BSOD: Binary-only Scalable fuzzing Of device Drivers

Fabian Toepfer
fabian.toepfer@posteo.de
TU Berlin
Berlin, Germany

Dominik Maier
dmaier@sect.tu-berlin.de
TU Berlin
Berlin, Germany

ABSTRACT

Operating system code interacting with the devices attached to our computers, device drivers, are often provided by their respective vendors. As they may run with kernel privileges, this effectively means that kernel code is written by third parties. Some of these may not live up to the high security standards the core kernel code abides by. A single bug in a driver can harm the complete operating system's integrity, just as if the bug was in the kernel itself. Attackers can exploit these bugs to escape sandboxes and to gain system privileges. Automated security testing of device drivers is hard. It depends on the attached device, and the driver code is not freely available. Dependency on a physical device increases the complexity even further. To alleviate these issues, we present BSOD, a fuzzing framework for high-complexity device drivers, based on KVM-VMI. BSOD retargets the well-known and battle-proven fuzzers, Syzkaller and AFL++, for binary-only drivers. We do not depend on vendor-specific CPU features and exceed 10k execs/sec on COTS hardware for coverage-guided kernel fuzzing. For evaluation, we focus on the highly complex closed-source drivers of a major graphics-card vendor for multiple operating systems. To overcome the strict hardware dependency of device driver fuzzing, making scaling impractical, we implement BSOD-fakedev, a virtual record & replay device, able to load a full graphics card driver without a physical device attached. It allows to scale fuzz campaigns to a large number of machines without the need for additional hardware. BSOD was able to uncover numerous bugs in graphics card drivers on Windows, Linux, and FreeBSD.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

Binary-Only, Fuzzing, Virtualization, Kernel Space, Drivers

ACM Reference Format:
Fabian Toepfer and Dominik Maier. 2021. BSOD: Binary-only Scalable fuzzing Of device Drivers. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3471621.3471863

RAID '21, October 6–8, 2021, San Sebastian, Spain

ACM ISBN 978-1-4503-9058-3/21/10…$15.00
https://doi.org/10.1145/3471621.3471863

1 INTRODUCTION

If users without administrative rights were able to freely execute kernel code on systems like Windows, Linux, or FreeBSD, they could alter their permissions directly in kernel memory to regain these rights. Hence, in major operating systems, kernel code has a trust boundary, shielding it from normal applications and their users. Still, the amount of code running in the kernel of a modern operating system is huge. While the core components of an operating system are usually well-tested, a large zoo of additional device drivers from various vendors runs within the kernel as well. Since device drivers must interact with user-mode processes and hardware devices, they open up a large attack surface. The drivers, often implemented as kernel modules, have extensive control inside the kernel, which means that bugs or other forms of misbehavior can have a significant impact on the overall system stability. The threat is present in real-world products. In the Linux kernel, device drivers constitute the majority of vulnerabilities [16]. In the past, vulnerabilities were also discovered in the closed sourced kernel components of graphic drivers [4, 18, 25, 28].

Since drivers are usually written in low-level languages like C and C++, they can contain high-severity bugs that could lead to memory corruptions. To uncover these issues, the security community had good success with fuzzing in recent years. Fuzzing kernels, however, is a more complex task than the common userspace fuzzing, since it involves running a full operating system [17]. Several kernel fuzzing approaches exist that depend on hand-crafted interface descriptions [11, 19] or utilize hardware-assisted coverage feedback [33, 40], but to the best of our knowledge, no fuzzers for complex closed-source device drivers, such as those of graphics cards, publicly exist.

Most personal computers and laptops contain GPUs from either NVIDIA, AMD, or Intel. The gaming platform *Steam* collects monthly data about what kind of hardware their customers are using, including the distribution of used GPUs by the vendor. Their Hardware Survey[1] from January 2021 states that 74.41% of users use an NVIDIA hardware. All of these customers are likely to run the official drivers to make full use of their hardware. A bug in one of the drivers, therefore, can be abused to attack a large share of the computing market.

To safeguard these drivers, we present BSOD, a framework that allows to fuzz binary-only drivers. After a researcher gains enough knowledge about the target's interfaces, they can set up an AFL++ or Syzkaller fuzzing campaign for the real device, using device passthrough. Then, they can record traces and replay them on other machines to parallelize fuzzing through BSOD-fakedev. This virtual device, together with a fast breakpoint-based coverage method originally proposed for userspace applications by Nagy and Hicks [27],

[1] https://store.steampowered.com/hwsurvey

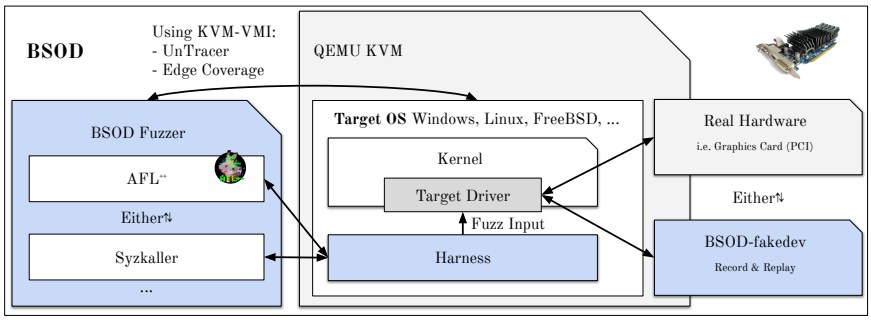

Figure 1: BSOD overview: The supported fuzzers create inputs on the host system and gather, interact with QEMU/KVM using KVM-VMI, and collect coverage feedback. The target OS executes each testcase against the driver, which either interacts with real hardware, or with the BSOD-fakedev, a virtual replay of the real graphics card.

allow us to scale the fuzzing campaign to any hardware that can run QEMU/KVM — without attaching a physical devices to all virtual machines. Figure 1 depicts an overview of the components of BSOD.

Contributions

- We develop and open-source *BSOD*, a framework for scalable binary-only device driver and kernel fuzzing.
- BSOD includes hardware-agnostic untracer-style coverage collection for kernels.
- BSOD allows fuzzing with AFL++ and Syzkaller. *BSOD* is the first public use of Syzkaller for binary-only driver fuzzing.
- BSOD-fakedev is a virtual device that records and replays PCI interactions. With it, we can load, and interact with, complete graphics drivers.
- We fuzz the NVIDIA graphics drivers for Linux, FreeBSD, and Windows, and uncovered multiple bugs.

2 BACKGROUND

Fuzzing is known to be an effective solution to uncover bugs by executing generated inputs on target binaries [13, 34, 39]. Most recent fuzzers rely on coverage information, which effectively guides them through the program. Whenever a test case hits a previously unseen program location, the fuzzer adds it to the corpus as a basis for newly generated test cases [11, 39]. If the program's source code is available compilers can add instrumentation during the build process [14, 34, 39]. For testing of binary-only targets, instrumentation can be either achieved dynamically at run-time through emulation with considerable overhead [39] or through binary rewriting [8].

In the following, we will further provide a quick introduction to the attack surface in drivers in general, the case-study NVIDIA driver in particular, as well as related work.

2.1 Kernel Driver Attack Surface

Device drivers are kernel modules and run in most privileged ring 0, which means they have full control over the system. A kernel module can execute arbitrary code and has arbitrary read and write primitives even for system-critical information that would render any security mechanisms useless.

User-space applications that access the hardware devices need to interface with the device drivers running in kernel space. For this purpose, there exists the Input/Output Control (ioctl) system call `int ioctl(int fd, unsigned long cmd, ...);`.

From an attacker's point of view, system calls are an interesting entry point since they allow executing code in kernel mode. Especially the ioctl system call provides a large attack surface since it is generally defined so that the expected data and the correct handling depend on the driver's implementation. It has a high chance to contain programming bugs that unexpected input data could trigger, leading to memory corruption or unintended behavior. An attacker could pass specifically crafted inputs to exploit these bugs to gain higher privileges, perform arbitrary memory reads or writes, or crash the system. When only proprietary device drivers are available for specific hardware devices, users have no other choice than to trust the hardware vendor's binaries. It is hard to know if these trusted modules contain security bugs.

In our work, we focus on the scenario in which the attacker is an unprivileged user that can access the driver's device files. Most components of the system have access to our target, the NVIDIA drivers. Potential code execution in a fundamental driver like that of a graphics card will, in most scenarios, grant an attacker full kernel code execution.

While another possible source of untrusted data into the driver can be the hardware device itself, we exclude it from this work. Several systems [21, 29, 32] already covered this scenario.

