* and *Overflow Flag*. Bits of other categories, especially of the control or system category, will not be clobbered during execution, except if those bits are intentionally modified by the instrumentation code - which is not the case. Therefore, this approach is suitable for kernel mode.

Listing 3.1 kAFL Linux x86-64 instrumentation assembly

```
1:      leaq -72(%rsp), %rsp      // resize stack
2:      /* ... */                 // save certain GPRs on stack
3:      seto %al                  // fast pushf
4:      lahf                      // fast pushf
5:      movq %rax, 64(%rsp)       // fast pushf (save on stack)
6:      movq $<value>, %rdi       // compile-time random
7:      call kafl_logger
8:      movq 64(%rsp), %rax       // fast popf (load from stack)
9:      addb $127, %al            // fast popf
10:     sahf                      // fast popf
11:     /* ... */                 // restore certain GPRs from stack
12:     leaq 72(%rsp), %rsp       // restore previous stack pointer
```

Similar to AFL, the `EFLAGS` register is not saved and restored by using `pushf` and `popf`. Instead, the instrumentation code uses the more efficient `sahf` and `lahf` approach, since `pushf` and `popf` are relatively slow compared to the other approach. This is due to a

quirk of the x86 architecture* and since this instruction saves mostly the entire content of `EFLAGS`, which is superfluous for our approach. Additionally, the `OVF` bit of the `EFLAGS` register is saved by using the `seto %ah` and `addb $128, %ah` approach. As already mentioned, other `EFLAGS` bits left untouched and therefore are not saved.

Unlike AFL, the `call` instruction does not point to a function, which is part of the instrumentation code and is included by the compiler wrapper. Instead, the instrumentation code calls a function, which is either part of the kernel core or part of a kernel module. This code is used for the inter-VM communication and must be explicitly included to the kernel source or, if included within a kernel module, be loaded during runtime. The use of function `kafl_logger()` is described in more detail in section 3.1.4.

Depending on the targeted kernel code to fuzz, it is required to integrate `kafl_logger()` into the kernel core, if the targeted code is executed even before the kernel module loader is able to load any kernel modules (e.g. file system driver). Otherwise, the code can be outsourced to a kernel module. In such case, the targeted kernel module will require to load the kernel module containing `kafl_logger()` beforehand.

3.1.3 kAFL Guest Device

Kernel Fuzzing can be divided into two different categories: *external fuzzing* via interfaces such as network devices or USB and *internal fuzzing* via a user mode application executing various syscalls. As already mentioned, kAFL currently only supports internal fuzzing and uses an inter-VM communication bridge for synchronization and communication with the fuzzing user mode application. To provide inter-vm communication, a special QEMU device was developed, that emulates a virtual PCI-device. This device is called *kAFL guest device*.

The kAFL guest device is based on the QEMU *ivshmem* (inter VM shared memory) device implementation [Mac11]. This QEMU device emulates a virtual PCI-device with several PCI memory regions, called PCI Base Address Registers (BAR), and a Memory-mapped I/O (MMIO) area. The kAFL guest device utilizes multiple PCI-BARs for shared-memory capabilities within the host and the virtualized guest system. By default, kAFL emulates the following PCI-BARs:

*https://reviews.llvm.org/D6629

Payload PCI-BAR (128 KB):

The *Payload PCI-BAR* stores the payload, which is used by the fuzzing application to be injected via syscalls into the targeted OS. This area is modified by the host after each iteration. To avoid race conditional memory accesses between the kAFL fuzzer and the fuzzing application inside the host, inter-VM synchronization mechanisms are used.

Program PCI-BAR (4 MB):

The fuzzing application is stored in the *Program PCI-Bar*. This memory region is only accessed once by the guest during the initialization processes. Afterwards the program is copied into the user space and executed once until the target OS crashes.

Arguments PCI-BAR (4 KB):

In addition to the fuzzing program, the *Argument PCI-BAR* stores additional arguments for the fuzzing program. This is especially useful for fuzzing applications such as filesystem fuzzer, whereas the filesystem type (e.g. EXT4) is specified in this PCI-BAR.

kAFL Bitmap PCI-BAR (64 KB):

The already mentioned `kafl_logger()` function manipulates the *kAFL Bitmap PCI-BAR* memory. This memory region represents the already shown AFL bitmap (see chapter 2.1.4) and is exactly as large as the original bitmap size.

Furthermore, the kAFL guest device emulates an additional MMIO-area for synchronization purposes and unidirectional communication from the guest to the host. The guest is able to use the following commands for unidirectional notification via writing into the MMIO-memory:

MMIO_REG_IRQ:

This command is used for the synchronization process. The guest notifies the host that the host's Interrupt Request (IRQ) was handled and the fuzzing application is about to initiate a fuzzing iteration. This is used to prepare for the host-side certain states.

MMIO_REG_ACQUIRE:

This command is sent by the guest to acquire the next payload. Afterwards, the host writes the requested payload into the Payload PCI-BAR and raises an IRQ to notify the guest.

MMIO_REG_PANIC:
If a kernel panic occurs, the guest notifies the host about the misbehavior via this command.

MMIO_REG_KASAN:
If KASan* is supported and an issue was detected by it, the guest notifies the host via this command.

MMIO_REG_READY:
This command is used only once during the initial setup process and notifies the host that the guest has prepared and launched the fuzzing application and is ready for the fuzzing process.

MMIO_REG_CR3:
This command is used for further Intel PT fuzzing and submits the CR3 value of the fuzzing application to the host. The host is then able to utilizes the CR3 filtering capability of Intel PT (see chapter 2.4.4).

The kAFL device driver, is able to send commands to the guest system via a PIN-based interrupt. This mechanism is only used for the synchronization process and therefore no additional commands are defined. Nevertheless, before raising a PIN-based interrupt, a notifier ID is written in the MMIO-region to avoid responsibility confusion between interrupt-handlers in the guest system due to IRQ-sharing†.

3.1.4 kAFL Guest Driver

For the usage of this virtual PCI-device, a kernel driver was developed, since the interaction with a PCI-device is a privileged operation and only possible in supervisor mode. This associated kernel driver for the virtualized target OS is called *kAFL guest driver*.

The kAFL guest driver implements the PCI enumeration and exposes all kAFL guest device resources to the guest's user mode. This driver also provides the `kafl_logger()` function and installs several hooks in the OS. This component is highly OS specific and is currently only implemented for Linux x86-64. Fuzzing of other OSs using kAFL requires a port of the kAFL guest driver. To expose kAFL guest device resources to

*KASan (Kernel Address Sanitizer) is fast memory error detector for the x86-64 Linux kernel. For further information see: https://kernel.org/doc/html/latest/dev-tools/kasan.html

†https://www.kernel.org/doc/Documentation/PCI/MSI-HOWTO.txt

the user mode, the kAFL guest driver provides an `ioctl()` interface and an additional `mmap()` interface. The `ioctl()` interface is accessible via a device node and includes the following commands:

KAFL_GUEST_ACQUIRE:
This function is called by the fuzzing application and blocks from within the kernel until a new payload is copied to the Payload BAR. The notification about the finished payload copy processes is submitted via an IRQ. This mechanism circumvents any concurrency issues.

KAFL_GUEST_GET_BAR{0-2}:
The user mode is able to acquire access to all PCI-BARs, except for the kAFL Bitmap PCI-BAR. The kAFL guest driver uses a Page Frame Number (PFN) remapping to provide this memory to user mode. Consequently, this results in a three way remapping of the associated page frames, which are accessible from within the host, the guest kernel and the guest user mode.

KAFL_GUEST_READY
Initially, a loader has to copy the program from the according PCI-BAR and launch the program. But before the fuzzing application is executed, the loader program notifies via this `ioctl()` command, that the fuzzing program is ready and is going to be launched.

In addition to this, the kAFL guest driver contains the `kafl_logger()` function, which uses the same hash function as introduced by AFL (see listing 2.3). In contrast to the AFL hash function, this implementation uses the kAFL bitmap PCI-BAR as the target memory. Hence, it is possible to access the memory of the kAFL bitmap even after a crash in the target OS has occurred.

To notify the host about misbehavior, the kAFL guest driver hooks during the on-load routine the Panic- and KASan-handler function. If a handler function is called during runtime, the execution is redirected to the following code in listing 3.2 (accordingly adapted for the KASan-handler). This code directly notifies the host and afterwards halts the vCPU. Beforehand, all interrupts are disabled (by using `cli` instruction), which leads to a stopped vCPU until the host loads a VM snapshot or restart the virtualized OS.

Listing 3.2 kAFL Linux x86-64 panic handler

```
1: void kafl_panic(void){
2:         asm volatile("cli\n\t");
3:         writel(MMIO_REG_PANIC, kafl_guest_dev.regs+status_reg);
4:         asm volatile("hlt\n\t");
5: }
```

3.1.5 Inter-VM Communication

To summarize the inter-VM communication model, an initial user mode loader application interacts with the kAFL guest driver and copies the fuzzing application from Program PCI-BAR (1). The kAFL guest driver provides access to all PCI-BAR, except for kAFL Bitmap PCI-BAR, via `mmap()` (2). Afterwards, the program is written to a file and executed by using `execve()`. Figure 3.3 illustrates this process. To simplify this illustration, the interaction with Argument PCI-BAR was left out.

