Moneta: Ex-Vivo GPU Driver Fuzzing by Recalling In-Vivo Execution States

Joonkyo Jung[*]
Department of Computer Science
Yonsei University

Jisoo Jang[*]
Department of Computer Science
Yonsei University

Yongwan Jo
Department of Computer Science
Yonsei University

Jonas Vinck
DistriNet
KU Leuven

Alexios Voulimeneas
CYS
TU Delft

Stijn Volckaert
DistriNet
KU Leuven

Dokyung Song[†]
Department of Computer Science
Yonsei University

***Abstract*—Graphics Processing Units (GPUs) have become an indispensable part of modern computing infrastructure. They can execute massively parallel tasks on large data sets and have rich user space-accessible APIs for 3D rendering and general-purpose parallel programming. Unfortunately, the GPU drivers that bridge the gap between these APIs and the underlying hardware have grown increasingly large and complex over the years. Many GPU drivers now expose broad attack surfaces and pose serious security risks.**

Fuzzing is a proven automated testing method that mitigates these risks by identifying potential vulnerabilities. However, when applied to GPU drivers, existing fuzzers incur high costs and scale poorly because they rely on physical GPUs. Furthermore, they achieve limited effectiveness because they often fail to meet dependency and timing constraints while generating and executing input events.

We present Moneta, a new ex-vivo approach to driver fuzzing that can statefully and effectively fuzz GPU drivers at scale. The key idea is (i) to recall past, in-vivo GPU driver execution states by synergistically combining snapshot-and-rehost and record-and-replay along with our proposed suite of GPU stack virtualization and introspection techniques, and (ii) to start parallel and stateful ex-vivo GPU driver fuzzing from the recalled states. We implemented a prototype of Moneta and evaluated it on three mainstream GPU drivers. Our prototype triggered deep, live GPU driver states during fuzzing, and found five previously unknown bugs in the NVIDIA GPU driver, three in the AMD Radeon GPU driver, and two in the ARM Mali GPU driver. These ten bugs were all confirmed by the respective vendors in response to our responsible disclosure, and five new CVEs were assigned.

I. Introduction

Graphics Processing Units (GPUs) are found ubiquitously across a broad spectrum of computing platforms, including mobile devices, desktops, and workstations. They serve an exploding demand for accelerated data-parallel computing infrastructure in areas such as rendering and machine learning. Unfortunately, the software stacks that power GPUs are immensely complex and bug-ridden. The device drivers at the lowest layer of the GPU software stacks are particularly concerning because they operate with kernel privileges, have large and often proprietary code bases, and expose attack surfaces that attackers can easily access through the system call interface, e.g., through web browsers and machine learning runtimes that use GPUs for accelerated rendering and computation on tensors, respectively.

Fuzzing is a proven method to identify vulnerabilities in device drivers [59], [57], [46], [39], [69], [32], and can, thus, mitigate the security risks GPU drivers incur. Prior work on device driver fuzzing has primarily tackled the following two well-known challenges: resolving dependencies between system calls (referred to as **P1**), and providing high-fidelity device-side inputs (**P2**). Addressing the first challenge, also known as the dependency challenge [26], requires fuzzers to generate sequences system calls that satisfy execution order (e.g., `read(fd)` should come after `open()`) and payload value constraints (e.g., `read(fd)` should use as an argument the return value of `open()`). This challenge is particularly pronounced in device driver fuzzing, due to (i) *proprietary* input formats (e.g., the arguments of `ioctl` system calls), and (ii) *long* chains of execution order and payload value dependencies between system calls. To address this, researchers have proposed a variety of static [17], [25] and dynamic [32] analysis solutions.

The second challenge involves providing high-fidelity device-side inputs to drivers during fuzzing. A straightforward way to address this is to use a physical instance of the device during fuzzing [62], [59], [57]. The physical device requirement imposed during fuzzing, however, severely restricts the usability and scalability of device driver fuzzers, because these techniques can only be used when at least one hardware device is accessible, and can scale only to the number of available hardware devices. To mitigate this problem, researchers proposed *ex-vivo* driver fuzzing approaches. In contrast to *in-vivo* approaches that require hardware *while* fuzzing drivers [59], [57], ex-vivo approaches either (i) do not require hardware at all via hardware access evasion [48] or static analysis [17], [56], [69], or (ii) use device hardware only while recording normal device driver executions which are reproduced later

[*]Equal contribution.
[†]Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230218
www.ndss-symposium.org

without hardware via replay [39], [32].

The latter line of work, which employs record-and-replay for a more effective ex-vivo driver fuzzing, is particularly promising: Record-and-replay can mitigate the dependency challenge (**P1**) to a large extent because recordings consist of a sequence of input events that already satisfy ordering and value dependencies, while also addressing the device-side input requirement problem (**P2**). However, achieving high-fidelity record-and-replay is challenging (referred to as **P3**) because of the well-known nondeterminism problems [32]. Any factors outside the scope of record-and-replay (e.g., precise timing of interrupts and input events) can easily cause discrepancies between recorded and replayed executions. Moreover, even though record-and-replay accurately captures inputs that satisfy all aforementioned constraints on the device side, it may not serve new, previously unseen I/O requests generated during fuzzing. Though it is theoretically possible to emulate all possible hardware behaviors in software, this entails substantial engineering efforts. Prior work only partially addresses this problem by deriving rules from recorded device I/O behaviors and performing rule-based I/O emulation [39].

This paper proposes Moneta, a new ex-vivo approach to GPU driver fuzzing that statefully fuzzes GPU drivers by accurately and quickly recalling past in-vivo driver execution states at scale. Our key insight is that we can overcome the fidelity challenges of record-and-replay (**P3**) by combining it with a deterministic past execution state recall technique: *snapshot-and-rehost*. Snapshot-and-rehost works by taking a live snapshot of the GPU driver normally operating with a real GPU and a GPU workload, and rehosting the snapshot in an ex-vivo, off-GPU environment to reproduce the state captured in the snapshot. Since the GPU driver states recalled from snapshots precisely mirror the outcomes of past in-vivo driver executions with real GPU applications and GPU hardware, they significantly alleviate the challenges of accurate record-and-replay and I/O emulation during ex-vivo fuzzing.

At a high level, Moneta runs GPU workloads such as rendering jobs, during which Moneta (i) takes a snapshot of the driver state, and (ii) records the input events (e.g., `ioctl` calls) delivered to the GPU driver immediately after taking the snapshot. Moneta starts fuzzing from past driver execution states it accurately recalls from these snapshots and post-snapshot recordings. These recalled execution states enable *stateful* and, therefore, more effective fuzzing of GPU drivers, effectively circumventing the dependency challenge of kernel driver fuzzing (**P1**) and the fidelity challenges of record-and-replay (**P3**). Moreover, Moneta, being an ex-vivo driver fuzzing approach, significantly mitigates the hardware requirement (**P2**), which facilitates GPU driver fuzzing at scale. While there exists prior work that addresses a *subset* of these problems, Moneta is the first work that addresses all of them simultaneously. In particular, Moneta diverges (i) from existing snapshot-based fuzzing approaches [32] in that Moneta creates snapshots in an *in-vivo* environment and restores them in an *ex-vivo* fuzzing environment, and (ii) from existing ex-vivo driver fuzzing approaches [48], [39] in that Moneta mitigates the need for fully accurate I/O emulation by restoring in-vivo driver snapshots.

The concrete design goals of Moneta are threefold: (i) ensuring applicability across a wide range of GPU drivers, including both desktop and mobile GPU drivers, (ii) supporting large-scale GPU driver fuzzing in a typical x86 server environment, and (iii) keeping driver execution states alive in an ex-vivo environment for stateful replay and fuzzing. To this end, Moneta first runs GPU drivers within a virtual machine with direct passthrough access to a GPU. We construct this virtual machine using our proposed *emulator-rehostable virtualization* principle that applies to both x86- and ARM-based commodity machines equipped with various GPUs. Moneta takes snapshots of the entire virtual machine to capture GPU driver execution states, which we then rehost in a virtual machine (possibly with an emulated CPU) that we can easily duplicate on typical x86 servers for large-scale GPU driver fuzzing. During rehosting, Moneta fully preserves the live driver state kept in the snapshot (i.e., the live contexts that the driver established with real GPU and GPU workloads). Finally, Moneta starts replaying and fuzzing from such a deep, live state of the driver recalled from its past executions to more effectively find bugs than existing fuzzers.

Our evaluation shows that Moneta can fuzz a wide range of GPU drivers, including NVIDIA, AMD Radeon, and even ARM Mali GPU driver. We conducted 128-instance parallel fuzzing of each driver on a 128-thread x86 server. The snapshots and post-snapshot recordings were effective at making fuzzing stateful. Using Moneta, we found 10 previously unknown bugs: five in NVIDIA GPU drivers, three in the AMD Radeon drivers, and two in the ARM Mali kernel-mode GPU drivers. Of these, there were four bugs we only found thanks to Moneta's capability of accurately recalling past in-vivo execution states for stateful fuzzing. Moneta increased the coverage of GPU drivers at the basic block level (by up to 137.8% on average when fuzzing the NVIDIA GPU driver) over baselines that are stronger than state-of-the-art.

To summarize, we make the following contributions:

- **Ex-Vivo Approach to GPU Driver Fuzzing:** Our ex-vivo approach dubbed Moneta synergistically combines snapshot-and-rehost and record-and-replay, which enables (i) stateful GPU driver fuzzing by using snapshots and recordings, and (ii) large-scale fuzzing by duplicating snapshots and recordings.
- **GPU Stack Virtualization & Introspection Techniques:** We present (i) GPU stack virtualization techniques (i.e., emulator-rehostable processor virtualization and passthrough I/O virtualization of GPUs with built-in IOMMUs) tailored to large-scale fuzzing of all the mainstream GPU drivers, and (ii) `ptrace`- and hypervisor-based GPU stack introspection techniques that enable stateful GPU driver fuzzing.
- **Real-World Impact:** We open-sourced our prototype implementation of Moneta[1], using which we found 10 bugs in 3 GPU drivers and significantly increased the coverage of the GPU software stack in the Linux kernel. The respective vendors confirmed all bugs after our responsible disclosure and have assigned 5 CVEs.

[1]Available at: https://github.com/yonsei-sslab/moneta

II. Background & Motivation

A. GPU Stack Overview

The software stack that supports modern GPU graphics and computing workloads contains several components a user-space application could, theoretically, interact with directly:

1) A **Graphics or Computing Library** that exposes an API such as OpenGL [54], Vulkan [61], or OpenCL [34]. This library serves as an intermediary between the application and the GPU driver by translating high-level API calls into lower-level calls the user-mode GPU driver can process.
2) The **User-Mode GPU Driver** translates the aforementioned calls into hardware-specific data and instructions, which it then submits to the kernel-mode GPU driver using system calls.
3) The **Kernel-Mode GPU Driver** manages the allocation and utilization of GPU resources such as buffer objects, texture units, and shader/compute units. It also coordinates safe access to the GPU, forwards the GPU data and instructions it receives through its user space-facing API to the GPU hardware, and, optionally, returns data it receives from the hardware.

As a performance optimization, most kernel-mode GPU drivers allow GPU-resident buffer objects to be mapped directly into the user mode-accessible portion of the application's address space using Memory-Mapped I/O (MMIO). This mechanism can dramatically improve the performance of certain rendering and computing workloads since it allows the user-mode stack to submit certain kinds of data directly to the GPU, completely bypassing the kernel-mode GPU driver. The graphics or computing API can expose this functionality directly to the application. OpenGL, for example, allows applications to temporarily or persistently map data buffers stored in GPU memory into their address space. However, the user-mode GPU stack can also use MMIO buffers internally, even without the application's explicit request.