2.2 NVIDIA Kernel Driver

In 2017, Google Project Zero [4] targeted the NVIDIA driver for the Windows operating system and uncovered multiple bugs that reside in the DxgkDdiEscape interface and in the exposed device nodes that are accessible from user-mode applications. Even though, in contrast to Windows, the Linux kernel is open source, the official NVIDIA driver for Linux is proprietary, too. This means the kernel components and the user-land implementations of the graphics API standards are distributed in closed-source binary form.

The proprietary driver package supports a wide range of device chips and generations. It consists of four kernel modules shown in Figure 2 with their respective dependencies and licenses.

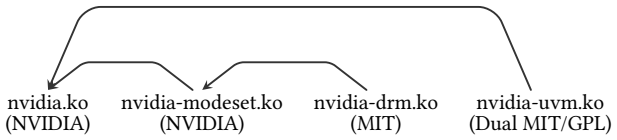

Figure 2: NVIDIA module dependencies on Linux

nvidia.ko. The main module comes with the closed-source binary `nv-kernel.o_binary` and only some source code needed to interface with the running Linux kernel. When the driver is loaded, it registers a control device and per GPU devices to be accessible from user-mode applications via the device files `/dev/nvidiactl` and `/dev/nvidia0`.

nvidia-modeset.ko. The module is responsible for retrieving and setting the appropriate display properties by reading Extended Display Identification Data (EDID) information from connected display devices. It comes bundled with another closed-source binary, `nv-modeset-kernel.o_binary`, and source code to interface with the kernel, similarly to the main module. It registers a control device that is accessible via the device file `/dev/nvidia-modeset` from user-mode applications.

nvidia-drm.ko. This module implements the Direct Rendering Infrastructure (DRM) interface that is typically used by the X window system. DRM exposes device files under `/dev/dri/card0` for control and `/dev/dri/renderD128` for rendering.

nvidia-uvm.ko. The module provides the unified virtual memory feature to use a single memory address space accessible by CPU and GPU, which is typically used by CUDA applications. It registers two devices that are accessible via the device files `/dev/nvidia-uvm` and `/dev/nvidia-uvm-tools` for user-mode applications.

According to our reverse engineering efforts, the drivers for all platforms, Windows, Linux, and FreeBSD appear to share a single codebase. For instance, when comparing the Linux and FreeBSD driver installation packages, they have a similar structure and include both the identical binary modules `nv-kernel.o_binary` and `nv-modeset-kernel.o_binary` together with a few source code that interfaces with the kernel. The Windows kernel driver `nvlddmkm.sys` also shares similarities in some functions and the included text strings.

2.3 Related Work

UnTracer. Coverage-guided fuzzers search for inputs that trigger previously unseen code paths to increase the coverage. Statistically, such inputs are typically infrequent since similar inputs often result in the same code paths. Unfortunately, the coverage tracing overhead is always the same, and inputs considered uninteresting are discarded.

To minimize the coverage tracing overhead while fuzzing, especially for the majority of uninteresting inputs, the authors proposed an implementation called *UnTracer* [27]. For the approach, they use two versions of the program under test, which are an interest oracle and a tracer binary. The purpose of the interest oracle is to determine whether a test case reached a previously unseen code location in the program. It's realized as a modified binary with inserted software breakpoints at the start of every basic block. Whenever a breakpoint triggers, the respective test case will be re-executed on the tracer binary with full coverage tracing enabled. Afterward, the system removes the breakpoints of all reached basic blocks from the interest oracle. To realize the concept, the authors created a customized version of AFL that generates the test cases for the interest oracle. The benchmarks have shown that this approach outperforms AFL's QEMU mode [39] and reaches nearly identical performance compared to fuzzing with Intel PT. Encouraged by these positive results, we implemented UnTracer-style coverage for BSOD.

RetroWrite. RetroWrite [8] is a method to add instrumentation to binaries to support AFL or ASAN through binary rewriting. It allows performing security analysis like fuzzing on closed-source targets. Binary rewriting is an involved process that requires recalculating and updating all pointer offsets so that the binary still executes correctly. The difficulty is to distinguish between reference and scalar constants. In our tests using NVIDIA's drivers, RetroWrite could not produce a usable kernel module. This is not surprising, given the scope of the driver.

kAFL. kAFL [33] is a kernel fuzzer for multiple operating systems that leverages Intel PT for hardware-assisted coverage feedback. It uses the KVM hypervisor and QEMU [2] to emulate the target operating system, in which the fuzzing takes place. To trace the guest VMs exclusively, they created customized versions of the KVM kernel component, namely KVM-PT, and the user-space system emulator QEMU, namely QEMU-PT. The KVM-PT component enables and disables the tracing via the host's CPU MSR registers on VM-Enter and VM-Exit calls, respectively. Inside the guests, specific agent programs wait for test cases and trigger the fuzzing loop. The way BSOD interacts with AFL++ resembles the concepts of kAFL. However, we chose UnTracer-style coverage collection over Intel PT, to be able to also fuzz drivers that only run on AMD CPUs, where Intel PT is not available, as well as to be able to scale to servers with ease.

Agamotto. Many techniques for fuzzing kernel-mode drivers exist, but most of them involve performance issues due to costly execution of kernel code, interference of test cases, or kernel crashes. Agamotto [36] introduces lightweight virtual machine checkpoints to improve the throughput of kernel driver fuzzing. Based on the observation that fuzzers frequently execute similar test cases in a row, the authors improved the performance by continuously creating checkpoints during the fuzzing execution and skipping identical parts of other test cases by restoring related checkpoints. Agamotto increased the speed of Syzkaller [11] by 66,6% on average when fuzzing 8 USB drivers, whereby it skipped 35,6% of test case executions. Additionally, the approach achieved a speed improvement of 21,6% when fuzzing PCI drivers with AFL [39].

Unicorefuzz. Unicorefuzz [24] allows the fuzzing of code in kernel space by leveraging CPU emulation based on the Unicorn engine. The setup uses QEMU [2] for the creation of the initial system state, AFL's Unicorn mode [13] for fuzzing, and Avatar2 [26] for

interaction with QEMU's exposed GDB stub. The system creates a breakpoint for that address so that the VM halts once the breakpoint gets triggered. After hitting the start address, the system synchronizes the CPU's state of the VM with the Unicorn engine that starts fuzzing the code right after the breakpoint. This methodology also allows to fuzz initialized drivers, albeit without further hardware interaction. On top, the emulated execution speed is rather low [24].

Difuze. Difuze [7] is a framework for interface recovery of kernel drivers for the Android platform. Its purpose is to enable interface-aware fuzzing on the ioctl interface of device drivers. The command and data structures of ioctl system calls are driver-dependent and can contain multiple pointers and substructures. From a security perspective, it has a high chance of containing vulnerabilities. To trigger deeper code paths these structures need to be valid, which requires meaningful fuzzing input choices. Otherwise, the target would likely early reject the inputs without reaching interesting program locations that might contain bugs. To extract the data structures, Difuze applies static code analysis of kernel drivers that requires to have access to the source code. Since the target devices are mobile devices running Android that contain vendor-specific hardware, such as GPS sensor, accelerometer, or camera, they have to include their drivers into the kernel tree that vendors must publicly release thanks to the GPL license. The process consists of three steps that are interface recovery by static analysis of the kernel source code, structure instance generation based on the prior extracted information, and execution of the generated instances on the actual device.

P2IM. Feng et al. proposed P^{2}IM [12], a firmware rehosting framework. P^{2}IM abstracts peripherals and handles firmware I/O to rehost the firmware of embedded devices fully automated. It enables peripheral-oblivious emulation and can be used to fuzz firmware of embedded devices with AFL. Of the 70 firmware and 10 real devices, P^{2}IM managed to executed 79% of the sample firmware without any manual assistance.

avatar2. Avatar2 [26] is a multi-target orchestration framework that allows dynamic analysis of embedded firmware images. Avatar2 provides interoperability between different dynamic binary analysis platforms, emulators, debuggers, and real hardware devices. The analyst can define topologies and specify events to transfer the state of the memory and CPU registers from one system to another. Partial emulation is achieved by executing the firmware via an emulator and forwarding the memory-mapped peripherals to the real hardware devices.

USBFuzz. USBFuzz [29] is a framework that targets the fuzzing of USB drivers inside QEMU [2]. Since hardware devices become costly when larger fuzzing campaigns require multiple instances, the authors created an emulated software USB device, configurable to imitate different USB devices by specifying device and vendor IDs. This allows drivers to recognize and bind the devices to properly initialize and operate. With this approach, the fuzzing scales way better, and the utilization of the system's resources is more efficient. The fuzzer operates from the device side, which means the device responds to requests with fuzzing inputs. To retrieve coverage feedback for Linux, the approach uses an instrumented kernel built with KCOV and KASAN features enabled, which limits coverage feedback to open-source drivers. Although the work targets USB devices, the approach shares similarities with our work. Instead of creating an emulated USB device, we need an emulated PCI device for improving the fuzzing process. Furthermore, our work is not limited to open-source drivers to retrieve coverage feedback and supports closed-source targets.