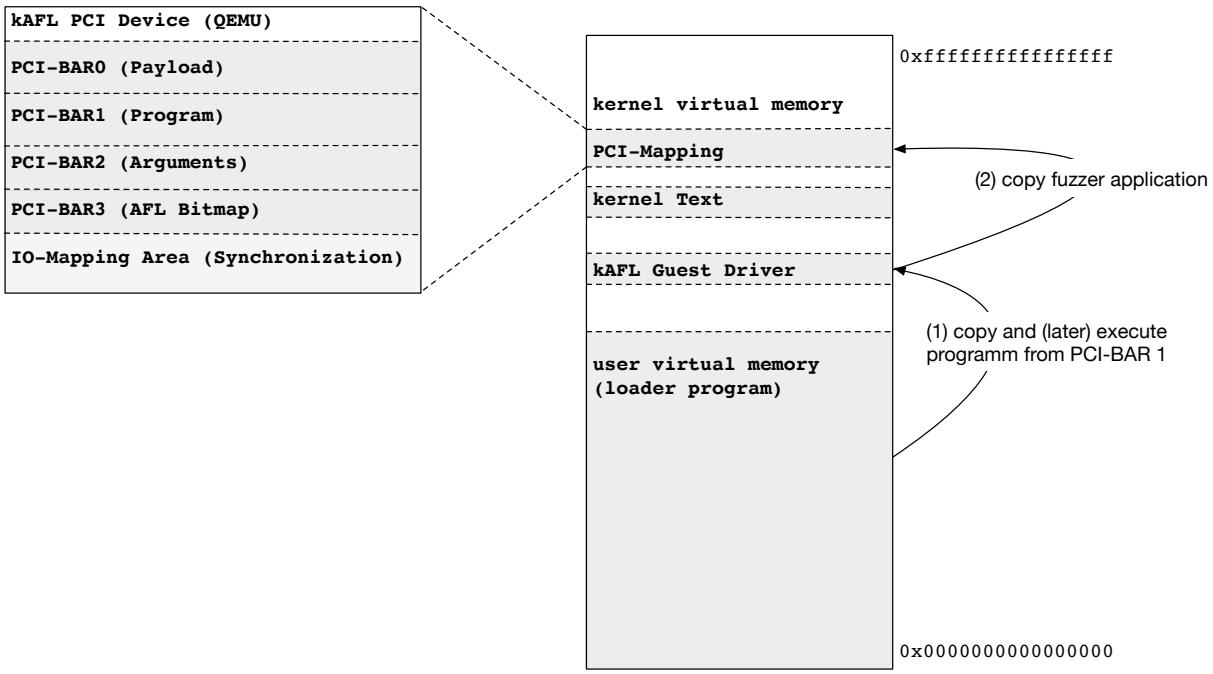

Figure 3.3: Guest User Mode Loader

Figure 3.4 illustrates the subsequent execution of the fuzzing application copied from Application PCI-BAR.
When the fuzzing application from Application PCI-BAR is ready, the application initiates the synchronization process via an `ioctl()` command `KAFL_GUEST_ACQUIRE` (1). This `ioctl()` command blocks until the payload is copied by the host and the host finishes the synchronization process via a raised IRQ (2). The fuzzing application copies the payload data and injects it via a specified fuzzing routine and a sequence of syscalls (3). Since the kernel is recompiled by using the kAFL compiler wrapper and thus partially instrumented, the execution of instrumented code will result in frequent calls of `kafl_logger()`. The function `kafl_logger()` sets coverage bits in kAFL Bitmap PCI-BAR (4).

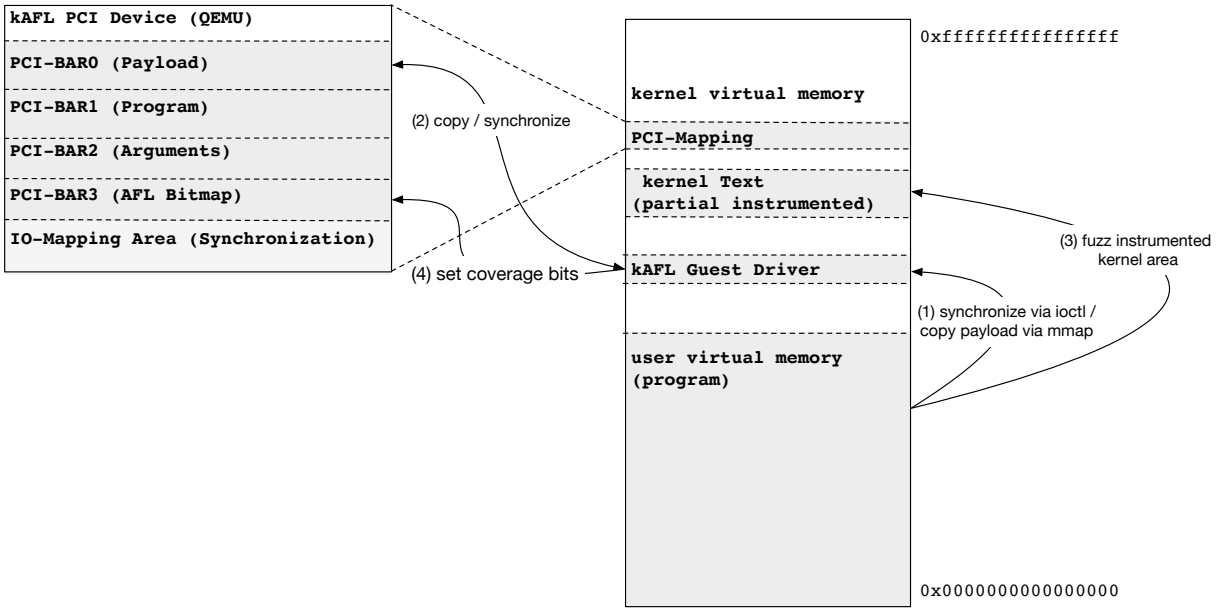

Figure 3.4: Guest Fuzzing Loop

This procedure is repeated for each fuzzing iteration. Since the fuzzing application is interchangeable from within the host, the host has only once to start the user mode loader application only once and creates a predefined VM snapshot as the starting point for the fuzzing process.

3.2 vmx_pt KVM Extension

The initial approach of the feedback-measurement essentially depends on compile-time instrumentations. Since closed-source operating systems such as Windows or macOS cannot be recompiled for this purpose, a more sophisticated approach is needed.

Utilizing Intel PT allows it to trace branch transitions without patching or recompiling the targeted kernel. Unfortunately, no publicly available driver was able to trace only VMX non-root operations using Intel PT at the time of this research. For instance, Simple-PT [Kle] does not support long-term tracing by design. The `perf`-subsystem [PPt] supports tracing of VMX non-root operations and long-term tracing. However, all publicly available Intel PT drivers for Linux are designed to trace logical CPUs*, not vCPUs. Even if VMX tracing would be supported, the data would be associated per logical CPU and not per vCPU. Therefore, the VMX context would be spread around all trace samples for each logical CPU and must be reassembled costly. An academic prototype driver, which is an extension of KVM, has been developed to fix these issues. This extension is called `vmx_pt` and provides a fast and reliable trace mechanism for KVM vCPUs. Moreover, this extension provides, such as KVM, an extensive user mode interface to expose this additional CPU feature to userland.

*Those Intel PT drivers implement *System-Wide* tracing (see figure 3.1).

3.2.1 Intel PT Aware Hypervisor

Trace samples should ideally contain only branch information of the associated vCPU. To achieve this objective, Intel has envisaged multiple trace models for Intel PT. Those trace models are described in the official PT documentation as "Common Usages of Intel PT and VMX" (see table 3.1).

Target Domain	**Output Consumer**	**Virtualize Output**	**TraceEN Configuration**
System-Wide (VMM + VMs)	Host	NA	WRMSR or XRSTORS by Host
VMM Only	Intel PT Aware VMM	NA	MSR load list to disable tracing in VM, enable tracing on VM exits
VM Only	Intel PT Aware VMM	NA	MSR load list to enable tracing in VM, disable tracing on VM exits
Intel PT Aware Guest(s)	Per Guest	VMM adds trace output virtualization	MSR load list to enable tracing in VM, disable tracing on VM exits

Table 3.1: Common Usages of Intel PT and VMX, table taken from [Intc]

Intel specifies for our best fitting trace model the target domain as *VM Only* and requires an *Intel PT Aware VMM*. For this project, KVM was chosen and extended as following to become an Intel PT Aware VMM: KVM ensures that during transitions from VMX root to VMX non-root, Intel PT tracing will be toggled and also handled. Figure 3.5 is an extended version of the already shown figure 2.2 and illustrates the above mentioned toggling mechanism.

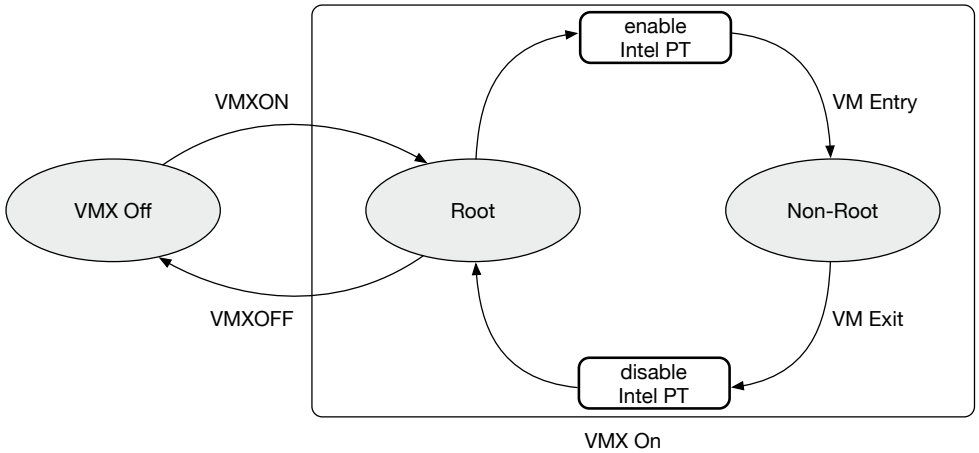

Figure 3.5: Intel PT VM Only Tracing

To enable Intel PT, software that runs within ring 0 has to modify the `TraceEn` bit of `IA32_RTIT_CTL_MSR` to 1 (see chapter 2.4). After Intel PT has been enabled, the logical CPU will trace any code that is executed if it satisfies configured filter options. The modification of `IA32_RTIT_CTL_MSR.TraceEn` has to be done before the CPU switches from VMX root to VMX non-root operation, since otherwise the CPU will execute guest code and thus is technically unable to modify any host MSRs. This procedure is also required the other way around after the CPU switches from VMX non-root to VMX root operation. However, enabling or disabling Intel PT will also result in collected trace data between the VMX mode transition and the MSR modification to enable or disable Intel PT. To circumvent the collection of unwanted trace data within the VMM, the hypervisor can exploit the MSR autoload capabilities of Intel VT-x (see chapter 2.2.5). MSR autoloading can be enabled by modifying the *VM-Entry Control Fields* as well as the *VM-Exit Control Fields*. This forces the CPU to load a list of preconfigured values for defined MSRs after either a VM-Entry or VM-Exit occurs. If MSR autoload is used, the hypervisor must instead of autonomously modifying `IA32_RTIT_CTL_MSR.TraceEn` before a VM-Entry or after a VM-Exit event occurs, just configure once the MSR autoload capability for `IA32_RTIT_CTL_MSR`. For this purpose, KVM provides the helper function `add_atomic_switch_msr()` to automatically prepare VM-Entry and VM-Exit Control for the autoload MSR feature.