Security Implications. GPU drivers have an exceptionally broad attack surface thanks to their layered architecture and memory-mapped GPU buffer optimizations. Whereas some peripheral drivers need only to worry about attacks via their system call interface, adversaries can target the GPU stack by (i) calling regular functions in the graphics/computing library or the user-mode GPU driver, (ii) corrupting internal state of the graphics/computing library or user-mode GPU driver, (iii) calling functions in the kernel-mode GPU driver using system calls, (iv) corrupting GPU-resident MMIO buffers, or (v) any combination thereof.

Not every attack on the GPU stack is equally dangerous, however. For example, attacks that only compromise the graphics/computing library or user-mode GPU driver [47] have a limited impact on the rest of the system. Attacks that compromise the kernel-mode GPU driver [13] are much more dangerous, as they give adversaries full control over the kernel and potentially the entire system. Somewhere in the middle are attacks that compromise the GPU, which only break isolation of GPU contexts. Adversaries could use the compromised GPU as a stepping stone to kernel compromise [27]. However, these types of attacks can be mitigated using IOMMU-based GPU isolation [42]. For this reason, we focus only on finding vulnerabilities that (i) are in the kernel-mode GPU driver, (ii) could be exploited from a user-space application, and (iii) do not require compromised GPU firmware (see §III).

Table I: Comparison of two methods that can recall past execution states of a program.

Execution State Recall ...	**Snapshot-and-Rehost**	**Record-and-Replay**
Fidelity	**High, deterministic**	Low, nondeterministic
Speed	**Fast**	Slow
Granularity	Coarse-grained	**Fine-grained**
Traceability	Low	**High**

B. Device Driver Fuzzing

In-Vivo Driver Fuzzing. A key challenge to device driver fuzzing is generating sequences of input payloads (e.g., system calls and I/O messages) that satisfy certain value and ordering constraints. For example, GPU driver-specific `ioctl` commands should be executed on the file descriptor returned by the `open` call on the GPU's device file. To address this so-called dependency challenge [26] (referred to as **P1**), one line of prior work uses *in-vivo* fuzzing [59], [57], [62], which fuzzes the target driver *while* it operates in a normal usage scenario, driven by the system calls and I/O messages generated by a real application and a real device. Charm [59] and PeriScope [57], for example, take an in-vivo approach to fuzz the system call and peripheral interface of device drivers, respectively. This approach effectively circumvents the dependency challenge for driver fuzzers, because the real input generators can easily lead the driver into various states. This advantage comes at the cost of usability and scalability, however: fuzzing can be conducted only when one or more hardware devices are available, and parallel fuzzing scales only to the number of available devices.

Ex-Vivo Driver Fuzzing. To mitigate the hardware requirement imposed by in-vivo approaches (referred to as **P2**), researchers have proposed *ex-vivo* approaches to device driver fuzzing [39], [32], [48]. Ex-vivo fuzzers operate in a controlled, device-free environment. One line of work replaces the device driver with a mock object [48], while another reproduces hardware behavior by recalling past observations of normal driver executions with real hardware [39], [32].

The observations could be generated in two distinct forms: (i) *recordings* of all the external inputs delivered to the driver during its normal execution [39], [32], and (ii) *snapshots* capturing complete driver execution states at specific points in time. One can then recall these observations either by replaying the recordings [39], [32], or by rehosting the captured snapshots (used by prior work primarily for embedded firmware analysis [50], [29]). Ex-vivo fuzzing facilitates large-scale parallel fuzzing of drivers, because, unlike physical instances of hardware, observations can easily be duplicated.

C. Recalling Past Driver Executions for Fuzzing

Ex-vivo fuzzers can recall past driver executions either via record-and-replay (RnR) or snapshot-and-rehost (SnR). We summarize their general strengths and weaknesses in Table I and elaborate on their use in fuzzers below.

Why (Not) Record-and-Replay? RnR typically works by (i) recording all external input events consumed by the driver (i.e., system calls and I/O messages delivered to the driver), and (ii) replaying these recorded events to reproduce the driver state observed while recording [39], [32]. RnR can, in theory, reproduce *all* the driver execution states observed during a single observation (i.e., recording) session by replaying a subsequence of the recorded input events; this means that fuzzing can start from a diverse set of driver execution states, and, therefore, it can trigger diverse driver code paths and bugs.

Precise *and* efficient record-and-replay, however, is challenging (referred to as **P3**). Although deterministic replay can accurately recall past executions [20], it requires recording and replaying *all* external and nondeterministic events such as peripheral device interrupts and time elapses, which, unfortunately, could significantly degrade fuzzing speed. Though it incurs less overhead than deterministic replay, nondeterministic replay cannot accurately recall past driver execution states because of the highly concurrent, timing-sensitive nature of device drivers [32].

Despite these unaddressed challenges, the traceability of RnR, i.e., the ability to trace back how a given execution state was derived, significantly enhances fuzzing, because recordings can be used as initial input corpora for evolutionary fuzzing [39], [32]. As long as the external input interface remains backward-compatible, the traceability also enables reusing recordings to fuzz differently configured targets (e.g., the driver configured with a kernel of a different version).

Why (Not) Snapshot-and-Rehost? The SnR approach works by (i) snapshotting the complete state of the driver after its normal operation with a physical device (e.g., GPUs), and (ii) rehosting the snapshot in an *off-device* environment. SnR, which became popular recently in the context of embedded firmware analysis [50], [29], can deterministically and quickly recall the captured driver state via snapshot restoration, and it can explore deep driver code paths by starting fuzzing from the recalled state.

There are limitations, however. First, SnR can recall only a *subset* of driver execution states, i.e., the driver's states captured at coarse-grained snapshot intervals. This means that, by solely using snapshots, one can only fuzz the driver from this subset of states, and not any other states between them. Second, SnR does not remember the lineage of (i.e., the sequence of input events that led to) the captured driver execution state, meaning that it alone cannot provide any input corpora that can facilitate evolutionary fuzzing. Third, since each snapshot captures the exact state of a specific version of the target software (e.g., the driver and kernel), it cannot be reused to reproduce the same state of the target configured differently.

III. Overview

We now introduce Moneta, a new ex-vivo approach to GPU driver fuzzing, which synergistically combines two methods for recalling past execution states at scale—snapshot-and-rehost (SnR) and record-and-replay (RnR)—and uses the recalled driver states for large-scale, stateful GPU driver fuzzing.

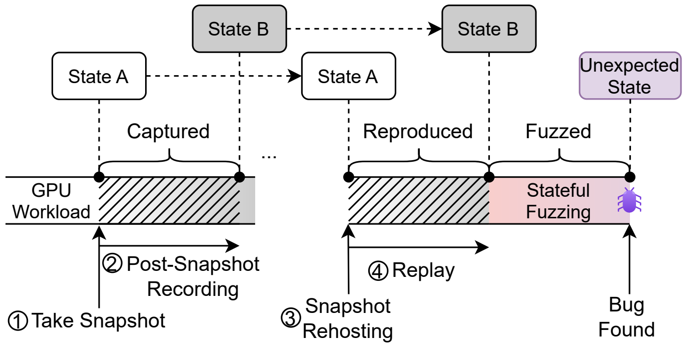


Fig. 1: GPU driver execution states—either captured, recalled, or fuzzed—under (a) a real GPU workload in our in-vivo observation environment, and (b) stateful fuzzing in our ex-vivo fuzzing environment.

We illustrate our approach in Figure 1: Moneta first captures various in-vivo GPU driver execution states while a GPU workload is executing on a real GPU, ① by discretely *snapshotting* driver states, and ② by continuously *recording* the external inputs subsequently delivered to the driver, thereby creating *snapshots* and *post-snapshot recordings*, respectively. Then, Moneta recalls these captured driver states in an off-GPU, ex-vivo environment, ③ by *rehosting* the snapshot, and then ④ by *replaying* its corresponding post-snapshot recording. During replay, Moneta fuzzes GPU drivers *statefully* from their past execution states as it recalls them at a fine-grained granularity.

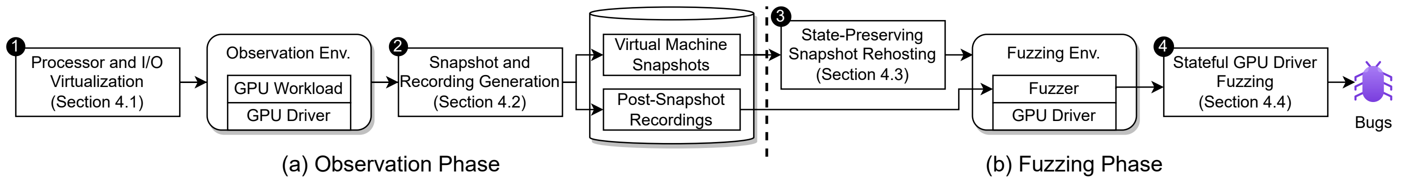
The concrete design goals of Moneta are threefold:

G1. Wide Applicability: Moneta should be a *generic* ex-vivo GPU driver fuzzer, which, unlike prior work [39], applies to all mainstream GPU drivers, including those for discrete and integrated GPUs.

G2. x86-Host Rehostability: Moneta should create an ex-vivo fuzzing environment that it can host on a typical multi-core x86 host to aid large-scale fuzzing.

G3. Live-State Reproducibility: Moneta should reproduce *live* states of the driver in an ex-vivo environment so that fuzzing can exercise stateful driver code paths.

Figure 2 depicts an overview of Moneta's four key components designed to achieve these goals. In the observation phase, ❶ Moneta first creates a virtual machine serving as the observation environment. The entire GPU software stack, including the target GPU driver, runs in this environment. Moneta uses *emulator-rehostable virtualization* to create the virtual machine, and gives the virtual machine passthrough access to the physical GPU whose driver we want to fuzz (see §IV-A). ❷ Moneta then captures various states of the GPU driver while executing a GPU workload on the physical GPU. Moneta uses both virtual machine- and process-level introspection mechanisms to automatically generate both virtual machine snapshots and their corresponding post-snapshot recordings (§IV-B). In the fuzzing phase, ❸ Moneta rehosts snapshots and brings the captured driver states back to life in a number of virtual machines on x86 hosts. This allows us to conduct fuzzing at scale and *without* the physical GPU. During this process, Moneta carefully transitions the input space



1
Processor and I/O Virtualization (Section 4.1)
Observation Env.
GPU Workload
GPU Driver
2
Snapshot and Recording Generation (Section 4.2)
Virtual Machine Snapshots
Post-Snapshot Recordings
3
State-Preserving Snapshot Rehosting (Section 4.3)
Fuzzing Env.
Fuzzer
GPU Driver
4
Stateful GPU Driver Fuzzing (Section 4.4)
Bugs
(a) Observation Phase
(b) Fuzzing Phase

Fig. 2: Interplay between Moneta's four key components.

Table II: Moneta's observation and fuzzing environment setup used when fuzzing a variety of GPU drivers. All observation environments use hardware-accelerated CPU virtualization. Fuzzing environments are all hosted on x86 servers without GPUs. This facilitates large-scale GPU driver fuzzing.

Target GPU Driver	**In-Vivo Driver Observation Environment**			**Ex-Vivo Driver Fuzzing Environment**		
	CPU	vCPU	GPU	CPU	vCPU	GPU
NVIDIA & AMD Radeon	x86	HW-accel. x86	Present	x86	HW-accel. x86	**Not Present**
ARM Mali	ARMv8	HW-accel. ARMv8	Present	**x86**	**Emulated ARMv8**	**Not Present**

of GPU drivers into a fuzzer-reachable one by substituting the driver's original input source with Moneta's own input providers, all while keeping the driver alive (§IV-C). ❹ Moneta finally performs *stateful* fuzzing of the GPU driver in these virtual machines by using the driver state restored from the snapshot as a starting point for fuzzing, and its post-snapshot recordings as initial corpora for evolutionary fuzzing (§IV-D).

