kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels

Sergej Schumilo, Cornelius Aschermann, and Robert Gawlik, *Ruhr-Universität Bochum;*
Sebastian Schinzel, *Münster University of Applied Sciences;*
Thorsten Holz, *Ruhr-Universität Bochum*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

This paper is included in the Proceedings of the
26th USENIX Security Symposium
August 16–18, 2017 • Vancouver, BC, Canada
ISBN 978-1-931971-40-9

kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels

Sergej Schumilo
Ruhr-Universität Bochum

Cornelius Aschermann
Ruhr-Universität Bochum

Robert Gawlik
Ruhr-Universität Bochum

Sebastian Schinzel
Münster University of Applied Sciences

Thorsten Holz
Ruhr-Universität Bochum

Abstract

Many kinds of memory safety vulnerabilities have been endangering software systems for decades. Amongst other approaches, fuzzing is a promising technique to unveil various software faults. Recently, feedback-guided fuzzing demonstrated its power, producing a steady stream of security-critical software bugs. Most fuzzing efforts—especially feedback fuzzing—are limited to *user space components* of an operating system (OS), although bugs in *kernel components* are more severe, because they allow an attacker to gain access to a system with full privileges. Unfortunately, kernel components are difficult to fuzz as feedback mechanisms (i.e., guided code coverage) cannot be easily applied. Additionally, non-determinism due to interrupts, kernel threads, statefulness, and similar mechanisms poses problems. Furthermore, if a process fuzzes its own kernel, a kernel crash highly impacts the performance of the fuzzer as the OS needs to reboot.

In this paper, we approach the problem of coverage-guided kernel fuzzing in an *OS-independent* and *hardware-assisted* way: We utilize a hypervisor and Intel's *Processor Trace* (PT) technology. This allows us to remain independent of the target OS as we just require a small user space component that interacts with the targeted OS. As a result, our approach introduces almost no performance overhead, even in cases where the OS crashes, and performs up to 17,000 executions per second on an off-the-shelf laptop. We developed a framework called *kernel-AFL* (kAFL) to assess the security of Linux, macOS, and Windows kernel components. Among many crashes, we uncovered several flaws in the *ext4* driver for Linux, the *HFS* and *APFS* file system of macOS, and the *NTFS* driver of Windows.

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 typically focus on user mode applications. This is likely 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 nowadays to exploit known vulnerabilities. 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 widely deployed in practice. This is due to more complex implementations, which may affect the system performance. Furthermore, some of them are not part of the official mainline code base or even require support for the latest CPU extension (e.g., SMAP / SMEP on x86-64). Additionally, when compromising 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 or to gain persistence for kernel-based rootkits.

For a long time, fuzzing has been a critical component in testing and establishing the quality of software. However, with the development of American Fuzzy Lop (AFL), smarter fuzzers have gained significant traction in the industry [1] as well as in research [8, 14, 35, 37]. This trend was further amplified by Google's *OSS Fuzz* project that successfully found—and continues to find—a significant number of critical bugs in highly security-relevant software. Finally, DARPA's Cyber Grand Challenge showed that fuzzing remains highly relevant for the state-of-the-art in bug finding. The latest generation of feedback-driven fuzzers generally uses mechanisms to learn which inputs are interesting and which are not. Interesting inputs are used to produce more inputs that may trigger new execution paths in the target. Inputs that did not trigger interesting behavior in the program are discarded. Thus, the fuzzer is able to "learn" the input format. This greatly improves efficiency and usability of fuzzers, especially by reducing the need for an oracle which generates semi-valid inputs or an extensive corpus that covers most paths in the target.

Unfortunately, AFL is limited to user space applications and lacks kernel support. Fuzzing kernels has a set of additional challenges when compared to userland (or *ring 3*) fuzzing: First, crashes and timeouts mandate the use of virtualization to be able to catch faults and continue gracefully. Second, kernel-level code has significantly more non-determinism than the average ring 3 program—mostly due to interrupts, kernel threads, statefulness, and similar mechanisms. This makes fuzzing kernel code challenging. Furthermore, there is no equivalent to command line arguments or `stdin` to interact with kernels or drivers in a generic way except for plain interrupt or `sysenter` instructions. In addition, the Windows kernel and many relevant drivers and core components (for Windows, macOS and even Linux) are closed source and cannot be instrumented by common techniques without a significant performance overhead.

Previous approaches to kernel fuzzing were not portable because they relied on certain drivers or recompilation [10, 34], were very slow due to emulation to gather feedback [7], or simply were not feedback-driven at all [11].

In this paper, we introduce a new technique that allows applying feedback fuzzing to arbitrary (even closed source) x86-64 based kernels, without any custom ring 0 target code or even OS-specific code at all. We discuss the design and implementation of *kernel-AFL* (kAFL), our prototype implementation of the proposed techniques. The overhead for feedback generation is very small (less than 5%) due to a new CPU feature: Intel's *Processor Trace* (PT) technology provides control flow information on running code. We use this information to construct a feedback mechanism similar to AFL's instrumentation. This allows us to obtain up to 17,000 executions per second on an off-the-shelf laptop (Thinkpad T460p, i7-6700HQ and 32 GB RAM) for simple target drivers. Additionally, we describe an efficient way for dealing with the non-determinisms that occur during kernel fuzzing. Due to the modular design, kAFL is extensible to fuzz *any* x86/x86-64 OS. We have applied kAFL to Linux, macOS, and Windows and found multiple previously unknown bugs in kernel drivers in those OSs.

In summary, our contributions in this paper are:

- **OS independence**: We show that feedback-driven fuzzing of closed-source kernel mode components is possible in an (almost) OS-independent manner by harnessing the hypervisor (VMM) to produce coverage. This allows targeting any x86 operating system kernel or user space component of interest.
- **Hardware-assisted feedback**: Our fuzzing approach utilizes Intel's Processor Trace (PT) technology, and thus has a very small performance overhead. Additionally, our PT-decoder is up to 30 times faster than Intel's `ptxed` decoder. Thereby, we obtain complete trace information that we use to guide our evolutionary fuzzing algorithm to maximize test coverage.
- **Extensible and modular design**: Our modular design separates the fuzzer, the tracing engine, and the target to fuzz. This allows to support additional x86 operating systems' kernel space and user space components, without the need to develop a system driver for the target OS.
- **kernel-AFL**: We incorporated our design concepts and developed a prototype called *kernel-AFL* (kAFL) which was able to find several vulnerabilities in kernel components of different operating systems. To foster research on this topic, we make the source code of our prototype implementation available at `https://github.com/RUB-SysSec/kAFL`.

2 Technical Background

We use the Intel Processor Trace (*Intel PT*) extension of *IA-32* CPUs to obtain coverage information for ring 0 execution of arbitrary (even closed-source) OS code. To facilitate efficient and OS-independent fuzzing, we also make use of Intel's hardware virtualization features (Intel VT-x). Hence, our approach requires a CPU that supports both *Intel VT-x* and *Intel PT*. This section provides a brief overview of these hardware features and establishes the technical foundation for the later sections.

2.1 x86-64 Virtual Memory Layouts

Every commonly used x86-64 OS uses a *split virtual memory layout*: The kernel is commonly located at the upper half of each virtual memory space, whereas each user mode process memory is located in the lower half. For example, the virtual memory space of Linux is typically split into kernel space (upper half) and user space (lower half) each with a size of 2^{47} due to the 48-bit virtual address limit of current x86-64 CPUs. Hence, the kernel memory is mapped to any virtual address space and therefore it is located *always* at the same virtual address. If an user mode process executes the `syscall/sysenter` instruction for kernel interaction or causes an exception that has to be handled by the OS, the OS will keep the current CR3 value and thus does not switch the virtual memory address space. Instead, the current virtual memory address space is reused and the kernel handles the current user mode process related task within the same address space.

2.2 Intel VT-x

The kernel fuzzing approach introduced in this paper relies on modern x86-64 hardware virtualization technology. Hence, we provide a brief overview of Intel's hardware virtualization technology, Intel VT-x.