3.2.2 ToPA Configuration

The *Table of Physical Addresses* (see chapter 2.4) is an essential part of `vmx_pt` and facilitates the implementation of long-term Intel PT tracing. The ToPA mechanism provides a hardware-assisted mechanism to notify software if a configured ToPA entry has been filled (using *ToPA Entry Field INT*) or to disable tracing automatically (using *ToPA Entry Field STOP*). By utilizing both capabilities, it is possible to implement a reliable long-term tracing mechanism. `vmx_pt` leverages both capabilities and creates the following ToPA configuration during usage.

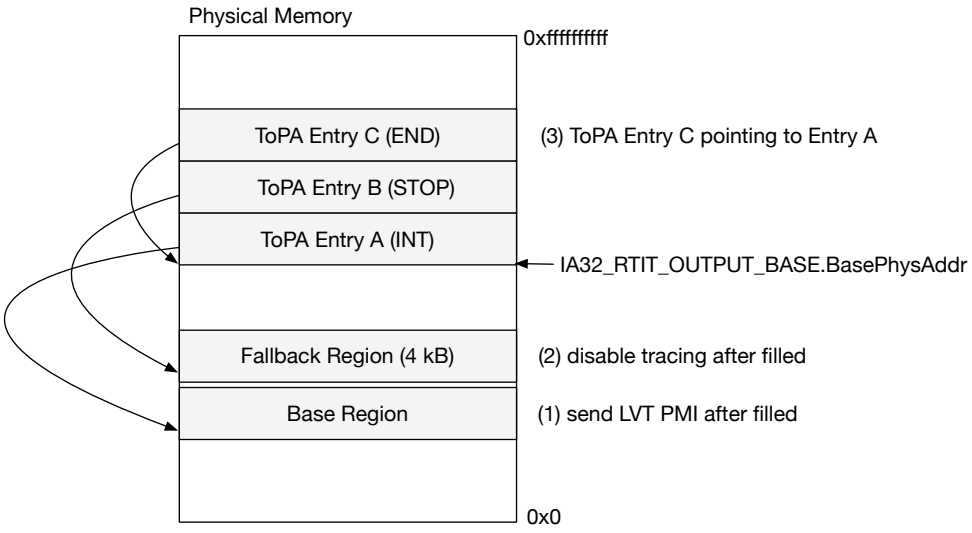

Figure 3.6: vmx_pt ToPA Configuration

As shown in figure 3.6, the first ToPA entry is pointing to the *Base Region* and sets the TopA entry field *INT*. The *Base Region* is the main buffer for arising trace data. If the associated LVT PMI is enabled, the processor will raise an interrupt if the *INT* marked entry region is filled. This LVT PMI is automatically activated by `vmx_pt` during the initial on-load routine and configured as a Non-Maskable Interrupt (NMI) according to the associated Intel Advanced Programmable Interrupt Controller (APIC) documentation [Int96]. Moreover, an interrupt handler will be registered for the LVT PMI. Since every raised NMI would result in a VM-Exit event, the PMI handler "handles" this NMI afterwards and a `vmx_pt` VM-Exit handler can save the trace data and reconfigure the ToPA for further usage. Therefore, the loss of trace data is impossible and long-term tracing would be guaranteed. Unfortunately, as already mentioned, Intel specifies this PMI as not *precise*. Thus, it cannot be ensured that the interrupt handler is called timely. To deal with this quirky issue and to avoid potential trace data losses, `vmx_pt` configures an additional fallback output region. This fallback region is by default 4 kB in size and ensures that the processor is able to keep tracing until the PMI is raised and tracing can be disabled. `vmx_pt` can later distinguish the number of written bytes within the fallback region by reading `IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset`.

In case of a *ToPA Stop* event, the processor will include an *Overflow Packet* (OVP) to the trace data after the tracing will be resumed again. During multiple empirical test runs*,

*This includes the tracing of a virtualized Linux 4.8.1 kernel during the compiling of the same kernel from source in user mode, whereas only the kernel mode was considered.

OVPs have never been found within the trace data. Therefore, it seems sufficient to use a fallback area which is 4 kB in size. At most, a fallback output area offset of nearly 1 kB has been measured during complex fuzzing test runs. In consequence, a ToPA overrun is *very unlikely*, but can be circumvented easily by increasing the size of the fallback output region.

3.2.3 Entry / Exit Handling

`vmx_pt` was designed to trace code execution between a VM-Entry and a VM-Exit event to provide coherent trace samples of the virtualized guest system. Furthermore, `vmx_pt` is also designed to allow long-term tracing. This implies that `vmx_pt` permanently checks if the ToPA base region has been overrun. Therefore, `vmx_pt` installs two function calls during the on-load-routine to be called before KVM switches the CPU to vmx non-root operations and after the CPU leaves the vmx non-root operations. Those functions are called `vmx_pt` entry and exit handler.

Listing 3.3 vmx_pt Entry Handler

```
 1: void vmx_pt_vmentry(struct vcpu_vmx_pt *vmx_pt){
 2:         if (enabled && vmx_pt && vmx_pt->configured){
 3:                 /* reset ToPA configuration if overrun occurred */
 4:                 if (vmx_pt->reset){
 5:                         vmx_pt->reset = false;
 6:                         prepare_topa(vmx_pt);
 7:                 }
 8:                 /* load vCPU specific configuration / previous ToPA state */
 9:                 vmx_pt_reconfigure_cpu(vmx_pt);
10:         }
11: }
```

Within the `vmx_pt` entry handler (see listing 3.3), `vmx_pt` checks if the base output region has been overflowed since the last guest mode execution (*line 4*). In such case, the user mode has to check if a ToPA overrun occurred and must copy or process the trace data in advance. Since the next time `vmx_pt` entry handler is called, the ToPA will be reconfigured and the current region and offset pointer be adjusted (*line 6*). As a result, the ToPA base region can be used again. Finally, the `vmx_pt` entry handler reconfigures the CPU.

Listing 3.4 vmx_pt Exit Handler

```
 1: void vmx_pt_vmexit(struct vcpu_vmx_pt *vmx_pt){
 2:         u64 topa_base, topa_mask_ptrs;
 3:         if (enabled && (vmx_pt != NULL)){
 4:                 if (vmx_pt->configured){
 5:                         /* save current ToPA state */
 6:                         rdmsrl(MSR_IA32_RTIT_OUTPUT_MASK_PTRS,
 7:                                 topa_mask_ptrs);
 8:                         WRITE_ONCE(vmx_pt->ia32_rtit_output_mask_ptrs,
 9:                                 topa_mask_ptrs);
10:                 }
11:         }
12: }
```

Unfortunately, frequent CPU reconfiguration is essential, since it is impossible to ensure that the CPU on which `vmx_pt` currently operates is the same as the last time the `vmx_pt` entry handler was called. The scheduler is able to arbitrary choose the best-fitting logical CPU for each guest mode switch. To deal with scheduling interferences, `vmx_pt` must ensure the integrity of the ToPA, Intel PT related MSR values, and other per-processor data before tracing is enabled. By saving the current state in the `vmx_pt` exit handler (*line 6 and 7 in listing 3.4*) and reconfiguring the associated MSRs in the `vmx_pt` entry handler, the Intel PT state of the associated CPU keeps consistent. Therefore, the `vmx_pt` entry handler has to reconfigure all previously saved Intel PT related MSRs each time. The `vmx_pt` exit handler saves the following per-CPU MSRs:

- `IA32_RTIT_OUTPUT_BASE`
- `IA32_RTIT_OUTPUT_MASK_PTRS`
- `IA32_RTIT_CR3_MATCH` (only if configured and used)
- `IA32_RTIT_ADDR[0,1,2,3]_[A,B]` (only if configured and used)

Besides that, the `vmx_pt` entry handler has to check also if `MSR_IA32_RTIT_STATUS.Error` was set before switching into the guest mode and resolve the issue if an error has occurred.

The vmx_pt entry handler as well as the vmx_pt exit handler are called in a critical code area. Fortunately, KVM also runs shortly before switching into guest mode into a critical code area and thus handles the synchronization. This is achieved by disabling preemption temporally before switching to guest mode and activate the preemption as soon as possible after the VMX exit event. To disable scheduling in critical code areas within KVM and avoid concurrent access of per-processor data, KVM executes preemption_disable() before and preemption_enable() afterwards*. vmx_pt exploits this mechanism and both handlers are called within this critical section. Calling the vmx_pt entry and exit handler within this section ensures that during the reconfiguration and saving procedure of all Intel PT related MSRs no concurrent access occurs.

3.2.4 Userspace Interface

To expose vmx_pt tracing capabilities to userspace, the established KVM ioctl()-interface (see chapter 2.3.3) was extended to serve this purpose. The userspace application has to acquire a new file descriptor via the ioctl()-command KVM_VMX_PT_SETUP_FD, which is sent to the vCPU related file descriptor. The kernel will then return a further vmx_pt specific file descriptor. This file descriptor provides a set of new ioctl()-commands, which provide all configurable Intel PT capabilities of the associated vCPU. To give an illustration, table 3.2 lists all new ioctl-commands.

Moreover, a mmap-interface was implemented to expose the tracing data in a zero-copy fashion. This mmap-interface directly maps both ToPA output buffers to the associated userspace virtual memory buffer via PFN remapping by using the kernel helper function remap_pfn_range()†. Since both ToPA output buffers are always 4 kB aligned, the mmap-interface is able to provide a coherent remapping of both ToPA buffers in userspace even if both ToPA output buffers are incoherently located in physical memory.