Periscope. PeriScope is a Linux kernel based probing framework that allows fine-grained analysis of device-driver interactions [35]. It allow researchers to monitor and log traffic between device drivers and hardware. On top of it, the authors built *PeriFuzz*, a fuzzer that is able to fuzz device drivers from the device side, uncovering bugs that can be exploited to exploit the kernel from rogue devices. The fuzzer found 15 unique vulnerabilities in the Wi-Fi drivers of two flagship Android smartphones, including 9 previously unknown ones.

3 BSOD DESIGN

BSOD is a framework to fuzz complex binary-only kernel device drivers using AFL++ and Syzkaller. It works for drivers with forwarded hardware devices, and offers the option to replace the hardware device with a virtualized record & replay device, BSOD-fakedev altogether. To make it cloud-friendly, we do not rely on strong hardware dependencies, like Intel PT. In this section, we will present BSODs design.

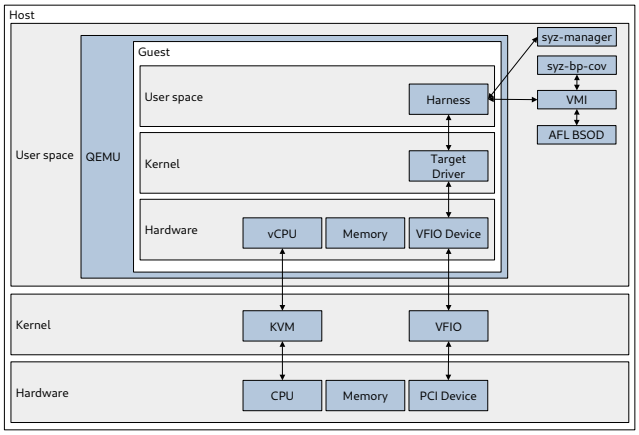

Figure 3: Detailed interactions of BSOD's components.

Figure 3 depicts an overview of the experimental environment used to analyze kernel drivers in conjunction with real hardware devices during our work. It uses virtualization-based on QEMU's full-system emulation [2] with Kernel-based Virtual Machine (KVM) acceleration to drive a guest running an operating system with native performance. To make physical hardware devices available inside the guest, we pass through host devices with the help of Virtual Function I/O (VFIO). To fuzz with BSOD the analyst has to determine how applications interact with target and what kind of data the driver expects as a one-time manual effort. To reveal the needed information, we trace the interactions and data exchanged between exemplary applications, the drivers, and the devices. Afterward, we inspect the collected data traces to infer parts of the

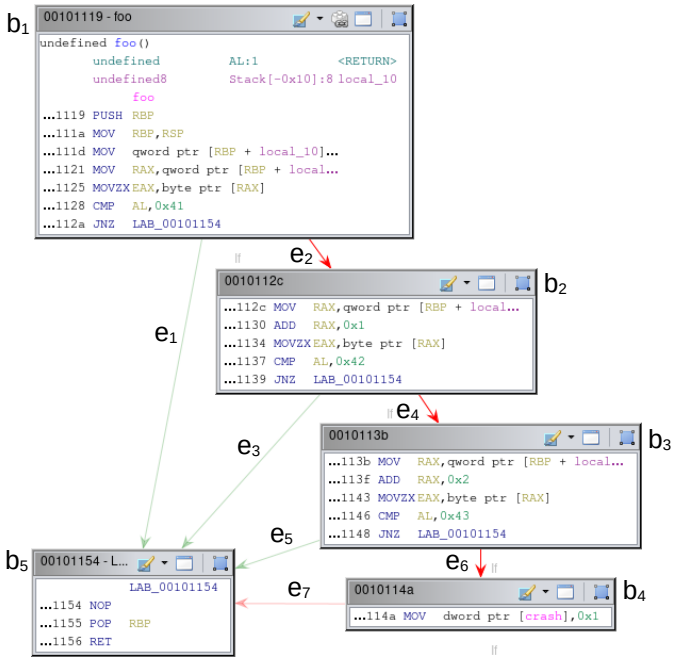

Figure 4: Control-flow graph extracted with GHIDRA [1]

drivers' functionalities and use these insights to set up the fuzzing procedure. As a final step, BSOD can eliminate the hardware dependency.

In the following, we introduce the used concepts. BSOD connects to our experimental environment by using `libvmi` [22], a library that provides an introspection API for different hypervisors. For the KVM hypervisor, the KVM-VMI [37] project forms the basis, which combines the KVM kernel module, QEMU, `libkvmi`, and `libvmi`. For the practicability of our setup, we modified the *libvmi* library to be usable without the *libvirt* virtualization API in between.

Since we cannot instrument binary-only drivers through recompilation, we need to establish a different way of getting coverage information.

```
1  void foo(char *input) {
2      if (input[0] == 'A') {
3          if (input[1] == 'B') {
4              if (input[2] == 'C') {
5                  crash = 1;
6              }
7          }
8      }
9  }
```

Listing 1: Exemplary program source code

Control-Flow Tracing. Listing 1 shows exemplary program source code and Figure 4 the related Control-Flow Graph (CFG) of the compiled binary. To trace the program flow during execution, we must determine the successive basic block when the CPU encounters a conditional control-flow instruction. In this example, the tracer replaces the first byte of the `JNZ` instructions at the offsets `0x112a`, `0x1139`, and `0x1148` with `0xcc`. When reaching one of these code locations, the tracer receives an interrupt to determine the following executed block, which depends on whether the branch will be taken or not that can be tracked by executing a single step. Before continuing execution, the tracer restores the original instruction byte to execute normally. The tracing of all the block transitions during the program's execution reveals the full path through the CFG.

Edge Coverage Mode. Tracking individual branches instead of only basic blocks is the coverage method of choice, where available [13]. In the following, we will explain how we implement edge coverage mode in BSOD.

(1) **Preprocess target module**
In the first step, we preprocess the `.text` section of the target kernel module to extract all the offsets of conditional control-flow instructions and both possible target basic block addresses into a file. We use *capstone* [31] to disassemble the binary module. This step is only needed once per target module.

(2) **Initialize fuzzing environment**
Then, the guest VM is booted, and the target kernel module is loaded. The beginning of the module's `.text` section in memory is determined by reading the module's load address from `/proc/modules`. Afterward, BSOD starts and takes the module load address and the file containing the extracted offsets of control-flow instructions as arguments. It connects to the introspection API, pauses the VM, and registers two events, namely `BREAKPOINT`, which triggers whenever encountering an INT 3 breakpoint exception and `SINGLESTEP` that triggers after executing an instruction in single-step mode. Then, it will back up and replace the first byte of every control-flow instruction according to the previously extracted offsets with a software interrupt. The VM will be resumed afterward and is prepared for the harness to start.

(3) **Execute test case**
Whenever a software interrupt occurs during the execution of the target function, BSOD catches and handles it according to the algorithm shown in Figure 4, in which M refers to the memory, B refers to the instruction backups, and COV refers to the collected coverage. It reads out the address of the encountered control-flow instruction of the CPU's `RIP` register and restores the `0xcc` byte to the saved opcode. When the current Process ID (PID) belongs to the harness, the CPU is switched into the single-step mode to trace which of the two possible paths are taken. After the single-step, BSOD receives an interrupt again and reads the new `RIP` of the CPU. The module load address is subtracted from both addresses to maintain consistent coverage information across reboots. Subsequently, both basic block addresses are linked using XOR to represent the edge and reported to the fuzzer.

UnTracer-Style Block Coverage Mode. Nagy and Hicks [27] show a method to get reasonably fast performance metrics out of purely breakpoint-based instrumentation if the fuzzer applys smart tracing. In their measurements, removing hit breakpoints after the first hit still performed well for coverage guided fuzzing, while reaching near-native execution speed after a while. Inspired by the presented fuzzer *UnTracer* for user-mode applications, we adapted the method for fuzzing modules in kernel space.