Threat Model. Our goal is to use Moneta to find vulnerabilities in *kernel-mode GPU drivers* that adversaries could feasibly trigger through their user-space-accessible interfaces. This is in line with prior work on finding vulnerabilities in GPU drivers [13], [39], and in other kernel-mode drivers [17], [25], [74]. The attacker could be either a local attacker who (i) has access to the device file interface exposed by the GPU driver modules, and (ii) wants to escalate their privileges to *root*, or a remote attacker who has remotely compromised a process that has an already established communication channel with the GPU driver, e.g., for accelerated rendering, video encoding/decoding, or machine learning.

IV. Design

A. Processor and I/O Virtualization

The goals of Moneta's observation phase are to generate snapshots of driver states and to record external inputs while executing realistic workloads on the GPU whose driver we want to fuzz. We achieve the former by running the driver inside a virtual machine and capturing driver states in the form of full virtual machine snapshots. Unlike previous driver fuzzers that instantiate driver-running virtual machines in an ex-vivo environment that lacks corresponding physical devices [59], [48], we instantiate this virtual machine initially on a host equipped with a physical GPU. By giving the virtual machine passthrough access to the physical GPU, Moneta can run native GPU drivers (i.e., the fuzzing target) and GPU workloads inside the virtual machine. To run the driver and the workload

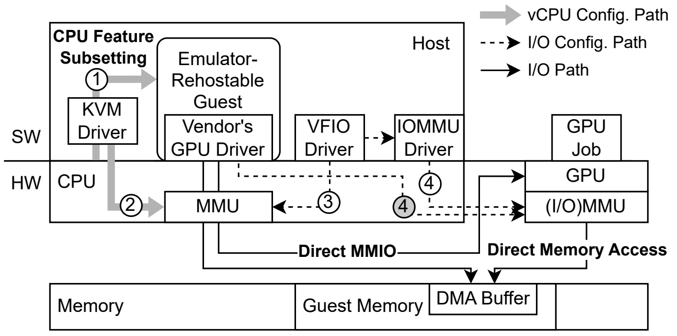


Fig. 3: Moneta's CPU, memory, and GPU (I/O) virtualization used in the observation phase.

at native speed, we use the *hardware-accelerated* CPU virtualization features available on most GPU-equipped hardware platforms. In the subsequent fuzzing phase, we rehost virtual machine snapshots generated from a GPU-equipped machine onto a GPU-free and potentially emulated-CPU environment (see Table II), which is more suitable for large-scale fuzzing.

Emulator-Rehostable Processor Virtualization. To facilitate large-scale fuzzing of a wide range of GPU drivers, including those for discrete *and* integrated GPUs, we propose a technique called *emulator-rehostable virtualization*. The idea is to create a virtualized environment on a GPU-equipped machine such that our generated virtual machine snapshots can be rehosted in an *emulator* with full system virtualization capabilities. Being able to rehost driver snapshots in an emulator allows us to fuzz the driver on machines that are far more powerful than the machines used in the observation phase. For example, we can take snapshots of the ARM Mali GPU driver running on an inexpensive ARM SoC equipped with an *SoC-integrated* ARM Mali GPU. Then, as shown in Table II, we can rehost the generated snapshots on a high-end x86 host with dozens of CPU cores to run several high-throughput fuzzer instances in parallel.

In theory, it should be possible to rehost snapshots of *fully virtualized* guests into virtualization environments with varying degrees of emulation support and capabilities, including CPU emulators that implement the CPU entirely in software. This is because an *ideal* full virtualization would provide guests with a complete abstraction of the *physical* hardware, meaning that guests are unaware of whether they are using emulated, virtualized, or physical hardware.

In practice, however, most CPU emulators do not provide such ideal full virtualization as they lack support for many CPU features available in physical production CPUs. ARM's

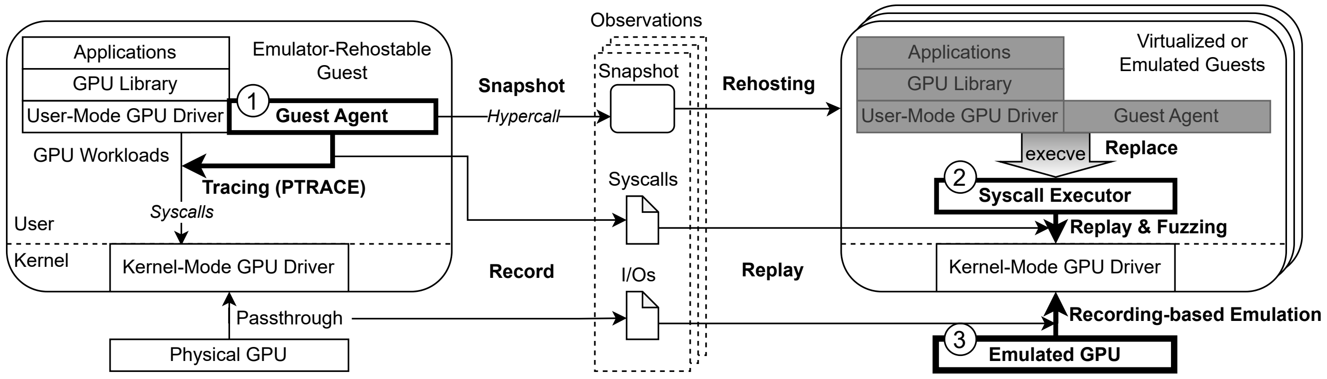


Fig. 4: Moneta's snapshot and recording generation (left), and state-preserving snapshot rehosting followed by stateful replay and fuzzing (right).

Fixed Virtual Platforms (FVPs) [6] are one exception since they accurately simulate most of the ARM CPU features. Unfortunately, FVPs are known to be substantially slower than other CPU emulators and, therefore, do not meet our design goals. To overcome the lack of CPU feature support in the emulators we could feasibly use in our fuzzing environment, we propose and implement *CPU feature subsetting*. The idea is to disable or restrict access to CPU features in our observation environment if said features are not available in the fuzzing environment (see ① in Figure 3). As we will show in §V, CPU feature subsetting allows us to rehost virtual machine snapshots, even those of KVM-accelerated ARM guests [18], onto purely-emulated ARM CPUs running on x86 hosts.

Passthrough GPU I/O Virtualization. To run target GPU drivers inside our emulator-rehostable virtual machine (i.e., the guest), Moneta gives the virtual machine passthrough access to the GPU hardware. Enabling passthrough access requires three steps. First, Moneta gives the guest's virtual CPU direct access to the GPU's memory-mapped I/O (MMIO) regions by creating MMU page table entries that translate guest-virtual addresses to the host-physical addresses of the GPU's MMIO region (see ③ in Figure 3). Second, Moneta forwards the GPU's physical interrupts as virtual interrupts to the guest virtual machine. Third, Moneta gives the GPU DMA access to the guest-physical memory and creates page table entries in the GPU's IOMMU. These entries translate I/O virtual addresses (IOVAs) into the host-physical addresses of the guest VM's DMA buffers, so that the GPU can directly access the guest's system memory using IOVAs.

The third step is not as straightforward as the first two since we need to consider the type and architecture of the IOMMU carefully. Discrete GPUs such as PCIe NVIDIA GPUs are typically behind a standalone IOMMU, which is controlled by an IOMMU driver running on the host (see ④ in Figure 3). In this case, we can enable DMA by making the host populate IOMMU entries that translate IOVAs into the host-physical memory addresses of the guest's DMA regions. This means that we can run unmodified GPU drivers inside the guest, when virtualizing GPUs with standalone IOMMUs.

In contrast, virtualizing SoC-integrated GPUs such as ARM Mali GPUs requires modification to their drivers, because integrated GPUs typically have a built-in IOMMU that is under the GPU driver's control (see ④ in Figure 3). This means that the GPU driver *inside the guest* must set up mappings from IOVAs to *host*-physical addresses, even though the guest has no access to the guest-physical to host-physical address mappings. To support these types of IOMMUs, we modify the GPU driver to invoke custom hypercalls to obtain *host*-physical memory addresses corresponding to guest virtual addresses when populating IOMMU page table entries. Moneta's virtual machine monitor in the host handles these custom hypercalls. To maintain rehostability of virtual machines, we implement our custom hypercall handlers, invoked by the guest during fuzzing, in the emulator. We detail our implementation of passthrough I/O virtualization of the ARM Mali GPU that has a built-in IOMMU in §V.

B. Snapshot and Recording Generation

Moneta observes the execution of the target GPU driver while it is executing inside an emulator-rehostable virtual machine that has a real, physical GPU assigned to it (see §IV-A). Inside this virtual machine, Moneta runs real GPU workloads such as a rendering job, during which we generate the following observations that capture various states of the target GPU driver: (i) virtual machine snapshots that capture the exact CPU and memory states, and (ii) recordings of system calls invoked by GPU workloads.

We use a *Guest Agent*, which leverages process and virtual machine introspection mechanisms to generate these observations automatically. This *Guest Agent*, shown as ① in Figure 4, is a user-mode process that is part of Moneta and runs in the guest virtual machine. The agent uses the Linux `ptrace` API to interpose itself between the guest kernel and the GPU process that generates the GPU workloads [2]. The agent can intercept, inspect, and possibly manipulate the GPU process' system calls. Moneta uses these interposition points (i) to instruct the hypervisor to generate snapshots, (ii) to record the system calls invoked by the workload process, and (iii) to instantiate a *Syscall Executor* in the guest user space during fuzzing (see §IV-C).

Virtual Machine Snapshot Generation. Moneta's *Guest Agent* can automatically generate virtual machine snapshots at each interposed system call by executing a hypercall that sends a snapshot request to the hypervisor. We configured the

Guest Agent to create snapshots at every fixed number of `ioctl` calls that operate on the GPU driver's device files. To minimize the storage space consumed by the generated snapshot images, Moneta runs the guest virtual machine with a copy-on-write file system image, so only the blocks modified with respect to the original image are serialized into each generated snapshot image.

Post-Snapshot Recording Generation. As explained in §II-C, one can only recall driver states at a coarse-grained granularity by restoring snapshots. For fine-grained driver execution recall past the snapshot point, Moneta also records external inputs provided to the GPU driver after each snapshot creation, thereby producing *post-snapshot* recordings. To capture external inputs, Moneta's *Guest Agent* records all the system calls (e.g., `ioctl` calls) made by the GPU workload process. In addition to the system calls, Moneta also records the I/O messages delivered to the GPU driver. Our current prototype implementation records I/O messages delivered through MMIO channels *and* through interrupts.

C. State-Preserving Snapshot Rehosting

After generating snapshots and their corresponding post-snapshot recordings, Moneta rehosts the snapshots in an ex-vivo environment created on x86 hosts.

Implanting Moneta's Input Executors. Upon resuming the guest, Moneta uses its *Guest Agent* to replace the original input executor processes (i.e., the GPU workload process and the *Guest Agent* itself) that drove the in-vivo execution of the GPU driver in the observation phase, with the *Syscall Executor* process shown as ② in Figure 4. The *Guest Agent* does this by either (i) forcing one of the GPU process' threads to invoke the `execve` syscall while killing the other threads and the *Guest Agent*, or (ii) by killing the entire GPU process and calling `execve` from the *Guest Agent* process itself. Simultaneously, Moneta creates an *emulated GPU* shown as ③ in Figure 4 in the virtual machine monitor in place of the physical GPU, either as a PCI device or a platform device depending on the bus that the GPU was originally attached to. Upon replacement, Moneta gains full control over the GPU driver's input space, enabling it to provide an arbitrary sequence of input events, whether they are recorded input events or fuzzer-generated ones, directly to the driver.

An alternative approach to delivering arbitrary input events to the driver is on-demand hooking [15], [57], where input events generated by an existing input provider, e.g., GPU applications, are hooked as they are executed and mutated on the fly. We chose a replacement-based approach over a hooking-based one because the latter does not allow arbitrarily inserting or deleting input events; it only allows mutating input events as generated by the existing input provider. Furthermore, hooking-based approaches would require the GPU hardware [57], i.e., the device-side input provider for the driver, which hinders large-scale fuzzing.