*See for further information: https://www.kernel.org/doc/Documentation/preempt-locking.txt
†For further information see: https://www.kernel.org/doc/Documentation/x86/pat.txt

Command (ioctl)	Argument	Description
KVM_VMX_PT_SETUP_FD	-	Acquire vmx_pt-fd per VCPU
KVM_VMX_PT_GET_TOPA_SIZE	-	Inquire size of ToPA region (excluding size of overflow region)
KVM_VMX_PT_CHECK_TOPA_OVERFLOW	-	Check if ToPA overflow occurs Returns overflow bytes
KVM_VMX_PT_ENABLE	-	Enable PT guest tracing for VCPU
KVM_VMX_PT_DISABLE	-	Disable PT guest tracing for VCPU
KVM_VMX_PT_ENABLE_CR3	-	Enable cr3 filtering
KVM_VMX_PT_DISABLE_CR3	-	Disable cr3 filtering
KVM_VMX_PT_CONFIGURE_CR3	Target PML4T	Configure cr3 Filtering
KVM_VMX_PT_CONFIGURE_ ADDR{0-3}_{A,B}	Target IP	Configure IP filtering for addr{0-3}_{a,b}
KVM_VMX_PT_ENABLE_ADDRN	addr{0-3}	Use addr{0-3} filtering configuration
KVM_VMX_PT_DISABLE_ADDRN	addr{0-3}	Ignore addr{0-3} filtering configuration

Table 3.2: vmx_pt ioctl() Interface

Consequently, the chosen file descriptor hierarchy is not only a clean and Unix-like interface, it does also provide full support for concurrency during vCPU tracing. Especially the separation of the allocation and management of ToPA areas, the pre- and post-handlers are the reason for this feasible implementation. As a result, vmx_pt allows tracing of an arbitrary number of concurrently executed virtual machines as well as an arbitrary number of virtual CPUs without losing the ability of generating continuous context-sensitive VMX trace samples.

3.3 QEMU PT

In order to make use of the KVM extension `vmx_pt`, a userspace counterpart is required. QEMU PT is an extension of QEMU and provides full support for `vmx_pt`. This is achieved via utilizing the newly introduced user mode interfaces. This also includes procedures to enable, disable, configure Intel PT during runtime and the periodic ToPA status check to avoid overruns.

3.3.1 vmx_pt Integration

As already mentioned in chapter 3.2.4, `vmx_pt` is accessible from within the user mode via `ioctl()`-commands and an additional `mmap()`-interface. The communication of QEMU and KVM is also based on `ioctl()`-commands and `mmap()` calls and is located within a more sophisticated implementation of the already shown KVM loop in chapter 2.3 (see listing 2.6).

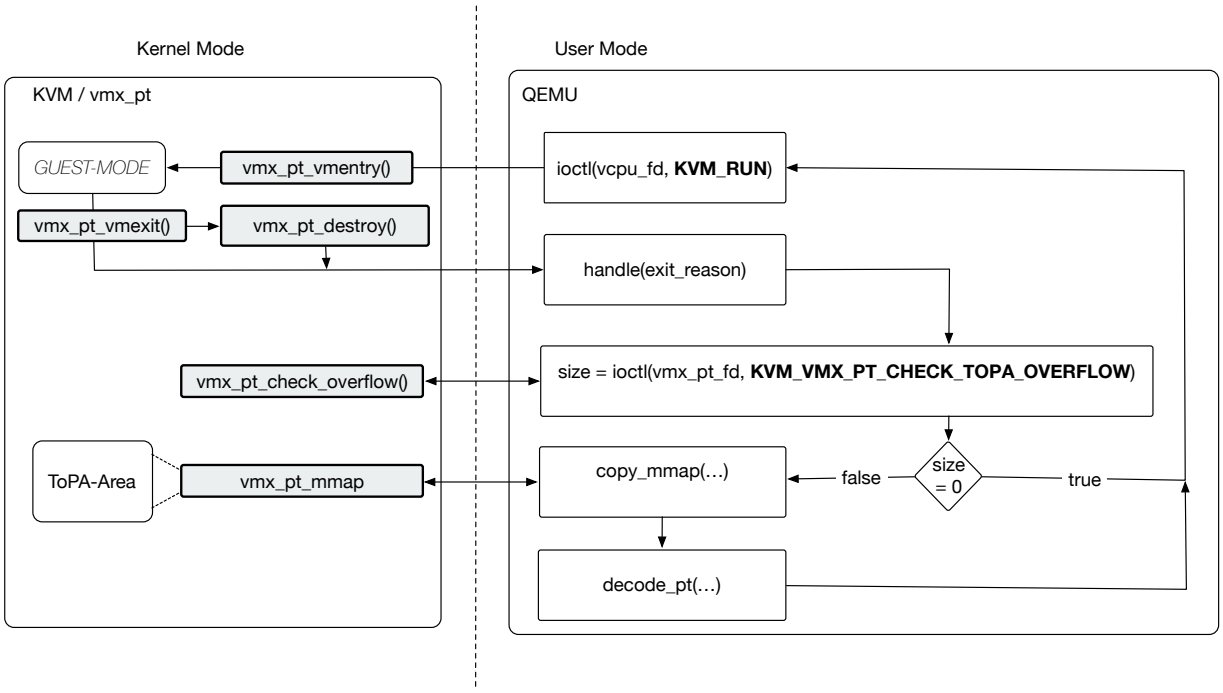

Figure 3.7: KVM / vmx_pt and QEMU Execution Loop

To configure `vmx_pt`, QEMU PT has to initially obtain a `vmx_pt` file descriptor from the vCPU file descriptor and send appropriate `ioctl()` commands via this file descriptor to configure `vmx_pt` once (see chapter 3.2.4).

To circumvent any loss of trace data, the user mode component has to check after each KVM_RUN iterations, if the ToPA Base region of vmx_pt is overflowed. In such case, the user mode has to process the data or copy data before KVM_RUN is called the next time. As illustrated in figure 3.7, QEMU PT has to check the ToPA Base region state immediately after the VM-Exit reason is handled.
If the ioctl() command KVM_VMX_PT_CHECK_TOPA_OVERFLOW returns a non-NULL value, the ToPA buffer is full and the trace data must be copied via mmap() or processed in-place. The used Intel PT software decoder, which is introduced in chapter 3.4, processes the data in-place and avoids any superfluous coping. This approach provides a zero-copy characteristic thanks to the mmap-mapping.

From a more technical perspective, the function kvm_cpu_exec(CPUState *cpu), which implements a more extensive version of the KVM-loop, was extended to provide full support for vmx_pt. Since this loop is processed each time KVM executes guest code and the VM-Exit reason is handled, function calls to a vmx_pt handlers at the prolog and epilog of the loop are installed (see listing 3.5). Via those vmx_pt handlers any configuration requests from the user or kAFL are processed in the prolog-handler.

Listing 3.5 Extended KVM-loop within ./kvm-all.c

```
1: int kvm_cpu_exec(CPUState *cpu)
2: {
3:         /* ... */
4:         do {
5:                 /* ... */
6:                 pt_pre_kvm_run(cpu); /* VMX_PT prolog handler */
7:                 /* ... */
8:                 run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
9:                 /* ... */
10:                switch (run->exit_reason) {
11:                        /* ... handle exit reasons */
12:                }
13:                pt_post_kvm_run(cpu); /* VMX_PT epilog handler */
14:        } while (ret == 0);
15:        /* ... */
16: }
```

In the `vmx_pt` handler, called at the epilog of the KVM-loop, the handler checks whether the main ToPA Buffer has overflowed or not. If the ToPA has been overflowed, the off-byte of the ToPA fallback region size will be provided and QEMU-PT will process the trace data.

To provide support for the instrumentation-based implementation of kAFL, the fuzzer interface of kAFL was adapted to support `vmx_pt`. In contrast to the old implementation, the fuzzer has to specify the desired IP-filter ranges. The kAFL guest driver of QEMU PT does also provide a kAFL bitmap. The kAFL bitmap is not represented by a PCI-BAR, but it is generated by the Intel PT software decoder based on traced branch transitions.

3.3.2 Management Interface

The QEMU Monitor, also known as QEMU Management Console*, allows the monitoring and controlling of running VMs. This text-oriented interface provides the ability to create or load VM snapshots during execution, hot-plug new peripheral devices or even hot-plug new vCPUs. To offer a direct way to communicate with the `vmx_pt` extension during execution, additional QEMU Management Console commands are added to QEMU PT. This includes the following commands:

Command (`hmp`)	**Arguments**	**Description**
`pt enable`	vCPU ID	Enable tracing for specified vCPU
`pt enable_all`	-	Enable tracing for presented vCPUs
`pt disable`	vCPU ID	Disable tracing for specified vCPU
`pt disable_all`	-	Disable tracing for presented vCPUs
`pt status`	vCPU ID	Print PT status information of the vCPU context (PT enable status, ToPA Overflows, Trace Size)
`pt status_all`	-	Print PT status information of all vCPU contexts
`pt ip_filtering`	vCPU ID, ADDRn ID, ADDR_A, ADDR_B	Configure PT IP filtering
`pt cr3_filtering`	vCPU ID, PML4T	Configure `cr3` Filtering
`pt set_file`	Trace File	Write trace data to specified files

Table 3.3: QEMU PT Management Console Commands

*https://en.wikibooks.org/wiki/QEMU/Monitor

This interface provides the ability to manually record Intel PT samples of vCPU activity for various purposes. This might be useful for further projects, which are based on `vmx_pt` and QEMU PT. The following listing shows a sample interaction with via QEMU Management Console.

Listing 3.6 Sample Interaction with the Management Console of QEMU PT

```
QEMU PT 2.6.0 monitor - type 'help' for more information
(qemu) pt set_file /tmp/pt_trace_data
(qemu) pt ip_filtering 0 0 0xffffffff83a00000 0xffffffff83a26160
CPU 0: ip filtering enabled...
(qemu) pt enable_all
CPU 0: processor trace enabled!
(qemu) pt status 0
Processor Trace Status (CPU 0)
       enabled:  yes
       ToPA overflows:  24
       trace data size: 12583936 (12MB)
       pt_ip_filter_0_a: 0xffffffff83a00000
       pt_ip_filter_0_b: 0xffffffff83a26160
(qemu) pt disable 0
CPU 0: processor trace disabled!
```

After executing the commands shown in listing 3.6, QEMU will create the following 3 files:

1. `/tmp/pt_trace_data_raw_cpu0`
2. `/tmp/pt_trace_data_decoded_cpu0`
3. `/tmp/pt_trace_data_code_cpu0_0`

All files indicate the associated vCPU by the postfix (`_cpu[id]`). The `raw` file contains the data copied from the ToPA output buffers without applied post-processing. The `decoded` file contains a list of executed *Conditional Branch* CoFI opcodes (see table 2.1). Finally, the `code` file stores the associated code, which was executed related to the configured IP filter (`0xffffffff83a00000` - `0xffffffff83a26160`) . If no IP filter was configured, only the `raw` file will be created.

3.4 JIT-Decoder

Extensive kernel fuzzing may generate several hundreds of megabytes of trace data per second. To deal with such large amounts of incoming data, the decoder must be implemented with a focus on efficiency. Otherwise, the decoder would become the major bottleneck during the fuzzing process and limit the effective performance. Nevertheless, the decoder must also be precise, as inaccuracies during the decoding process would result in further errors. This is due to the nature of Intel PT decoding, since the decoding process is sequential and is affected by previously decoded packets.