Algorithm 1 Breakpoint coverage

```
while wait for new event do
    event ← readevent
    if event == breakpoint then
        M[rip] ← B[rip]
        if pid == tracepid then
            singlestep ← true
            COV ∪ rip − base ⊕ prev_loc − base
            prev_loc ← rip
        end if
    else if event == singlestep then
        M[prev_loc] ← 0xcc
        singlestep ← false
        COV ∪ rip − base ⊕ prev_loc − base
        prev_loc ← rip
    end if
end while
```

We modified the described edge coverage mode above to reduce the amount of reoccurring interrupts that are costly to handle and consequently lower the tracing accuracy to improve the fuzzing throughput. Here, BSOD sets breakpoints on the offsets of control-flow instructions and both reachable basic blocks. Whenever a breakpoint triggers, it reports the current `RIP` register for coverage feedback but omits the reinjection of the breakpoint so that it only triggers once.

Therefore, the granularity of the coverage feedback per test case reduces to previously unseen basic blocks, which lowers the number of inputs considered interesting. For example, when an input encounters a new edge whose origin and target blocks were already reached previously, it can not be spotted as new interesting behavior.

3.1 BSOD-AFL

We implemented an AFL++ proxy [13] that integrated AFL++ into BSOD using libvmi. Initially, it was derived from the *kernel-fuzzer-for-xen* [23] project and retargeted for KVM. Figure 3 shows the overall system overview of the fuzzing setup. The host system runs AFL and QEMU, while the fuzzing takes place in the guest system, which runs the target kernel module. Based on the collected information, the analyst must create a target-specific harness that implements the respective API functions to fuzz. Listing 2 shows the minimal harness boilerplate code. At first, the harness allocates a buffer and issues a hypercall to share the buffer's address and length. We implemented the hypercalls as software interrupts with prepared register values, which are handled by BSOD accordingly. The `rax` register contains a specific magic value that encodes the command, and the registers `rbx` and `rcx` hold the arguments that are the address and length, respectively. BSOD translates the buffer's virtual address to the physical address through page-table lookup. After starting the harness inside the guest, it will first allocate a buffer and share the address and length via a hypercall. Since the scheduling of other processes can happen while a whole operating system is running, it could be possible that other processes hit breakpoints during the execution of a test case. To ensure to collect only the related coverage of the harness, BSOD determines the respective PID of the current process by reading it from the `struct task_struct`. Afterward, the harness enters the fuzzing loop and issues at first in every iteration a hypercall to request a new test case into the created buffer. Then, it calls the target function and fills the arguments with the buffer's content.

We create breakpoints at the basic block addresses of the Linux kernel error handlers, such as `oops_begin`, `panic`, and `kasan_report` to detect faults that we report to AFL. When the system has to halt due to a critical fault, the VM needs to reboot and initialize again to continue the fuzzing process.

```c
#include <stdlib.h>
#include <string.h>

#define HYPERCALL_BUFFER    0x1337133713371338
#define HYPERCALL_TESTCASE  0x1337133713371337
#define LENGTH              0x10000

int main(int argv, char **argc) {
    // Allocate input buffer
    char *buffer = malloc(LENGTH);
    memset(buffer, 0, LENGTH);

    // Hypercall to signal buffer address and length
    asm("int $3" ::"a"(HYPERCALL_BUFFER), "b"(buffer), "c"(LENGTH));

    // Fuzzing loop
    while (1) {
        // Hypercall to request new test case
        asm("int $3" ::"a"(HYPERCALL_TESTCASE));

        // Call target function
        target_function(buffer);
    }
}
```

Listing 2: BSOD-AFL Harness boilerplate

3.2 BSOD-Syzkaller

Syzkaller [11] is a system call kernel fuzzer, predestined in the context of device driver fuzzing on the ioctl interface. It uses virtualization, based on QEMU [2], to run one or multiple worker guests in which the fuzzing takes place.

When the fuzzer inside the guest executes a program that reaches new program locations, according to coverage, it gets re-executed multiple times to verify functionality, reduce noise, and minimize the testcase, in the so-called triage phase. The minimized program and the coverage information are sent to the manager on the host. The manager adds the new program to the input corpus from which the fuzzer subsequently generates the programs through mutation of the parameters and updates the overall reached coverage. The reached basic block addresses are mappable to the respective source code lines of the kernel image.

In the case of closed-source targets, the common KCOV kernel feature can not provide coverage information for binary modules. To overcome this hurdle, we hook up BSOD as new coverage source, that prototypically emulates the functionality of KCOV based on software breakpoints in a very similar way to the setup described in subsection 3.1.

BSOD includes a tool named *syz-bp-cov* to run alongside the Syzkaller manager on the host. It also relies on the introspection capabilities of VMI to receive interrupts and to access the guest's memory. We modified the *libvmi* library and removed the *libvirt* dependency, an extensive virtualization API, for seamless integration in Syzkaller, since the manager unit already controls the QEMU processes. We extended the startup routine of Syzkaller to start the tool before it executes the fuzzer inside the guest. In the first step,

the tool connects to the introspection API of the guest VM and replaces the target's control-flow instructions with software breakpoints those offsets also require to be extracted beforehand. In the next step, it registers the two events `BREAKPOINT` and `SINGLESTEP` that are used for control-flow tracing according to Figure 4 from the first setup.

We modified the Syzkaller executor in terms of dropping out the interactions with the KCOV interface. Instead, it allocates a coverage buffer and signals its memory location via a hypercall realized as a software interrupt with prepared register values. Whenever the CPU reaches a breakpoint during the execution of a Syzkaller program, the basic block address is reported according to the current `RIP` register value and written into the dedicated coverage buffer at the position of the current size value that is increased afterward. Since the module load addresses vary across reboots, the tool rebases the reached basic block addresses into a fixed address range per module. To avoid having to manage coverage buffers for each thread and the corresponding mapping for each basic block reached, the experimental setup is limited to a single fuzzer process that runs in non-threaded mode per VM.

In BSOD-Syzkaller, reached breakpoints are not reactivated and only trigger once, which requires some changes to the Syzkaller logic. Since Syzkaller typically uses edge coverage that chains two successive basic blocks together, it would likely create edges depending on the state of enabled breakpoints that are not existing. This effect influences the assessment of whether an input triggers new behavior and is added to the corpus. To solve the problem, we modified Syzkaller to use block coverage that lowers the accuracy to maintain correct functionality. Consistent coverage feedback is required to function correctly to keep the triage phase intact for inputs that reach new program locations. Therefore, when entering this phase, the fuzzer issues a hypercall dedicated to *syz-bp-cov*. It activates all breakpoints again and continuously reactivates them when they are triggered so that it traces the full set of reached basic block addresses accordingly until the fuzzer leaves the triage phase and signals it via another hypercall. Fuzzing with Syzkaller requires the tester to provide interface descriptions for the target.

```
1  resource fd_nvidiactl[fd]
2  openat$nvidiactl(fd const[AT_FDCWD], file ptr[in, string["/dev/nvidiactl"]],
       flags flags[open_flags], mode const[0]) fd_nvidiactl
3  ioctl$NVRM_IOCTL_CREATE(fd fd_nvidiactl, cmd const[0xc020462b], arg ptr[inout
       , nvrm_ioctl_create_t])
4
5  nvrm_ioctl_create_t {
6      cid int32 (in)
7      par int32 (in)
8      handle  int32 (inout)
9      cls int32 (in)
10     ptr ptr64[in, int32]
11     status  int32 (out)
12     _pad  int32
13 }
```

Listing 3: Syzkaller ioctl system call description

Based on recovered ioctl structures, the creation of equivalent Syzkaller descriptions is straightforward, exemplary shown in Listing 3. The first line defines a resource to hold the file descriptor of the device node, which will be opened in the next line. The third line defines an ioctl system call according to the function definition which takes the file descriptor, the `cmd` parameter, and a pointer of an expected parameter structure as arguments. The definition of the parameter structure starts from the fifth line, and each line between the brackets defines a field with name and type.

4 PCI DEVICE RECORD & REPLAY

Fuzzing of device drivers requires running an operating system containing a supported hardware device so that the target driver can initialize and operate. Unfortunately, when fuzzing with physical hardware devices via PCI passthrough, it requires one separate device for each guest, in which the fuzzing takes place. Furthermore, the fuzzer could trigger operations that bring the device into an unstable state or even break it. To overcome these issues, we create a QEMU virtual device, able to replay large parts of the real PCI devices. It enables fuzzing execution on systems that do not include the physical device and allows scaling up using virtualization technology to utilize all the system resources more efficiently. Several similar attempts address this issue for different hardware families.[6, 15, 29, 30, 41]

4.1 PANDA-based Deterministic Recordings

Analyzing any driver behavior in full-system emulation involves challenges due to the nondeterminism of the operating system and the hardware, due to the state differences of the OS and the physical device, the memory addresses, order of MMIO operations, interrupt timings, and context switches change.