We differentiate between three kinds of CPUs: physical CPUs, logical CPUs, and virtual CPUs (vCPUs). A physical CPU is a CPU that is implemented in hardware. Most modern CPUs support mechanisms to increase multithreading performance without additional physical CPU cores on the die (e.g., *Intel Hyper-Threading*). In this case, there are multiple logical CPUs on one physical CPU. These different logical CPUs share the physical CPU and, thus, only one of them can be active at a time. However, the execution of the different logical CPUs is interleaved by the hardware and therefore the available resources can be utilized more efficiently (e.g., one logical CPU uses the arithmetic logic unit while another logical CPU waits for a data fetch) and the operating system can reduce the scheduling overhead. Each logical CPU is usually treated like a whole CPU by the operating system. Finally, it is possible to create multiple hardware-supported virtual machines (VMs) on a single logical CPU. In this case, each VM has a set of its own vCPUs.

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

To provide full hardware-assisted virtualization support, Intel VT-x adds two additional execution modes to the well-known protection ring based standard mode of execution. The default mode of executions is called *VMX OFF*. It does not implement any hardware virtualization support. When using hardware-supported virtualization, the CPU switches into the *VMX ON* state and distinguishes between two different execution modes: the higher-privileged mode of the hypervisor (VMX root or VMM), and the lower privileged execution mode of the virtual machine guest (VMX non-root or VM).

When running in guest mode, several privileged actions or reasons (execution of restricted instructions, expired VMX-preemption timer, or access to certain emulated devices) in the VM guest will trigger a VM-Exit event and transfer control to the hypervisor. This way, it is possible to run arbitrary software that expects privileged access to the hardware (such as an OS) inside a VM. At the same time, a higher authority can meditate and control the operations performed with a small performance overhead.

To create, launch, and control a VM, the VMM has to use a virtual machine control structure (VMCS) for each vCPU [28]. The VMCS contains all essential information about the current state and how to perform VMX transitions of the vCPU.

2.3 Intel Processor Trace

With the fifth generation of Intel Core processors (Broadwell architecture), Intel has introduced a new processor feature called *Intel Processor Trace (Intel PT)* to provide execution and branch tracing information. Unlike other branch tracing technologies such as *Intel Last Branch Record (LBR)*, the size of the output buffer is no longer strictly limited by special registers. Instead, it is only limited by the size of the main memory. If the output target is repeatedly and timely emptied, we can create traces of arbitrary length. The processor's output format is packet-oriented and separated into two different types: general execution information and control flow information packets. Intel PT produces various types of control flow related packet types during runtime. To obtain control-flow information from the trace data, we require a decoder. The decoder needs the traced software to interpret the packets that contain the addresses of conditional branches.

Intel specifies five types of control flow affecting instructions called *Change of Flow Instruction (CoFI)*. The execution of different CoFI types results in different sequences of flow information packets. The three CoFI types relevant to our work are:

1. **Taken-Not-Taken (TNT):** If the processor executes any conditional jump, the decision whether this jump was taken or not is encoded in a TNT packet.

2. **Target IP (TIP):** If the processor executes an indirect jump or transfer instruction, the decoder will not be able to recover the control flow. Therefore, the processor produces a TIP packet upon the execution of an instruction of the type `indirect branch`, `near ret` or `far transfer`. These TIP packets store the corresponding target instruction pointer executed by the processor after the transfer or jump has occurred.

3. **Flow Update Packets (FUP):** Another case where the processor must produce a hint packet for the software decoder are asynchronous events such as interrupts or traps. These events are recorded as FUPs and usually followed by a TIP to indicate the following instruction.

To limit the amount of trace data generated, Intel PT provides multiple options for runtime filtering. Depending on the given processor, it might be possible to configure multiple ranges for instruction-pointer filtering (*IP Filter*). In general, these filter ranges only affect virtual addresses if paging is enabled; this is always the case in x86-64 long-mode. Therefore, it is possible to limit trace generation to selected ranges and thus avoid huge amounts of superfluous trace data. In accordance to the IP filtering mechanism, it is possible to filter traces by the current privilege level (CPL) of the protection ring model (e.g ring 0 or ring 3). This filter allows us to select only the user mode ($CPL > 0$) or kernel mode ($CPL = 0$) activity. kAFL utilizes this filter option to limit tracing explicitly to kernel mode execution. In most cases, the focus of tracing is not the whole OS within all user mode processes and their kernel interactions. To limit trace data generation to one specific virtual memory address space, software can use the *CR3 Filter*. Intel PT will only produce trace data if the CR3 value matches the configured filter value. The CR3 register contains the pointer to the current page table. The value of the CR3 register can thus be used to filter code executed on behalf of a certain ring 3 process, even in ring 0 mode.

Intel PT supports various configurable target domains for output data. kAFL focuses on the Table of Physical Addresses (ToPA) mechanism that enables us to specify multiple output regions: Every ToPA table contains multiple *ToPA entries*, which in turn contain the physical address of the associated memory chunk used to store trace data. Each ToPA entry contains the physical address, a size specifier for the referred memory chunk in physical memory, and multiple type bits. These type bits specify the CPU's behavior on access of the ToPA entry and how to deal with filled output regions.

3 System Overview

We now provide a high-level overview of the design of an OS-independent and hardware-assisted feedback fuzzer before presenting the implementation details of our tool called kAFL in Section 4.

Our system is split into three components: the fuzzing logic, the VM infrastructure (modified versions of QEMU and KVM denoted by QEMU-PT and KVM-PT), and the user mode agent. The fuzzing logic runs as a ring 3 process on the host OS. This logic is also referred to as kAFL. The VM infrastructure consists of a ring 3 component (QEMU-PT) and a ring 0 component (KVM-PT). This facilitates communication between the other two components and makes the Intel PT trace data available to the fuzzing logic. In general, the guest only communicates with the host via hypercalls. The host can then read and write guest memory and continues VM execution once the request has been handled. A overview of the architecture can be seen in Figure 1.

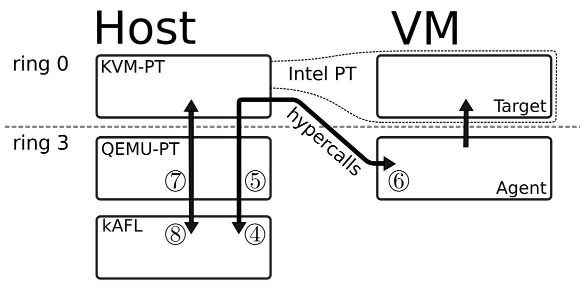

Figure 1: High-level overview of the kAFL architecture. The setup process (①-③) is not shown.

We now outline the events and communication that take place during a fuzz run, as depicted in Figure 2. When the VM is started, the first part of the user mode agent (the loader) uses the hypercall `HC_SUBMIT_PANIC` to submit the address of the kernel panic handler (or the *BugCheck* kernel address in Windows) to QEMU-PT ①. QEMU-PT then patches a hypercall calling routine at the address of the panic handler. This allows us to get notified and react fast to crashes in the VM (instead of waiting for timeouts / reboots).

Then the loader uses the hypercall `HC_GET_PROGRAM` to request the actual user mode agent and starts it ②. Now the loader setup is complete and the fuzzer begins its initialization. The agent triggers a `HC_SUBMIT_CR3` hypercall that will be handled by KVM-PT. The hypervisor extracts the CR3 value of the currently running process and hands it over to QEMU-PT for filtering ③. Finally, the agent uses the hypercall `HC_SUBMIT_BUFFER` to inform the host at which address it expects its inputs. The fuzzer setup is now finished and the main fuzzing loop starts.