Preserving Live Driver-Internal Contexts. However, a challenge to the replacement approach is preserving the *driver-internal* state captured in the observation environment across replacements. This state includes (i) context information and metadata for the user-mode process that generated the GPU

Algorithm 1 Moneta's stateful GPU driver fuzzing.

```
 1: procedure FUZZ($s$, $R_s^{(k)}$)
 2:     ▷ $s$: A virtual machine snapshot
 3:     ▷ $R_s^{(k)}$: A sequence of $k$ syscalls recorded after creating $s$
 4:     $Corpus \leftarrow R_s^{(k)}$                ▷ Init. the corpus with a recording
 5:     $Cov = \emptyset$                           ▷ Init. coverage as an empty set
 6:     $LiveCtx \leftarrow$ RESTORE($s$)           ▷ Recall driver state using $s$
 7:     RUNEXECUTOR($LiveCtx$)                       ▷ Replace input executors
 8:     checkpoint                                   ▷ Create a checkpoint
 9:         if $mutation$ then                       ▷ Fuzzing loop iteration starts
10:             $R_s^{(n|0 \leq n \leq k)}||F \leftarrow$ MUTATE(SELECT($Corpus$))
11:             EXECUTE($R_s^{(n)}$)                 ▷ Recall driver state using $R_s^{(n)}$
12:         else if $generation$ then
13:             $R_s^{(n|n=0)}||F \leftarrow$ GENERATE
14:         end if
15:         $NewCov \leftarrow$ EXECUTE($F$)         ▷ Exec. fuzzer-mutated or
            -generated input events
16:         if $NewCov \cap Cov \neq \emptyset$ then
17:             $Corpus = Corpus \cup (R_s^{(n)}||F)$
18:             $Cov = Cov \cup NewCov$
19:         end if
20:     restore                                      ▷ Go back to the checkpoint
21: end procedure
```

workloads (*User-Mode Context*), and (ii) context information and metadata for the physical GPU (*GPU Context*). Moneta transfers this state to the fuzzing environment and needs to ensure that the state remains valid after the transfer. If we inadvertently invalidate or discard any of the state, then the fuzzing phase could effectively start *statelessly* from the initial state of the driver.

1) *User-Mode Context:* Moneta captures the user-mode context using its *Guest Agent*. Among other things, this agent maintains a list of open file descriptors for the Moneta-controlled user-mode process that runs in the observation environment. During rehosting, the *Guest Agent* replaces the GPU process and the agent with Moneta's *Syscall Executor*, and it transfers file descriptors that refer to the GPU driver's device files to the *Syscall Executor* process.
2) *GPU Context:* While replacing the physical GPU with an emulated GPU, Moneta creates an illusion that the GPU has remained the same to preserve the driver's context established with the physical GPU. To this end, Moneta instantiates the emulated GPU at the same address on the same bus to which the physical GPU was originally attached, and suppresses the device detach and re-attach events on the bus when the snapshot is restored. The GPU driver can then continue to execute, unaware of the GPU replacement. During replay and fuzzing, Moneta redirects all interactions between the driver and the GPU to the emulated GPU (see §IV-D). We emulate MMIO and IRQ responses in the emulated GPU using post-snapshot recordings, as detailed in §V.

D. Stateful GPU Driver Fuzzing

Moneta performs *stateful fuzzing*[2] of GPU drivers to find stateful bugs in them [7]. Moneta achieves this by fuzzing

[2]Following Ba et al. [7], by stateful fuzzing, we refer to fuzzing tailored to find "stateful" bugs, which require executing a specific sequence of input events.

```
r1 = fetch_preserved_fd$nvidia()
r2 = fetch_preserved_fd$nvidia()
ioctl$nvidia(r1, 0xc020462a,
↪  &(0x7f0000072ec0)="0200d0c10200d0c13601000000...")
ioctl$nvidia(r1, 0xc020462a,
↪  &(0x7f0000074f00)="0200d0c10200005c0c22802000")
ioctl$nvidia(r1, 0xc020462a,
↪  &(0x7f0000074f40)="0200d0c10200005c1022802000")
ioctl$nvidia(r1, 0xc020462a,
↪  &(0x7f0000076f80)="0200d0c10200005c0230802000...")
sysinfo(&(0x7f0000078fc0)=""/134)
```

Fig. 5: A simplified `syzkaller` program representing a post-snapshot syscall trace of NVIDIA's GPU driver. Moneta triggered a previously unknown slab-out-of-bounds bug by mutating the third argument of one of the `ioctl` calls.

the driver from states recalled via SnR and RnR. We formally describe Moneta's fuzzing algorithm in Algorithm 1, and provide a detailed description below.

Deterministic Live Driver State Recall (Line 6 – Line 7). Moneta's fuzzing loop starts by deterministically recalling in-vivo driver execution states via snapshot restoration (see Line 6). During this process, Moneta preserves the GPU and user-mode contexts the GPU driver expects, as explained in §IV-C (Line 7). Moneta uses these live driver-internal contexts, captured in each snapshot and preserved by Moneta across rehosting, to make fuzzing stateful. In each iteration of the fuzzing loop, Moneta uses `syzkaller` to generate a program to run after the snapshot restoration. In most cases, Moneta cannot run this program as-is because it may refer to the driver-internal context whose file descriptor handles have changed during rehosting. For example, the `syzkaller`-generated program might execute system calls on files whose file descriptor changed when Moneta transferred them to the *Syscall Executor* process. This can happen when a file's original file descriptor, as recorded in the observation phase, is unavailable in the context of the *Syscall Executor*.

To solve this problem, Moneta generates each program by (i) first inserting pseudo system calls that fetch the correct file descriptor handles by consulting metadata recorded during the observation phase, and (ii) then generating the rest using these fetched handles. Figure 5 shows an example program, which contains two such pseudo system calls at the beginning. Subsequent system calls then use the resources these pseudo system calls produce. This allows Moneta (i) to faithfully express post-snapshot traces of system calls, e.g., `ioctl` calls invoked on *existing* file descriptors, and also (ii) to generate new stateful system calls that use existing resources as their arguments during fuzzing.

Fine-Grained Driver State Recall & Stateful Fuzzing (Line 9 – Line 19). Moneta records the sequence of system calls invoked immediately after each snapshot creation, as this is likely a valid sequence of system calls that have the dependencies between them satisfied (thus addressing **P1**). Moneta uses these post-snapshot recordings as an initial seed corpus of Moneta's evolutionary fuzzing. When a post-snapshot recording is selected from the corpus and mutated during fuzzing (see Line 10 in Algorithm 1), the mutated program comprises a prefix of the recording (i.e., n out of k recorded input events, denoted by $R_s^{(n)}$ in Algorithm 1) followed by a fuzzer-mutated or fuzzer-generated part (denoted by F). Observe that any prefix of a recording, once executed (see Line 11), can reproduce a certain driver state observed while recording. Moneta then *statefully* fuzzes the driver from the past execution state reproduced via replay by executing the fuzzer-mutated or -generated part (Line 15).

State-Resetting Fuzzing Loop via Checkpoint-Restore (Line 8 & Line 20). Moneta constructs a state-resetting fuzzing loop by employing checkpoint-restore. When a new fuzzing campaign starts, Moneta creates a checkpoint of the entire virtual machine (Line 8) immediately after a successful state-preserving snapshot rehosting (see §IV-C), which installs the *Syscall Executor* controlled by Moneta's fuzzer, but before the executor consumes any fuzzer-generated input. This checkpoint, unlike the snapshots we generate during the observation phase, is ephemeral; that is, it is used only for the duration of each fuzzing campaign. Moneta restores this ephemeral checkpoint at the end of each fuzzing loop iteration (Line 20), so the next input can always execute from a clean, recalled state of the driver.

V. Implementation Details

CPU Feature Subsetting for ARMv8-to-x86 Rehosting. We implemented our emulator-rehostable virtualization technique (see §IV-A) for the ARMv8-to-x86 virtual machine rehosting scenario; that is, we rehosted virtual machine snapshots taken from an ARMv8 SoC onto an x86 (64-bit) machine running QEMU 4.0.0's ARMv8 CPU emulator called `max` [12]. We configured the ARMv8-based observation environment to use hardware-accelerated virtualization via Linux's Kernel Virtual Machine (KVM) [44]. We found that `max` lacks many features pertaining to performance monitoring and security enhancements. Specifically, it does not (fully) emulate the following features of ARMv8: Performance Monitoring Unit (PMU), Privilege Access Never (PAN), User Access Override (UAO), and Interrupt Translation Service (ITS) provided by ARMv8's Generic Interrupt Controller v3 (GICv3). We implemented our proposed CPU feature subsetting by removing all of the above features from our observation environment at the hypervisor level, either by removing them from the KVM abilities, or by modifying the configuration registers of the virtual CPUs.

NVIDIA/AMD GPU Passthrough. GPU passthrough requires configuring the guest virtual machine such that (i) the guest has direct access to the GPU's MMIO regions, (ii) GPU's interrupts are forwarded to the guest when raised, and (iii) GPU's IOMMU translates IOVAs to the physical memory addresses assigned to the guest (see §IV-A). NVIDIA and AMD GPUs are discrete GPUs attached to the PCIe bus and are behind a standalone IOMMU. Moneta assigns them to guests using Linux's VFIO mechanism [60]. The VFIO driver running in the host kernel configures the IOMMU driver such that the full system memory used by the GPU-assigned guest is exposed to the GPU for DMA.

ARM Mali GPU Passthrough. Unlike discrete GPUs that can use the IOMMU functionality provided by PCIe's root complex, an SoC-integrated GPU such as an ARM Mali GPU typically uses its own built-in IOMMU, and this built-in IOMMU is managed by the GPU driver running *inside* the guest (see §IV-A). We repurposed the built-in IOMMU for GPU passthrough by modifying the GPU driver. Specifically, we reprogrammed the translation entries for the built-in IOMMU such that the IOVAs the GPU uses for DMA translate to the host-physical address space rather than the guest-physical address space. To this end, Moneta introduces a new hypercall that takes as an argument a guest-physical address and returns its corresponding host-physical address after pinning the page at that address. Our modified ARM Mali GPU driver invokes this hypercall and uses the host-physical address returned by the hypercall when constructing page table entries for a given IOVA range.

Generating Snapshots & Post-Snapshot Recordings. We implemented Moneta's *Guest Agent* (see §IV-B) based on `strace` to monitor and manipulate system call invocations of the GPU application process [4]. In the observation phase, the agent serves two primary purposes at each system call: it either passively observes and records the system call, or it takes a snapshot of the guest by invoking a hypercall. After creation, we associate each snapshot with the recording of system calls that follow. GPU drivers pass references to vendor-specific (often nested) objects to `ioctl` calls. We added a minimal parsing routine in the *Guest Agent* that records these objects by copying them into a byte array. This implementation required a fairly modest amount of manual effort. Generally, we only needed to figure out the size of each type of object. For `ioctl` arguments that refer to nested objects, we also had to customize our parsing routine to serialize and copy the entire nested object into the byte array. In the worst case, for a specific `ioctl` operation in the NVIDIA drivers, this meant we had to serialize and copy three levels of objects.

Stateful GPU Driver Replay and Fuzzing. We based our implementation of Moneta's stateful fuzzing of the system call interface of GPU drivers (see §IV-D) on `syzkaller` [22]. We added support for replaying recorded system calls in `syzkaller` by translating the system call recordings into `syzkaller` programs so that `syzkaller`'s input executor can execute (or replay) them. For device-side inputs, we (i) recorded all MMIO and IRQ accesses made by the GPU driver by hooking their calls to corresponding access functions, (ii) derived rules (i.e., read-only and read-write rules [39]) based on the recording such that the rules capture the behavior of each MMIO location of the physical GPU, and (iii) performed rule-based MMIO and IRQ emulation via Moneta's emulated GPU device in the hypervisor. To implement our state-resetting fuzzing loop (see §IV-D), we used existing checkpoint-restore mechanisms. When fuzzing the NVIDIA and AMD GPU drivers, we used an existing checkpoint-restore implementation for KVM-accelerated x86 virtual machines [58]. When fuzzing ARM Mali GPU driver, we used QEMU's virtual machine snapshot and restoration mechanism.

VI. Evaluation

A. Snapshot and Recording Generation

Fuzzing Targets: Kernel-Mode GPU Drivers. We target the following three widely-used kernel-mode GPU drivers:

1) **NVIDIA GPU's Linux driver:** We used version 530.41.03 of the official NVIDIA Linux open GPU kernel modules[3]. We chose the open-source module to facilitate compile-time source code instrumentation.
2) **AMD Radeon GPU's Linux driver:** We used AMD GPU's Direct Rendering Manager (DRM) driver available in the upstream Linux kernel (v6.8).
3) **ARM Mali GPU's Linux driver:** We used the vendor-modified version (version number `g11p0-01eac0`) of ARM's official Mali Driver Development Kit (DDK) (version number `r25p0-01eac0`). As explained in §IV-A, we modified this driver to add support for GPU passthrough using a built-in IOMMU.

Table III: Our target kernel-mode GPU drivers and their size (in Lines of Code).

GPU	Kernel-Mode Driver Source Code	LoC
NVIDIA	Official, out-of-tree module[1]	925,157
AMD Radeon	`drivers/gpu/drm/amd/amdgpu` (Upstream)	225,341
ARM Mali	`drivers/gpu/arm/bifrost`[2] (Downstream[3])	83,260

[1] https://github.com/NVIDIA/open-gpu-kernel-modules
[2] This driver supports both Bifrost- and Valhall-architecture GPUs.
[3] https://github.com/khadas/linux

Table IV: The user-mode GPU drivers, libraries, and the physical GPUs used to capture the execution states of each target kernel-mode driver.

Kernel-Mode GPU Driver	Physical GPU (HW Architecture)	User-Mode GPU Drivers & Libraries
NVIDIA	GTX 1650 Super (Turing)	`libnvidia-egl*.so` `libEGL_nvidia.so` `libnvidia-opencl.so` `libcuda.so`
	RTX 3060 (Ampere)	
	RTX 4060 Ti (Ada Lovelace)	
AMD Radeon	Radeon RX 580 (Polaris 20)	`libGL*.so` `libEGL(_mesa).so` `lib(Mesa)OpenCL.so`
ARM Mali	Mali-G610 MP4 (Valhall)	`libmali-...-wayland.so`[1]

[1] `libmali-valhall-g610-g15p0-wayland.so`

To maximize fuzzing effectiveness, we instrumented these drivers with two kernel sanitizers: AddressSanitizer [36] and UndefinedBehaviorSanitizer [51]. Since we insert the necessary instrumentation at compile time, we need source code access to all drivers we fuzz. Generally, however, we do not need to modify the source code. The only exception is when we need to add paravirtualization support to drivers for GPUs with built-in IOMMUs (e.g., ARM Mali GPUs), as we explained in §IV-A.

User-Mode GPU Driver/Library & GPU Configurations. In the observation phase of Moneta, we ran the three target kernel-mode GPU drivers with the user-mode drivers/libraries and physical GPUs shown in Table IV. We used either vendor-specific, closed-source user-mode drivers and libraries that we installed separately, e.g., `libEGL_nvidia.so`, or the generic ones included in the guest OS distribution (Debian 11 codenamed "bullseye") we used, e.g., `libOpenCL.so`. On the GPU side, we used the following: For the AMD and ARM GPU

[3]In NVIDIA's proprietary driver package, there are two kernel modules: one open-source and one closed-source module. The open-source kernel module in the version we used is missing some functionalities (e.g., GPU power management).

drivers, we used a dedicated PCIe AMD Radeon RX 580 GPU, and ARM Mali-G610 GPU integrated into the RK3588S SoC, respectively. For the NVIDIA GPU driver, we used three recent generations of the PCIe NVIDIA GPUs with different processor architectures. This variety of NVIDIA GPUs allowed us to capture a wider range of states of the NVIDIA GPU driver, because the driver contains many hardware-specific code paths.

GPU Workload Configurations. We ran several graphics and compute workloads in our observation phase.

1) As a **graphics workload**, we chose a WebGL [30] application running in the Chromium browser [1]. Specifically, we visited a website that renders an aquarium scene [5]. We chose this workload because recent work showed that many GPU driver bugs are exploitable via the WebGL interface [71].
2) For a **compute workload**, we used an OpenCL program that generates a fractal image [3]. We chose a single simple program because most compute workloads exercise similar code paths in the kernel driver. Among different options for GPU-accelerated compute libraries, we chose OpenCL [34], because most GPU vendors support it.

We used multiple classes of GPU workloads to cover various kernel-mode driver code paths. Different classes of GPU workloads use (i) different sets of user-mode libraries (e.g., OpenGL [54] and OpenCL [34]), (ii) different user-mode proprietary GPU drivers, and often (iii) different kernel-mode driver modules as well. For example, running compute workloads on an NVIDIA GPU triggers code paths in an NVIDIA kernel driver module called `nvidia-uvm`, which are different from the ones covered by graphics workloads.

Generated Snapshot & Post-Snapshot Recordings. Using the GPU stack and workload configurations detailed above, we generated a number of snapshots and their post-snapshot recordings for each driver we targeted. For the NVIDIA and AMD Radeon GPU driver, we generated 128 unique snapshots for each driver using a combination of the graphics and compute workloads. For the ARM Mali GPU driver, we only generated snapshots using the compute workload, because the vendor-customized WebGL-supported Chromium did not work in our guest environment. For each snapshot, we included 200 system calls in its post-snapshot recording. This was a conservative choice, because only fewer than a dozen (out of 200) contributed to increased coverage, because the driver execution started to diverge from the recorded execution after the dozen calls due to nondeterminism. We used the I/O recordings generated for the entire workload duration to derive MMIO and IRQ rules, as described in §V.

B. Fuzzing Effectiveness

We evaluated Moneta's effectiveness on two major axes: (i) its coverage of kernel-mode GPU stack (including the target GPU drivers), and (ii) bug finding capability.

Experimental Setup. Following our threat model (see §III), we fuzzed the system call interface of the target GPU drivers using Moneta. We ran three twenty-four-hour fuzzing campaigns, one for each target driver. In each fuzzing campaign, we fuzzed a single GPU driver by leveraging all of the snapshots along with their post-snapshot recordings generated for that driver (see §VI-A). All the fuzzing campaigns, including the ones targeting the ARM Mali GPU driver, were executed on a machine equipped with two 64-thread Intel Xeon 8358 CPUs (a total of 128 threads) and 1TB of RAM.

We ran 128 fuzzing instances in parallel in each campaign. We assigned each snapshot (and its post-snapshot recording) of the driver to one or more fuzzing instances and gave each fuzzing instance a single snapshot only. This means that every fuzzing instance always fuzzes the target driver from the state recalled from a single snapshot and its corresponding post-snapshot recording.

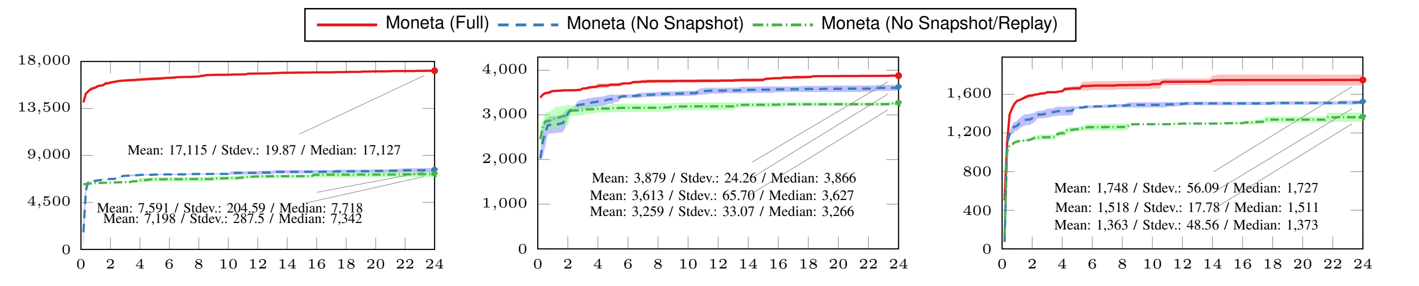
We injected (i) user-mode-originating inputs to drivers by replaying post-snapshot recordings of system calls, and (ii) device-originating inputs by emulating the GPU with the rules derived from I/O recordings, as described in §V. We performed coverage-guided, evolutionary fuzzing, by giving each fuzzing instance its designated post-snapshot system call recording as an initial corpus. To prioritize fuzzing the target GPU driver and not other kernel components, we limited our coverage instrumentation only to (i) the target GPU driver and (ii) two kernel subsystems for GPU acceleration, namely `drivers/gpu` and `drivers/video`. We also enabled the Linux kernel's AddressSanitizer [36] and UndefinedBehaviorSanitizer [51] instrumentation to detect bugs more reliably when triggered by Moneta.

Baseline Configurations. Since the key contribution of Moneta is combining RnR and SnR for a more effective GPU driver fuzzing, we first ablate Moneta's capability of rehosting in-vivo snapshots; we refer to this configuration as No Snapshot. This first baseline effectively simulates an enhanced version of a state-of-the-art GPU driver fuzzer that uses record-and-replay [39], with the following enhancements: (i) Moneta's RnR at the system call interface, (ii) Moneta's SnR albeit using a single snapshot taken after a successful probing of the driver, and (iii) Moneta's ARM-to-x86 SnR. We introduce another configuration referred to as No Snapshot/Replay, which additionally ablates the syscall-side RnR capability of Moneta. This second baseline reflects common usage scenarios of `syzkaller`, in which evolutionary fuzzing is conducted without a starting corpus derived from in-vivo driver executions.

We emphasize that, to conservatively (and therefore accurately) evaluate the effectiveness of Moneta's past execution state recall (i.e., SnR and RnR) on GPU driver fuzzing, the baselines were all configured to use (i) the same driver-specific `ioctl` grammars we manually wrote for Moneta, and (ii) Moneta's device-side emulation. We evaluate the impact of Moneta's device-side emulation separately in §VI-D.

Code Coverage Results. We depict the basic block coverage of the target GPU drivers obtained during our fuzzing campaigns in Figure 6. Moneta outperforms the two strong baselines consistently across all three GPU drivers we targeted. This means that Moneta can indeed sidestep the problems of RnR caused by nondeterminism by using SnR.

Our further investigation revealed that there are many ways nondeterminism affects the fidelity of RnR, and its



(a) NVIDIA GPU driver. (b) AMD Radeon GPU driver. (c) ARM Mali GPU driver.

Fig. 6: Basic block coverage (y-axis) measured every ten minutes during twenty-four hours (x-axis) of fuzzing each GPU driver. We ran Moneta three times in each configuration (i.e., Full, No Snapshot, and No Snapshot/Replay), reporting the geometric mean and the standard deviation of coverage using lines and shading, respectively. In each experiment (including ARM Mali GPU driver fuzzing), we performed 128-instance parallel fuzzing on a 128-thread x86 server.

effectiveness at enhancing fuzzing. For example, the system call payloads of the NVIDIA GPU driver (where Moneta performs best) often include (i) specific virtual address values (which could be *randomized* due to ASLR or *uncontrolled* by RnR), and (ii) specific driver-internal handle values assigned to various driver-controlled resources (which can change when other resource creation or deletion is not fully controlled). These values could, in theory, be included in the abstraction of RnR. However, given that the `ioctl` input space of GPU drivers is huge and proprietary, doing so would require significant engineering effort. Moneta does not require such efforts and can recall in-vivo execution states where (i) those values are deterministically restored, and (ii) the timing constraints imposed on replaying recorded inputs are easier to satisfy.

Similar nondeterminism issues complicate the device-side input generation as well. Even though we derive rules from I/O recordings that capture real I/O behavior, rule-based I/O emulations based on recordings (including those of BSOD [39] and Moneta) cannot be as accurate as I/O behavior of real GPUs. Moneta's SnR also helps mitigate this challenge of high-fidelity I/O emulation from the GPU side, because the driver states recalled via Moneta's SnR precisely replicate the outcomes of past in-vivo driver executions that interact with a real GPU. Although GPU I/O behavior may still be inaccurately emulated after restoring an in-vivo snapshot, we can bypass such hard-to-emulate behavior by using another snapshot captured after the real GPU has demonstrated that behavior.

When comparing the two baselines, we see that our first baseline representing RnR (denoted by No Snapshot), outperforms the other (No Snapshot/Replay). It achieves higher coverage, benefiting from the initial corpus program, but only marginally higher coverage, due mainly to the aforementioned nondeterminism challenges of RnR.

Bug Finding Results. Our fuzzing campaigns with Moneta uncovered, in total, 10 previously unknown bugs across all the drivers we fuzzed. As summarized in Table V, we found 5 bugs in the NVIDIA's, and 3 in AMD Radeon's, and 2 in ARM Mali's GPU driver.

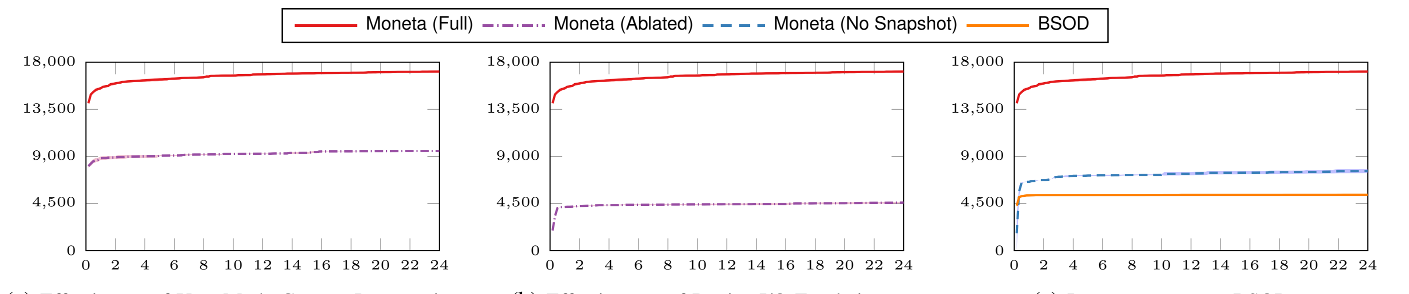
Since the GPU emulation in our prototype is not fully accurate, Moneta can, in theory, find false bugs. For each GPU driver bug we triggered during fuzzing, we verified that it can indeed be triggered under our threat model, by invoking a minimized bug-triggering sequence of system calls (disclosed in §A) on a machine equipped with a real GPU. Although theoretically possible, we have not encountered any false bugs during our experiments.

Table V: Bugs found while fuzzing our target GPU drivers using Moneta. For each bug, we show the snapshot point in terms of the number of system calls, and the minimized bug triggering program, which indicate how stateful the bug is.

GPU Driver	Id.	Bug Type	*Minimized* Bug Trigger[1]		Status
			Snapshot Point	Program Size	
NVIDIA	1	Slab-out-of-bounds write	339	2	CVE-2024-0090
	2	Slab-out-of-bounds write	9,441	2	
	3	Slab-out-of-bounds read	341	2	CVE-2024-0075[2]
	4	Null pointer dereference	341	2	
	5	Null pointer dereference	0	5	
AMD Radeon	6	Use-after-free	0	2	CVE-2024-26656
	7	Slab-use-after-free	0	3	CVE-2024-27400[2]
	8	Null pointer dereference	0	3	CVE-2024-26657
ARM Mali	9	Shift out-of-bounds	0	4	Confirmed
	10	Shift out-of-bounds	0	4	Confirmed

[1] The bug triggering programs are disclosed in §A.
[2] These bugs were concurrently found by the respective vendor or other researchers.

We examined all the bugs found by Moneta to assess the effectiveness of Moneta's in-vivo driver state recall capability. Among these bugs, four bugs in the NVIDIA GPU driver were only found by using Moneta's SnR. Triggering them requires deep, in-vivo driver execution states that are challenging to recall via RnR alone, as indicated in the ***Minimized* Bug Trigger** columns in Table V. Of these four bugs, we detail two high-severity bugs in §VI-F. While two bugs in the ARM Mali driver were also discovered by using RnR alone, these bugs still demonstrate the usefulness of Moneta's ARM-to-x86 rehosting capability; that is, x86 servers, which are much more accessible than high-end ARM servers, can be used to find ARM-SoC-attached GPU driver bugs, by using Moneta.



(a) Effectivenss of User-Mode Context Preservation. (b) Effectiveness of Device I/O Emulation. (c) Improvement over BSOD.

Fig. 7: Block-level coverage of the NVIDIA GPU driver (y-axis) measured over 24 hours (x-axis), obtained by (a) Moneta without its user-mode context preservation, (b) Moneta without its device I/O emulation, and (c) BSOD, each compared with Full and/or No Snapshot configuration of Moneta.

Table VI: The number of inputs (i) executed in total during fuzzing, and (ii) kept in the corpus at the end, averaged (geometric mean) over multiple runs of the same experiment.

GPU Driver	Moneta Configuration	Total Executions (# of Execs/Minute)	Corpus Size (Total BB Coverage)
NVIDIA	Full	**50,781,942 (35,265)**	**1,378 (17,115)**
	No Snapshot	30,667,828 (21,297)	334 (7,588)
	No Snapshot/Replay	38,754,625 (26,913)	262 (7,198)
AMD Radeon	Full	28,726,479 (19,949)	**690 (3,879)**
	No Snapshot	25,485,670 (17,698)	508 (3,612)
	No Snapshot/Replay	**31,084,091 (21,586)**	452 (3,258)
ARM Mali	Full	3,358,603 (2,332)	**235 (1,748)**
	No Snapshot	3,942,659 (2,738)	209 (1,518)
	No Snapshot/Replay	**4,312,300 (2,995)**	202 (1,363)

C. Fuzzing Throughput

Table VI shows, for each experiment, (i) the number of inputs executed and (ii) the number of inputs kept in the corpus at the end of our coverage-guided fuzzing. When fuzzing the NVIDIA GPU driver, Moneta significantly outperforms the baselines in both measures. However, when fuzzing the AMD Radeon and ARM Mali GPU driver, the fuzzing speed becomes lower in the full Moneta configuration than the two baselines. Even with a smaller number of executions, however, Moneta covered more driver code paths than all the baselines in all GPU drivers, and, consequently, more inputs were kept in the corpus with coverage-guided fuzzing.

Low fuzzing speed is a common argument against using an emulator for fuzzing. Moneta uses CPU emulators for ARM Mali GPU driver fuzzing and can rehost snapshots taken from a GPU-equipped ARM platform onto a non-native (x86) platform with an emulated ARM CPU. Though slower than fuzzing x86 drivers, we argue that fuzzing ARM's Mali GPU driver on a non-native platform still achieves a reasonable fuzzing speed (up to millions of inputs in twenty-four hours), as shown in Table VI. This can be attributed to Moneta's ability to massively parallelize fuzzing on a multi-core server CPU, such as the 128-thread x86 server we used for fuzzing ARM Mali GPU's driver.

D. Other Ablation Studies

Although we showed in §VI-B that Moneta can improve GPU driver fuzzing via a synergistic combination of SnR and RnR over multiple baselines that represent prior state-of-the-art in kernel (driver) fuzzing [39], [22], we conduct additional studies to quantify the contribution of Moneta's individual components. We measure their effectiveness by counting the basic block coverage, and targeting NVIDIA's GPU driver mainly because it is most complex among the target drivers, and because Moneta was most effective when fuzzing it.

Impact of State-Preserving Rehosting. We first show that, without state-preserving rehosting (see §IV-C), the driver effectively disregards the deep, interesting states that were captured using real GPU workloads. We create a configuration of Moneta, which ablates its user-mode context preservation (see §IV-C), and run the same fuzzing experiment using this configuration. Figure 7a depicts the result, in comparison to the full configuration of Moneta. We find the configuration that does not preserve user-mode contexts performs significantly worse than those that do, demonstrating the effectiveness of Moneta's state-preserving rehosting.

Impact of Device I/O Emulation. We run the same fuzzing experiment with Moneta but without its device emulation, and depict the result in Figure 7b. The result shows that, without I/O emulation, GPU driver fuzzing through the system call interface performs worse. In fact, during the fuzzing campaigns, we observed many MMIO accesses issued by the GPU software stack. One of them, for example, is a check for GPU attachment status. If this MMIO read is not emulated, the driver refuses to continue execution.

E. Baseline Representativeness

To support our claim that Moneta's No Snapshot configuration is a stronger version of BSOD [39], we compare this baseline configuration with the open-source version of BSOD[4]. Although BSOD is a state-of-the-art GPU driver fuzzer that offers several key contributions, our comparison specifically focuses on its ability to record-and-replay the device-side inputs of GPU drivers, namely MMIO and IRQ. Since the open-source version of BSOD is tailored for fuzzing NVIDIA GPU drivers operating with a different set of NVIDIA GPUs,

[4]Available at: https://github.com/0xf4b1/bsod-kernel-fuzzing

we adjusted BSOD such that it uses the same I/O recordings that Moneta's device-side RnR used to emulate GPU-side inputs. To ensure fairness, we also gave BSOD the same the NVIDIA driver `ioctl` grammars we used. With these adjustments, we performed 128-instance fuzzing of the same target, i.e., the same NVIDIA driver running inside the same guest kernel. The result depicted in Figure 7c confirms that our own No Snapshot baseline outperforms BSOD.

F. Case Studies: Two OOB Writes in NVIDIA's Driver

Of the 4 bugs in the NVIDIA GPU driver that require the deterministic recall capability of Moneta via its snapshot rehosting, we detail the two slab out-of-bounds (OOB) write bugs, identified by Id. 1 and 2 in Table V. The vendor rated the severity of these bugs as high, noting that their successful exploitation could potentially lead to code execution, denial of service, escalation of privileges, information disclosure, and data tampering.

Moneta triggered each of these two bugs by restoring a snapshot taken after a number of system calls, followed by a single `ioctl` call on an existing file descriptor preserved across rehosting. The payload of each bug-triggering `ioctl` call contains three handle values (whose type is `NvHandle`) that refer to driver-controlled resources, which must all be valid handle values to trigger the bug. However, it is challenging to reproduce valid handle values through RnR alone due to nondeterminism, or through fuzzing. Also, even if we use deterministic RnR, triggering these bugs would require long fuzzing time without SnR.

Moneta triggered the first OOB bug by restoring a snapshot taken after the GPU application invoked 339 system calls, and subsequently invoking a fuzzer-crafted `ioctl` call. The OOB access is triggered specifically in a graphics context (i.e., `vidmemConstruct_IMPL`), and, therefore, can only be triggered by restoring snapshots obtained from graphics workloads. The post-snapshot recordings of the graphics workload we used contained hundreds of the bug-triggering `ioctl` that invokes the bug-triggering function, while the same `ioctl` was rarely observed (up to only once) in the recordings obtained from compute workloads. This bug could provide a *constrained* write-where-what primitive. By crafting an `ioctl` payload, the attacker can first overflow the size argument of a memory allocation function. By controlling the amount of this unsigned integer overflow, the attacker can allocate a memory region whose size is smaller than the object it holds. The driver subsequently accesses various fields of this object, most of which go out-of-bounds overwriting the victim object adjacent to the allocated memory region. With heap manipulation, the attacker can (i) allocate this region right before the victim object (implies *where* in write-where-what), and (ii) partially control the value written to this adjacent object as the `ioctl` payload influences this value (implies *what*).