Non-Determinism Hinders Fuzzer Development. As valuable improvement during development, we eliminate non-determinism by recording the execution of the operating system for repeatable analysis on top of PANDA, a QEMU-fork [10]. We can use the QEMU's VFIO device to pass through real hardware devices to PANDA. When starting a recording, PANDA relies on the snapshot capabilities of QEMU, which require devices to be migratable, but migration is not possible with the VFIO device. We defined the device to be migratable so that it is added to the snapshot and stubbed out the device initialization routines that are called when a snapshot is loaded. With these changes in place, the record and replay capabilities of PANDA with the VFIO device are working as expected for MMIO operations and interrupts, except for DMA transfers. Virtual devices typically use a specific function to map memory for DMA transfers, and PANDA adds these addresses to the watchlist. This method does not work for real hardware devices, which means the DMA pages need to be collected differently.

BSOD Panda Plugins. To cover the recording of the missing DMA operations without customizing PANDA itself, we implemented a PANDA plugin. During recording, the plugin tracks the allocated DMA pages by using PANDA's memory callbacks [10]. The plugin directly hooks the functions responsible for allocating the DMA buffers inside the kernel driver module to retrieve the page addresses. Furthermore, an event is registered that triggers before any memory read access occurs by using the `PANDA_CB_PHYS_MEM_BEFORE_READ` callback. Whenever a read access relates to a monitored DMA page, the plugin checks whether the content changed by comparing the checksum of the current bytes with the previously stored checksum. If the content is changed, the page is dumped and stored together with the current instruction counter. During replay, the plugin also uses the previously used callback to write changed pages due to DMA just-in-time before the driver accesses the data according to the stored instruction counter. This eliminates nondeterminism and improves debugging by having GDB attached

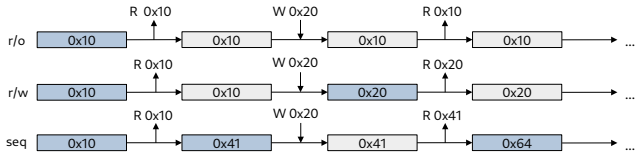

Figure 5: Memory replay

during replay. Furthermore, it enables the use of expensive analysis plugins that no longer influence the execution.

Next, we created a plugin that intercepts ioctl system calls directed to the driver. It is realized by hooking the related ioctl handler function in the driver to dump the request parameters and passed data structures of any performed ioctl system call during the replay into files. It also supports taint analysis [9, 38] by labeling the input bytes to the ioctl system call invocation to track the propagation of the data. We can determine the influenced program locations by using the tainted branch and tainted instruction plugins.

4.2 BSOD-fakedev

To remove the hardware device dependency for the fuzzing process, we implement a virtual replay device, BSOD-fakedev, that can be plugged into any QEMU instance. We determine the properties by reading out the PCI configuration space of the device. The `lspci` utility allows showing the information in a human-friendly readable format. First, we determine the values for the vendor ID, device ID, and class ID, and specify them in the class properties of our device implementation. Then, we must determine the layout of the Base Address Registers (BARs) with its sizes and access properties. In our QEMU device implementation, we define appropriately sized memory areas and IO operations for each BAR. Afterward, we define an IO memory region for each BAR with the respective size and IO operations by using QEMU's defined `memory_region_init_io` function. Furthermore, we define the BAR for our device with the respective access types and bind it to the respective memory region by using QEMU's `pci_register_bar` function.

At this point, we have created our virtual device according to the device properties without implementing any functionality. For the handling of memory events, we need to implement the handlers of the IO operations for read and write accesses. To approximate the device's functionality, we implement a memory replay logic that works on previously captured trace data. We define the memory access length to 4 bytes and classify each addressable memory location to be either of type *read-only*, *read-writable*, or *sequential*.

To demonstrate the concept, Figure 5 shows an exemplary excerpt of device memory. For read-only memory locations, our implementation discards any writing attempt so that they will always remain the initial value. Read-writeable memory locations are treated as typical memory and served from the allocated memory areas of the virtual device. For memory locations of type sequential, we implemented a special treatment that serves read accesses with sequential data. Here, successive read events return the values according to the previously captured trace data, and write attempts are discarded. Interrupts are bound to the previously occurred read or write event and triggered after the event occurred.

We link our implementation to the handler functions of the IO operations for the memory regions. The handler functions are extendable to implement more device-specific custom logic, which is useful when the replay concept does not work for single memory addresses.

For the data collection, we use our presented experimental setup shown in Figure 3 in section 3. We use the VFIO device, located between the physical device and the guest, as a proxy to intercept Memory-Mapped I/O (MMIO) events and interrupts by leveraging its tracing capabilities. We set the VFIO option `x-no-mmap=true`, which disables the direct mapping of the device memory regions into the guest to enable the trace events `vfio_region_read` and `vfio_region_write` that trigger on each memory access into the BARs. Furthermore, we enable the option `x-no-kvm-intx=true` to collect interrupts in KVM mode. Then, we boot our virtual machine and trigger device actions, such as executing an exemplary application that interacts with the device.

Afterward, we analyze the trace data in the following way:

(1) **Preprocess trace data**
In the first step, we parse the QEMU trace log and extract the data of the occurred memory and interrupt events.
(2) **Split memory regions**
In the next step, we split the preprocessed events into the respective memory regions.
(3) **Extract initial RAM image**
Then, we extract an initial RAM image for every memory region. For each memory address, the value of the first read event indicates the initial value.
(4) **Identify register types**
In the following, we assign the appropriate class for each memory address of the memory region.
Read-Only: We consider a memory address read-only when the read value never changes throughout the trace for that region, regardless of whether there were write accesses with different values to the address.
Read-Writable: We consider a memory address read-writable when the read value always represents the last written value.
Sequencial: We consider a memory address sequential when we observe two successive read events that return different values or when we notice a read value that does not reflect the last written value to that address.
We consider memory addresses that don't appear in the trace data as read-writeable.

After we have specified the device properties and prepared the trace data, we can add the virtual device model via the QEMU command line as a replacement for the VFIO device. The device model will prototypically act as the physical device while it serves memory read and write operations and interrupts according to the traced data. Read-only memory locations can hold fixed data, such as device properties or binary data like the BIOS. More interestingly, they could implement some functionality that is triggered when they are written with a specific value but effectively, from a memory perspective, not store the value. Read-writeable memory locations are independent of the captured trace data and could diverge across executions when they are not related to the sequential registers.

Memory locations identified as sequential could implement counters or timers, provide status information, or deliver data streams similar to pipes.

A simple device model and replay will never accurately emulate the full functionality of a complex device like the graphics card. Different branches in the driver may lead to non-recorded behavior. Nevertheless, it is sufficient to drive deterministic parts properly, such as driver initialization and other functionalities, as long as the read access event order remains the same for the sequential addresses. Therefore, we must verify our findings whether they also occur with the physical hardware device.

5 EVALUATION

During this chapter, we evaluate our presented fuzzing approaches by creating different experiments. The results of the experiments respond to the following questions:

(1) **Performance**
What are the fuzzing throughput and reached coverage achieved of both approaches?
(2) **Scaling**
How well do the approaches scale to make efficient use of available system resources?
(3) **Findings**
Can real-world bugs be found with the approaches?
(4) **Emulated device model**
Does the achieved coverage differ when using the device model compared to the physical hardware device?

We performed the evaluation on a host system with an Intel Core i5-4570 and 8 GB RAM across the experiments. The QEMU guests are started with the options `-cpuhost,kvm=off,migratable=off` and `-snapshot`. The `host` option specifies to copy the host's CPU configuration, which is best practice and results in the highest performance [5].

Harnessing NVIDIA drivers for BSOD. The NVIDIA driver performs simple hypervisor detection and refuses to initialize for devices that are not certified to run virtualized. It is actually no technical restriction since the more expensive device counterparts are using the same chipsets. We hide the KVM hypervisor by using the `kvm=off` option to bypass the virtualization detection of the driver.

By default, QEMU ensures the migratability of guest VMs, whereby incompatible CPU features are turned off and lower the possible performance. It enables creating snapshots of the VMs that could be loaded for quick initialization and state restore when crashes occur. However, we decided to set the `migratable=off` flag to disable this feature in favor of higher throughput and take the costs of re-initialization by rebooting the guests. Furthermore, when fuzzing with real hardware via the VFIO device, migration is not supported anyway, which would prevent comparability during the evaluation.

The `-snapshot` option allows using the file system in a copy-on-write manner, which effectively discards any performed changes during the execution after stopping the guest. Another advantage of this option is the ability to boot multiple Virtual Machine (VM) instances using the same file system image [5].

For interface recovery, we utilized available information from the envytools [20] repository together with tracing and analyzing exemplary applications.