During the main loop, the agent requests a new input using the `HC_GET_INPUT` hypercall ④. The fuzzing logic produces a new input and sends it to QEMU-PT. Since QEMU-PT has full access to the guest's memory space, it can simply copy the input into the buffer specified by the agent. Then it performs a VM-Entry to continue executing the VM ⑤. At the same time, this VM-Entry event enables the PT tracing mechanism. The agent now consumes the input and interacts with the kernel (e.g., it interprets the input as a file system image and tries to mount it ⑥). While the kernel is being fuzzed, QEMU-PT decodes the trace data and updates the bitmap on demand. Once the interaction is finished and the kernel handed control back to the agent, the agent notifies the hypervisor via a `HC_FINISHED` hypercall. The resulting VM-Exit stops the tracing and QEMU-PT decodes the remaining trace data ⑦. The resulting bitmap is passed

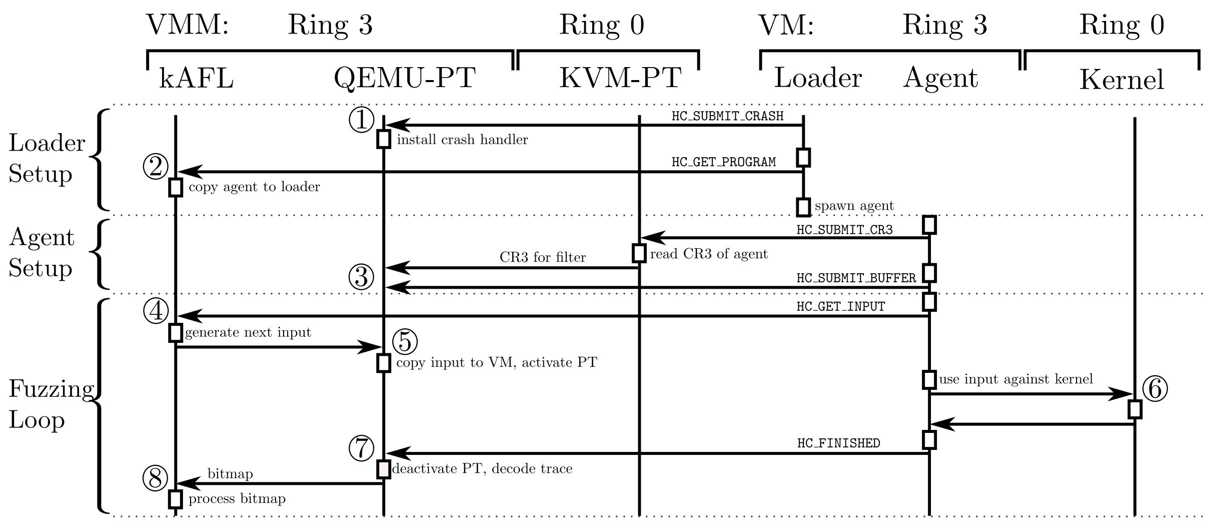

Figure 2: Overview of the kAFL hypercall interaction.

to the logic for further processing ⑧. Afterwards, the agent can continue to run any untraced clean-up routines before issuing another HC_GET_INPUT to start the next loop iteration.

3.1 Fuzzing Logic

The fuzzing logic is the command and controlling component of kAFL. It manages the queue of interesting inputs, creates mutated inputs, and schedules them for evaluation. In most aspects, it is based on the algorithms used by AFL. Similarly to AFL, we use a bitmap to store basic block transitions. We gather the AFL bitmap from the VMs through an interface to QEMU-PT and decide which inputs triggered interesting behaviour. The fuzzing logic also coordinates the number of VMs spawned in parallel. One of the bigger design differences to AFL is that kAFL makes extensive use of multiprocessing and parallelism, where AFL simply spawns multiple independent fuzzers which synchronize their input queues sporadically[1]. In contrast, kAFL executes the deterministic stage in parallel, and all threads work on the most interesting input. A significant amount of time is spent in tasks that are not CPU-bound (such as guests that delay execution). Therefore, using many parallel processes (upto 5-6 per CPU core) drastically improves performance of the fuzzing process due to a higher CPU load per core. Lastly, the fuzzing logic communicates with the user interface to display current statistics in regular intervals.

[1]AFL recently added experimental support for distributing the deterministic stage, see `https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt#L60-L66`.

3.2 User Mode Agent

We expect a user mode agent to run inside the virtualized target OS. In principle, this component only has to synchronize and gather new inputs by the fuzzing logic via the hypercall interface and use it to interact with the guest's kernel. Example agents are programs that try to mount inputs as file system images, pass specific files such as certificates to kernel parser or even execute a chain of various syscalls.

In theory, we only need one such component. In practice, we use two different components: The first program is the *loader component*. Its job is to accept an arbitrary binary via the hypercall interface. This binary represents the *user mode agent* and is executed by the loader component. Additionally, the loader component will check if the agent has crashed (which happens often in case of syscall fuzzing) and restarts it if necessary. This setup has the advantage that we can pass any binary to the VM and reuse VM snapshots for different fuzzing components.

3.3 Virtualization Infrastructure

The fuzzing logic uses QEMU-PT to interact with KVM-PT to spawn the target VMs. KVM-PT allows us to trace individual vCPUs instead of logical CPUs. This component configures and enables Intel PT on the respective logical CPU before the CPU switches to guest execution and disables tracing during the VM-Exit transition. This way, the associated CPU will only provide trace data of the virtualized kernel itself. QEMU-PT is used to interact with the KVM-PT interface to configure and toggle Intel PT from user space and access the output buffer to decode the trace data. The decoded trace data is directly translated into a stream of addresses of executed *conditional branch instructions*. Moreover, QEMU-PT also filters the stream of executed addresses—based on previous knowledge of non-deterministic basic blocks—to prevent false-positive fuzzing results, and makes those available to the fuzzing logic as AFL-compatible bitmaps. We use our own custom Intel PT decoder to cache disassembly results, which leads to significant performance gains compared to the off-the-shelf solution provided by Intel.

3.4 Stateful and Non-Deterministic Code

Tracing operating systems results in a significant amount of non-determinism. The largest source of non-deterministic basic block transitions are interrupts, which can occur at any point in time. Additionally, our implementation does not reset the whole state after each execution since reloading the VM from a memory snapshot is costly. Thus we have to deal with the stateful and asynchronous aspects of the kernel. An example for stateful code might be a simple call to `kmalloc()`: Depending on the number of previous allocations, `kmalloc()` might simply return a fresh pointer or map a whole range of pages and update a significant amount of metadata. We use two techniques to deal with these challenges.

The first one is to filter out interrupts and the transition caused while handling interrupts. This is possible using the Intel PT trace data. If an interrupt occurs, the processor emits a TIP instruction since the transfer is not visible in the code. To avoid confusion during an interrupt occurring at an indirect control flow instruction, the TIP packet is marked with FUP (flow update packet) to indicate an asynchronous event. After identifying such a signature, the decoder will drop all basic blocks visited until the corresponding `iret` instruction is encountered. To link the interrupts with their corresponding `iret`, we track all interrupts on a simple call stack. This mechanism is necessary since the interrupt handler may itself be interrupted by another interrupt.

The second mechanism is to blacklist any basic block that occurs non-deterministically. Each time we encounter a new bit in the AFL bitmap, we re-run the input several times in a row. Every basic block that does not show up in all of the trials will be marked as non-deterministic and filtered from further processing. For fast access, the results are stored in a bitmap of blacklisted basic block addresses. During the AFL bitmap translation, any transition hash value—which combines the current basic block address and the previous basic block address—involving a blacklisted block will be skipped.

3.5 Hypercalls

Hypercalls are a feature introduced by virtualization. On Intel platforms, hypercalls are triggered by the `vmcall` instruction. Hypercalls are to VMMs as syscalls are to kernels. If any ring 3 process or the kernel in the VM executes a `vmcall` instruction, a VM-Exit event is triggered and the VMM can decide how to process the hypercall. We patched KVM-PT to pass through our own set of hypercalls to the fuzzing logic if a magic value is passed in `rax` and the appropriate hypercall-ID is set in `rbx`. Additionally, we also patched KVM-PT to accept hypercalls from ring 3. Arguments for specific hypercalls are passed through `rcx`. We use this mechanism to define an interface that user mode agent can use to communicate with the fuzzing logic. One example hypercall is `HC_SUBMIT_BUFFER`. Its argument is a guest pointer that is stored in `rcx`. Upon executing the `vmcall` instruction, a VM-Exit is triggered and QEMU-PT stores the buffer pointer that was passed. It will later copy the new input data into this buffer (see step ⑤ in Figure 2). Finally, the execution of the VM is continued.

```
cli
mov rax, KAFL_MAGIC_VALUE
mov rbx, HC_CRASH
mov rcx, 0x0
vmcall
```

Listing 1: Hypercall crash notifier.