To limit effort to implement an Intel PT software decoder, Intel provides its own decoding engine called `libipt`*. `libipt` is a general-purpose Intel PT decoding engine, but it does not fit our purposes very well, since `libipt` decodes trace data in order to provide execution data and flow information. Furthermore, `libipt` does not cache disassembled instructions and provides an overall poor performance compared to our approach†.

Since kAFL relies only on flow information and the fuzzing process is applied on the same "application", it is possible to optimize the decoding process. The developed Intel PT software decoder acts like an "JIT-Decoder", which means that code sections are only considered if they are executed according to the decoded trace data. To optimize further lookups, all disassembled code sections are cached.

*https://github.com/01org/processor-trace

†According to Shlomi Oberman and Ron Shina [OS16], the decoding process of *several hundreds megabytes* of trace data using `libipt` would require *several hours* of processing.

3.4.1 Decoding of Trace Data

kAFL only requires information about the executed control flow, whereas Intel PT also provides execution information such as TSC, MNT and CBR. For our use, it is sufficient to focus on flow-information packets only. Thus, Intel PT packets are not considered for further processing. However, the decoding process itself is required to determine the offset to the following packet, since Intel PT packets do not have a uniform packet size. The following algorithm represents the decoding process of this Intel PT JIT-Decoder:

Algorithm 2: Decoder Algorithm

```
1  last_IP ← 0
2  while pkt = decode_packet(buffer) do
3      if pkt.type == TNT then
4          TNT_cache.append(pkt.value)
5      else if pkt.type == TIP_PGE then
6          last_ip = pkt.value
7      else if pkt.type == TIP or
8      pkt.type == TIP_PGD or
9      pkt.type == TIP_FUP then
10         follow_and_dissemble(last_IP)
11         last_ip = pkt.value
12 end
```

The buffer containing the Intel PT trace sample is decoded into a sequence of Intel PT packets. Only a small subset of Intel PT packets are considered and handled. The values of TNT packets are always cached (*line 3-4*), since, depending on the presented CPU model, the CPU will cache those packets as well. Therefore, it might be necessary to cache TNT packets for later use. TIP.PGE indicates the beginning of a trace session and the IP value is saved for later use (*line 5-6*). If a TIP, TIP.PGD or TIP.FUP packet is decoded, the decoder executes the function `fetch_and_follow()` and processes the previously saved IP (`last_IP`) and all cached TNT packets (*line 7-11*).

3.4.2 Binary Disassembling and Model Transfer

The JIT-Decoder has to decode a large number of trace samples related to the same disassembly during the fuzzing process. Using a general-purpose Intel PT decoding engine is non-optimal, since the engine would disassemble the code on every appearance, even if the same basic blocks are processed. Instead, it is more efficient to cache *interesting* instructions and disassemble them only once. This comes at the cost of a much higher memory consumption, but significantly reduces CPU load during further processing and look up. However, the higher memory consumption was found to be tolerable, since the JIT-Decoder only focuses on CoFIs and the targeted kernel code area is relatively small* in most use cases.

To cache CoFIs, the decoder has to disassemble unknown code sections and transfer CoFIs into a special data structure (see listing 3.7). The JIT-Decoder uses a data structure (`cofi_list`) to represent an object of a linked list for memory management purposes and to track all allocated CoFI objects (`list_header`) in memory. Such objects are generated during runtime, each time an unseen code section is disassembled. Another pointer references to the next decoded CoFI object (`cofi_ptr`), which is used to fetch the next CoFI if a jump instruction was not "taken". The counterpart pointer (`taken_ptr`) is set during runtime if a jump was actually taken, otherwise this pointer is set to zero.

Listing 3.7 JIT-Decoder Data Structures

```
typedef struct {
        uint64_t addr;                /* opcode address */
        uint64_t taken_addr;          /* (jmp) target address */
        cofi_type type;               /* CoFI Type */
} cofi_header;

typedef struct cofi_list {
        struct cofi_list *list_ptr;  /* Pointer to the next list element */
        struct cofi_list *cofi_ptr;  /* Pointer to the next CoFI */
        struct cofi_list *taken_ptr; /* Pointer to the "taken" CoFI */
        cofi_header *cofi;           /* Pointer to the CoFI data */
} cofi_list;
```

*To give an example: The core of the Linux kernel (`Linux Debian 4.8.5-1-amd64`) is $\approx 13MB$ in size. This does not include additional kernel modules.

Each time the disassembler decodes an unknown code section, every CoFI is translated into those objects. The decoding process is terminated if the decoder is unable to decode the next instruction or the end of a function appears. During this process, every CoFI is categorized into one of the following CoFI types:

1. `CONDITIONAL_BRANCH`
2. `COFI_TYPE_UNCONDITIONAL_DIRECT_BRANCH`
3. `COFI_TYPE_INDIRECT_BRANCH`
4. `COFI_TYPE_NEAR_RET`
5. `COFI_TYPE_FAR_TRANSFERS`

There is one exception for non-CoFIs. If a regular instruction is located on an unprocessed target address, an object is also created with the `cofi_ptr` reference to the next CoFI in the disassembled binary. Non-CoFI occurrences are typed as `NO_COFI_TYPE`. Furthermore, `COFI_TYPE_INDIRECT_BRANCH`, `COFI_TYPE_NEAR_RET`, `COFI_TYPE_FAR_TRANSFERS` are heuristics to indicate the end of a function.

In parallel, a hash map is created and filled with all `cofi_list` objects referenced by the opcode address as key value. This hash map is based on kHash* and is used for fast lookups of CoFI target addresses, which are unknown during the disassembly process. This includes the occurrence of `COFI_TYPE_INDIRECT_BRANCH`, `COFI_TYPE_NEAR_RET` and `COFI_TYPE_FAR_TRANSFERS`. If an object is not found in the hash table, the located code section is not yet disassembled and processed. To limit the number decoded and processed CoFIs during runtime, the taken address of `CONDITIONAL_BRANCH` is not considered. However, if a taken address is resolved and disassembled during runtime, a reference in `cofi_list` is set to prevent further effort (using `taken_ptr`).

Figure 3.8 illustrates this process. This example is based on the application shown in listing 2.7 and the trace data shown in listing 2.8. The `cofi_header` objects are ordered relatively to the opcode address, whereas the `cofi_list` objects are numbered chronological order. Note that the `NO_COFI_TYPE` objects are allocated later during runtime, since those objects are only generated if the decoder follows an address which is not a CoFI.

*https://github.com/attractivechaos/klib/blob/master/khash.h

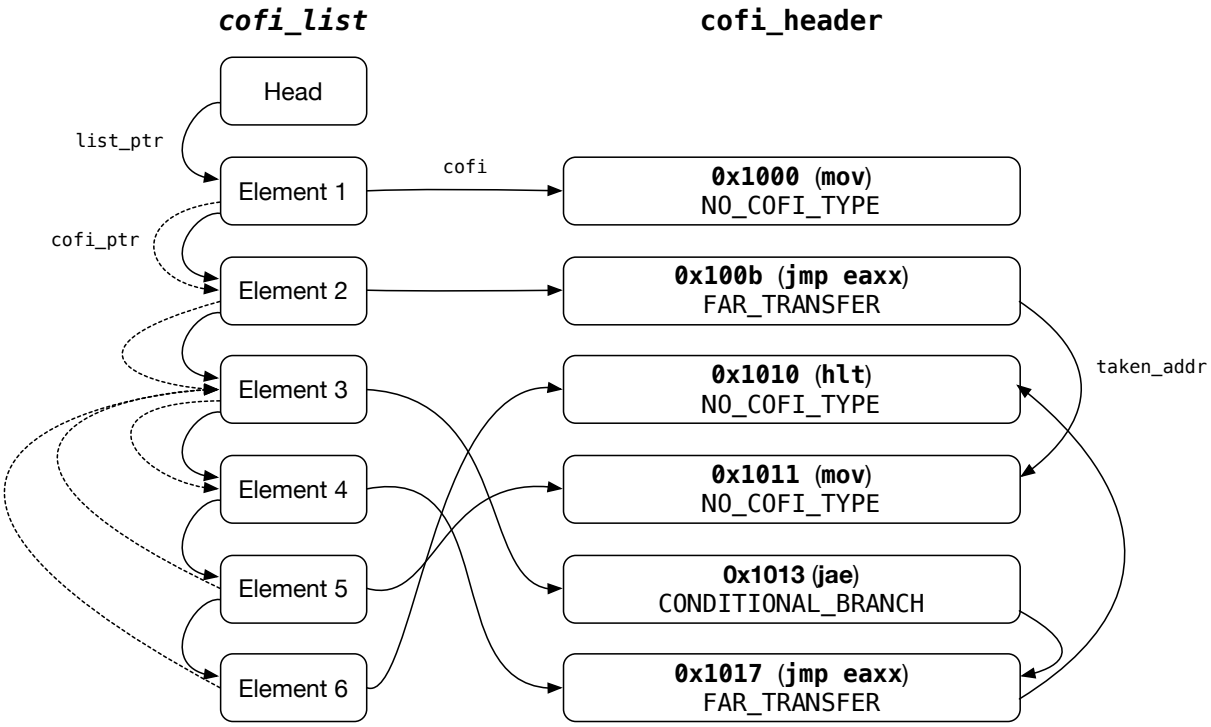

Figure 3.8: JIT-Decoder Data Structure

To disassemble x86-64 code, the Capstone Engine* was used. Moreover, if kAFL configures one of four possible IP ranges (using Intel PT IP-filter options) during runtime, the guest's memory located at this IP-range is carved out and considered as the associated application for Intel PT decoding. Since every commonly used x86-64 OS uses a *split virtual memory layout*†, the kernel memory is mapped to any virtual address space and therefore is located *always* at the same virtual address.

*http://www.capstone-engine.org

†This means that the kernel is commonly located at the upper half of the virtual memory space. The virtual memory space of linux is typically split into kernelspace (upper half) and userspace (lower half) to the size of 2^{47} each, due to the 48-bit virtual address limit of current x86-64 CPUs.

3.4.3 Fetch and Follow

Since every code section is disassembled and translated into another data representation once, the JIT-Decoder is able to apply a *fetch and follow* approach on this data. This means, that the decoder has to fetch the next CoFI object address based on the new data representation and follow this address to get the next CoFI object. Depending on the CoFI type, other procedures are used to fetch the following CoFI object address. The following algorithm describes those procedures depending on the CoFI type based on the data representation shown in listing 3.7.

Algorithm 3: follow_and_dissemble() Algorithm

Input: last_IP

```
1  obj = get_or_disas(last_IP)
2  while true do
3      if obj.type == COFI_TYPE_CONDITIONAL_BRANCH then
4          tnt = tnt_cache_get()
5          if tnt.type == TNT_CACHE_EMPTY then
6              exit
7          set_kAFL_bitmap(obj.addr)
8          if tnt.type == TNT_CACHE_TAKEN then
9              obj = get_or_disas(obj.taken)
10         if tnt.type == TNT_CACHE_NOT_TAKEN then
11             obj = obj.cofi_ptr
12     else if obj.type == COFI_TYPE_UNCONDITIONAL_DIRECT_BRANCH then
13         obj = get_or_disas(obj.taken)
14     else if obj.type == COFI_TYPE_INDIRECT_BRANCH or
15     obj.type == COFI_TYPE_NEAR_RET or
16     obj.type == COFI_TYPE_FAR_TRANSFERS then
17         exit
18     else if obj.type == NO_COFI_TYPE then
19         obj = obj.cofi_ptr
20 end
```

The function `get_or_disas()` is called to fetch an already existing CoFI object based on a virtual address of the guest's memory (not the pointer to the next CoFI object) from

the hash map or otherwise disassemble and processes the code section. Conditional Branch instructions and a cached *not-taken* TNT value would result in following the next CoFI instruction using `cofi_ptr` (*line 10-11*). In case of a *taken* TNT value, the next CoFI object will be obtained via `get_or_disas()` (*line 8-9*). To minimize wasteful hash map accesses, the `taken_ptr` will be set during the first time executing `get_or_disas()` for next iterations. If the TNT-Cache is empty, the function is exited and the JIT-Decoder has to decode more Intel PT packets in order to reconstruct the control flow (*line 5-6*). The Unconditional Direct Branch instruction obviously just have one pointer to follow, due to the always taken jump (*line 12-13*). The `taken_ptr` will also be set during the first call of `get_or_disas()` to avoid hash map accesses. Indirect Branch, Near Ret, and Far Transfers CoFIs are handled differently. Since the target addresses are only obtainable during runtime, those CoFI types relies on TIP, TIP.PGD or FUP Intel PT packets, which provides the target address. Therefore, the function is exited, since the decoder must find such Intel PT packet first(*line 14-17*). Finally, if a `NO_COFI_TYPE` object is processed, the algorithm follows the `cofi_ptr` to the next CoFI object (*line 18-19*).

3.4.4 Bitmap Translations

During the decoding process, the function `set_AFL_bitmap()` is called on occurrences of `COFI_TYPE_CONDITIONAL_BRANCH` typed CoFIs and submits the opcode address of the CoFI (*line 7, algorithm 3*).

Every `COFI_TYPE_CONDITIONAL_BRANCH` typed CoFI indicates a new basic block transition. Since the non-altering Intel PT approach does not provide *compile-time random values*, it is not possible to reuse the traditional AFL bitmap approach. Nevertheless, since opcode addresses within the kernel space are always unique, those are a suitable alternative for the required compile-time random. The following listing represents the modified AFL hash function. Note that only the 16 least significant bits of the opcode address are considered due the limited bitmap size of 64 kB (2^{16}).

Listing 3.8 Modified AFL hash function

```
1: #define BITMAP_SIZE 1<<16 /* 64 KB */
2: uint64_t bitmap_last_key;
3: uint8_t* bitmap = NULL;
4:
5: /* ... */
6:
7: static void pt_bitmap(uint64_t addr){
8:         if(bitmap){
9:                 bitmap[(addr ^ bitmap_last_key) & 0xffff]++;
10:                bitmap_last_key = addr >> 1;
11: }
```

This approach for kernel fuzzing was initially proposed by Vegard Nossum*. Originally, this approach with a more sophisticated hash function was also used for the *Binary-only instrumentation* fuzzing of AFL[Zal]. As demonstrated by the NCC group†, it might be necessary to increase the bitmap size to circumvent hash collisions during kernel fuzzing.

*http://lkml.iu.edu/hypermail/linux/kernel/1605.2/03665.html

†ProjectTriforce is based on the same approach and uses bitmap that is 1MB (2^{20}) in size to avoid too much hash collisions during fuzzing [HN16b].

3.4.5 Trace Data Sanitization

During extensive kernel fuzzing, it is possible that non-deterministic code areas are traced and thus part of the trace data. As already mentioned in chapter 2.1.7, tracing non-deterministic code activities would clobber the kAFL bitmap and result in false positive path detections. To circumvent bitmap clobbering, the trace data must be sanitized during the decoding process.

Code Execution in Interrupt Context

One of the major part of the sanitization post-processing is to detect and opt out occurring asynchronous events from the decoding process, such as soft Interrupt Service Routines (ISRs). The most commonly known software ISR is the task-switch, which may happen during user mode or even kernel mode execution. Because the processor produces a FUP Intel PT packet after the occurrence of an ISR, it is possible to determine the beginning of an ISR execution based on the Intel PT trace data. The following Intel PT signature can be used as a heuristic to determine the beginning of an ISR:

Listing 3.9 Interrupt / Asynchronous Event (Intel PT Signature)

```
1: FUP <Address of the previously executed instruction>
2: TIP <Address of the handler for the occurred asynchronous event>
```

After the occurrence of an asynchronous event, the decoder has to keep decoding the trace data and find the exit of the ISR. During this process, the decoder does not set any bits in the kAFL bitmap to circumvent bitmap clobbering. Typically, an ISR is leaved by executing the `iret` instruction (*Interrupt Return*). If the decoder processes an `iret` instruction, the ending of the ISR is found. Note that the x86-64 platform also supports *ISR nesting*. Therefore, the decoder must track the level of nested ISRs and thus find the corresponding number of `iret` instructions.

Code Execution Entry Point Consideration

If more than one IP-filtering range is configured, especially if a widely used kernel helper function is within one of this area, bitmap clobbering might be possible. This could happen, if a kernel module and a kernel helper function, which is frequently called by other kernel modules, is within IP-filter ranges and thus are traced.

To circumvent bitmap clobbering during the execution of kernel helper functions, the decoder must track the previous address before the helper function is entered. This allows it to distinguish if the helper function is called by the traced kernel module or not. Due to the previous address, it is possible to decide whether the traced execution of the kernel helper function should be considered or not.

Non-Deterministic Code Execution in Non-Interrupt Context

The execution of stateful kernel code could lead to bitmap clobbering, since kAFL does only reload a VM snapshot if a misbehavior occurs due to performance reasons. Therefore, a mechanism is required to spot non-deterministic code execution in non-interrupt context. This includes, for instance, Linux kernel helper functions such as `kmalloc()` and `schedule()` (in non-interrupt context). Since Intel PT does not provide further information about the execution of non-deterministic code areas, the detection of such areas is difficult to achieve.

A promising approach would be the frequent sampling of the same fuzzing iteration using the same input. Based on this data, the decoder can calculate a symmetric difference set of all occurred branch transition and use this set as heuristic to not consider those areas in further iterations. This process must be repeated for each input, which potentially result in a new path transition. However, this approach does not ensure that also deterministic areas falsely marked as non-deterministic and opt out from further consideration. Unfortunately, the challenge of implementing a reliable detection of non-deterministic code areas is currently not solved.

Chapter 4

Evaluation

The evaluation of this thesis discusses the performance of all kAFL components. This includes comparisons with related projects. The performance overhead of `vmx_pt` is discussed in chapter 4.1. Secondly, a performance comparison of the JIT-Decoder and an Intel implementation of a software decoder is given in chapter 4.2. Finally, the overall fuzzing performance of kAFL is compared to ProjectTriforce, which is based on the emulation backend of QEMU instead of hardware-assisted virtualization and Intel PT (see chapter 4.3). Moreover, chapter 4.4 provides an overview of all reported vulnerabilities, which were found during the development process by kAFL.

The following benchmarks are performed on a desktop system based on an Intel i5-6500 processor and 8GB DDR4 RAM. Furthermore, all benchmarks are performed on a ram disk to avoid high influences of poor I/O performance.

As of time of writing, the development of kAFL is not finished and still ongoing. Therefore, it is hard to predict, whether the following benchmark results could be improved by further optimization or not. Furthermore, it can be assumed that even more vulnerabilities will be found by using kAFL in the future, since the presented vulnerabilities are found unintentionally during development.

4.1 vmx_pt Overhead

The KVM extension `vmx_pt` adds an overhead to the raw execution of KVM. Therefore, the performance overhead was compared with several `vmx_pt` setups. This includes `vmx_pt` in combination with the JIT-Decoder, `vmx_pt` without the JIT-Decoder but processing a frequent ToPA state checks (using `KVM_VMX_PT_CHECK_TOPA_OVERFLOW`, see chapter 3.2.4) and `vmx_pt` without any ToPA consideration. For this benchmark, a 13MB kernel code range was configured via IP-filtering ranges and traced with one of the aforementioned setups of `vmx_pt`. Those benchmarks consider only the kernel core, but neither any kernel module. During `vmx_pt` execution only supervisor mode was traced.

To generate Intel PT load, `QEMU 2.6.0` was compiled within a traced VM*. This benchmark was executed on a single vCPU. The resulting compile-time was measured and compared. The following figure illustrates the relative overhead compared to KVM execution without `vmx_pt`.

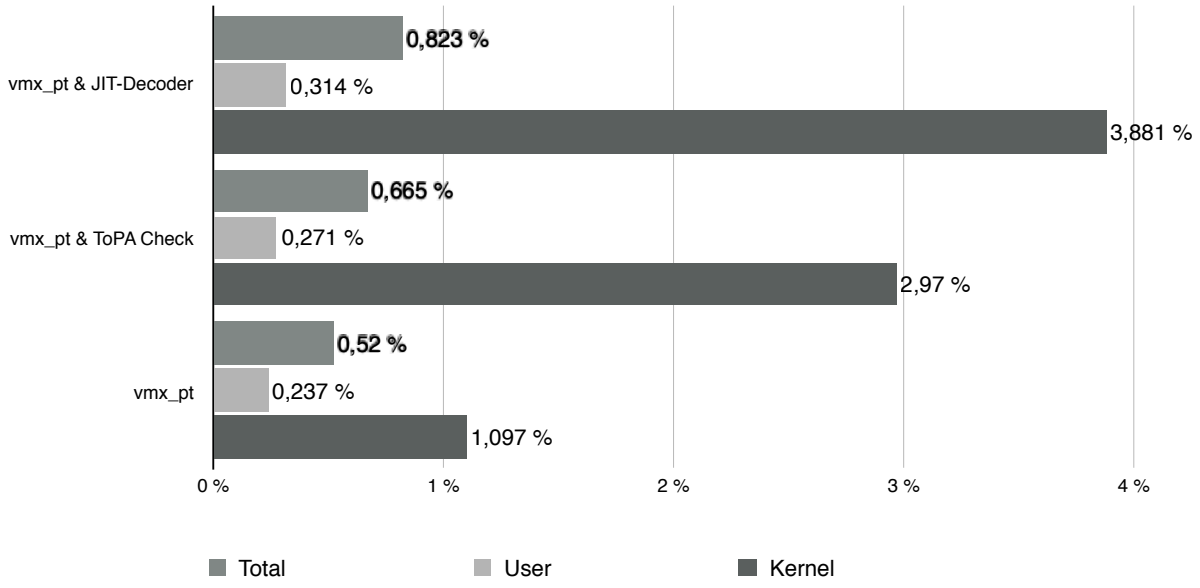

Figure 4.1: `vmx_pt` Overhead Comparison (Compiling QEMU-2.6.0)

Intel describes a performance penalty of < 5 % compared to execution without enabled Intel PT. During our benchmarks, an overhead in between of 1% - 4% was measured.

*The `./configure` option `--target-list=x86_64-softmmu` was used.

Since the resulting overhead is so small, it is not expected that it has major influence in the overall fuzzing performance.

4.2 Decoder Engine

In contrast to `vmx_pt`, the decoder has a significant influence on the overall performance of the fuzzing process, since the decoding process is, other than Intel PT and hence `vmx_pt`, not hardware-accelerated. Therefore, this process is costly and has to be as efficient as possible. Consequently, the performance of the developed JIT-Decoder was compared to `ptxed`*. This decoder is Intel's example implementation of an Intel PT software decoder and is based on `libipt` and Intel XED†. To compare both decoder engines, a small Intel PT trace sample was generated by executing

```
find / > /dev/null 2> /dev/null
```

within a Linux VM traced by `vmx_pt`‡. The generated sample is 9.4*MB* in size and contains over 431,650 TNT packets, each represents up to 7 branch transitions. Furthermore, the sample also contains over 100,045 TIPs. To compare different aspects of the decoder engines, the sample was used as a *raw* copy and a *sanitized* copy. The raw copy of the sample represents the raw content, whereas the sanitized sample only contains *flow information packets* (see chapter 2.4.2). The file is hence smaller and only 5.3*MB* in size. This is used to avoid any influence of decoding large amount of *execution information packets* (see chapter 2.4.1), since those are not considered by the JIT-Decoder. Furthermore, both samples were used to generated larger files, whereas those files contains the sample data multiple times. This includes test cases for 1, 5, 10, 50 and 250 copies per file. This is used to demonstrate the effectiveness of the *fetch and follow* approach of the JIT-Decoder. The following diagram illustrates the measured *speedup* of the JIT-Decoder compared to `ptxed`:

*https://github.com/01org/processor-trace

†https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library

‡Using the following kernel: `Linux debian 4.8.0-1-amd64`. Moreover, only code execution in supervisor mode was traced.

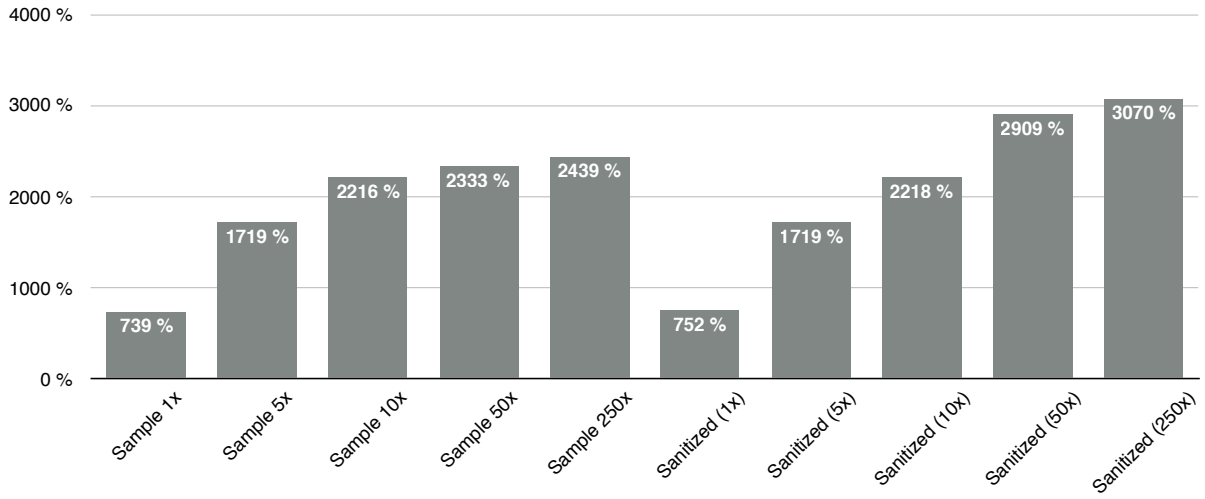

Figure 4.2: JIT-Decoder and Intel `ptxed` Performance Comparison

Figure 4.2 shows that the JIT-Decoder easily outperforms the Intel decoder implementation - even if the JIT-Decoder processes data for the very first time. This could be due to the use of the Capstone Engine as the instruction decoding backend. Furthermore, the decoding process of multiple sample copies within the same file provides even better performance. Since this is a common fuzzing use case, due to the generation of inputs triggering the same or similar path within the traced code section, it also shows that the JIT-Decoder is more suitable. Eventually, the *fetch and follow* approach outperforms Intel's implementation and is up to 25 to 30 times faster.

4.3 Fuzzing Performance

Finally, the overall performance of kAFL is compared to the ProjectTriforce kernel fuzzer. Unfortunately, it was not possible to compare Oracle's file system fuzzer due to technical issues. To compare ProjectTriforce with kAFL, the associated *TriforceLinuxSyscallFuzzer*[*] was slightly modified to work with a special kernel module. This kernel module was used for this benchmark and creates during the on-load routine the file `/proc/vuln`. Any write attempt to this file will execute the code shown in listing 4.1.

[*]https://github.com/nccgroup/TriforceLinuxSyscallFuzzer

Listing 4.1 Sample Kernel Module for Benchmark Purposes

```
 1: if (copy_from_user(i, buff, len))
 2:         return -EFAULT;
 3:
 4: if(i[0] == 'K' && i[1] == 'E' && i[2] == 'R' && i[3] == 'N' && i[4] == 'E' &&
 5:    i[5] == 'L' && i[6] == 'A' && i[7] == 'F' && i[8] == 'L')
 6:                 panic("Oops..."); /* testcase #1 */
 7:
 8: if(i[0] == 'F' && i[1] == 'U' && i[2] == 'Z' && i[3] == 'Z' && i[4] == 'I' &&
 9:    i[5] == 'N' && i[6] == 'G')
10:                 panic("Oops..."); /* testcase #2 */
11:
12: if(i[0] == 'K' && i[1] == 'A' && i[2] == 'S' && i[3] == 'A' && i[4] == 'N')
13:                 kfree(array); array[0] = 1234; /* use-after-free (KASan testcase) */
```

The input is compared bytewise and in case of one of the following three inputs, the kernel will crash*: `KERNELAFL` (*line 4-6*), `FUZZING` (*line 8-10*) or `KASAN` (*line 12-13*). The following diagram in figure 4.3 shows the overall performance (measured in *tests per second*) of kAFL based on compile-time instrumentations and kAFL using `vmx_pt` compared to ProjectTriforce. To conclude, the hardware-accelerated virtualization and tracing provides up to 50 times better performance compared to QEMU's CPU emulation. This applies to single process execution as well as for multi process execution. Besides that, the instrumentation-based approach is up to 40% slower than the Intel PT accelerated approach.

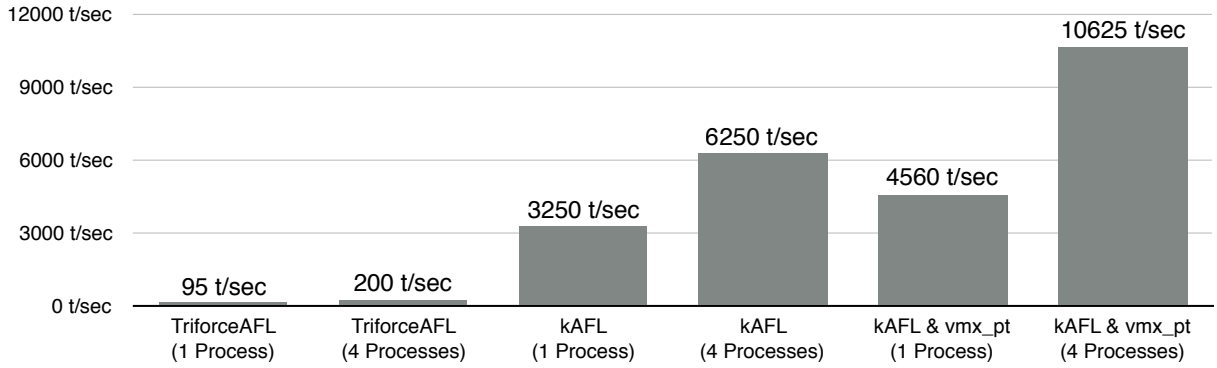

Figure 4.3: Performance Comparison

*Except for the last input, since this case is used to represent a use-after-free vulnerability and is usually only detectable if KASan is enabled.

4.4 Vulnerabilities

During the development process of kAFL, multiple security vulnerabilities were found. In total, 3 bugs were reported to Red Hat to this day:

1. Local DoS: Linux Kernel Nullpointer Dereference via keyctl* (CVE-2016-8650†)
2. Local DoS: Linux Kernel EXT4 Memory Corruption‡
3. Local DoS: Linux Kernel EXT4 Error Handling§

Red Hat has assigned a CVE number for the first reported security flaw, which triggers a null pointer deference and a partial memory corruption in the kernel ASN.1 parser if a RSA certificate with a zero exponent is presented. For the second reported vulnerability, which triggers a memory corruption in the EXT4 file system, a mainline patch was proposed¶. The last reported vulnerability, which calls in the EXT4 error handling routine `panic()` and hence results in a kernel panic, was currently not investigated any further.