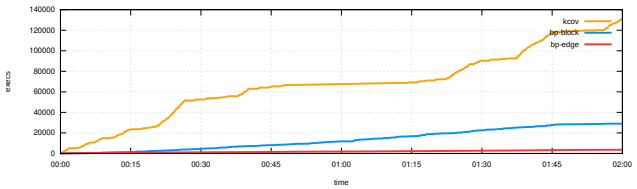

Figure 6: Syzkaller fuzzing executions for KCOV/bp-edge/bp-block coverage modes.

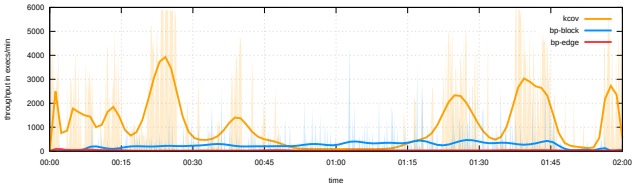

Figure 7: Syzkaller fuzzing throughput for KCOV/bp-edge/bp-block coverage modes.

Based on the information, we created a test-harness for AFL and use the collected ioctl data structures extracted from an exemplary execution replay as meaningful initial test cases.

For Syzkaller, we have to provide the target's system call interface descriptions by transforming the previously implemented structures for the harness into equivalent Syzkaller descriptions.

For both setups, we start the fuzzing guests with either the physical hardware device using PCI pass-through via VFIO, or with the derived device model, to bring the driver into a working state.

Table 1 shows our test matrix of targets per fuzzing setup and operating system. For Linux, the fuzzing has been performed inside guests, running an at the time of writing up-to-date installation of Arch Linux together with a custom kernel where we enabled Kernel Address Sanitizer (KASAN) to identify memory corruption bugs in kernel space. We installed the target NVIDIA kernel modules via the `nvidia-dkms` package, which builds the most recent driver version `460.56` against the running kernel. We configured the guest to print kernel logs to serial output and enabled Secure Shell (SSH) for interaction.

For preparing the target driver at system boot, we built a startup routine that creates the device files by using the `mknod` utility. Since the device initialization takes a significant amount of time when an application opens the GPU's device descriptor the first time, we open it in the startup routine and keep it open. New file descriptors can now instantly be created and used in conjunction with other system calls, which increases the throughput.

5.1 Performance

BSOD-Syzkaller. To determine the overhead of our used coverage method, we compare the throughput of Syzkaller when using KCOV and breakpoint coverage. Since KCOV only supports open-source targets, we chose to target the closely related nouveau driver for this experiment together with the physical hardware device. We further compare the breakpoint coverage throughput when using edge and block modes.

Setup / OS	Windows	Linux	FreeBSD
BSOD-AFL	nvlddmkm.sys	nvidia	nvidia
BSOD-Syzkaller	-	nvidia, nvidia-modeset, nvidia-drm, nvidia-uvm, nouveau	nvidia, nvidia-modeset

Table 1: Test matrix

	KCOV	bp-edge	bp-block
total progs	131091	3585	28994
mean progs/min	1072	30	246
coverage	11025	7381	7041

Table 2: Syzkaller fuzzing statistics by coverage mode.

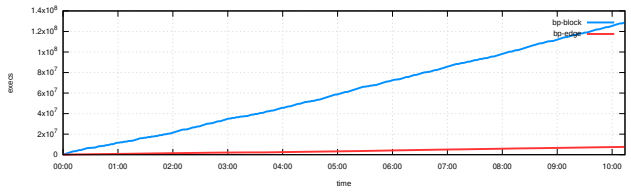

Figure 8: BSOD-AFL fuzzing executions for bp-edge/bp-block coverage modes.

Table 2 shows the statistics for the three tested modes in this experiment and Figure 6 depicts the number of program executions. According to the total executions, breakpoint coverage in edge mode causes an overhead of 36x compared to KCOV, whereby the block mode reduces the overhead to 4.5x. As expected, the block mode involves a slower start at the beginning compared to the edge mode due to the overhead of switching between triage phases but quickly improves over time.

Figure 7 shows the respective program execution speed for the three tested modes. High variance is noticeable for the KCOV curve that is related to system reboots due to encountered crashes. Because of the higher execution speed, the fuzzer triggered crashes more often, which resulted in a considerable penalty of multiple reboots for that instance.

The overhead of breakpoint coverage compared to statically instrumented binaries is quite large, but we expected this status as it also applies to user space fuzzers. We decreased the overhead by lowering the tracing accuracy to improve the throughput to an acceptable value.

BSOD-AFL. We performed another evaluation to measure the effectiveness of block coverage to reduce tracing overhead. This time, we execute the BSOD-AFL setup on the main kernel module with the emulated device model and compare the edge tracing and block tracing modes.

Table 3 shows the statistics of the experiment and Figure 8 shows the number of executions for both modes respectively. When comparing the total executions, we can observe the primary advantage of block coverage since it outperforms the edge coverage mode by a factor of 17.

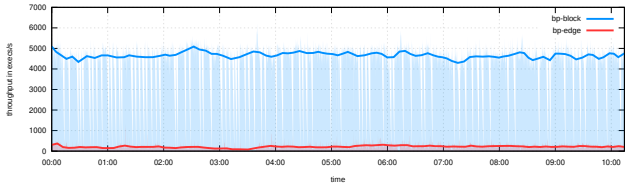

Figure 9: BSOD-AFL fuzzing throughput for bp-edge/bp-block coverage modes.

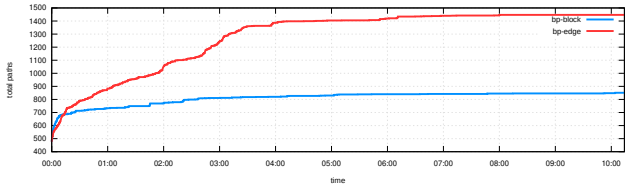

Figure 10: BSOD-AFL paths for bp-edge/bp-block coverage modes.

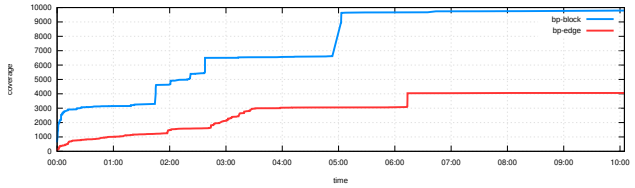

Figure 11: BSOD-AFL coverage for bp-edge/bp-block modes.

	bp-edge	bp-block
total execs	7.58M	129.29M
mean exec/s	213	4664
total paths	1447	852
coverage	4058	9800

Table 3: BSOD-AFL fuzzing statistics for bp-edge/bp-block coverage modes.

Figure 9 shows the execution speed for both modes respectively. The mean execution speed in block coverage mode is significantly higher compared to the edge coverage mode. Both modes show a high variance in the execution speeds, which is related to the code depth of the test cases generated by the fuzzer.

For block coverage mode, the tracing cost mainly reduces down to the overhead of providing new test cases between two executions since each reached code location is only reported once. We can

clearly see when the fuzzer has caused the guest to crash based on the times when the throughput temporarily drops to zero.

Figure 10 shows the total paths found for both modes respectively. We provided 467 input seeds of dumped ioctl structures, which defines the starting value in the origin. Both curves share an increased slope at the beginning that continuously decreases over time, which can be explained with an increased probability to find new paths when starting the experiment that decreases over time since there are fewer new paths to discover.

Generally, the number of found paths in block coverage mode is lower than in edge coverage mode, which does not necessarily mean that the block mode explores fewer paths. The reason is the lower tracing granularity since the tracer reports only previously unreached basic blocks and cannot distinguish between block transitions and identify the paths. The curve for the block coverage mode effectively shows the number of test cases that reached at least one new code location.

Figure 11 depicts the achieved coverage for block and edge modes. We can observe that both curves increase with jumps depending on whether a test case was executed that reaches a certain number of new code locations. If we compare total paths and coverage, we can observe that as coverage increases, the number of total paths increases, which is expected by definition.

It seems contradictory that the block mode explores about half the amount of paths but more than twice the number of coverage compared to the edge mode, due to its reduced tracing granularity. The greatly increased throughput of the block mode pays off as it achieves faster and more coverage compared to the edge mode.

Interestingly, the coverage in block mode suddenly jumps at about 2 hours and 5 hours of the experiment. At the same time, the total paths increase only minimally, which means that only a few test cases were necessary to reach many previously unseen code locations.

The overall achieved coverage is below the coverage reached when running our sample application, whose exchanged data we used as seeds for AFL. Since the minimal application already triggered above 400 highly dependent ioctl system calls, it becomes very unlikely for the fuzzer to trigger similar behavior in short time.