Another use case for this interface is to notify the fuzzing logic when a crash occurs in the target OS kernel. In order to do so, we overwrite the kernel crash handler of the OS with a simple hypercall routine. The injected code is shown in Listing 1 and displays how the hypercall interface is used on the assembly level. The `cli` instruction disables all interrupts to avoid any kind of asynchronous interference during the hypercall routine.

4 Implementation Details

Based on the design outlined in the previous section, we built a prototype of our approach called kAFL. In the following, we describe several implementation details. The source code of our reference implementation is available at `https://github.com/RUB-SysSec/kAFL`.

4.1 KVM-PT

Intel PT allows us to trace branch transitions without patching or recompiling the targeted kernel. To the best of our knowledge, no publicly available driver is able to trace only guest executions of a single vCPU using Intel PT for long periods of time. For instance, Simple-PT [29] does not support long-term tracing by design. The `perf`-subsystem [5] supports tracing of VM guest operations and long-term tracing. However, it is designed to trace logical CPUs, not vCPUs. Even if VMX execution is traced, the data would be associated with logical CPUs and not with vCPUs. Hence, the VMX context must be reassembled, which is a costly task.

To address these shortcomings, we developed KVM-PT. It allows us to trace vCPUs for an indefinite amount of time without any scheduling side effects or any loss of trace data due to overflowing output regions. The extension provides a fast and reliable trace mechanism for KVM vCPUs. Moreover, this extension exposes, much like KVM, an extensive user mode interface to access this tracing feature from user space. QEMU-PT utilizes this novel interface to interact with KVM-PT and to access the resulting trace data.

4.1.1 vCPU Specific Traces

To enable Intel PT, software that runs within ring 0 (in our case KVM-PT) has to set the corresponding bit of a model specific register (MSR) (`IA32_RTIT_CTL_MSR.TraceEn`) [28]. After tracing is enabled, the logical CPU will trace any executed code if it satisfies the configured filter options. The modification has to be done before the CPU switches from the host context to the VM operation; otherwise the CPU will execute guest code and is technically unable to modify any host MSRs. The inverse procedure is required after the CPU has left the guest context. However, enabling or disabling Intel PT manually will also yield a trace containing the manual MSR modification. To prevent the collection of unwanted trace data within the VMM, we use the MSR autoload capabilities of Intel VT-x. MSR autoloading can be enabled by modifying the corresponding entries in the VMCS (e.g., *VM_ENTRY_CONTROL_MSR* for VM-entries). This forces the CPU to load a list of preconfigured values for defined MSRs after either a VM-entry or VM-exit has occurred. By enabling tracing via MSR autoloading, we only gather Intel PT trace data for one specific vCPU.

4.1.2 Continuous Tracing

Once we have enabled Intel PT, the CPU will write the resulting trace data into a memory buffer until it is full. The physical addresses of this buffer and how to handle full buffers is specified by an array of data structures called Table of Physical Addresses (ToPA) entries.

The array can contain multiple entries and has to be terminated by a single END entry ③. There are two different ways the CPU can handle an overflow: It can stop the tracing (while continuing the execution—thus

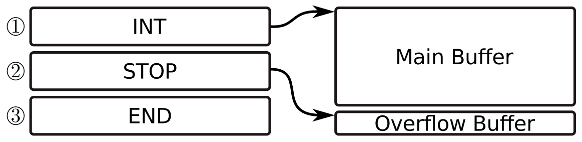

Figure 3: KVM-PT ToPA configuration.

resulting in incomplete traces) or it can raise an interrupt. This interrupt causes a VM-exit since it is not maskable. We catch the interrupt on the host and consume the trace data. Finally, we reset the buffers and continue with the VM execution. Unfortunately, this interrupt might be raised at an unspecified time after the buffer was filled[2]. Our configuration of the ToPA entries can be seen in Figure 3. To avoid losing trace data, we use two different ToPA entries. The first one is the main buffer ①. Its overflow behavior is to trigger the interrupt. Once the main buffer is filled, a second entry is used until the interrupt is actually delivered. The ToPA specifies another smaller buffer ②. Overflowing the second buffer would lead to the stop of the tracing. To avoid the resulting data loss, we chose the second buffer to be about four times larger than the largest overflowing trace we have ever seen in our tests (4 KB).

In case the second buffer also overflows, the following trace will contain a packet indicating that some data is missing. In that case the size of the second buffer can simply be increased. This way, we manage to obtain precise traces for any amount of trace data.

4.2 QEMU-PT

To make use of the KVM extension KVM-PT, an user space counterpart is required. QEMU-PT is an extension of QEMU and provides full support for KVM-PT's user space interface. This interface provides mechanisms to enable, disable, and configure Intel PT at runtime as well as a periodic ToPA status check to avoid overruns. KVM-PT is accessible from user mode via `ioctl()` commands and an `mmap()` interface.

In addition to being a userland interface to KVM-PT, QEMU-PT includes a component that decodes trace data into a form more suitable for the fuzzing logic: We decode the Intel PT packets and turn them into an AFL-like bitmap.

4.2.1 PT Decoder

Extensive kernel fuzzing may generate several hundreds of megabytes of trace data per second. To deal with

[2]This is due to the current implementation of this interrupt. Intel specifies the interrupt as *not precise*, which means it is likely that further data will be written to the next buffer or tracing will be terminated and data will be discarded.

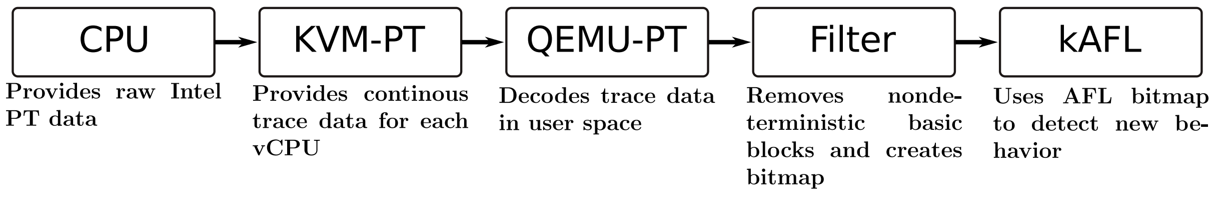
CPU
Provides raw Intel PT data
KVM-PT
Provides continous trace data for each vCPU
QEMU-PT
Decodes trace data in user space
Filter
Removes nondeterministic basic blocks and creates bitmap
kAFL
Uses AFL bitmap to detect new behavior

Figure 4: Overview of the pipeline that converts Intel PT traces to kAFL bitmaps.

such large amounts of incoming data, the decoder must be implemented with a focus on efficiency. Otherwise, the decoder may become the major bottleneck during the fuzzing process. 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 ease efforts to implement an Intel PT software decoder, Intel provides its own decoding engine called `libipt` [4]. `libipt` is a general-purpose Intel PT decoding engine. However, it does not fit our purposes very well because `libipt` decodes trace data in order to provide execution data and flow information. Furthermore, `libipt` does not cache disassembled instructions and has performed poorly in our use cases.

Since kAFL only relies on flow information and the fuzzing process is repeatedly applied to the same code, it is possible to optimize the decoding process. Our Intel PT software decoder acts like a *just-in-time decoder*, which means that code sections are only considered if they are executed according to the decoded trace data. To optimize further look-ups, all disassembled code sections are cached. In addition, we simply ignore packets that are not relevant for our use case.

Since our PT decoder is part of QEMU-PT, trace data is directly processed if the ToPA base region is filled. The decoding process is applied in-place since the buffer is directly accessible from user space via `mmap()`. Unlike other Intel PT drivers, we do not need to store large amounts of trace data in memory or on storage devices for post-mortem decoding. Eventually, the decoded trace data is translated to the AFL bitmap format.

4.3 AFL Fuzzing Logic

We give a brief description of the fuzzing parts of AFL because the logic we use to perform scheduling and mutations closely follows that of AFL. The most important aspect of AFL is the bitmap used to trace which basic block transitions where encountered. Each basic block has a randomly assigned ID, and each transition from basic block A to another basic block B is assigned an offset into the bitmap according to the following formula:

$$(id(A)/2 \oplus id(B)) \ \% \ \texttt{SIZE\_OF\_BITMAP}$$

Instead of the compile-time random, kAFL uses the addresses of the basic blocks. Each time the transition is observed, the corresponding byte in the bitmap is incremented. After finishing the fuzzing iteration, each entry of the bitmap is rounded such that only the highest bit remains set. Then the bitmap is compared with the global static bitmap to see if any new bit was found. If a new bit was found, it is added to the global bitmap and the input that triggered the new bit is added to the queue. When a new interesting input is found, a deterministic stage is executed that tries to mutate each byte individually.

Once the deterministic stage is finished, the non-deterministic phase is started. During this non-deterministic phase, multiple mutations are performed at random locations. If the deterministic phase finds new inputs, the non-deterministic phase will be delayed until all deterministic phases of all interesting inputs have been performed. If an input triggers an entirely new transition (as opposed to a change in the number of times the transition was taken), it will be favored and fuzzed with a higher priority.

5 Evaluation

Based on our implementation, we now describe the different fuzzing campaigns we performed to evaluate kAFL. We evaluate kAFL's fuzzing performance across different platforms. Section 5.5 provides an overview of all reported vulnerabilities, crashes, and bugs that were found during the development process of kAFL. We also evaluate kAFL's ability to find a previously known vulnerability. Finally, in Section 5.6 the overall fuzzing performance of kAFL is compared to *ProjectTriforce*, the only other OS-independent feedback fuzzer available. TriforceAFL is based on the emulation backend of QEMU instead of hardware-assisted virtualization and Intel PT. The performance overhead of KVM-PT is discussed in Section 5.7. Additionally, a performance comparison of our PT decoder and an Intel implementation of a software decoder is given in Section 5.8.

If not stated otherwise, the benchmarks were performed on a desktop system with an Intel i7-6700 processor and 32GB DDR4 RAM. To avoid distortions due to poor I/O performance, all benchmarks are performed on a RAM disk. Similar to AFL, we consider a crashing input to be *unique* if it triggered at least one basic block transition which has not been triggered by any previous crash (i.e., the bitmap contains at least one new bit). Note this does not imply that the underlying bugs are truly unique.

5.1 Fuzzing Windows

We implemented a small Windows 10 specific user mode agent that mounts any data chunk (fuzzed payload) as NTFS-partitioned volume (289 lines of C code). We used the *Virtual Hard Disk* (VHD) API and various IOCTLS to mount and unmount volumes programmatically [31, 32]. Unfortunately, mounting volumes is a slow operation under Windows and we only managed to achieve a throughput of 20 executions per second. Nonetheless, kAFL managed to find a crash in the NTFS driver. The fuzzer ran for 4 days and 14 hours and reported 59 unique crashes, all of which were division by zero crashes. After manual investigation we suspect that there is only one unique bug. While it does not allow code execution, it is still a denial-of-service vulnerability, as for example, a USB stick with that malicious NTFS volume plugged into a critical system will crash that system with a blue screen. It seems that we only scratched the surface and NTFS was not thoroughly fuzzed yet. Hence, we assume that the NTFS driver under Windows is a valuable target for coverage-based feedback fuzzing.

Furthermore, we implemented a generic system call (syscall) fuzzing agent that simply passes a block of data to a syscall by setting all registers and the top stack region (55 lines of C and 46 lines of assembly code). This allows to set parameters for a syscall with a fuzzing payload *independent* of the OS ABI. The same fuzzer can be used to attack syscalls on different operation systems such as Linux or macOS. However, we evaluated it against the Windows kernel given the proprietary nature of this OS. We did not find any bugs in 13 hours of fuzzing with approx 6.3M executions since many syscalls cause the userspace agent to terminate: Due to the coverage-guided feedback, kAFL quickly learned how to generate payloads to execute valid syscalls, and this led to the unexpected execution of user mode callbacks via the kernel within the fuzzing agent. These crashes require rather expensive restarts of the agent and therefore we only achieved approx. 134 executions per second, while normally kAFL achieves a throughput of 1,000 to 4,000 tests per second (see Section 5.2). Additionally, the Windows syscall interface has already received much attention by the security community.

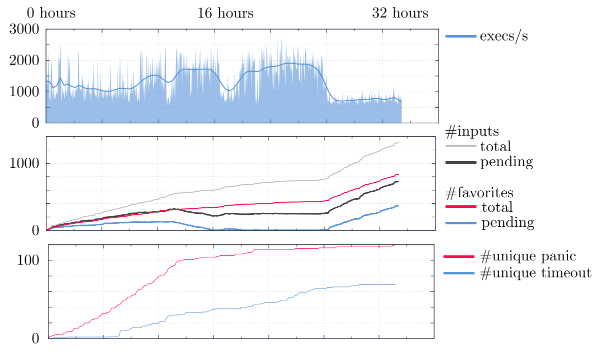

Figure 5: Fuzzing the ext4 kernel module for 32 hours.

5.2 Fuzzing Linux

We implemented a similar agent for Linux, which mounts data as ext4 volumes (66 lines of C code). We started the fuzz campaign with a minimal 64KB ext4 image as initial input. However, we configured the fuzzer such that it only fuzzes the first two kilobytes during the deterministic phase. In contrast to Windows, the Linux mount process is very fast, and we reached 1,000 to 2,000 tests per second on a Thinkpad laptop with a i7-6700HQ@2.6GHz CPU and 32GB RAM. Due to this high performance, we obtained significantly better coverage and managed to discover 160 unique crashes and multiple (confirmed) bugs in the ext4 driver during a twelve-day fuzzing campaign. Figure 5 shows the first 32 hours of another fuzzing run. The fuzzing process was still finding new paths and crashes on a fairly regular basis after 32 hours. An interesting observation is that there was no new coverage produced between hours 16 and 25, yet the number of inputs increased due a higher number of loop iterations. After hour 25, a truly new input was found that unlocked significant parts of the codebase.

5.3 Fuzzing macOS

Similarly to Windows and Linux, we targeted multiple file systems for macOS. So far, we found approximately 150 crashes in the HFS driver and manually confirmed that at least three of them are unique bugs that lead to a kernel panic. Those bugs can be triggered by unprivileged users and, therefore, could very well be abused for local denial-of-service attacks. One of these bugs seems to be a use-after-free vulnerability that leads to full control of the `rip` register. Additionally, kAFL found 220 unique crashes in the APFS kernel extension. All 3 HFS vulnerabilities and multiple APFS flaws have been reported to Apple.

5.4 Rediscovery of Known Bugs

We evaluated kAFL on the `keyctl` interface, which allows a user space program to store and manage various kinds of key material in the kernel. More specifically, it features a DER (see RFC5280) parser to load certificates. This functionality had a known bug (CVE-2016-0758[3]). We tested kAFL against the same interface on a vulnerable kernel (version 4.3.2). kAFL was able to uncover the same problem and one additional previously unknown bug that was assigned CVE-2016-8650[4]. kAFL managed to trigger 17 unique KASan reports and 15 unique panics in just one hour of execution time. During this experiment, kAFL generated over 34 million inputs, found 295 interesting inputs, and performed nearly 9,000 executions per second. This experiment was performed while running 8 processes in parallel.