Moneta triggered the second OOB bug by invoking an `ioctl` call after restoring a snapshot taken after 9,441 system calls. Similar to the first OOB bug, triggering this bug begins with the attacker allocating a memory region that is smaller than the object it is supposed to contain, caused by the same unsigned integer overflow. Unlike the first OOB bug, however, this object is an array, and the overflow occurs in a different context (i.e., `pmaRegmapScanDiscontiguous`). A loop in this context iterates over this array, with its termination condition also determined by the `ioctl` payload, eventually leading to out-of-bounds access. With the help of heap layout manipulation, the attacker can allocate this object right before the victim object (implies *where* in write-where-what). The attacker can then trigger the out-of-bounds write bug, thereby overwriting the victim object. Also, this attacker could control the value which is written to the victim object (implies *what*), because it is also part of the `ioctl` payload.

VII. Discussion & Limitation

GPU-Side Attack Surface. While our work focuses on finding vulnerabilities in kernel-mode GPU drivers that can be triggered by user-mode adversaries, attacking the GPU driver from the GPU side is also possible. GPUs could directly be compromised from user-mode adversaries without compromising their kernel-mode drivers, which could then be used as a stepping stone to attack kernel-mode drivers [27]. Alternatively, an attacker could physically attach a rogue PCIe device to the victim machine, which advertises itself as a GPU, using PCIe slots or even Thunderbolt ports [40] in "drive-by" (e.g., evil maid) attack scenarios. Upon attaching a rogue GPU, its corresponding GPU driver would be loaded on the CPU side, which could then be attacked by sending maliciously crafted I/O messages to the driver. We intend to explore this GPU-side attack surface in the future by fuzzing the I/O interface of the GPU drivers.

Remote Attack Surface. Our work fuzzes kernel-mode GPU drivers, assuming full access to the user-mode-exposed interface of the drivers. This interface could further be exposed to remote untrustworthy users, albeit indirectly, through, e.g., WebGL [30] for accelerated rendering of web content, WebGPU [43] for accelerated general computation through the web, or H.264-on-GPU for accelerated decoding of video content [63]. These remote contents typically execute in a sandbox [72], and, because of the sandbox, they may not be able to trigger low-level GPU driver bugs found by Moneta. We argue, however, that the bugs Moneta can find are still valuable, because they are essential in constructing a full remote exploit chain that compromises the kernel. Moreover, most of the Moneta's design except for the fuzzer implantation part can also be applied to fuzzing the WebGL, WebGPU, and H.264 interfaces directly exposed to remote contents as well. In our future work, we intend to augment Moneta's capability to directly fuzz these interfaces to the GPU software stack.

Fuzzing GPU Drivers in Non-Linux OSs. Our current implementation of Moneta targets kernel-mode GPU drivers running in Linux. We expect that, with additional engineering effort, our approach can be applied to fuzzing GPU drivers running in other OSs as well. When targeting GPU drivers running in non-Unix-like OSs such as Windows, most of the engineering effort would be spent on reconciling the differences in the system call interface. The `ptrace`-based implementation of Moneta's *Guest Agent* would also be replaced with a comparable process introspection mechanism offered by non-Unix-like OSs. Other components of Moneta implemented at the hypervisor level, e.g., CPU feature subsetting, would seamlessly work across OSs.

Synergy with Other Dynamic Analysis. Moneta's GPU driver execution recall capability could also fulfill dynamic analysis demands other than fuzzing, such as interactive debugging and exploit development. When Moneta identifies bugs in a GPU driver through fuzzing, one could interactively debug them using Moneta. Interactive debugging could help with developing a proper fix for the underlying issue, and assessing the exploit potential of the bug. If the bug can be exploited, Moneta could aid even interactive exploit development as well. Besides, Moneta's stateful fuzzing can benefit from other dynamic bug finding techniques as well. Though we only applied KASAN [36] and KUBSAN [51] to the GPU drivers, other tools such as memory leak detectors and data race bug detectors could also be employed with Moneta to find other kinds of bugs.

VIII. Related Work

Device-Free Driver Analysis. To put Moneta into a broader context of *device-free* driver analysis, we further discuss non-fuzzing approaches and other device-free fuzzing approaches. Non-fuzzing approaches use static pointer/pattern analysis [10], [38], [9], [8], [41] or symbolic execution [49], [37], [19], [45], [23] to statically reason about the behavior of drivers. Another line of work uses static analysis to enhance driver fuzzing, via symbolic execution [56], [69], input format inference [14], [25], and other driver-tailored static analysis [17], [76]. All of these techniques analyze drivers without requiring hardware devices, but could suffer from the precision and scalability challenges of static analysis. Also, since GPUs are omnipresent, one can easily use GPU-in-the-loop approaches such as Moneta to complement static analysis.

Unlike prior work relying on static analysis, SATURN [70] uses fuzzing to enhance USB driver fuzzing. The idea is to configure a pair of USB drivers, one on the host side and the other on the gadget (i.e., device) side, letting them interact with each other. SATURN simultaneously fuzzes both sides to trigger interesting (or buggy) behaviors on their opposite sides. This approach does not trivially extend to GPU driver fuzzing, however, due to the difficulty of fuzzing GPU (i.e., device-side) firmware.

Firmware Rehosting and Fuzzing. Firmware rehosting is a technique that runs firmware in an emulated environment without requiring hardware [68]. Due to its key advantage of not requiring hardware, researchers proposed many firmware rehosting solutions that use a variety of static and dynamic analysis techniques to increase the fidelity of firwmare execution in a rehosting environment [24], [50], [21], [16], [33], [75], [55], [52], [29]. Moneta distinguishes itself from this line of work, in that it targets GPU-equipped hardware platforms, which are much more capable than embedded devices that firmware rehosting targets. In particular, Moneta uses hardware-assisted virtualization available in most of GPU-equipped hardware platforms, and introduces new virtualization primitives that facilitate rehosting virtual machine snapshots on a powerful x86 machines that lack GPU hardware.

Snapshot Rehosting for Fuzzing. The idea of employing a snapshot-and-rehost approach for fuzzing is inspired by prior work on firmware rehosting that uses bare-metal memory snapshots [50], [29]. This line of prior work is different from ours in the following ways: (i) prior work takes bare-metal memory snapshots whereas ours takes virtual machine snapshots, (ii) prior work uses snapshots only, while ours uses snapshots in combination with record-and-replay, which requires careful state preservation, and (iii) prior work targets a related but different problem of firmware fuzzing whereas ours targets GPU device driver fuzzing. EASIER also uses a snapshot technique to facilitate ex-vivo Android device driver fuzzing [48]; however, it creates custom snapshots by manually dumping portions of kernel memory and CPU registers, requiring significant engineering efforts to rehost those partial snapshots in an emulator. Additionally, unlike Moneta, EASIER uses neither record-and-replay nor does it conduct stateful fuzzing.

System Call Interposition for Kernel Fuzzing. System call interposition has been proved useful in kernel fuzzing [15], [11]. Moneta also uses system call interposition after each in-vivo snapshot creation and ex-vivo snapshot restoration. Specifically, Moneta uses the Linux `ptrace` API to interpose the system calls invoked by the original GPU workload process, (i) recording them after each snapshot creation, and (ii) implanting the input executor by forcefully changing one of them to `execve` after snapshot restoration. A well-known disadvantage of `ptrace`, however, is its run-time overhead, which could slow down post-snapshot recording creation, and the GPU workload process itself. To address this limitation, alternative system call interposition mechanisms optimized for lower overhead could be used [65], [35], [64], [67], [66], [73], [31], [28], [53].

IX. Conclusion

We presented Moneta, a new type of device driver fuzzer that combines SnR and RnR to deterministically recall deep driver states while remaining capable of evolutionary fuzzing. In its observation phase, Moneta observes applications that interact with the target driver in a virtual machine that implements our proposed emulator-rehostable virtualization technique. Moneta periodically generates system snapshots and system call traces that serve as the corpora for its fuzzing phase. This full set of techniques enables Moneta to fuzz efficiently and effectively at large scale, using powerful testing infrastructure without physical instances of the devices whose drivers we want to fuzz. We thoroughly evaluated Moneta and demonstrated its unique capabilities on a set of widely used open-source GPU drivers. We found and responsibly disclosed 10 previously undiscovered bugs.

Acknowledgment

The authors would like to thank the anonymous shepherd and reviewers for their valuable feedback. This material is based upon work partly supported by National Research Foundation (NRF) of Korea under grant nr. RS-2024-00334395, and by Institute of Information & Communications Technology Planning & Evaluation (IITP) of the Korea government's Ministry of Science and ICT (MSIT) under grant nr. RS-2024-00439762. This work is also partially funded by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and the Fund for Scientific Research - Flanders (FWO) under grant nr. G033520N. We also gratefully acknowledge the awards from the Yonsei University Research Fund of 2022 (nr. 2022-22-0103) and 2023 (nr. 2023-22-0108).

REFERENCES

[1] "The Chromium projects – Chromium." [Online]. Available: https://www.chromium.org/Home

[2] "ptrace." [Online]. Available: http://man7.org/linux/man-pages/man2/ptrace.2.html

[3] "Simple OpenCL^{TM} samples – Julia Set." [Online]. Available: https://github.com/bashbaug/SimpleOpenCLSamples/tree/main/samples/04_julia

[4] "strace." [Online]. Available: https://man7.org/linux/man-pages/man1/strace.1.html

[5] "WebGL samples – Aquarium." [Online]. Available: https://webglsamples.org/aquarium/aquarium.html

[6] Arm Limited, "Fast models fixed virtual platforms (FVP) reference guide," 2023.

[7] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *Proceedings of the USENIX Security Symposium (Security)*, 2022.

[8] J.-J. Bai, T. Li, K. Lu, and S.-M. Hu, "Static detection of unsafe DMA accesses in device drivers," in *Proceedings of the USENIX Security Symposium (Security)*, 2021.

[9] X. Bai, L. Xing, M. Zheng, and F. Qu, "iDEA: Static analysis on the security of Apple kernel drivers," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[10] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2006.

[11] I. Beer, "pwn4fun spring 2014 - Safari - part II," 2014. [Online]. Available: https://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html

[12] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005.

[13] O. Chang, "Attacking the Windows NVIDIA driver," 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/02/attacking-windows-nvidia-driver.html

[14] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "SyzGen: Automated generation of syscall specification of closed-source macOS drivers," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2021.

[15] J. Choi, K. Kim, D. Lee, and S. K. Cha, "NTFUZZ: Enabling type-aware kernel fuzzing on Windows with static binary analysis," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2021.

[16] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *Proceedings of the USENIX Security Symposium (Security)*, 2020.

[17] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[18] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the Linux ARM hypervisor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[19] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and precise symbolic analysis of concurrency bugs in device drivers," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[20] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[21] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proceedings of the USENIX Security Symposium (Security)*, 2020.

[22] Google, "syzkaller - kernel fuzzer." [Online]. Available: https://github.com/google/syzkaller

[23] R. Gupta, L. P. Dresel, N. Spahn, G. Vigna, C. Kruegel, and T. Kim, "POPKORN: Popping Windows kernel drivers at scale," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2022.

[24] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[25] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "SyzDescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2023.

[26] Y. Hao, H. Zhang, G. Li, X. Du, Z. Qian, and A. A. Sani, "Demystifying the dependency challenge in kernel fuzzing," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2022.

[27] B. Hawkes, "Attacking the Qualcomm Adreno GPU," 2020. [Online]. Available: https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html

[28] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[29] Z. Hu and B. Dolan-Gavitt, "IRQDebloat: Reducing driver attack surface in embedded devices," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2022.

[30] D. Jackson and J. Gilbert, "WebGL 2 specification," Apr. 2017. [Online]. Available: https://registry.khronos.org/webgl/specs/2.0.0

[31] A. Jacobs, M. Gülmez, A. Andries, S. Volckaert, and A. Voulimeneas, "System call interposition without compromise," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024.

[32] J. Jang, M. Kang, and D. Song, "ReUSB: Replay-guided USB driver fuzzing," in *Proceedings of the USENIX Security Symposium (Security)*, 2023.

[33] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *Proceedings of the USENIX Security Symposium (Security)*, 2021.

[34] Khronos® OpenCL Working Group, "The OpenCL^{TM} specification," Apr. 2023. [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html

[35] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[36] A. Konovalov and D. Vyukov, "KernelAddressSanitizer (KASan): A fast memory error detector for the Linux kernel," *LinuxCon North America*, 2015.

[37] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with DDT," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[38] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr. Checker: A soundy analysis for Linux kernel drivers," in *Proceedings of the USENIX Security Symposium (Security)*, 2017.

[39] D. Maier and F. Toepfer, "BSOD: Binary-only scalable fuzzing of device drivers," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[40] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. M. Watson, "Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[41] Microsoft, “Static driver verifier,” 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier

[42] ——, “IOMMU-based GPU isolation,” 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/iommu-based-gpu-isolation

[43] K. Ninomiya, B. Jones, and J. Blandy, “WebGPU W3C working draft,” Nov. 2023. [Online]. Available: https://www.w3.org/TR/2023/WD-webgpu-20231115

[44] Open Virtualization Alliance, “Linux kernel virtual machine.” [Online]. Available: https://www.linux-kvm.org

[45] J. Patrick-Evans, L. Cavallaro, and J. Kinder, “POTUS: Probing off-the-shelf USB drivers with symbolic fault injection,” in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[46] H. Peng and M. Payer, “USBFuzz: A framework for fuzzing USB drivers by device emulation,” in *Proceedings of the USENIX Security Symposium (Security)*, 2020.

[47] H. Peng, Z. Yao, A. A. Sani, D. J. Tian, and M. Payer, “GLeeFuzz: Fuzzing WebGL through error message guided mutation,” in *Proceedings of the USENIX Security Symposium (Security)*, 2023.

[48] I. Pustogarov, Q. Wu, and D. Lie, “Ex-vivo dynamic analysis framework for Android device drivers,” in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[49] M. J. Renzelmann, A. Kadav, and M. M. Swift, “SymDrive: Testing drivers without devices,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[50] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, “Frankenstein: Advanced wireless fuzzing to exploit new Bluetooth escalation targets,” in *Proceedings of the USENIX Security Symposium (Security)*, 2020.

[51] A. Ryabinin, “UBSan: Run-time undefined behavior sanity checker,” 2014. [Online]. Available: https://lwn.net/Articles/617364

[52] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, “Fuzzware: Using precise MMIO modeling for effective firmware fuzzing,” in *Proceedings of the USENIX Security Symposium (Security)*, 2022.

[53] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, “Jenny: Securing syscalls for PKU-based memory isolation systems,” in *Proceedings of the USENIX Security Symposium (Security)*, 2022.

[54] M. Segal and K. Akeley, “The OpenGL® graphics system: A specification,” May 2022. [Online]. Available: https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf

[55] L. Seidel, D. Maier, and M. Muench, “Forming faster firmware fuzzers,” in *Proceedings of the USENIX Security Symposium (Security)*, 2023.

[56] Z. Shen, R. Roongta, and B. Dolan-Gavitt, “Drifuzz: Harvesting bugs in device drivers from golden seeds,” in *Proceedings of the USENIX Security Symposium (Security)*, 2022.

[57] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, “PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[58] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, “Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints,” in *Proceedings of the USENIX Security Symposium (Security)*, 2020.

[59] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, “Charm: Facilitating dynamic analysis of device drivers of mobile systems,” in *Proceedings of the USENIX Security Symposium (Security)*, 2018.

[60] The kernel development community, “VFIO - “virtual function I/O”,” 2021. [Online]. Available: https://www.kernel.org/doc/html/v5.14/driver-api/vfio.html

[61] The Khronos® Vulkan Working Group, “Vulkan® 1.3.270 - a specification (with all registered vulkan extensions),” Nov. 2023. [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/index.html

[62] R. Van Tonder and H. Engelbrecht, “Lowering the USB fuzzing barrier by transparent two-way emulation,” in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[63] W. R. Vasquez, S. Checkoway, and H. Shacham, “The most dangerous codec in the world: Finding and exploiting vulnerabilities in H.264 decoders,” in *Proceedings of the USENIX Security Symposium (Security)*, 2023.

[64] J. Vinck, B. Abrath, B. Coppens, A. Voulimeneas, B. D. Sutter, and S. Volckaert, “Sharing is caring: Secure and efficient shared memory support for MVEEs,” in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2022.

[65] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz, “Secure and efficient application monitoring and replication.” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[66] A. Voulimeneas, D. Song, P. Larsen, M. Franz, and S. Volckaert, “dMVX: Secure and efficient multi-variant execution in a distributed setting,” in *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2021.

[67] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, “You shall not (by)pass! practical, secure, and fast PKU-based sandboxing,” in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2022.

[68] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, “Challenges in firmware re-hosting, emulation, and analysis,” *ACM Computing Survey (CSUR)*, vol. 54, no. 1, Jan. 2021.

[69] Y. Wu, T. Zhang, C. Jung, and D. Lee, “DEVFUZZ: Automatic device model-guided device driver fuzzing,” in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2023.

[70] Y. Xu, H. Sun, J. Liu, Y. Shen, and Y. Jiang, “SATURN: Host-gadget synergistic USB driver fuzzing,” in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2024.

[71] Z. Yao, Z. Ma, Y. Liu, A. Amiri Sani, and A. Chandramowlishwaran, “Sugar: Secure GPU acceleration in web browsers,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[72] Z. Yao, S. Mirzamohammadi, A. Amiri Sani, and M. Payer, “Milkomeda: Safeguarding the mobile GPU interface using WebGL security checks,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[73] K. Yasukata, H. Tazaki, P.-L. Aublin, and K. Ishiguro, “zpoline: a system call hook mechanism based on binary rewriting,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2023.

[74] T. Yin, Z. Gao, Z. Xiao, Z. Ma, M. Zheng, and C. Zhang, “KextFuzz: Fuzzing macOS kernel extensions on Apple silicon via exploiting mitigations,” in *Proceedings of the USENIX Security Symposium (Security)*, 2023.

[75] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, “AVATAR: A framework to support dynamic security analysis of embedded systems’ firmwares,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[76] W. Zhao, K. Lu, Q. Wu, and Y. Qi, “Semantic-informed driver fuzzing without both the hardware devices and the emulators,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2022.

Appendix A
Bug Reproduction Steps

We disclose the steps to reproduce the bugs in Table VII. After minimizing the fuzzer-generated, bug-triggering programs, we used our *Guest Agent* to reproduce the bugs. For the bugs that were discovered only through snapshot restoration, we used the *Guest Agent* to execute the workload up to the specified syscall count before executing the bug-triggering program.

Table VII: Steps to reproduce the 10 bugs we found using Moneta. We verified that each bug can be triggered when there is a physical GPU instance, and all of them were confirmed by the respective vendors.

Id.	Bug Trigger

1	**Execute the workload up to 339 syscalls.**
	`r1 = fetch_preserved_fd$nvidia()`
	`ioctl$nvidia(r1, 0xc0b8464a,`
	`↪ &(0x7f0000000140)="0200d0c10100005c020000000500005c000000000000000000000000000000d0c10700005c00000000`
	`95c4030300110003000000000000009e6000d87d1eced000000000000000000000000000008000fb00004000000000000000000000000`
	`0040500000000000000000000000000000000000000000000008aea171300000000000000000000000004000000000000000000000`
	`000000000000000000cd410000000000000000000000000000000000000000000000faff00")`

2	**Execute the workload up to 9,441 syscalls.**
	`r1 = fetch_preserved_fd$nvidia()`
	`ioctl$nvidia(r1, 0xc0b8464a,`
	`↪ &(0x7f00001f0c00)="0a00d0c10300efbe0600000000000000000000000000002000000000000000000000000000000000000000`
	`004c47320000000000000000dc003000004000004fffffff23824e9e030084e2f00000000ff030000000000000000000201e0000000`
	`0000000000000000000000000000000000000000000000000000000000000400000000000000000000ffffffffffffff422700`
	`00000000000000000000000000000000000002000")`

3	**Execute the workload up to 341 syscalls.**
	`r1 = fetch_preserved_fd$nvidia()`
	`ioctl$nvidia(r1, 0xc020462a, &(0x7f00001f0c00)="0200d0c10100005c0214800000000000bd2a24fbfd7f0000030300")`

4	**Execute the workload up to 341 syscalls.**
	`r1 = fetch_preserved_fd$nvidia()`
	`ioctl$nvidia(r1, 0xc0384657, &(0x7f000001e600)="0a00d0c10300efbe2300f0ca19000000000000000000000000000000200")`
	`ioctl$nvidia(r1, 0xc020462a,`
	`↪ &(0x7f000001e640)="0a00d0c10400efbe10018020000000008097e97d6c7f00008400000000000000000")`

5	`mknodat$nvidia(0xffffffffffffff9c, &(0x7f0000069200)='/dev/nvidiactl\x00', 0x21b6, 0xc3ff)`
	`r1 = openat$nvidia(0xffffffffffffff9c, &(0x7f0000069200)='/dev/nvidiactl\x00', 0x80802)`
	`ioctl$nvidia(r1, 0xc020462b, &(0x7f0000069c40))`
	`ioctl$nvidia(r1, 0xc020462b, &(0x7f000006ae00))`
	`ioctl$nvidia(r1, 0xc020462a, &(0x7f0000070700)="0200d0c10200005c0201802000000000507cea8dfd7f0000fc0100")`

6	`r1 = openat$amdgpu(0xffffffffffffff9c, &(0x7f00000cb540)='/dev/dri/renderD128\x00', 0x0, 0x0)`
	`ioctl$amdgpu(r1, 0xc1186451, &(0x7f0000000e00)={0xffffffffffff0000, 0x80000000, 0x7})`

7	`r1 = openat$amdgpu(0xffffffffffffff9c, &(0x7f0000173000)='/dev/dri/renderD128\x00', 0x0, 0x0)`
	`ioctl$amdgpu(r1, 0xc1206440, &(0x7f0000001b80)={0x8, 0x0, 0x4, 0x9})`
	`ioctl$amdgpu(r1, 0xc0206440, &(0x7f0000000140)={0x7fffffff, 0x0, 0x4, 0x9})`

8	`r1 = openat$amdgpu(0xffffffffffffff9c, &(0x7f0000002c80)='/dev/dri/renderD128\x00', 0x0, 0x0)`
	`ioctl$amdgpu(r1, 0xc0106442, &(0x7f0000000b80)={0x1})`
	`ioctl$amdgpu(r1, 0xc0206449, &(0x7f0000000100)={0x0, 0x2000000000000, 0x9, 0x0, 0x0, 0x1})`

9	`r1 = openat$mali(0xffffffffffffff9c, &(0x7f0000098380)='/dev/mali0\x00', 0x80802)`
	`ioctl$mali(r1, 0xc0048034, &(0x7f0000098440))`
	`ioctl$mali(r1, 0x40048001, &(0x7f0000098480))`
	`ioctl$mali(r1, 0x40288028, &(0x7f00000000c0)="0809cb02a62d86010501400084d5")`

10	`r1 = openat$mali(0xffffffffffffff9c, &(0x7f0000098380)='/dev/mali0\x00', 0x80802)`
	`ioctl$mali(r1, 0xc0048034, &(0x7f0000098440))`
	`ioctl$mali(r1, 0x40048001, &(0x7f0000098480))`
	`ioctl$mali(r1, 0x40018037, &(0x7f0000000140)="99")`