The performance gain from using the block mode in this approach is higher compared to Syzkaller that must use a mixture of block and edge modes due to the triage phase.

5.2 Scaling Using BSOD-fakedev

To run BSOD-fakedev, the tester needs to determine the PCI specifications needed: the target's hardware vendor and device IDs and the layout of the BARs. The test device used during this work is an NVIDIA GeForce GTX 760 with a GK104 chipset (Kepler) and 2 GB VRAM and contains three memory regions. To work around initial state problems with the replay device, we needed to implement custom logic for some memory addresses to avoid unexpected data for the driver.

To evaluate the functionality of BSOD-fakedev, the emulated hardware device, we compare the coverage in terms of reached basic block addresses and the throughput when fuzzing with the physical hardware device via VFIO and the emulated device model.

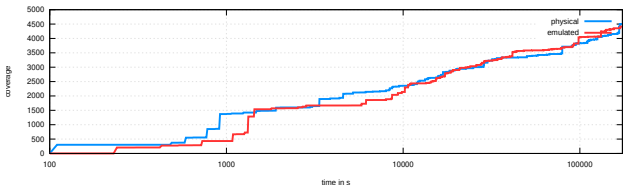

Figure 12: Syzkaller coverage for physical/emulated hardware device.

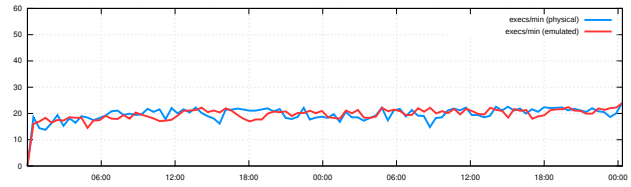

Figure 13: Syzkaller fuzzing throughput for physical/emulated hardware device.

	VFIO	Device Model
total progs	57450	57024
mean progs/min	17	17
overall coverage	4445	4411
exclusive coverage	451	417

Table 4: Syzkaller fuzzing statistics for physical/emulated hardware device.

For this purpose, we have chosen the Syzkaller setup that targets all the modules coming from the proprietary driver package.

Figure 12 shows the reached coverage for both devices in 48 hours. Both graphs progress nearly identical, starting with an increased slope in the first 30 minutes until it reaches a coverage value of about 1500. An increased slope at the beginning is typical since all defined system calls should trigger new behavior. Afterward, the slope continuously reduces for both graphs. Figure 13 shows the execution speed of Syzkaller programs with the physical device and the emulated model during the same experiment. In this case, also both graphs behave very similarly and reach a mean throughput of 17 programs per minute, which shows that the virtual device model involves no further overhead. Table 4 shows the final results of the experiment. The first three rows indicate that both experiments ran very similarly. The last row reveals the number of exclusively reached basic blocks for the experiment after executing the test cases in the other configuration, which indicates that some functionality deviates depending on the used device. With 3994 blocks in common, both setups reach about 90% of the same coverage in this experiment. These insights show that the emulated device model is suitable to replace a physical device for fuzzing purposes. In the following, we use BSOD-fakedev for fuzzing with Syzkaller and AFL.

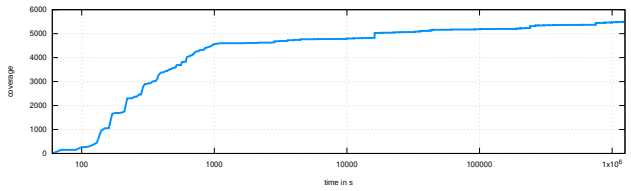

Figure 14: Syzkaller coverage for emulated device and bp-block in 15 days (log scale).

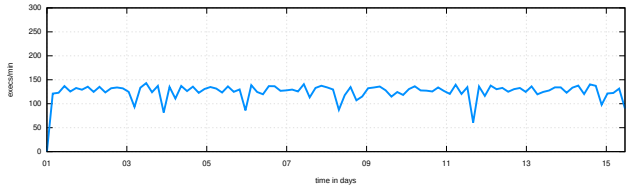

Figure 15: Syzkaller fuzzing throughput for emulated device and bp-block in 15 days.

BSOD-Syzkaller & fakedev. As the previous evaluations have shown, the emulated device model reaches similar behavior without overheads. It allows performing fuzzing on multiple VMs in parallel without the need for specific hardware devices. We utilized this feature and started a fuzzing campaign targeting the kernel driver modules using the Syzkaller approach over 15 days. We executed the fuzzing environment inside a guest on a server running an AMD EPYC 7281 16-Core Processor. Inside the guest, we executed Syzkaller with ten worker instances.

Figure 14 shows the reached coverage during the period of the experiment. The coverage value exceeded 5000 within the first day. From then on, it increased further only very slowly but continuously over time, even on the last day of the experiment up to a total coverage value of 5497.

Figure 15 depicts the execution speed of the programs. The mean fuzzing execution speed during the experiment is 125 progs/min. It shows a high variance, which has several reasons that depend on the generated programs and active worker guests. Syzkaller restarts the guests after running for one hour without a crash to initialize a clean system state that results in short downtimes. Additionally, when the fuzzer triggered previously unseen crash reports, Syzkaller tries to reproduce them on several worker guests from the pool. The points at which the execution speed drops noticeable down are explainable by a reduced number of active worker guests due to crash reproduction. Overall, while running ten instances simultaneously, we scaled up the throughput compared to a single instance, as shown in Figure 13, although we must consider a penalty due to nested-virtualization.

BSOD-AFL & fakedev. We applied BSOD-AFL on the main kernel module driver by setting up multiple AFL instances in parallel mode. As [13] have stated during their evaluation, the optimal parameters used for fuzzing depend heavily on the target. Therefore, using variations of the available parameters across fuzzing instances is beneficial. Variations in this context include selecting different power schedules, edge and block coverage modes, and usage of physical and emulated devices.

On the test system, we have four cores available and thus used four fuzzing instances. Since we have one physical GPU device present, we attached it to the master fuzzing instance and executed the remaining three instances with the emulated device model. We executed the master node with the `exploit` power schedule, and the secondary nodes with `mmopt`, cut-off-exponential `coe`, and quadratic `quad`.

The power schedule `coe` and `quad` depend on a parameter, which indicates the number of generated inputs that exercise the same path as the compared seed. Additionally, `coe` depends on a parameter, which is the average number of generated inputs that exercise a path [3]. We execute these two instances with edge coverage mode to provide meaningful values for these parameters. We chose to execute the other two instances with `exploit` and `mmopt` schedules in block coverage mode for high throughput.

The next chapter presents some findings that we encountered during the evaluation.

5.3 Findings

Linux. During the evaluation of BSOD-AFL with the emulated device model, we encountered multiple kernel crashes of type general protection fault in `__kmalloc`, as shown in Listing 4.

```
1  general protection fault, probably for non-canonical address 0
       x7baaeaf18dfcdf85: 0000 [#1] PREEMPT SMP KASAN NOPTI
2  RIP: 0010:__kmalloc+0x18b/0x420
```

Listing 4: Excerpt of crash report

The bug is reproducible on the host system with the physical hardware device present and causes the system to freeze and no longer respond. The stack traces vary across multiple testings, which indicates the root cause of the bug is triggered at an earlier point in time and causes faults for different tasks depending on the state of the operating system. Additionally, BSOD-AFL triggered multiple instances of page faults and bugs in `__vunmap` that cause recursive faults. During the evaluation of Syzkaller with the emulated device model, it was able to uncover multiple crashes due to NULL pointer dereference bugs.

```
1  BUG: kernel NULL pointer dereference, address: 0000000000000008
2  RIP: 0010:_nv018026rm+0x158/0x1520 [nvidia]
```

Listing 5: NULL pointer dereference

Excerpts of the kernel logs are shown in Listing 5 and Listing 6. Syzkaller successfully reproduced the crashes and generated minimized C programs that trigger the bugs. We tested the reproducers on the host system with the physical hardware device present and verified them. The NULL pointer dereference bug causes the operating system to crash and requires a reboot for recovery.

```
1  BUG: KASAN: use-after-free in nv_match_dev_state+0x124/0x130 [nvidia]
2  Read of size 8 at addr ffff88801e835380 by task crash/369
```

Listing 6: KASAN Use-After-Free

FreeBSD. We set up FreeBSD version 12.2-RELEASE and also installed the most recent NVIDIA driver version `460.56`. Since the driver package for FreeBSD contains the same binary blobs as the

	Windows	Linux	FreeBSD
BSOD-AFL	NULL pointer dereferences	General Protection Faults	-
BSOD-Syzkaller	-	Use-After-Free, NULL pointer dereferences	NULL pointer dereferences

Table 5: Found bug classes per fuzzer and operating system.