5.5 Detected Vulnerabilities

During the evaluation, kAFL found more than a thousand unique crashes. We evaluated some manually and found multiple security vulnerabilities in all tested operating systems such as Linux, Windows, and macOS. So far, eight bugs were reported and three of them were confirmed by the maintainers:

- Linux: keyctl Null Pointer Dereference[5] (CVE-2016-8650[6])
- Linux: ext4 Memory Corruption[7]
- Linux: ext4 Error Handling[8]
- Windows: NTFS Div-by-Zero[9]
- macOS: HFS Div-by-Zero[10]
- macOS: HFS Assertion Fail[10]
- macOS: HFS Use-After-Free[10]
- macOS: APFS Memory Corruption[10]

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 an 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 Linux vulnerability, which calls in the ext4 error handling routine `panic()` and hence results in a kernel panic, was at the time of writing not investigated any further. The NTFS bug in Windows 10 is a non-recoverable error condition which leads to a blue screen. This bug was reported to Microsoft, but has not been confirmed yet. Similarly, Apple has not yet verified or confirmed our reported macOS bugs.

5.6 Fuzzing Performance

We compare the overall performance of kAFL across different operating systems. To ensure comparable results, we created a simple driver that contains a JSON parser based on `jsmn`[11] for each aforementioned operating system and used it to decode user input (see Listing 2). If the user input is a JSON string starting with `"KAFL"`, a crash is triggered. We traced both the JSON parser as well as the final check. This way kAFL was able to learn correct JSON syntax. We measured the time used to find the crash, the number of executions per second, and the speed for new paths to be discovered on all three target operating systems.

```
jsmn_parser parser;
jsmntok_t tokens[5];
jsmn_init(&parser);

int res=jsmn_parse(&parser, input, size, tokens, 5);
if(res >= 2){
    if(tokens[0].type == JSMN_STRING){
        int json_len = tokens[0].end - tokens[0].
            start;
        int s = tokens[0].start;
        if(json_len > 0 && input[s+0] == 'K'){
        if(json_len > 1 && input[s+1] == 'A'){
        if(json_len > 2 && input[s+2] == 'F'){
        if(json_len > 3 && input[s+3] == 'L'){
            panic(KERN_INFO "KAFL...\n");
    }}}}}
}
```

Listing 2: The JSON parser kernel module used for the coverage benchmarks.

We performed five repeated experiments for each operating system. Additionally we tested TriforceAFL with the Linux target driver. TriforceAFL is unable to fuzz Windows and macOS. To compare TriforceAFL with kAFL, the associated *TriforceLinuxSyscallFuzzer* was slightly modified to work with our vulnerable Linux kernel module. Unfortunately, it was not possible to compare kAFL against Oracle's file system fuzzer [34] due to technical issues with its setup.

During each run, we fuzzed the JSON parser for 30 minutes. The averaged and rounded results are displayed in Table 1. As we can see, the performance of kAFL is very similar across different systems. It should be remarked that the variance in this experiment is rather high

[3] `https://access.redhat.com/security/cve/cve-2016-0758`
[4] `https://access.redhat.com/security/cve/cve-2016-8650`
[5] `http://seclists.org/fulldisclosure/2016/Nov/76`
[6] `https://access.redhat.com/security/cve/cve-2016-8650`
[7] `http://seclists.org/fulldisclosure/2016/Nov/75`
[8] `http://seclists.org/bugtraq/2016/Nov/1`
[9] Reported to Microsoft Security.
[10] Reported to Apple Product Security.
[11] http://zserge.com/jsmn.html

	Execs/Sec (1 Process)	Execs/Sec (8 Processes)	Time to Crash (1 Process)	Time to Crash (8 Processes)	Paths/Min (1 Process)	Paths/Min (8 Processes)
TriforceAFL	150	320	-[a]	-[a]	10.08	-[b]
Linux (initramfs)	3000	5700	7:50	6:00	15.84	15.62
Debian 8	3000	5700	4:55	6:30	16.20	16.00
Debian 8 (KASan)	4300	5700	9:20	6:00	16.22	15.90
macOS (10.12.4)	5100	8100	7:43	5:10	14.50	15.06
Windows 10	4300	8700	4:14	4:50	11.50	12.02

[a] Not found during 30-minute experiments.

[b] This value cannot be obtained since TriforceAFL does not synchronize in such short time frames.

Table 1: kAFL and TriforceAFL fuzzing performance on the JSON sample driver.

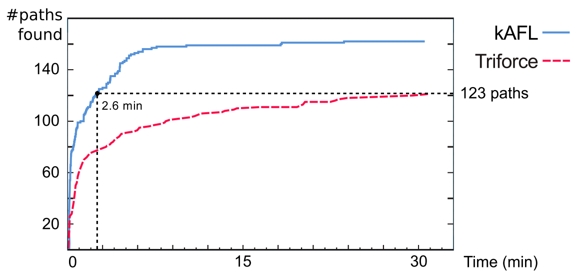

Figure 6: Coverage comparison of kAFL (initramfs) and TriforceAFL. kAFL takes less than 3 minutes to find the same number of paths as TriforceAFL does in 30 minutes (each running 1 process).

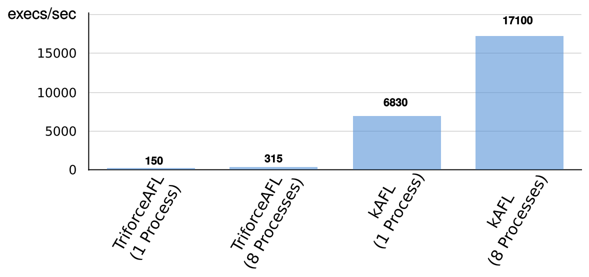

Figure 7: Raw execution performance comparison.

and explains some of the surprising results. This is due to the stochastic nature of fuzzing, since each fuzzer finds vastly different paths, some of which may take significantly longer to process, especially crashing paths and loops. One example for high variance is the fact that on Debian 8 (initramfs), the multiprocessing configuration on average needed more time to find the crash than one process.

TriforceAFL We used the JSON driver to compare kAFL and TriforceAFL with respect to execution speed and code coverage. However, the results where biased heavily in two ways: TriforceAFL did not manage to find a path that triggers the crash within 30 minutes (usually it takes approximately 2 hours), making it very hard to compare the code coverage of kAFL and TriforceAFL. The number of discovered paths is not a good indicator for the amount of coverage: With increasing running time, it becomes more difficult to discover new paths. Secondly, the number of executions per second is also biased by slower and harder to reach paths and especially crashing inputs. The coverage reached over time can be seen in Figure 6. It is obvious from the figure that kAFL found a significant number of paths that are very hard to reach for TriforceAFL. kAFL mostly stops finding new paths around the 10-15 minute mark, because the target driver simply doesn't contain any more paths to be uncovered. Therefore, the coverage value in Table 1 (stated as *Paths/Min*) is limited to the first 10 minutes of each 30-minute run.

We also compare raw execution performance instead of overall fuzzing performance, which is biased because of the execution of different paths, the sampling process for the non-determinism-filter, and various synchronization mechanisms. Especially on smaller inputs, these factors disproportionately affect the overall fuzzing performance. To avoid this, we compared the performance during the first havoc stage. Figure 7 shows the raw execution performance of kAFL compared to TriforceAFL during this havoc phase. kAFL provides up to 54 times better performance compared to TriforceAFL's QEMU CPU emulation. Slightly lower performance boosts are seen in single-process execution (48 times faster).

syzkaller We did not perform a performance comparison against syzkaller [10]. This has two reasons: First of all, syzkaller is a highly specific syscall fuzzer that encodes a significant amount of domain knowledge and is therefore not applicable to other domains such as filesystem images. On the other hand, syzkaller would most likely generate a significantly higher code coverage even without any feedback since it knows how to generate

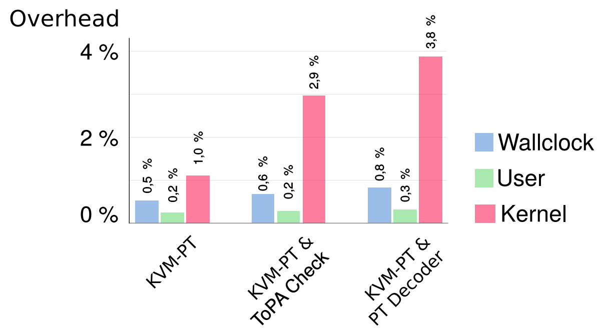

Figure 8: Overhead for compiling QEMU-2.6.0 in a traced VM.

valid syscalls and hence is able to trigger valid paths without any learning. Therefore, the coverage comparison would be highly misleading unless we implement the same syscall logic; a task that is out of the scope of this paper. Additionally, the coverage collection via kcov is highly specific to Linux and not applicable to closed-source targets.

5.7 KVM-PT Overhead

Our KVM extension KVM-PT adds overhead to the raw execution of KVM. Therefore, the performance overhead was compared with several KVM-PT setups on an i5-6500@3.2Ghz desktop system with 8GB DDR4 RAM. This includes KVM-PT in combination with the PT decoder, KVM-PT without the PT decoder but processing frequent ToPA state checks, and KVM-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 KVM-PT. These benchmarks consider only the kernel core, but neither considers any kernel module. During KVM-PT execution only supervisor mode was traced.

To generate Intel PT load, `QEMU-2.6.0` was compiled within a traced VM using the `./configure` option `--target-list=x86_64-softmmu`. We restricted tracing to the whole kernel address space. 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 KVM-PT (see Figure 8). We ran three experiments to determine the overhead of the different components. In each experiment, we measured three different overheads: wall-clock time, user, and kernel. The difference in overall time is denoted by the wall-clock overhead. Additionally, we measured how much more time is spent in the kernel and how much time is spend only in user space. Since we only trace the kernel, we expect the users space overhead to be insignificant. Intel

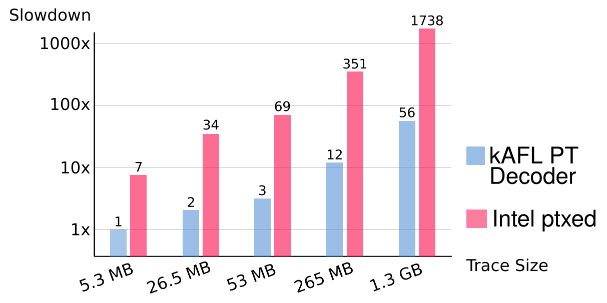

Figure 9: kAFL and `ptxed` decoding time on multiple copies of the same trace (kAFL is up to 30 times faster).

describes a performance penalty of < 5 % compared to execution without enabled Intel PT [30]. Accordingly, we expect approximately 5% of kernel overhead. In the first experiment, the traces were discarded without further analysis (KVM-PT). In the second experiment (KVM-PT & ToPA Check), we enabled repeated checking and clearing of the ToPA buffers. In the final experiment (KVM-PT & PT decoder), we tested the whole pipeline including our own decoder and conversion to an AFL bitmap.

During our benchmarks, an overhead between 1% – 4% was measured empirically. Since the resulting overhead is small, we do not expect it to have a major influence on the overall fuzzing performance.

5.8 Decoder Engine

In contrast to KVM-PT, the decoder has significant influence on the overall performance of the fuzzing process since the decoding process is—other than Intel PT and hence KVM-PT—not hardware-accelerated. Therefore, this process is costly and has to be as efficient as possible. Consequently, the performance of our developed PT decoder was compared to that of `ptxed`. This decoder is Intel's example implementation of an Intel PT software decoder and is based on `libipt`. 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 (`Linux debian 4.8.0-1-amd64`) traced by KVM-PT. This performance benchmark was processed on an i5-6500@3.2Ghz desktop system with 8GB DDR4 RAM. Only code execution in supervisor mode was traced. The generated sample is 9.4MB in size and contains over 431,650 TNT packets, each representing up to 7 branch transitions. The sample also contains over 100,045 TIPs. We sanitized the sample by removing anything but *flow information packets* (see Section 2.3) to avoid any influence of decoding large amount of *execution information packets*, since those are not considered by our PT decoder. The result is a $5.3MB$ trace file. To test the effectiveness of the caching approach of our PT decoder, we created cases containing 1, 5, 10, 50, and 250 copies of the trace. This is a realistic test case, since during fuzzing we see the same (or very similar) paths repeatedly. Figure 9 illustrates the measured speedup of our PT decoder compared to `ptxed`.

The figure also shows that our PT decoder easily outperforms the Intel decoder implementation, even if the PT decoder processes data for the very first time. This is most likely due to the fact that even a single trace already contains a significant amount of loops. Another possible factor is the use of Capstone [2] as the instruction decoding backend. As we decode more and more copies of the same trace, it can be seen that our decoder becomes increasingly faster (only using 56 times as much time to decode 250 times that amount of data). The *caching* approach outperforms Intel's implementation and is up to 25 to 30 times faster.

6 Related Work

Fuzzers are often classified according to the amount of interaction with the target program. For black-box fuzzers, the fuzzer does not use any information about the target program at all. White-box fuzzers typically use advanced program analysis techniques to uncover interesting properties of the target. Somewhere in the middle are so called gray-box fuzzers that will typically use some kind of feedback from the target (such as coverage information) to guide their search, without analyzing the logic of the target program itself. In this section, we provide a brief overview of the work performed in the corresponding areas of fuzzing.

6.1 Black-Box Fuzzers

The oldest class of fuzzers are black-box fuzzers. These fuzzers typically have no interaction with the target program beyond executing it on newly generated inputs. To increase effectiveness, a number of assumptions are usually made: Either a large corpus of good coverage inputs get mutated and recombined repeatedly. Examples for this class are Radamsa [3] or zzuf [12]. Or, the programmer needs to specify how to generate new semi-valid input files that almost look like real files. Examples including tools like Peach [6] or Sulley [9]. Both approaches have one very important drawback: It is a time-consuming task to use these tools.

To improve the performance of black-box fuzzers, many techniques have been proposed. Holler et al. [27] introduced learning interesting parts of the input grammar from old crashing inputs. Others even sought to infer the whole input grammar from program traces [13,24,38]. The selection of more interesting inputs was optimized by Rebert et al. [36]. Similar approaches have been used to optimize the mutation rate [17,40].

6.2 White-Box fuzzers

To reduce the burden on the tester, techniques where introduced that apply insights from program analysis to find more interesting inputs. Tools like SAGE [23], DART [22], KLEE [15], SmartFuzz [33], or Mayhem [16] try to enumerate complex paths by using techniques such as symbolic execution and constraint solving. Tools like TaintScope [39], BuzzFuzz [21] and Vuzzer [35] utilize taint tracing and similar dynamic analysis techniques to uncover new paths. These tools are often able to find very complicated code paths that are hidden behind checksums, magic constants, and other constraints that are very unlikely to be satisfied by random inputs. Another approach is to use the same kind of information to bias the search towards dangerous behavior instead of new code paths [26].

The downside is that these techniques are often significantly harder to implement, scale to large programs, and parallelize. To the best of our knowledge, there are no such tools for operating system fuzzing.

6.3 Gray-Box Fuzzers

Gray-box fuzzers try to retain the high throughput and simplicity of black-box fuzzers while gaining some of the additional coverage provided by the advanced mechanics in white-box fuzzing. The prime example for gray-box fuzzing is AFL, which uses coverage information to guide its search. This way, AFL voids spending additional time on inputs that do not trigger new behaviors. Similar techniques are used by many other fuzzers [8,25].

To further increase the effectiveness of gray-box fuzzing, many of the tricks already used in black-box fuzzing can be applied. Böhme et al. [14] showed how to use the insight gained from modelling gray-box fuzzing as a walk on a Markov chain to increase the performance of gray-box fuzzing by up to an order of magnitude.

6.4 Coverage-Guided Kernel Fuzzers

A project called syzkaller was released by Vyukov; it is the first publicly available gray-box coverage-guided kernel fuzzer [10]. Nossum and Casanovas demonstrate that most Linux file system drivers are vulnerable to feedback-driven fuzzing by using an adapted version of AFL [34]. This modified AFL version is based on glue code to the kernel consisting of a driver interface to measure feedback during fuzzing file system drivers of the kernel and expose this data to the user space. This fuzzer runs inside the targeted OS; a crash terminates the fuzzing session.

In 2016, Hertz and Newsham released a modified version of AFL called TriforceAFL [7]. 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 a control flow altering instruction has been executed. In theory, their fuzzer is able to fuzz any OS emulated in QEMU. In practice, the TriforceAFL 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, TriforceAFL is currently not able to fuzz closed-source operating systems such as macOS or Windows.

7 Discussion

Even though our approach is general, fast and mostly independent of the underlying OS, there are some limitations we want to discuss in this section.

OS-Specific Code. We use a small amount (usually less than 150 lines) of OS-dependent ring 3 code that performs three tasks. First, it interacts with the OS to translate the inputs from the fuzzing engine to interactions with the OS (e.g., mount the data as a partition). Second, it obtains the address of the crash handler of the OS such that we can detect crashes faster than it would take to wait for the timeout. Third, it can return the addresses of certain drivers. These addresses can be used to limit tracing to the activity of said drivers, which improves performance when only fuzzing individual drivers.

None of these functions are necessary and only improve performance in some cases. The first use case can be avoided by using generic syscall fuzzing. In that case a single standard C program which does not use any platform-specific API would suffice to trigger `sysenter/syscall` instructions. We do not strictly need the address of the crash handler, since there are numerous other ways to detect whether the VM crashed. It would also be quite easy to obtain crash handlers dynamically by introducing faults and analyzing the obtained traces. Finally, we can always trace the whole kernel, taking a slight performance hit (mostly introduced by the increased amount of non-determinism). In cases such as syscall fuzzing, we need to trace the whole kernel, therefore syscall fuzzing would not be impacted if this ability was missing. In summary, this is the first approach that can fuzz arbitrary x86-64 kernels without any customization and a near-native performance.

Supported CPUs. Due to the usage of *Intel PT* and *Intel VT-x*, our approach is limited to certain Intel CPUs supporting these extensions. Virtually all modern Intel CPUs support *Intel VT-x*. Unfortunately, Intel is rather vague as to which CPUs exactly support process trace inside of VMs and various other extensions (such as IP filtering and multi-entry ToPA). We tested our system on the following CPU models: Intel Core i5-6500, Intel Core i7-6700HQ, and Intel Core i5-6600. We believe that at the time of writing, most Skylake and Kabylake CPUs have the necessary hardware support.

Just-In-Time Code. 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 control flow information 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 is unable to exactly restore the runtime control flow. To bypass this limitation, the decoder requires information about all modifications applied to the program instead of an ordinary memory dump or the executable file. As Deng et al. [18] have shown, this is possible by making use of EPT violations when executing written pages. Another, somewhat more old-fashioned, method to achieve the same is to use shadow page tables [19]. Once one it is possible to hook the execution of modified code, self-modifying code can be dumped. Reimplementing this technique was out of the scope of this work. It should be noted though that fuzzing kernel JIT code is a very interesting topic since kernel JIT components, such as the BPF JIT in Linux, have often been part of serious vulnerabilities.

Multibyte Compares. Similar to AFL, we are unable to effectively bypass checks for large magic values in the inputs. However, we support specifying dictionaries of interesting constants to improve performance if such magic values are known in advance (e.g., from RFCs, source code, or disassembly). Some solutions involving techniques such as concolic execution (e.g., Driller [37]) or taint tracking (e.g., Vuzzer [35]) have been proposed. However, none of these techniques can easily be adapted to closed-source operating system kernels. Therefore it remains an open research problem how to deal with those situations on the kernel level.

Ring 3 Fuzzing. We only demonstrated this technique against kernel-level code. However, the exact same technique can be used to fuzz closed-source ring 3 code as well. Since our approach has a very modest tracing overhead, we expect that this technique will outperform current dynamic binary instrumentation based techniques for feedback fuzzing of closed-source ring 3 programs such as *winAFL* [20].

8 Conclusion

The latest generation of feedback-driven fuzzing methods has proven to be an effective approach to find vulnerabilities in an automated and comprehensive fashion. Recent work has also demonstrated that such techniques can be applied to 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 a lack of portability due to the need for compile-time instrumentations.

In this paper, we presented a novel mechanism to utilize the latest CPU features for a feedback-driven kernel fuzzer. As shown in the evaluation, combining all components provides the ability to apply kernel fuzz testing to any target OS with significantly better performance than the alternative approaches.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0592K HWSec). We would like to thank our shepherd Suman Jana for his support in finalizing this paper and the anonymous reviewers for their constructive and valuable comments. Furthermore, we would also like to thank Ralf Spenneberg and Hendrik Schwartke from OpenSource Security for supporting this research. Finally, we would like to thank Ali Abbasi, Tim Blazytko, Teemu Rytilahti and Christine Utz for their valuable feedback.

References

[1] Announcing oss-fuzz: Continuous fuzzing for open source software. `https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html`. Accessed: 2017-06-29.

[2] Capstone disassembly framework. `http://www.capstone-engine.org/`. Accessed: 2017-06-29.

[3] A general-purpose fuzzer. `https://github.com/aoh/radamsa`. Accessed: 2017-06-29.

[4] Intel Processor Trace Decoder Library. `https://github.com/01org/processor-trace`. Accessed: 2017-06-29.

[5] Linux 4.8, perf Documentation. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/tools/perf/Documentation/intel-pt.txt?id=refs/tags/v4.8`. Accessed: 2017-06-29.

[6] Peach. `http://www.peachfuzzer.com/`. Accessed: 2017-06-29.

[7] 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/`. Accessed: 2017-06-29.

[8] Security oriented fuzzer with powerful analysis options. `https://github.com/google/honggfuzz`. Accessed: 2017-06-29.

[9] Sulley. `https://github.com/OpenRCE/sulley`. Accessed: 2017-06-29.

[10] syzkaller: Linux syscall fuzzer. `https://github.com/google/syzkaller`. Accessed: 2017-06-29.

[11] Trinity: Linux system call fuzzer. `https://github.com/kernelslacker/trinity`. Accessed: 2017-06-29.

[12] zzuf. `https://github.com/samhocevar/zzuf`. Accessed: 2017-06-29.

[13] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[14] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[15] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[16] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy*, 2012.

[17] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[18] Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[19] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[20] Fratric, Ivan. WinAFL: A fork of AFL for fuzzing Windows binaries. `https://github.com/ivanfratric/winafl`, 2017.

[21] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)*, 2009.

[22] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[23] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20, 2012.

[24] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. Technical report, January 2017.

[25] P. Goodman. Shin GRR: Make Fuzzing Fast Again. `https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again/`. Accessed: 2017-06-29.

[26] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, 2013.

[27] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, 2012.

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

[29] A. Kleen. simple-pt: Simple Intel CPU processor tracing on Linux. `https://github.com/andikleen/simple-pt`.

[30] A. Kleen and B. Strong. Intel Processor Trace on Linux. Tracing Summit 2015, 2015.

[31] Microsoft. FSCTL_DISMOUNT_VOLUME. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa364562(v=vs.85).aspx`, 2017.

[32] Microsoft. VHD Reference. `https://msdn.microsoft.com/en-us/library/windows/desktop/dd323700(v=vs.85).aspx`, 2017.

[33] D. Molnar, X. C. Li, and D. Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[34] V. Nossum and Q. Casasnovas. Filesystem Fuzzing with American Fuzzy Lop. Vault 2016, 2016.

[35] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[36] A. Rebert, S. K. Cha, T. Avgerinos, J. M. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[37] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[38] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning. Experiences with model inference assisted fuzzing. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[39] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[40] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.