Those bugs were found by the instrumentation-based kAFL implementation. Currently only a small subset of all found bugs are reported. More bug reports will follow in the near future.

*http://seclists.org/fulldisclosure/2016/Nov/76
†https://access.redhat.com/security/cve/cve-2016-8650
‡http://seclists.org/fulldisclosure/2016/Nov/75
§http://seclists.org/bugtraq/2016/Nov/1
¶http://www.spinics.net/lists/linux-ext4/msg54572.html

Chapter 5

Future Work

The current implementation of kAFL has several limitations. Potential approaches to bypass these limitations are research fields for future work.

5.1 Sanitization of Non-Deterministic Code Traces

As already mentioned in chapter 3.4.5, tracing of non-deterministic code sections would result in a large amount of false positive path findings, due to bitmap clobbering. To deal with this issue, it might be sufficient to black-list code areas. This task would required manual work and comprehensive knowledge about non-deterministic code sections of the targeted OS. Since this approach contradicts the general-purpose aim of kAFL, another more sophisticated approach is desired. Currently multiple promising approaches are evaluated and developed to deal with this issues in an automated fashion.

5.2 Inter-VM Communication via Hypercalls

The rely on additional kernel code for the PCI-enumeration of the kAFL guest device makes it currently difficult to apply kAFL on other operating systems than linux. Otherwise, if no kAFL guest driver is available, the fuzzer will not be able to communicate with the targeted OS and cannot apply *internal* kernel fuzzing. A very promising approach would be the use of a Hypercall interface, instead. Hypercalls are usually triggered by executing the `vmcall` instruction. Moreover, it does not make a difference, whether `vmcall` was executed in kernel or user mode. Therefore, a Hypercall interface is more feasible, since only the host VMM has to be adapted once and it would further only require the development of specific user mode *drivers* instead of full-blown kernel

drivers for each OS. For this purpose, a proof of concept implementation has already been developed.

Chapter 6

Conclusion

Latest feedback-driven fuzzing methods have proven as an effective approach to find vulnerabilities in an automated and comprehensive fashion. Recent work has also demonstrated that such techniques can be applied to the kernel space. While previous feedback-driven kernel fuzzers were able to find a large amount of security flaws in certain operating systems, their benefit was either limited by poor performance due to CPU emulation or the lack of portability due to the need of compile-time instrumentations.

In this thesis a novel feedback-driven kernel fuzzer was presented, which utilizes latest CPU features. Combining all components provides a much higher performance than other approaches and the ability to apply kernel fuzz testing on any target domain.

As shown in the evaluation of this thesis, kAFL provides much higher performance than other kernel fuzzers. The Intel PT accelerated approach, which also includes the need of simultaneous trace decoding, is even faster than an instrumentation-based kernel fuzzing approach. Since the development of this kernel fuzzer is still ongoing, it can be assumed that the performance will improve even more.

Nevertheless, the Intel PT approach comes also with some drawbacks. This includes the rely on additional kernel code to interact with the kAFL fuzzer, which makes it difficult and costly to apply kAFL on other OSs than Linux. Furthermore, the filtering of non-deterministic code in non-interrupt context is currently not fully solved and therefore it limits the applicability of kAFL. Since for both issues potential solutions are proposed, the further evolutions and development will show how they perform.

Acronyms

AFL American Fuzzy Lop.
APIC Advanced Programmable Interrupt Controller.

BAR Base Address Registers.
BLOB Binary Large Object.

CoFI Change of Flow Instructions.
CoW Copy-on-Write.
CPL Current Privilege Level.

EPT Extended Page Tables.

GPR General Purpose Register.

Intel PT Intel Processor Trace.
IP Instruction Pointer.
IRQ Interrupt Request.
ISR Interrupt Service Routine.

kAFL kernel AFL.
KVM Kernel-based Virtual Machine.

LVT PMI Local Vector Table Performance Monitor Interrupt.

MMIO Memory-mapped I/O.
MSR Model Specific Register.

NMI Non-Maskable Interrupt.

OS Operating System.

PFN Page Frame Number.
PML4T Page Map Level 4 Table.

QEMU Quick Emulator.

ToPA Table of Physical Addresses.

vCPU virtual CPU.
VM Virtual Machine.
VMCS Virtual Machine Control Structure.
VMM Virtual Machine Monitor.
VMX Virtual Machine Extension.

Bibliography

[PG74] Gerald J Popek and Robert P Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33.12 (1990), pp. 32–44. ISSN: 00010782. DOI: 10.1145/96267.96279. URL: http://ftp.cs.wisc.edu/paradyn/technical%7B%5C_%7Dpapers/fuzz.pdf.

[Int96] Intel Intel. *Intel® 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC), May 1996)*. 1996. URL: http://www.intel.com/design/chipsets/datashts/29056601.pdf.

[RI00] John Scott Robin and Cynthia E Irvine. "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor". In: *Proceedings of the USENIX Security Symposium* 8.3 (2000), p. 10. ISSN: 15407993. DOI: 10.1109/MSP.2010.92. URL: http://portal.acm.org/citation.cfm?id=1251316.

[Luk+05] Chi-Keung Luk et al. "Pin: Building customized program analysis tools with dynamic instrumentation". In: *Proceedings of the Conference on Programming Language Design and Implementation*. 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: http://doi.acm.org/10.1145/1065010.1065034.

[Kiv+07] Avi Kivity et al. "kvm: the Linux virtual machine monitor". In: *Proceedings of the Linux Symposium* 1 (2007), pp. 225–230. ISSN: 1465-6906. DOI: 10.1186/gb-2008-9-1-r8. URL: https://www.kernel.org/doc/mirror/ols2007v1.pdf%7B%5C#%7Dpage=225.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX conference on Operating systems*

design and implementation (2008), pp. 209–224. ISSN: <null>. DOI: 10.1.1.142.9494. URL: http://portal.acm.org/citation.cfm?id=1855756.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David a. Molnar. "Automated Whitebox Fuzz Testing". In: *Search* 9.July (2008), pdf. ISSN: 1064-3745. DOI: 10.1007/978-3-642-02652-2_1. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.151.9430%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[Rus08] Rusty Russell. "Virtio". In: *ACM SIGOPS Operating Systems Review* 42 (2008), pp. 95–103. ISSN: 01635980. DOI: 10.1145/1400097.1400108.

[VMw08] VMware VMware. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. 2008. Chap. Software Technique: Binary Translation. URL: http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf.

[AEW09] Andrea Arcangeli, Izik Eidus, and Chris Wright. "Increasing memory density by using KSM". In: *Proceedings of the linux symposium*. 2009, pp. 19–28. URL: http://www.kernel.org/doc/ols/2009/$%5Cbackslash$nhttps://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf.

[VMw09] VMware VMware. *Software and Hardware Techniques for x86 Virtualization*. 2009. Chap. Software Technique: Binary Translation. URL: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/software_hardware_tech_x86_virt.pdf.

[Mac11] A. Cameron Macdonell. "Shared-memory Optimizations for Virtual Machines". AAINR89468. PhD thesis. Edmonton, Alta., Canada, 2011. ISBN: 978-0-494-89468-2.

[SSS14] Sergej Schumilo, Ralf Spenneberg, and Hendrik Schwartke. "Don't trust your USB! How to find bugs in USB device drivers". In: (2014). URL: https://www.blackhat.com/docs/eu-14/materials/eu-14-Schumilo-Dont-Trust-Your-USB-How-To-Find-Bugs-In-USB-Device-Drivers-wp.pdf.

[Cha15] Baruch Chaikin. "Micro VMMs and Nested Virtualization". 2015. URL: http://tce.webee.eedev.technion.ac.il/wp-content/uploads/sites/8/2015/09/BC_Micro-VMMs-and-Nested-Virtualization.pdf.

[HN16a] Jesse Hertz and Tim Newsham. *Project Triforce: AFL + QEMU + kernel = CVEs! (or) How to use AFL to fuzz arbitrary VMs*. https://github.com/nccgroup/TriforceAFL/blob/master/slides/ToorCon16_TriforceAFL.pdf. 2016.

[HN16b] Jesse Hertz and Tim Newsham. *Project Triforce: Run AFL on Everything!* https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Blog. 2016.

[Jur16] Mateusz Jurczyk. *A year of Windows kernel font fuzzing #2: the techniques*. https://googleprojectzero.blogspot.de/2016/07/a-year-of-windows-kernel-font-fuzzing-2.html. Blog. 2016.

[NC16] Vegard Nossum and Quentin Casasnovas. *Filesystem Fuzzing with American Fuzzy Lop*. Vault 2016. 2016. URL: https://events.linuxfoundation.org/sites/events/files/slides/AFL%20filesystem%20fuzzing,%20Vault%202016_0.pdf.

[OS16] Shlomi Oberman and Ron Shina. "COFI Break: Breaking Exploits with Practical Control Flow Integrity". In: (2016). URL: http://gsec.hitb.org/sg2016/sessions/cofi-break-breaking-exploits-with-practical-control-flow-integrity/.

[Ste+16] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: *Ndss*. February. 2016, pp. 21–24. ISBN: 189156241X. DOI: 10.14722/ndss.2016.23368.

[Inta] Intel Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual (Order number: 325384-058US, April 2016)*. Chap. 23. Introduction to Virtual Machine Extensions.

[Intb] Intel Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual (Order number: 325384-058US, April 2016)*. Chap. 24. Virtual Machine Control Structures.

[Intc] Intel Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual (Order number: 325384-058US, April 2016)*. Chap. 36. Intel®Processor Trace.

[Kle] Andi Kleen. *simple-pt - Simple Intel CPU processor tracing on Linux*. https://github.com/andikleen/simple-pt.

[PPt] *Linux 4.8, perf Documentation, `Linux/tools/perf/Documentation/intel-pt.txt`*. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/tools/perf/Documentation/intel-pt.txt?id=refs/tags/v4.8.

[Swi] Robert Swiecki. *A general-purpose, easy-to-use fuzzer with interesting analysis options.* https://google.github.io/honggfuzz/.

[Vyu] Dmitry Vyukov. *syzkaller - linux syscall fuzzer*. https://github.com/google/syzkaller.

[Zal] Michael Zalewski. *Technical "whitepaper" for afl-fuzz*. http://lcamtuf.coredump.cx/afl/technical_details.txt.