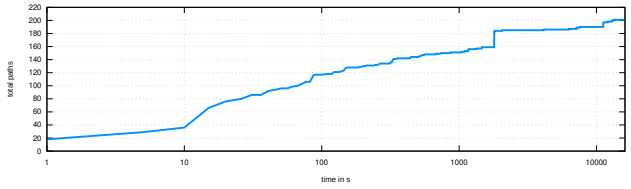

Figure 16: BSOD-AFL paths for fuzzing nvlddmkm.sys on Windows

Linux driver package, the preparation of the needed Syzkaller descriptions was convenient because we could reuse the descriptions we created for Linux. Syzkaller was able to trigger the identical NULL pointer dereference bug that we already found in the Linux kernel driver, as shown in Listing 5. We expected this case due to the identical binary blobs.

Windows. We have also applied the BSOD-AFL approach for fuzzing the most recent NVIDIA kernel driver version `461.72` for the Windows operating system, namely `nvlddmkm.sys`. The practicability without any required adaptions of the setup shows the independence of the target operating system. It does solely require the creation of a target-specific harness for the Windows operating system. The created harness is kept very simple and only targets the device node `\\.\NvAdminDevice` that is accessed via *DeviceIoControl*.

By using an exemplary application that accesses the device, we captured some typical data inputs. These data revealed at which memory offsets pointer values need to be replaced by the harness. Furthermore, we used these data again as input seeds for AFL.

We performed the fuzzing with the physical hardware device in one single guest with all four available cores so that AFL was able to execute test cases with a mean execution speed of 10.7k execs/s. BSOD-AFL uncovered a bug after about 45 minutes of the experiment, which caused the system to crash by triggering a Blue Screen of Death (BSoD). We investigated the input data and classified the fault as a NULL pointer dereference bug that a user-mode application can trigger via a single ioctl call. Then, we manually reduced the input to the necessary bytes to create a reproducer.

Interestingly, we triggered the bug initially in driver version `441.12` that was released over a year ago. After updating to version `456.71`, the found input no longer triggered the crash. When we started fuzzing again and added the crashing input to the initial seeds, it took only seconds to trigger again. The new crashing input only differs from the previous one by a single increased byte value and is still present in the most recent driver version.

To avoid triggering the same bug multiple times during fuzzing with the cost of rebooting the system each time, we blocklisted the respective combination of bytes that we already identified for the reproducer and continued fuzzing. Figure 16 shows the explored paths over time of this experiment.

The NULL pointer deference bugs are not exploitable to gain higher privileges but can be abused to cause a Denial of Service (DoS). The presented findings proved that both approaches find real existing bugs in closed-source kernel drivers to answer the initial question.

6 CONCLUSION

BSODs instrumentation allows us to fuzz binary-only drivers with real devices in virtual machines. We reach decent execution speed and coverage without the need for certain hardware features, such as Intel PT. Fuzzing the complex drivers of graphics cards yielded a range of bugs on all tested operating systems. On top of fuzzing with real attached devices, we were able to record and replay graphics card traces using BSOD-fakedev. The virtual device emulates parts of the device's behavior by replaying traced MMIO interactions. It enabled us to scale to additional fuzzing instances in the cloud without the need for additional physical devices. The results of our experiments are convincing, yielding good fuzzing speeds and driver exploration with Syzkaller and AFL++ respectively. As depicted in Table 5, we found multiple bugs in Windows, FreeBSD, and Linux drivers.

AVAILABILITY

All relevant source code for BSOD is avaliable open-source at https://github.com/0xf4b1/bsod-kernel-fuzzing.

DISCLOSURE PROCESS

We reported all found bugs to the NVIDIA Product Security Incident Response Team. The coordinated disclosure process is ongoing.

ACKNOWLEDGMENTS

The authors would like to thank Jiska Classen for valuable feedback.

REFERENCES

[1] National Security Agency. 2019. *GHIDRA.* https://ghidra-sre.org
[2] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.
[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428
[4] Oliver Chang. 2017. *Attacking the Windows NVIDIA Driver.* https://googleprojectzero.blogspot.com/2017/02/attacking-windows-nvidia-driver.html
[5] H.D. Chirammal, P. Mukhedkar, and A. Vettathu. 2016. *Mastering KVM Virtualization.* Packt Publishing. https://books.google.de/books?id=fAjVDQAAQBAJ
[6] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer.

2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1201–1218. https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[7] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3133956.3134069

[8] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. 1497–1511. https://doi.org/10.1109/SP40000.2020.00009

[9] Brendan Dolan-Gavitt, Josh Hodosh, P. Hulin, T. Leek, and R. Whelan. 2014. Repeatable Reverse Engineering for the Greater Good with PANDA.

[10] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop* (Los Angeles, CA, USA) *(PPREW-5)*. Association for Computing Machinery, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/2843859.2843867

[11] David Drysdale. 2016. *Coverage-guided kernel fuzzing with syzkaller.* https://lwn.net/Articles/677764/

[12] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1237–1254. https://www.usenix.org/conference/usenixsecurity20/presentation/feng

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[14] Google. [n.d.]. *honggfuzz.* https://honggfuzz.dev

[15] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 135–150. https://www.usenix.org/conference/raid2019/presentation/gustafson

[16] Sami Tolvanen Jeff Vander Stoep. 2018. *Android Kernel Security.* https://events19.linuxfoundation.org/wp-content/uploads/2017/11/LSS2018.pdf

[17] Tim Newsham Jesse Hertz. 2016. *Project Triforce.* https://raw.githubusercontent.com/nccgroup/TriforceAFL/master/slides/ToorCon16_TriforceAFL.pdf

[18] Richard Johnson. 2017. *Evolutionary Kernel Fuzzing.* https://www.fuzzing.io/Presentations/Evolutionary%20Kernel%20Fuzzing-BH2017-rjohnson-FINAL.pdf

[19] Michael Kerrisk. 2013. *LCA: The Trinity fuzz tester.* https://lwn.net/Articles/536173/

[20] Marcelina Kościelnicka. 2013. *envytools.* https://envytools.readthedocs.io/en/latest/index.html

[21] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. 2010. Testing Closed-Source Binary Device Drivers with DDT. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 12.

[22] Tamas K Lengyel. 2015. *LibVMI: Simplified Virtual Machine Introspection.* https://libvmi.com

[23] Tamas K Lengyel. 2020. *VM Forking and Hypervisor-based Fuzzing.* https://static.sched.com/hosted_files/xen2020/bc/XPDS2020%20-%20VM%20forking%20and%20Hypervisor%20Based%20Fuzzing.pptx

[24] Dominik Maier, Benedikt Radtke, and Bastian Harren. 2019. Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/woot19/presentation/maier

[25] MITRE. 2017. *Common Vulnerabilities and Exposures.* https://cve.mitre.org

[26] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar 2 : A Multi-Target Orchestration Platform. https://doi.org/10.14722/bar.2018.23017

[27] S. Nagy and M. Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 787–802. https://doi.org/10.1109/SP.2019.00069

[28] NIST. 2017. *NVD National Vulnerability Database.* https://nvd.nist.gov

[29] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2559–2575. https://www.usenix.org/conference/usenixsecurity20/presentation/peng

[30] I. Pustogarov, Q. Wu, and D. Lie. 2020. Ex-vivo dynamic analysis framework for Android device drivers. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1088–1105. https://doi.org/10.1109/SP40000.2020.00094

[31] Nguyen Anh Quynh. 2014. *Capstone.* http://www.capstone-engine.org

[32] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. 2012. SymDrive: Testing Drivers without Devices. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 279–292. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/renzelmann

[33] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 167–182.

[34] Kostya Serebryany. [n.d.]. *libFuzzer – a library for coverage-guided fuzz testing.* https://llvm.org/docs/LibFuzzer.html

[35] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/periscope-an-effective-probing-and-fuzzing-framework-for-the-hardware-os-boundary/

[36] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2541–2557. https://www.usenix.org/conference/usenixsecurity20/presentation/song

[37] Mathieu Tarral. 2019. *KVM-based Virtual Machine Instrospection.* https://github.com/KVM-VMI

[38] Ryan Whelan, Tim Leek, and David Kaeli. 2013. Architecture-Independent Dynamic Information Flow Tracking. In *Compiler Construction*, Ranjit Jhala and Koen De Bosschere (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 144–163.

[39] Michał Zalewski. 2013. *american fuzzy lop.* https://lcamtuf.coredump.cx/afl/

[40] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min. 2018. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access* 6 (2018), 37302–37313. https://doi.org/10.1109/ACCESS.2018.2851237

[41] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng