MORPHUZZ: Bending (Input) Space to Fuzz Virtual Devices

Alexander Bulekov, *Boston University and Red Hat;* Bandan Das and Stefan Hajnoczi, *Red Hat;* Manuel Egele, *Boston University*

https://www.usenix.org/conference/usenixsecurity22/presentation/bulekov

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

MORPHUZZ: Bending (Input) Space to Fuzz Virtual Devices

Alexander Bulekov*† Bandan Das† Stefan Hajnoczi† Manuel Egele*
*Boston University †Red Hat
alxndr@bu.edu, bsd@redhat.com, stefanha@redhat.com, megele@bu.edu

Abstract

The security of the entire cloud ecosystem crucially depends on the isolation guarantees that hypervisors provide between guest VMs and the host system. To allow VMs to communicate with their environment, hypervisors provide a slew of virtual-devices including network interface cards and performance-optimized VIRTIO-based SCSI adapters. As these devices sit directly on the hypervisor's isolation boundary and accept potentially attacker controlled input (e.g., from a malicious cloud tenant), bugs and vulnerabilities in the devices' implementations have the potential to render the hypervisor's isolation guarantees moot. Prior works applied fuzzing to simple virtual-devices, focusing on a narrow subset of the vast input-space and the state-of-the-art virtual-device fuzzer, Nyx, requires precise, manually-written, specifications to exercise complex devices.

In this paper we present MORPHUZZ, a generic approach that leverages insights about hypervisor design combined with coverage-guided fuzzing to find bugs in virtual device implementations. Crucially MORPHUZZ does not rely on expert knowledge specific to each device. MORPHUZZ is the first approach that automatically elicits the complex I/O behaviors of the real-world virtual devices found in modern clouds. To demonstrate this capability, we implemented MORPHUZZ in QEMU and bhyve and fuzzed 33 different virtual devices (a superset of the 16 devices analyzed by prior work). Additionally, we show that MORPHUZZ is not tied to a specific CPU architecture, by fuzzing 3 additional ARM devices. MORPHUZZ matches or exceeds coverage obtained by Nyx, for 13/16 virtual devices, and identified a superset (110) of all crashes reported by Nyx (44). We reported all newly discovered bugs to the respective developers. Notably, MORPHUZZ achieves this without initial seed-inputs, or expert guidance.

1 Introduction

While the cloud unveils unique opportunities to IT businesses, it presents a host of fundamental security issues. From a technical standpoint, *virtualization* is the core technology powering cloud-infrastructure. Virtualization Hypervisors (or VMMs) multiplex the hardware resources of a physical machine (the host), between multiple Virtual Machines (VMs or guests). Cloud-ready hypervisors are complex pieces of software, tasked with isolating the software running inside a VM (i.e., a guest), from the other guests, and the hypervisor itself. Beyond the cloud, hypervisors are commonly used to sandbox applications (e.g., for malware research), and for desktop use, to run applications not supported by the host OS. Regardless the application, hypervisors are trusted with providing a layer of isolation between virtual machines and the host OS. Crucially, to provide their functionality to guests, hypervisors include a slew of implementations for *virtual devices*, and the code for these devices commonly executes at the privilege level of the hypervisor itself. Virtual devices play a critical role in ensuring that the guest is isolated, but due to the complexity of these devices, it can be difficult to safely implement their functionality in software. Unfortunately exploits compromising this layer of isolation (and specifically the virtual devices) are a tangible reality. In 2015, VENOM [14] was highly publicized as a *VM-Escape* vulnerability, which allows an attacker running within an untrusted guest to compromise the underlying hypervisor and execute code outside the security confines of the VM. VENOM is certainly not a unique example, and security researchers have identified many vulnerabilities leading to potential *VM-Escape*. Ranked by the size of bug bounties, VM-escapes are considered among the most critical classes of vulnerabilities, along with iOS, Android, and browser bugs [58]. Though VM-escape attacks can take advantage of weaknesses in other hypervisor components, such as shadow page tables, our work focuses on virtual-devices which are responsible for the vast majority of reported VM-escape vulnerabilities [37].

Software fuzz testing has proven to be a versatile technique, capable of exposing vulnerabilities in a wide range of software [3, 12, 15, 19, 23, 24, 29, 30, 44, 50, 54, 57]. As virtual devices are software components, it seems natural to apply fuzzing techniques to identify lingering vulnerabilities therein. Typically a "fuzzer" is tasked with providing randomized inputs to software through an interface such as a file, or a

command-line argument. Unfortunately, hypervisors present a series of unique challenges, which make it difficult to apply "off-the-shelf" fuzzers. For example, the virtual devices do not consume inputs as files or command-line arguments, but rather as a sequence of memory read and write operations at precise locations and pointers to memory buffers which should abide by tight semantic constraints. As a result, the cumulative input space for virtual-devices grows to gigabytes or exabytes in size. Thus, without augmentations, this input-space is intractable for an off-the-shelf fuzzer such as AFL, which is well-suited to providing small inputs to parsers.

Recognizing the importance of hypervisor security, multiple works have implemented tailored fuzz-testing for the virtual-devices provided by various hypervisors. As in the physical world, VMs interact with virtual devices through a combination of Port-mapped IO (PIO), Memory Mapped IO (MMIO), or Direct Memory Access (DMA). IOFuzz [32] only fuzzes the relatively-compact PIO address space. VDF [18] uses selective coverage-guided fuzzing, combined with seed traces recorded during normal VM usage, to fuzz devices that receive inputs through PIO and MMIO.

Recently, Hyper-Cube [43] is the first fuzzer to consider the last-remaining, and most sophisticated, widespread device interface: Direct Memory Access (DMA). Fuzzing DMA devices is critical to protecting Cloud infrastructure, since devices used on the cloud rely heavily on DMA. For example, all four PCI devices (for disk, network, VM memory "ballooning", and random-number-generation) connected to a standard Google Compute Engine virtual-machine rely on DMA. To elicit DMA behavior, Hyper-Cube first writes random data to a small scratch-buffer in memory. Then, by writing the address of the scratch-buffer to the enumerated PIO and MMIO regions, Hyper-Cube tries to trigger DMA activity. Unfortunately, this approach is capable of triggering only the simplest types of DMA transactions.

Recognizing this limitation, the state of the art virtual device fuzzer, Nyx [42], a follow-up to Hyper-Cube, augments the unguided Hyper-Cube with coverage information collected via Intel-PT. However, Nyx' coverage-guided nature alone does not resolve the issues encountered when fuzzing complex virtual-devices. Thus, Nyx proposes a framework for developing *precise user-provided specifications* for fuzzing complex devices. While this allows Nyx to boost the coverage for virtual-devices such as USB controllers, and VIRTIO block-devices, it essentially turns core-aspects of Nyx into a grammar-based fuzzer with all its advantages and disadvantages. That is, provided a grammar, the fuzzer can elicit "deep" behavior of virtual devices. However, expert knowledge is required to create the grammar in a laborious and error-prone process. Furthermore, the fuzzer loses the ability to identify bugs that exist outside the confines of the grammar. Even worse, the grammar is developed based on abstract specifications. Yet, the actual device code has ample leeway to implement said specification. As such, disconnects between distilled grammars and actual implementations are highly likely, potentially placing dangerous bugs outside of the fuzzer's reach. Considering the fact that the specifications require a custom mutation engine, and took days of effort to create for individual devices, this approach faces scalability issues. At the time of this paper's acceptance, Hyper-Cube/Nyx source code had not been released, however, in our own analysis, we found that DMA activity is not conducive to heuristic-based fuzzers, such as Hyper-Cube, that rely on a single scratch buffer (see Section 3.2). Furthermore, we found new bugs in all the devices for which Nyx had specifications, further illustrating the difficulty of writing accurate and complete specifications.

We identified the core challenge faced by virtual-device fuzzers is the fact that virtual devices rely heavily on *semantic dependencies*, that are not conducive to off-the-shelf guided fuzzers. These dependencies exist between data written to PIO/MMIO addresses and the location of DMA buffers, but more importantly within the structure of DMA buffers themselves (e.g., buffers can contain pointers to further buffers, queues, rings, etc.). To satisfy the semantic constraints required by virtual-device implementations we leverage insights from hypervisor design. Specifically, we observe that modern virtualization strategies require hypervisors to implement functionality to mediate PIO, MMIO, and DMA activity. Combined with the fact that the hypervisor must know about the input-space(s) for all its virtual-devices, we devise a methodology called MORPHUZZ that leverages this information to fulfill any semantic constraint arising from device I/O *on-demand*. In addition to transparently fulfilling semantic constraints, this on-demand approach allows MORPHUZZ to leverage battle-tested off-the-shelf fuzzers (e.g., libFuzzer) to reach "deep" virtual-device functionality. Importantly, MORPHUZZ obviates the need for specifications and seed traces. Based on the insights provided by the hypervisor, MORPHUZZ *dynamically reshapes* (i.e., bends) the input-space to conform to the strict boundaries of the currently-accessible PIO, MMIO and, most-importantly, DMA regions.

The input-space for virtual-devices is vast, encompassing all the VM's memory and port addresses. However, only a small subset of these addresses is engaged in device I/O, at any time. Furthermore, through technologies such as DMA, and PCI, device I/O can happen at arbitrary, dynamically changing, addresses. Hence, a random fuzzer would waste most of its time interacting with addresses that are unrelated to device I/O. By bending the input-space, the I/O activity generated by MORPHUZZ, precisely, and exclusively interacts with the PIO/MMIO and DMA regions engaged in device I/O. Though, MORPHUZZ only fuzzes small areas of the entire input-space at any time, these areas are representative of the addresses engaged in I/O at any moment. With dynamic reshaping, MORPHUZZ can interact with *any address*, as long as the address is related in to device I/O. Additionally, MORPHUZZ does not rely on architecture-specific knowledge to

operate. Observing the growing influence of non-x86 architectures, we applied MORPHUZZ to fuzz ARM virtual-devices, with no changes to the MORPHUZZ code.

Finally, in a head-to-head evaluation, MORPHUZZ outperformed Nyx' manually written specifications for xHCI and VIRTIO, and found all 44 bugs that Nyx reported in these devices. Furthermore, MORPHUZZ identified an additional 66 bugs. Beyond the number of bugs, MORPHUZZ also consistently outperforms prior work, with respect to coverage. MORPHUZZ produces DMA behaviors that are more complex than anything that can probabilistically be triggered by Hyper-Cube. Moreover MORPHUZZ achieves this without the manual effort required to fuzz complex devices with Nyx. In summary, this paper makes the following contributions:

- We describe MORPHUZZ – our generic method that leverages inherent characteristics of hypervisor designs to *reshape* the virtual-device input space, making it amenable to fuzzing without seeds or specifications (§4). Our methodology enables targeted, coverage-guided, fuzzing of virtual devices in production hypervisors.
- We implement MORPHUZZ for the popular open-source QEMU hypervisor (§4). Our implementation, QMORPHUZZ produces self-contained and deterministically reproducible results (i.e., crashing inputs). In addition to typical memory-corruption bugs, MORPHUZZ identifies bugs that are characteristic of DMA virtual-devices, such as data-races and double-fetches. For bugs that are not double-fetches MORPHUZZ automatically generates reproducers for recreating issues in an unmodified build of QEMU. To demonstrate the ease of applying MORPHUZZ to other hypervisors, we also ported MORPHUZZ to bhyve.
- As MORPHUZZ fuzzes virtual-device implementations, we evaluate MORPHUZZ on 28 virtual-devices (§5) in QEMU, a strict superset of the 15 devices analyzed in prior work. Our experiments demonstrate that MORPHUZZ effectively identifies previously known, as well as, new bugs in a wide range of virtual device implementations.
- By working directly with the QEMU developer community, we reported 61 bugs identified by QMORPHUZZ, 22 of which have already been fixed, by their corresponding maintainers. Nine of these issues have been assigned CVE IDs, and published. We also responsibly disclosed the issues found by MORPHUZZ in bhyve.
- A fully functional implementation of virtual-device fuzzing, based on QMORPHUZZ, is already contained in the QEMU repository, where it is used to continuously fuzz virtual-device implementations via Google's OSS-Fuzz [33].

2 Background

In this section, as background for our work, we describe relevant aspects of hypervisors, virtual devices, and fuzzers.

Computers normally communicate with the outside world through peripheral devices (e.g. hard disk, network and USB controllers). Since the operating systems running in VMs expect to communicate through peripherals too, hypervisors implement *virtual-devices* in software that create the illusion of peripherals for the guest. However, physical hardware is often designed around a different set of constraints than software. Thus, software virtual devices that mimic physical hardware, often achieve sub-optimal performance. As VMs became ubiquitous and VM performance inefficiencies resulted in monetary costs, hypervisor developers designed and implemented paravirtual devices, such as VIRTIO-type devices. These devices are not based on any physical device. Instead, they are designed and optimized specifically for virtualization.

2.1 Interfacing with (virtual) devices

Common computer architectures provide some, or all of the following interfaces for the CPU to interact with devices.

Port-mapped IO (PIO) is an x86-specific interface that relies on special CPU instructions (`in/out` on x86) to pass data from registers or main memory to a separate address-space dedicated to devices. PIO is constrained to a limited address space (64k total addresses) and the small size (1-4 bytes) of each transfer. PIO often serves as a primary interface for low-bandwidth communication with devices, such as timers, interrupt controllers, or serial ports.

Memory-Mapped IO (MMIO) sets aside regions of physical main memory for device I/O. Unlike regular `read/write` access to physical memory, operations to memory-mapped regions are forwarded by the memory-controller to the peripheral. Similar to PIO, transferring data to and from the device via MMIO is a blocking/synchronous operation for the CPU.

Direct Memory Access (DMA) provides peripherals with rapid, direct access to physical memory, bypassing the CPU. When the CPU wishes to signal to a device that some data is available for DMA transfer, it can simply communicate the location of the data in physical memory to the device using PIO or MMIO. The device can then, independently, access the data in memory. Notably, the CPU is not involved in the actual transfer of the data. From the perspective of the CPU, it simply signaled a pointer to some data, by writing to a PIO/MMIO register. As we will see in Section 3.2, a single PIO/MMIO command can result in many levels of DMA-accesses, since the DMA data can itself contain the locations of additional data. The format and protocol of DMA buffers is prescribed by the implementation of the virtual-device. Due to its performance benefits, DMA is used for high-throughput devices such as storage controllers, network, and graphics adapters. While DMA-capabilities require additional hardware/costs for physical devices (i.e., DMA controllers), paravirtual-devices are not constrained by hardware costs, and

rely on DMA extensively for "free".

The pointers and data-structures involved in DMA transfers, are by no means the sole semantic dependency that occur in device-usage. For example, PCI-based devices are unavailable, until the CPU interacts with the PCI controller to configure each device's *base-address-registers (BARs)*, which specify the location of each MMIO/PIO region associated with the device. Thus, the MMIO/PIO regions for PCI devices are semantically tied to prior interactions with the PCI controller. Prior systems rely on heuristics, or expert-knowledge to address these semantic connections, and typically prevent the fuzzer from tampering with the PCI state, after it has been initialized. On the other hand, MORPHUZZ can transparently fuzz the PCI controller, and PCI devices.

2.1.1 Emulating Virtual Devices

In a VM, when the guest attempts to access a device using PIO, or MMIO, the hypervisor must stop the virtual CPU to handle the access within a virtual-device. Hypervisors implement a mechanism to "trap" on PIO and MMIO accesses. For example, by unmapping the memory-pages that correspond to the guest's physical MMIO regions, the hypervisor ensures that any IO activity within the guest results in an exception. Similarly, PIO relies on privileged instructions which raise exceptions, when the guest invokes them, providing the hypervisor with control over execution. The hypervisor handles the exception, calling into the virtual-device code corresponding to the PIO/MMIO access. After the virtual-device handles the request, the hypervisor updates the register/memory state of the guest, and resumes the virtual CPU.

2.2 Virtual Device Vulnerabilities

Virtual-device code has been found to contain bugs such as buffer-overflows, heap-overflows, stack-overflows, use-after-frees, use of uninitialized memory, infinite loops, and premature terminations [7–11, 38, 39]. Additionally, we highlight two types of issues that are characteristic of virtual-devices: reentrancy vulnerabilities and double-fetch bugs. Virtual DMA can introduce guest-exploitable reentrancy and data-race issues. These bugs arise from insufficient precautions when handling IO commands that originate from the guest. For example, DMA data often contains sensitive information, such as buffer-lengths, or element-count. Virtual-devices must take care when handling this metadata, since a malicious guest can modify the metadata *while a virtual-device is processing a request* to exploit a *double-fetch bug*. In other cases, if a device negligently performs a DMA access, it can, inadvertently touch a region of memory that is mapped to its own MMIO space, which may not be designed to handle multiple concurrent accesses, resulting in a guest-triggerable *reentrancy bug* in the implementation of the virtual-device.

2.3 Testing Virtual Devices

Hypervisor developers have recognized the importance of testing virtual-devices. For example, QEMU's testing framework, *qtest*, provides an interface for directly performing memory/IO through an API, rather than through assembled CPU instructions. This allows *qtest* to run test-suites against virtual device implementations without relying on a purpose-built testing guest-OS. Usually a guest interacts with devices using CPU instructions such as `inb %Port_Number`, or `movl %MMIO_Address,value`, which trap into the hypervisor code and are handled by virtual-device code. With *qtest*, the developer specifies I/O requests using a simple ASCII protocol. *qtest* interprets the ASCII instructions using the same APIs that are used to route vCPU I/O accesses that trapped into the hypervisor. For example, the trace in Figure 1 can be directly provided to *qtest* to reproduce a crash. Some hypervisors, such as bhyve, do not provide testing facilities. However, we were able to trivially implement the subset of *qtest* operations helpful for fuzzing bhyve in 90 lines of code.

3 Motivation

In this section, we outline the challenges facing a virtual-device fuzzer. First, we describe the randomized inputs provided to a virtual-device fuzzer by a fuzzing engine. Next, we describe an output of the virtual-device fuzzer - an I/O sequence used to crash a hypervisor. Finally, we provide a summary of the challenges faced by a fuzzer aiming to convert randomized inputs into I/O sequences that crash hypervisors.

3.1 Fuzzer Inputs

Fuzzer engines provide randomized bytes that can be fed as inputs to a program. Most modern fuzzers typically generate discrete inputs. This makes it simple to store inputs that resulted in interesting behaviors (e.g. crashes). One benefit of separate inputs is that a fuzzer can operate over external inputs provided by the developer. For example, a developer fuzzing an image library, can provide the fuzzer with "seed" inputs, which the fuzzer can use as the basis for future mutations - increasing the chance of triggering interesting code over a purely-randomized approach. Coverage guided fuzzers further augment the mutation process by gathering coverage feedback from the target program, used to guide future mutation of inputs.

Complex interfaces such as code compilers and operating-systems require highly structured inputs, that off-the-shelf fuzzers are unlikely to produce. For such targets, even a rich corpus of valid inputs might not be enough for effective fuzzing, since small mutations often compromise the structural validity of the input, producing inputs that are unlikely to reach interesting code-paths within the target. To fuzz such complex targets, developers usually implement additional layers of machinery between the fuzzer and the actual target. For example, the fuzzer's random bytes can be

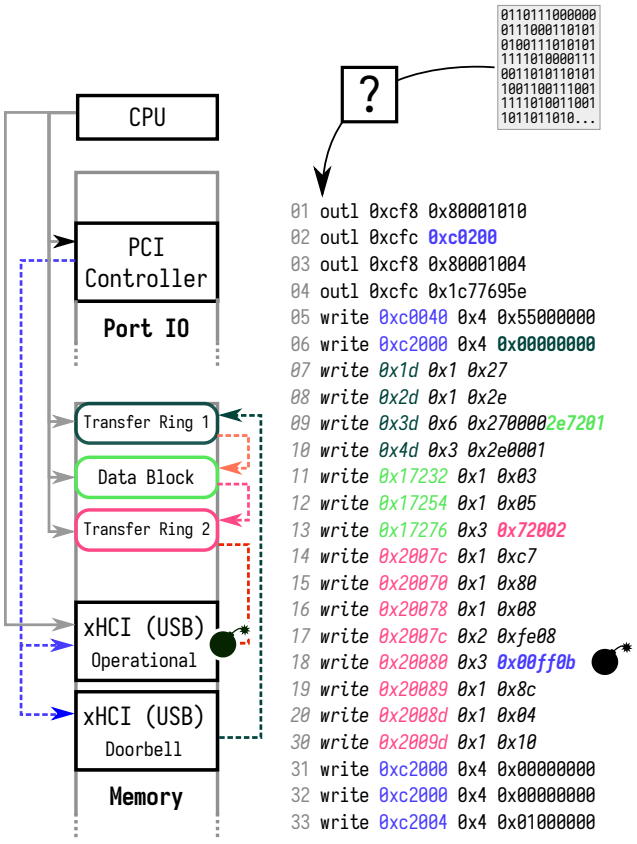

Figure 1: On the right, a crashing input found by MORPHUZZ for the xHCI (USB) controller. In color, we highlight addresses that depend semantically on previously written values (in bold). Note that the addresses may have swapped-endianness. In italics, we highlight the lines that serve to populate DMA regions in memory. In the diagram, the regions with rounded edges represent DMA buffers that could be located *anywhere* in memory. The solid lines represent the Port/Memory accesses performed by the virtual CPU. The dashed lines represent implicit semantic dependencies.

used to seed a program generator, the output of which is, in turn, used to fuzz a compiler. Commonly, fuzzer developers rely on fuzzing-grammars and structure-aware fuzzers, which covert random bytes, into a structured input based on manually-created grammars or descriptions. Such grammars are used by the renowned syzkaller OS-fuzzer.

3.2 Crashing Virtual Devices

As we are interested in finding potential VM-escape vulnerabilities in virtual-device code, the fuzzer must be capable of interacting with virtual-devices over the major device I/O interfaces (PIO/MMIO and DMA). To illustrate the complexity of producing complex virtual-device interactions, we describe a reentrancy issue found by MORPHUZZ in QEMU's xHCI(USB) controller. Figure 1 features the crash reproducer generated by QMORPHUZZ, along with a diagram. Instead of attempting to summarize the 645-page xHCI specification, we highlight the *five layers* of semantic-dependencies that the fuzzer addressed to produce a crashing input.

The xHCI controller is a PCI-based device. As such, the first I/O interactions (lines 01-04) use PIO `outl` instructions to communicate with the PCI controller (on ports 0xCF8 and 0xCFC) and configure the xHCI's first BAR to 0xC0200, which is automatically page-aligned to 0xC0000 (line 03). Once the BAR is configured and enabled, the xHCI device accepts MMIO requests at the specified address. In particular, the crash requires interacting (lines 05-06, 31-33) with the xHCI's operational range (based at 0xC0040) and the doorbell range (Based at 0xC2000). Note that only five lines interact with the xHCI controller's MMIO range. The vast majority of the operations (lines 07-30) configure data in three distinct DMA buffers (starting at 0x1D, 0x17232, and 0x2007C). These three buffers correspond to segments of a xHCI Transfer Ring (containing pointers to DMA buffers describing transfers), and an Input Data Block, all of which are represented in main memory, and accessed via DMA.

As mentioned in Section 2.1, CPUs can provide the address of DMA buffers by writing pointers over PIO/MMIO. In our example, the CPU only communicates *one* DMA location to the xHCI device over MMIO (the location of Transfer Ring 1 sent at line 06). Subsequently, the location of the Data Block is derived from Transfer Ring 1 (line 09). The location of Transfer Ring 2 is derived from the Data Block (line 13). There are, in-fact, *four* valid DMA addresses required to reproduce the crash. The address of the final buffer (a pointer to an Output Data Block) is derived from Transfer Ring 2. Instead of pointing to some free space in memory, though, the address of the output Data Block (written on line 18) points to a location (0xBFF00) that is near the xHCI's "Operational" MMIO Region. Thus, when the xHCI device attempts to write 3,584 bytes to the output Data Block, it *inadvertently writes to its own MMIO registers*. By writing to its own MMIO region, the device initiates a new DMA transfer, while the original request is still in progress. As the implementation of the xHCI controller is not reentrant, this nesting of xHCI activity frees resources that are still referenced by the original request. Thus, when the nested MMIO access is complete, the device triggers a use-after-free.

Note that in addition to the appearance of all major modes of device I/O (PIO, MMIO and DMA), there are *five* layers to the implicit semantic dependencies involved in triggering the crash in Figure 1 (follow the dashed arrows). For a grammar-based fuzzer to find such a crash, its grammar must accurately reason about each of the five levels of xHCI semantics. Each inaccurate assumption, or heuristic, greatly reduces the fuzzer's ability to reach the next layer of interaction.

3.3 Summary of Challenges

In summary, fuzzers face several challenges in finding bugs such as the one detailed in Fig. 1:

1. The virtual device fuzzer must be capable of producing inputs that operate across all the modes of device I/O.

That is, a fuzzer attached to an off-the-shelf fuzzing engine, must ingest randomized bytes and convert them into virtual-device IO.

2. As the input space accessible through each mode of I/O is enormous, the fuzzer must identify and target only the I/O regions that are associated with virtual-devices.

3. The fuzzer must be aware of the implicit semantic dependencies that occur in device I/O.

One straightforward solution to address these challenges is to create a grammar specification for each device - essentially creating specialized fuzzers for each device. However, as virtual-devices specifications are often complex (and sometimes inaccurate), creating quality specifications requires significant time investment and expert domain-knowledge.

4 MORPHUZZ

MORPHUZZ closes the existing gap that inhibits fuzzers from exercising virtual-devices that implement complex I/O protocols, without days of manual specification effort, for each device. The two core tenets of our design are the *reshaping of input space* to account for dynamic changes in I/O memory layout and *optimizing for arbitrary DMA activity*. To this end, MORPHUZZ leverages fundamental characteristics of VMMs to precisely and exclusively fuzz memory regions associated with virtual-devices. First, we describe how MORPHUZZ works with a fuzzing mutation backend to produce sequences of IO operations. Then we explain how MORPHUZZ isolates PIO and MMIO regions, to guarantee that each fuzzer-produced PIO/MMIO interaction triggers virtual-device code (§4.2). Then, we cover MORPHUZZ's strategy for leveraging the hypervisor's DMA APIs to transparently produce inputs that satisfy complex semantic dependencies (§4.3). Combining these capabilities yields a generic fuzzing approach (MORPHUZZ) that reshapes the virtual-device input-space and successfully fuzzes virtual device implementations. Importantly, the generic nature of MORPHUZZ ensures that no prior knowledge in the form of I/O protocols (such as PCI enumeration), seed inputs, or explicit specifications is required. If MORPHUZZ identifies a crash in a virtual-device, the final step "unbends" the reshaped input-space into a standalone reproducer (§4.4.3). This capability is particularly useful to communicate identified crashes to virtual-device developers who can use the reproducer to deterministically trigger the bug in a regular build of the hypervisor. Figure 2 presents a conceptual overview of our MORPHUZZ design.

We describe QMORPHUZZ, our implementation of virtual-device fuzzing for QEMU. We explain our automated system for reproducing crashes found by QMORPHUZZ in an unmodified version of QEMU (§4.4.3). To emphasize the applicability of MORPHUZZ's techniques to other hypervisors, we describe the straightforward steps necessary to adapt MORPHUZZ for this task. We also provide an account of how we followed these steps to rapidly deploy fuzzing with MORPHUZZ for bhyve (§4.5.1).

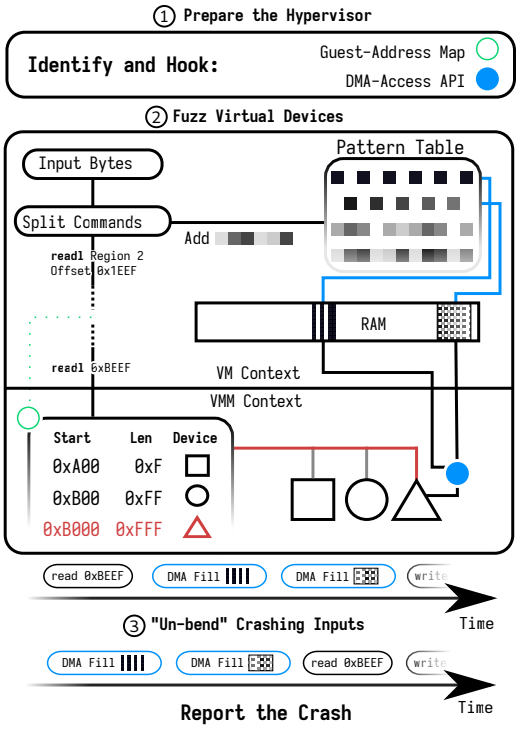

Figure 2: ① We identify and provide MORPHUZZ with insight into the DMA-access API and Guest-Address Map, reshaping the input-space. ② MORPHUZZ fuzzes virtual-devices over PIO, MMIO and DMA, in a precise, and targeted manner. ③ We "unbend" the crashes found by MORPHUZZ to report bugs.

4.1 The Interpreter

As we mentioned in Section 3, fuzzers such as AFL and libFuzzer provide inputs in the form of buffers. While such inputs are well-suited for fuzzing e.g., image-libraries with interfaces that operate on buffers, virtual-devices have no such interface. Instead, virtual-device interactions involve many individual PIO/MMIO and DMA operations.

To address this, MORPHUZZ follows canonical guidelines for splitting fuzzer inputs [22] to implement an interpreter. MORPHUZZ's interpreter relies on a simple opcode language. Individual *operations* in the input are divided by a 4-byte "separator"[1]. For MORPHUZZ to perform two operations, the fuzzing backend must provide a byte-buffer that contains one "separator". As an added benefit, the separator ensures that small changes, such as bit-flips within an instruction, do not result in a cardinally different set of I/O operations. That is, similar inputs are likely to result in similar behaviors.

The first byte of each operation acts as the "opcode". The interpreter examines the byte, and executes the corresponding

[1]Modern fuzzing backends, such as libFuzzer, automatically identify the separator and add it to their "dictionary" of byte patterns to be used when generating new inputs.

opcode handler. MORPHUZZ's operations can be of variable-length. If the operation contains too few bytes to encode all of the parameters required by the opcode, MORPHUZZ discards the corresponding operation and moves on to the next decoded operation in the input. Conversely, if an operation contains too many bytes, MORPHUZZ reads only the number of operands specified by the opcode's implementation and discards the excess bytes.

The interpreter ingests all of the fuzzer inputs. However, it is the actual types of operations that enable MORPHUZZ to effectively fuzz virtual-devices. Throughout the rest of this section, we will introduce these operations.

4.2 Distilling the MMIO and PIO Input Space

As mentioned in Section 2.1, the CPU can communicate directly with virtual-devices by reading/writing to Memory-mapped and Port-mapped addresses. For simple devices, such as keyboards, and timers, the locations of the ports are usually standardized across the architecture. However, complex PCI devices provide the guest OS with flexibility to configure the locations of PIO and MMIO regions. For example in Figure 1, the first four instructions (with ports `0xCF8` and `0xCFC`) are programming the PCI controller, to specify PIO and MMIO locations and enable the xHCI device.

At system-startup, PCI-device MMIO and PIO regions are not accessible. Instead, low-level kernel drivers are responsible for *dynamically*, configuring and enabling PCI-devices. To effectively fuzz virtual-devices, MORPHUZZ must have an accurate view of the locations of active PIO and MMIO regions at all times, to avoid interacting with PIO and Memory Regions that are not mapped to devices. Fortunately, there is a fundamental solution to this problem: hypervisors must *themselves* keep track of active PIO and MMIO regions, so they can trap-and-emulate upon accesses to those regions and call the proper virtual-device handlers. As such, hypervisors keep a Guest-Address Map - a mapping of guest physical addresses to the underlying virtual hardware. When I/O regions are created and removed, the mapping is updated. When a guest I/O access raises an exception, the hypervisor compares the exception details against the Guest-Address Map to route the I/O request to the corresponding virtual-device handler.

MORPHUZZ monitors changes to the Guest-Address Map, tracking the IO regions associated with virtual-devices. Leveraging these hooks, MORPHUZZ provides opcodes that allow the fuzzer input to interact with devices over PIO and MMIO:

in[b,w,l](addr) perform a PIO read
out[b,w,l](addr, value) perform a PIO write
read[b,w,l,q](addr) perform an MMIO read
write[b,w,l,q](addr, value) perform an MMIO write

For these operations, the **b**`yte`,**w**`ord`,**l**`ong`,**q**`uad` suffixes specify the size of the access. Devices, such as the VIRTIO family handle I/O differently, depending on the size of the access, making this distinction important.

MORPHUZZ consults the Guest-Address Map every time the fuzz-input initiates an MMIO/PIO access. Instead of requiring the mutation engine to produce inputs containing actual MMIO/PIO addresses, MORPHUZZ's `addr` parameters are [index, offset] tuples. MORPHUZZ simply uses these tuples to select a region from the PIO and the MMIO regions listed in the Guest-Address Map, and to choose an offset within the region. By bending the input space, MORPHUZZ ensures that this address falls within an active I/O region listed within the Guest-Address Map. As such MORPHUZZ guarantees that each fuzzer PIO and MMIO access results in a hypervisor trap, with a subsequent call to a software virtual-device handler. Since, immediately after boot, PCI devices provide no PIO/MMIO regions the initial Guest-Address Map contains only a few entries (including the PCI controller registers). Once MORPHUZZ discovers a fuzz-input that configures BARs, the hypervisor updates the Guest-Address Map. As MORPHUZZ keeps track of the Guest-Address Map, subsequent I/O actions produced by the same fuzz-input can interact with the newly-mapped PIO/MMIO regions. Essentially MORPHUZZ's inputs quickly learn to configure and enable virtual-devices, automatically.

4.3 On-demand Fuzzing of DMA Accesses

At first glance, the DMA input-space is intractable for a fuzzer. DMA buffers can reside *anywhere* in guest memory, and the VM *does not trap*, when the CPU accesses them. As such, hypervisors do not need to keep track of DMA buffers, unlike for PIO/MMIO regions. Instead, virtual devices interpret values written over MMIO and PIO as DMA buffer addresses[2] and use them to index into the guest's physical memory. This massive input-space is a clear roadblock for virtual-device fuzzing. As a result, prior approaches have, ignored DMA entirely [18], relied on coarse heuristics of DMA accesses [43], or required precise specifications for each device [42]. Recognizing the flexibility, and corresponding complexity introduced by DMA, MORPHUZZ avoids guessing the proper format and location of DMA buffers, altogether. Instead, MORPHUZZ produces precise DMA-based interactions, by leveraging intrinsic behaviors of hypervisors, as follows.

[2]Furthermore, as we will see in Section 5.2.3 devices can transform the values received over MMIO or PIO before accessing the actual guest-physical memory (e.g., interpreting the value as a page-frame-index rather than treating it as a physical address proper).

When the vCPU provides a virtual device with a pointer to a DMA buffer over MMIO, or PIO, the hypervisor traps into the virtual-device code. As the implementation of the virtual device executes in the context of the hypervisor (i.e., in *host-virtual memory*), the device code cannot simply dereference a pointer into *guest-physical memory*. Instead, the pointer must first be "translated" into the hypervisor's address space. To provide this "translation" functionality to virtual-devices, hypervisors expose a *DMA-access API*, which cross-references the address of the DMA access against the hypervisor's internal representation of the guest's memory.

Hence, the DMA-access API provides a lightweight and clear mechanism for fuzzing the data communicated over DMA channels. MORPHUZZ's interpreter implements two opcodes to fuzz DMA access. However, reflecting the device-initiated nature of DMA, these opcodes are simply used to prime MORPHUZZ with fuzzer-provided data in anticipation of future DMA accesses. These opcodes maintain a ring of patterns in MORPHUZZ's internal memory:

add_dma_pattern(offset, stride, pattern[]) Add a pattern to MORPHUZZ's DMA pattern-ring.
clear_dma_patterns() Clear the pattern-ring.

To put these patterns to use, MORPHUZZ hooks all calls into the DMA-access API and populates the guest-memory corresponding to DMA regions with a repeating pattern from the pattern-ring, on-demand, for the device to read. Internally, each time MORPHUZZ hooks a DMA access and fills the corresponding memory with a pattern, it advances the pattern-ring pointer, so that the next DMA access is filled with a different DMA pattern (if one is available). Patterns can be of arbitrary length to represent data of any complexity. However, MORPHUZZ's add_dma_pattern operation also enables even short patterns to encode interesting structures using the "offset" and "stride" parameters. When these parameters are specified, each time MORPHUZZ repeats the pattern, it increments the "offset-th" byte by "stride". This allows even short patterns to represent complex DMA data-structures, such as pointer-rings with many unique addresses[3].

Note that, as explained in Section 3.2, virtual-device implementations frequently impose stringent semantic constraints on the contents of DMA buffers (e.g., buffers must contain pointers to other *valid* buffers). As MORPHUZZ responds to DMA accesses on-demand, such nested accesses are transparently handled by the exact same mechanism. This on-demand operation is crucial and allows MORPHUZZ to reach the "deep" and nested code paths in virtual device implementations. Section 5 provides quantitative evidence for this claim.

Through its awareness of the fundamental Guest-Address Map and DMA-access API, MORPHUZZ achieves a uniquely-precise view of active PIO, MMIO and DMA regions.

[3] E.g. add_dma_pattern(1, 2, `10 01 00 00`) expands out to `10`**`01`**`0000 10`**`03`**`0000 10`**`05` ...

4.4 Fuzzing QEMU

We focused on implementing MORPHUZZ for QEMU because QEMU is a popular open-source hypervisor, commonly used for cloud-applications. Due to its large number of virtual-devices, and their complexity, QEMU has been the primary benchmark used for evaluating prior works [18, 42, 43]. QEMU features a large selection of devices, ranging from legacy PC components, to models of complex network and storage controllers, and paravirtual VIRTIO interfaces.

MORPHUZZ' design calls for tapping into the Guest-Address Map and DMA-access API that are implemented in hypervisors. Locating the implementation of these APIs within QEMU is a simple task: QEMU provides developer-documentation about the DMA-access API, and implements an `mtree` command that prints information about all currently-configured regions in the Guest-Address Map.

4.4.1 Initializing QEMU

QMORPHUZZ is implemented as a build of QEMU augmented with additional code for interpreting fuzzer inputs, and a thin layer of hooks tapping into QEMU's Guest-Address Map and DMA-access API. For QMORPHUZZ we use libFuzzer as our fuzzing-backend. The interface between QMORPHUZZ and libFuzzer consists of an `init()` and an `exec(input)` function. This minimal `init/exec` interface is also supported by other fuzzers, such as AFL[++], and hongfuzz, making it straightforward to swap the fuzzing-engine. `init()` is executed once, prior to fuzzing, and `exec()` invokes the opcode interpreter used to translate individual fuzzing inputs into sequences of I/O commands.

To initialize QEMU, QMORPHUZZ simply passes a user-provided set of command-line arguments to QEMU's `main` function. This is necessary as the virtual hardware available to VM's in QEMU is configured through command-line options. For example, adding `-nic tap,model=ne2k_pci` to the command-line arguments, configures the VM to have a NE2000 network-interface card connected to the PCI bus, relying on a TAP-based network-backend. As QEMU initializes the VM, it registers I/O regions (such as the PCI controller ports) using its Guest-Address Map. QMORPHUZZ hooks registration and de-registration of I/O regions in the Guest-Address Map and keeps a record of them in an *I/O region vector*. QMORPHUZZ also configures QEMU with the *qtest* CPU backend. Instead of booting binary CPU code, *qtest* exposes a small text-based API that directly accesses memory and PIO (see Section 2.3 for details). Once the VM is initialized and awaiting *qtest* commands, QMORPHUZZ passes control to libFuzzer, which begins its fuzzing loop.

4.4.2 Executing Inputs

For each input, libFuzzer calls the `exec(input)` callback, implemented by QMORPHUZZ. In its default configuration, QMORPHUZZ forks the process, to ensure that each input exe-

cution is independent of all other executions. In Section 4.4.4, we describe alternatives to forking that we experimented with.

To produce complex I/O behavior, MORPHUZZ feeds each input into its interpreter (described in § 4.1).

When MORPHUZZ's interpreter parses a PIO/MMIO operation, it issues the corresponding *qtest* instruction. In the case of the DMA-related instructions, there is no need to perform any immediate I/O operation; instead, QMORPHUZZ simply updates the internal ring of DMA-patterns.

As virtual-devices handle PIO/MMIO requests, they may invoke the DMA-access API to copy data to and from the VM's main memory, over DMA. QMORPHUZZ hooks to each of these API functions. When the virtual-device code invokes the DMA API to read data from guest-physical memory, QMORPHUZZ examines the location and size of the access, and fills the corresponding guest-physical memory with a pattern previously added to the DMA pattern-ring. Filling the memory is as simple as invoking the `qtest_memwrite` API.

Once QMORPHUZZ finishes executing the commands specified by the fuzzer's input, it destroys the process. The parent process waits on the child to exit, and returns control to libFuzzer. libFuzzer examines the coverage data collected during the child's execution and determines whether the input triggered new behaviors and should be added to the input corpus. Finally, libFuzzer's mutation engine generates a new input, calls `exec()`, and hence continues the fuzzing cycle.

4.4.3 "Unbending" Inputs

Reproducing crashes with MORPHUZZ is trivially possible by simply providing the same input to the opcode interpreter that led to the crash in the first place. However, virtual-device developers should not be forced to acquaint themselves with the fuzzer just to be able to reproduce a bug. That is, a self-contained and stand-alone representation of a crashing input that is readily usable with a regular build of QEMU is strictly preferable. Since *qtest* is widely used by QEMU developers, MORPHUZZ "unbends" each crash into a standalone *qtest* reproducer. MORPHUZZ replays the crashing input through the opcode interpreter and logs the resulting linear sequence of MMIO/PIO and DMA-related device I/O commands, in the order they were issued. Since real VMs do not populate DMA buffers on-demand, MORPHUZZ annotates all I/O commands used to fulfill DMA accesses in the log, with a prefix. Then, MORPHUZZ simply re-arranges the logged I/O commands so that each command filling a DMA request precedes the direct PIO/MMIO command that triggered it. This process is illustrated at the bottom of Figure 2.

The result is a linear *qtest* API trace, which can be piped into a standard QEMU process to reproduce the crash. The *qtest* trace can be sent to virtual-device developers along with the command-line used to specify the connected virtual-devices, as a simple, self-contained, and straightforward way to reproduce crashes. Additionally, MORPHUZZ's automatically-generated reproducers have been used as regression-tests, alongside manually-written test-cases.

We "unbend" all crashes, except double-fetches which are difficult to consistently reproduce without instrumentation. Due to their time-sensitivity, there is no straightforward way to automatically and reliably reproduce double-fetches in an unmodified hypervisor. However, the developer can still *reliably reproduce* crashes due to double-fetches with a build of MORPHUZZ and the crashing input.

4.4.4 Resetting State

Hypervisors are large, stateful, applications. For fuzzing such applications, it is important to roll-back the state-changes caused by a fuzz-input, prior to the next execution. Otherwise, the same input can lead to different program-behaviors (nondeterminism). MORPHUZZ is not tied to any particular method for resetting state. In practice, QMORPHUZZ supports resetting the hypervisor in between inputs by either rebooting the VM, VM-Snapshotting, or using process-level forking. While rebooting VMs and Snapshotting are often more performant than forking a large process, they require a correct implementation of those features to guarantee fuzzer stability. While many vital devices in QEMU have precise rebooting and snapshotting handlers, others have lacking implementations. Relying on incomplete snapshotting implementations exposes the fuzzer to the state leakage between individual inputs and, therefore, nondeterminism. As such, throughout our evaluation, we use MORPHUZZ in a process-level forking mode, to ensure consistent results. Note that MORPHUZZ's approach is compatible with emerging systems for full-system snapshot-based fuzzing, such as Agamotto, and Nyx [42, 48].

4.4.5 Implementation Complexity

In total, QMORPHUZZ consists of 714 lines of C/C++ code. Of these, 68 lines are changes to existing QEMU code. The rest implement QMORPHUZZ's `init` and `exec` functions. QMORPHUZZ is already integrated into QEMU's code-base and available upstream where it continuously fuzzes QEMU's virtual devices on Google's OSS-Fuzz.

4.5 Beyond QEMU

We described in Sections 4.2 and 4.3, that trap-and-emulate hypervisors implement a DMA-access API and Guest-Address Map. Interactions with guest memory are tightly coupled to the particular VM's configuration. As such, individual virtual-devices are obligated to use centralized APIs for interacting with the guest's address-spaces. Since calls to these APIs pervade virtual-devices, we were able to easily identify them for popular hypervisors. Table 2 in the Appendix lists the DMA-access APIs for well-known hypervisors. MORPHUZZ's hooks are non-invasive, and can be implemented using an external debugger.

4.5.1 Implementing MORPHUZZ for bhyve

Bhyve is a hypervisor shipped as part of FreeBSD. We ported MORPHUZZ to bhyve. In total, it took a researcher with no prior bhyve experience 4 hours to implement fuzzing with MORPHUZZ. Our implementation reuses the core fuzzing code from QMORPHUZZ. To tap into bhyve's Guest-Address Map and DMA-access API, we added a total of four lines of C code. Bhyve does not provide a testing framework so, we implemented an API mimicking QEMU's *qtest*, in 90 lines of C code. This API simply calls into the same functions that bhyve traps into upon MMIO and PIO accesses. In total, our implementation required 451 lines of code (including the 357 lines of fuzzing code copied directly from QMORPHUZZ). As bhyve does not have an upstream qtest-like interface, we unbend bhyve crashes into a standalone guest-kernel image (called ReprOS) which simply invokes the CPU instructions that correspond to the I/O operations that caused the crash.

5 Evaluation

We evaluate MORPHUZZ's fuzzing capabilities to answer the following research questions.

- **RQ1** Can MORPHUZZ discover and reproduce bugs in virtual devices? (see § 5.2, § 5.2.1 and Table 1)
- **RQ2** Is MORPHUZZ's implementation generic? Can MORPHUZZ be used to find crashes in multiple hypervisors, and architectures? (see § 5.2.2)
- **RQ3** How does MORPHUZZ's performance compare to the state-of-the-art hypervisor fuzzers? (see Table 1)
- **RQ4** How does QMORPHUZZ's choice to isolate each input (reset state) affect performance? (see § 5.4)
- **RQ5** Can MORPHUZZ stimulate complex DMA behaviors? (discussion in § 5.3 and case studies in § 5.2.3)

5.1 Experimental Setup

We performed our experiments in Debian 10 VMs on a university cluster. The underlying hosts were a mix of small (2 Xeon CPUs with 16 logical cores and 128 GB of RAM) and larger (2 Xeon CPUs with 28 logical cores and 384 GB of RAM) machines. We assigned two cores and 4gb of RAM to each VM. Our experiments were conducted against QEMU version 5.0, and bhyve 12.1 – the same versions used by recent related work. In addition to fuzzing all of the devices covered by prior work, we fuzzed VMs configured with DMA-heavy devices, such as USB controllers, and VIRTIO devices (almost twice as many QEMU devices as prior works).

We dedicated each VM on the cluster to fuzzing a QEMU/bhyve guest configured with each virtual-device in our evaluation set. Following standard practice, we ran the fuzzer with AddressSanitizer enabled and disabled (one fuzzing process each) [31]. ASAN instrumentation allows MORPHUZZ to detect more memory-corruption bugs, but adds considerable overhead (especially when forking).

5.2 Bug-finding

MORPHUZZ found *all*[4] 16 QEMU and 28 bhyve bugs reported by Nyx. Furthermore, MORPHUZZ discovered 61 unique new Bugs in QEMU and 5 in bhyve, for a total of 110 bugs. We manually confirmed each new bug (Appendix C lists all 66 *new* bugs). The final column in Table 1 shows the devices for which QEMU found bugs. Notably, MORPHUZZ found new bugs for every single Block, Network, and USB controller fuzzed – all devices that are particularity DMA-intensive. In total, across the x86 QEMU and bhyve machines, MORPHUZZ found 7 use-after-free, 7 buffer-overflow, 8 stack-overflow, 8 segfaults, 3 resource-exhaustion, 29 abort issues, and 4 miscellaneous crashes.

5.2.1 Reproducing the Crashes

For each of these crashes, MORPHUZZ recorded the *qtest* commands produced by MORPHUZZ and "unbent" the traces, as explained in §. 4.4.3. The resulting order of commands in the *qtest* recording ensures that DMA buffers are filled *prior* to the command that triggers the DMA access.

We successfully reproduced the 61 QEMU bugs in an unmodified build of QEMU 5.0 by simply replaying the *qtest* recordings. For the 5 bhyve bugs found by MORPHUZZ, we converted each *qtest* recording into a separate build of ReprOS, which successfully reproduced all the bhyve bugs. Note that though the number of bugs found for bhyve is significantly lower than those found for QEMU, this is likely due to the fact that bhyve's codebase is less than 2% the size of QEMU. Including the bugs already reported by Nyx, MORPHUZZ found 33 bugs in bhyve. Furthermore Nyx relied of descriptions to fuzz the VIRTIO and USB devices in bhyve, whereas QMORPHUZZ had no such aids.

5.2.2 Strength of MORPHUZZ's generic design

As stated in Section 4, the core insights of bending the input space and fulfilling DMA requests on-demand are independent of any given hypervisor. Previously, we discussed our implementation and evaluation of MORPHUZZ for bhyve. To further illustrate the generic applicability of MORPHUZZ we performed cross-pollination experiments and assessed MORPHUZZ's applicability to architectures other than x86/x86-64.

Bugs in ARM Devices Without changing a single line of code, QMORPHUZZ can fuzz devices only available to VMs targeting a CPU architecture other than x86/x86-64. Specifically, MORPHUZZ found 5 bugs in ARM-specific devices. These bugs include a heap-use-after-free(write) on the ARM Global-Interrupt-Controller, included in all ARM QEMU VMs targeting the cloud. This bug was introduced over 13 years ago. Note that prior works that rely on custom-built operating systems to perform fuzzing (e.g., Hyper-Cube and

[4] Though prior work did not release reproducers for the security bugs, we manually confirmed that MORPHUZZ found bugs matching the Nyx paper's bug descriptions (indicating the type of bug, and location in the source code)

Nyx) would require a re-implementation of the kernel before they can fuzz a different CPU architecture. Since ARM was outside of the scope of all prior works, to allow for a head-to-head comparison, we do not count the 5 ARM-specific QEMU bugs, when tallying the 110 bugs found by MORPHUZZ.

In summary, MORPHUZZ is highly successful at finding and reproducing bugs in virtual-devices (**RQ1**). Furthermore, MORPHUZZ's generic approach transparently applies across VM/CPU architectures and hypervisors (**RQ2**).

5.2.3 Case-Studies

In addition to the example in Section 3.2, here we showcase two bugs found by QMORPHUZZ. These bugs are representative of MORPHUZZ's focus on generically fuzzing the entire virtual-device PIO/MMIO and DMA input-space, while producing inputs that satisfy the complex semantic dependencies required by individual devices.

5.2.3.1 Double-fetch in PCNET The PCNET network adapter is emulated by hypervisors such as QEMU and VirtualBox, with drivers available for all major operating systems. The device accesses two ring-buffers over DMA (one each for sending/receiving packets). Each ring-buffer consists of a set of descriptors that contain pointers to the actual data along with a length field. Prior to sending a packet from the ring-buffer, PCNET checks that the descriptor's length fits alignment requirements. Instead of reading the entire descriptor, before checking the length, the virtual PCNET performs two DMA reads to access the same length data, creating a double-fetch issue. The fuzzer found inputs that leverage this issue to trigger a heap-overflow. Finding and reproducing this timing-sensitive crash would be nearly impossible, without MORPHUZZ's instrumentation of the DMA API.

5.2.3.2 Reentrancy Problem in virtio-gpu The virtio-gpu is a paravirtual adapter designed to provide graphics support with optional hardware-acceleration. The virtio-gpu device relies on a set of "Virtqueues", accessed over DMA, to communicate with the guest. The virtio-gpu is designed with performance and reentrancy in mind, so it splits I/O jobs into lightweight "top-halves", and deferred "bottom-halves" that handle the bulk of the I/O processing. In theory, this approach can combat reentrancy issues, since the nested call into the device simply schedules a deferred bottom-half. In practice, it is difficult to consider all possible reentrancy cases. The fuzzer found inputs that provide a page-index to the virtio-gpu device, via MMIO. The device uses this index to locate a virtqueue in the guest's memory via DMA. The fuzzer proceeds to notify the virtio-gpu of a new request waiting on the command virtqueue. The virtio-gpu schedules and executes the request in a deferred, asynchronous manner. The input provides an address for outputting the response for the I/O, but instead of specifying an address in RAM, the fuzzer provides a generic VIRTIO MMIO register which resets the device. Since the original I/O request still refers to some global virtio-gpu data, this reset triggers a use-after-free. The device failed to consider the VIRTIO MMIO reset register when accounting for reentrancy. Even though Nyx relied on a manually-written VIRTIO spec, it did not find this crash. In fact, the Nyx paper reported no VIRTIO bugs found in QEMU.

When compared with prior works, MORPHUZZ' ability to reshape the input-space was essential for identifying the complex issues presented here (**RQ5**). Furthermore, as demonstrated by the re-entrancy issues found in virtio-gpu and xHCI (described in Section 3.2), we observe another weakness of grammar-based approaches. While a detailed grammar has potential to reach some deep code-paths, it can overlook discrepancies between the specification (encoded in the grammar) and the actual virtual-device implementation. While Nyx' specifications have not been released, MORPHUZZ likely found a superset of the bugs reported by Nyx precisely because its inputs can encode complex, unspecified datastructures that would not be represented in a grammar.

5.3 Coverage

To gauge MORPHUZZ's capability to exercise device-code, we collected branch-coverage and compared MORPHUZZ's performance with prior-work. Table 1 presents these comparative results. Since, at the time of writing, neither VDF, Hyper-Cube, nor Nyx have released fuzzer source-code, we had to rely on the numbers published in each paper for this comparison. Note that, VDF [18] and Hyper-Cube [43] presented coverage data for QEMU 2.2, a version released in 2014. Since then, QEMU's C code has more than doubled in size. The Nyx paper, by virtue of being by the same authors, provides Hyper-Cube coverage, updated for a 24-hour experiment over QEMU 5.0, so we present those numbers in Table 1, rather than those found in the original paper. Unfortunately, there is no up-to-date coverage data for VDF, so we provide the original numbers collected for QEMU 2.2.

To determine whether the coverage increase demonstrated by MORPHUZZ actually arises from it's approach to DMA, rather than some unrelated discrepancy, we performed identical experiments for two modified versions of MORPHUZZ. Essentially, as the source-code for prior systems is not public, we simulated their approaches to DMA within MORPHUZZ. We present the coverage results in Table 1. For the *No-DMA* experiment, we ran QMORPHUZZ with all of the DMA hooks disabled, to simulate a fuzzer that only interacts with devices over Port-IO and MMIO (such as VDF). The *Scratch-Buffer* experiment is identical to *No-DMA*, however, we filled the first 3 GB of the guest's RAM with bytes randomly generated from a seed, and added fuzzer opcodes that write pointers within this random buffer to the virtual device's PIO and MMIO ranges. As a result, the fuzzer has a high chance of providing devices with a pointer to randomized data that can be accessed over DMA. However, the actual randomized data is not controlled by the fuzzer's mutations; the fuzzer can

Device	VDF‡ 25-65 Days	Hyper-Cube* 24 Hours	Nyx‡ 24 Hours	*No-DMA* 24 Hours	*Scratch-Buffer* 24 Hours	QMORPHUZZ 24 Hours	Bug
Source File	Cov.	Cov.	Cov.	Cov.	Cov.	Cov.	
Audio							
ac97	53.0%	**100**%	98.92%	96.38%	96.38% (0.00)	96.38% (0.00)	
cs4231a	56.0%	74.76%	74.76%	92.20%	92.20% (0.00)	**92.20**% (0.00)	
es1370	72.7%	91.38%	91.38%	93.66%	93.66% (0.00)	**93.66**% (0.00)	✔
intel-hda	58.6%	79.17%	78.33%	78.18%	79.10% (**+0.92**)	**81.18**% (**+2.08**)	✔
sb16	81.0%	83.80%	81.34%	86.88%	86.88% (0.00)	86.68% (-0.20)	✔
IBM PC							
fdc	70.5%	84.51%	83.10%	85.43%	86.12% (**+0.69**)	**88.19**% (**+2.07**)	✔
parallel	**42.9**%	38.61%	38.61%	38.61%	38.61% (0.00)	38.61% (0.00)	
serial	44.6%	73.76%	73.76%	73.76%	73.76% (0.00)	73.76% (0.00)	
Block							
ide/core	27.5%	74.87%	74.69%	72.32%	73.55% (**+1.23**)	**78.63**% (**+5.08**)	✔
ahci				55.62%	57.36% (**+1.74**)	**80.86**% (**+23.50**)	✔
sdhci	**90.5**%	81.15%	88.93%	79.65%	80.55% (**+0.90**)	84.8% (**+4.25**)	✔
virtio-blk				52.12%	54.12% (**+2.00**)	**68.51**% (**+14.39**)	✔
virtio-scsi				52.32%	55.80% (**+3.48**)	**66.78**% (**+10.98**)	✔
megasas				26.41%	34.52% (**+8.11**)	**88.41**% (**+53.89**)	✔
sd				64.47%	66.43% (**+1.96**)	**70.11**% (**+3.68**)	✔
scsi-disk				62.36%	65.44% (**+3.08**)	**74.09**% (**+8.65**)	✔
Network							
eepro100	75.4%	83.82%	83.82%	87.13%	87.13% (0.00)	**89.26**% (**+2.13**)	✔
e1000	81.6%	66.08%	54.55%	65.77%	66.14% (**+0.37**)	**89.23**% (**+23.09**)	✔
e1000e_core				75.24%	75.84% (**+0.60**)	**90.54**% (**+14.70**)	✔
ne2000	71.7%	71.89%	71.89%	82.95%	83.47% (**+0.52**)	**98.71**% (**+15.24**)	✔
pcnet	36.1%	78.71%	89.49%	71.38%	72.72% (**+1.34**)	**96.35**% (**+23.63**)	✔
rtl8139	63.0%	74.68%	79.28%	81.78%	84.92% (**+3.14**)	**94.01**% (**+9.09**)	✔
vmxnet3				45.37%	47.63% (**+2.26**)	**63.89**% (**+16.26**)	✔
virtio-net				50.67%	51.58% (**+0.91**)	**60.23**% (**+8.65**)	✔
Graphics							
virtio-gpu				37.64%	39.11% (**+1.47**)	**70.40**% (**+31.29**)	✔
cirrus_vga				90.55%	90.56% (**+0.01**)	90.55% (-0.01)	✔
USB							
hcd-ehci				49.28%	58.09% (**+8.81**)	**78.94**% (**+20.85**)	✔
hcd-xhci		64.40%	87.7%†	60.72%	62.10% (**+1.38**)	**90.52**% (**+28.42**)	✔
ARM							
arm_gic				67.94%	67.94% (0.00)	**67.94**% (0.00)	✔
smc91c111				92.14%	92.14% (0.00)	**92.14**% (0.00)	✔
xgmac				56.00%	64.50% (**+8.50**)	**94.40**% (**+29.90**)	
bhyve							
pci_xhci				48.18%	50.35% (**+2.17**)	**71.32**% (**+20.97**)	✔
virtio_block				54.60%	62.31% (**+7.71**)	**74.36**% (**+12.05**)	✔
Average (Nyx devices)§	61.67%	76.35%	78.16%	77.93%	78.58%	**85.76**%	
Average (All devices)				67.51%	69.42%	**81.08**%	

Table 1: QMORPHUZZ Coverage results side-by-side with results reported by prior work. Empty cells indicate that prior work did not include the corresponding device in its evaluation. In parentheses, we indicate the increase in coverage over the previous column. The final column indicates whether we found bugs, for each device.
*Numbers for Hyper-Cube are taken from the Nyx paper, since the original paper only presents data for 10-minute experiments, while Nyx provides data for 24-hour periods, which matches our experiments.
‡Source is not available. The numbers are copied from the respective papers.
†Nyx relied on a manually-written specification to achieve this coverage
§For this row, we provide the average over 15 VDF devices. The rest of the columns represent averages over the 16 devices evaluated in Nyx [42].

only point the virtual-devices with different offsets within the randomized buffer. This functions similarily to the approach described in Hyper-Cube and Nyx (without virtual-device descriptions). Note that though these experiment employ the strategies for fuzzing DMA described in prior works, the actual implementation details (such choice of fuzzing engine, use of a guest OS, PIO/MMIO range enumeration) differ.

QMORPHUZZ achieves equal or higher coverage for most (13/16) of the devices tested in prior work. Notably, MORPHUZZ, achieved higher coverage over the DMA-intensive xHCI controller than Nyx, despite the manually-written spec necessary for Nyx. Additionally, for 24/33 configurations, QMORPHUZZ outperformed the *Scratch-Buffer* and *No-DMA* configurations which mimic the DMA approaches of prior works. Manually examining the code of the remaining 9 devices, we confirmed that none of them are controlled by DMA data. As the configurations are identical to QMORPHUZZ except for the DMA strategy, it is clear that MORPHUZZ's increased coverage and bug-finding capability is primarily due its treatment of DMA as a first-class I/O transport.

MORPHUZZ performed especially well for devices with complex DMA-interactions, such as block and network-devices. Prior work achieved matching or higher coverage than MORPHUZZ for four legacy devices and devices with low DMA complexity. Interestingly, though MORPHUZZ achieves less coverage over the SDHCI device, we found long-standing SDHCI issues, that were not reported by prior works.

We manually sampled the code that MORPHUZZ did not cover. We found that coverage is limited for devices that contain code which is executed only when certain command-line options are configured. For example, the virtio-net device provides 69 configurable options on the command line; a large part of the code in the virtio-net.c file is only accessible, when the corresponding set of options is configured. Additionally we found that many virtual devices dedicate a sizable amount of code for hot-plugging and live-migration support, which cannot be reached by interacting with virtual-devices from a VM. Instead these functions are only reachable from the hypervisor, and hence are out of scope for our threat model.

In summary, MORPHUZZ found a superset of the bugs reported by prior works that target virtual device implementations. Importantly, MORPHUZZ achieved significantly higher coverage for complex devices that required the fuzzer to perform PCI configuration, PIO, MMIO and multiple layers of DMA communication to reproduce a single crash. MORPHUZZ's ability to effectively fuzz devices across all of these modes of I/O, without seeds or specifications, is unparalleled by prior approaches (**RQ3**).

5.4 Throughput

We measured MORPHUZZ's performance in terms of qtest instructions/second. QMORPHUZZ's *qtest* instructions are guaranteed to interact with address ranges associated with virtual devices (through PIO and MMIO) or DMA buffers read by these devices. We ran the fuzzer for 10 minutes on a single core and calculated the number of *qtest* instructions executed per second in three configurations. We found that in the standard configuration, QMORPHUZZ executes 4,170 *qtest* instructions per second. Additionally, we experimented with other state-resetting mechanisms, and found that though they improve throughput, though, usually, at the cost of stability(**RQ4**). We provide further details in Appendix B.

MORPHUZZ's reports have already led to 22 issues fixed in QEMU (9 of these have been assigned CVEs). In fact, MORPHUZZ had the second-most report credits in QEMU's 5.2 release (the first spot is held by an automated system that reports errors encountered during compilation and unit-testing). MORPHUZZ has, since, been deployed on OSS-Fuzz, where it has found and reported additional bugs. In total, QMORPHUZZ has found 81 QEMU bugs on OSS-Fuzz, that are in various stages of triage. The QEMU repository docu-

ments the trivial steps for configuring QMORPHUZZ to fuzz additional virtual-devices, on OSS-Fuzz.

6 Discussion

Despite QMORPHUZZ's positive results, we briefly discuss limitations and avenues of further improvement.

Hypervisor Configurations. MORPHUZZ fuzzes a given "hardware"-configuration of a VM. Our current implementation aggregates a set of virtual-device configurations by executing QEMU's test suite. However, these tests do not elicit the complete spectrum of viable QEMU configurations (such as the 78 virtio-net options). We found that certain devices were not configured by any of the test cases, and other devices had certain features disabled. A dedicated analysis (e.g., over QEMU's command line parser) could enrich the available set of configurations and directly benefit QMORPHUZZ.

Independent Virtual-Devices. With KVM, users can complement QEMU's virtual-devices, with kernel-modules. For example, vhost-net is a host-kernel-driver that accelerates the virtio-net device by avoiding expensive context-switches to user-space QEMU that occur each time the guest accesses a virtual-device. In the future, for security purposes, hypervisors may run virtual-devices in isolated processes. MORPHUZZ relies on libFuzzer, which is designed to collect coverage of a single user-space process. As these approaches gain traction, they will open new opportunities for virtual-device fuzzers.

Reproducing Double-Fetches As we mentioned in Section 4.4.3, MORPHUZZ does not "unbend" double-fetch bugs into standalone reproducers. That is, though double-fetches can be consistently reproduced in an instrumented version of the hypervisor, we have no reliable way to automatically re-create them in an unmodified build. The underlying reason for this is that with MORPHUZZ's hooks, DMA becomes a synchronous operation from the fuzzer's point of view - the fuzzer does not have to race against the device to overwrite guest memory since the DMA access is effectively paused until the fuzzer's hooking code returns. Once the fuzzer and the hooks are removed, we are faced with the DMA's natural asynchronicity. In the future, it may be possible to use a brute-force approach, or instrumentation such as ThreadSanitizer to consistently reproduce double-fetches in hypervisors.

7 Related Work

Since its appearance in the '80s, fuzzing has gained widespread attention in the academic community. A major catalyst reviving interest, was the release of the American Fuzzy Lop (AFL) [57] fuzzer, which popularized, coverage-guided, fuzzing for a wide range of software. Researchers have focused on improving fuzzing performance, with advancements in input scheduling [24, 41, 52], mutation algorithms [4,30,40], and input feedback [1,16,59]. Other systems focus on applying concolic execution [25,26,56] to overcome roadblocks, such as comparisons against "magic constants", and checksums [35]. Fuzzers such as AFL with laf-intel [27] and libFuzzer [46] have applied source-code instrumentation to identify comparisons against magic bytes and produce inputs that can pass them. Other works have adapted fuzzers to complex targets such as code-interpreters [17,21,50,55], compilers [5,28,29], and network-protocols [2,12,15]. Operating system kernels have received widespread attention within the academic community, with systems purpose-built for kernel-drivers [6], kernel race-conditions [23], file-systems [54], and the system-call interface [13, 19, 44]. Other works, such as Periscope [47] examine MMIO and DMA communication between a kernel and peripherals to identify vulnerabilities in a kernel exposed to a compromised device. Periscope, operates from a different viewpoint that MORPHUZZ, by reacting to DMA activity initiated by guest drivers running within Linux. Unlike Periscope, MORPHUZZ constructs entire device "converstations", starting with PCI configuration and continuing with DMA buffer-setup, and PIO/MMIO activity. This allows MORPHUZZ to exercise code paths, that well-behaved guest-drivers would never touch. Similarly, VIA [20] fuzzes OS-drivers to identify bugs that could compromise of security-guarantees in a confidential-computing environment, where virtual-device code is not trusted. Works have also applied static [51,53] and dynamic [45] techniques to detect and analyze double-fetch issues in software.

Most related to our work, others have identified that virtual devices pose many of the same challenges for fuzzers as kernel system-calls and drivers. IOFuzz [32] finds bugs in virtual devices, by writing random values to port-mapped IO. VDF [18] collects MMIO and PIO traces, and uses these traces as seeds for coverage-guided fuzzing. VDF does not fuzz DMA traffic or fully reset state between input executions. Hyper-Cube [43] and Nyx [42] are state of the art virtual device fuzzers, based on a custom-built guest operating system. Hyper-Cube uses PCI enumeration to identify IO ranges mapped to virtual devices and sets up a single scratch buffer which it uses to trigger DMA activity, by writing addresses within the scratch buffer to MMIO and PIO ranges. These properties make Hyper-Cube portable to various hypervisors supporting x86, but limit Hyper-Cube on virtual-devices which rely on magic constants, and, importantly, DMA semantics. Nyx augments Hyper-Cube, by using a full-system snapshotting and hardware-assisted coverage framework. To fuzz complex DMA devices, Nyx requires a manually-written specification. Unlike Nyx, MORPHUZZ reshapes the virtual-device input-space in response to feedback collected from fundamental hypervisor APIs. This allows MORPHUZZ to achieve high coverage over complex devices without specifications or seed-inputs and intrinsically enables MORPHUZZ to generate device interactions that lie outside the manually-crafted specification. Concurrently and independently, V-Shuttle [34] developed a method to fuzz the DMA input-space in a targetted fashion, by replacing DMA access calls with reads from a file generated by AFL. Notably, V-Shuttle performs a static-analysis to annotate DMA accesses

and treats and fuzzes each one using a separate fuzzer mutator. Unlike V-Shuttle, MORPHUZZ does not require any preliminarty analysis or manual work to convert crashes into reproducers.

In an industrial setting, there has been work to fuzz virtual devices [49] using a minimal OS connected to AFL. MORPHUZZ builds upon the lessons of prior-works to present the first design that can fuzz arbitrary device across all major modes of I/O.

8 Conclusion

MORPHUZZ is the first generic and coverage-guided fuzzer capable of interacting with virtual devices that implement complex DMA interactions. To this end, MORPHUZZ leverages insights from hypervisor design (e.g., inferring IO ranges from Guest-Address Maps) and adds support for on-demand DMA buffer identification and provision thereof. MORPHUZZ features an opcode interpreter that leverages the hypervisor's APIs to directly communicate with virtual devices, obviating the need for a custom-built guest OS. The evaluation of our QMORPHUZZ prototype shows that it is highly performant reaching an average 81% branch coverage over 33 virtual device implementations. QMORPHUZZ is particularly successful when analyzing devices that rely on complex DMA interactions. The system identified 66 new and unique crashing bugs, which we reported upstream. MORPHUZZ is already included in QEMU's code-base where it continually fuzzes virtual device implementations via OSS-Fuzz.

Acknowledgements

We would like to thank our anonymous reviewers for their insightful comments and feedback. Additionally, we are pleased to acknowledge Darren Kenny, Paolo Bonzini, Philippe Mathieu-Daudé, Qiuhao Li, Thomas Huth, and the rest of the QEMU community for their help with upstreaming and maintaining the fuzzer. We acknowledge that the computational work reported in this paper was performed on the Shared Computing Cluster, administered by Boston University's Research Computing Services. This material is based upon work supported by the NSF under award number CNS-1942793.

References

[1] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. IJON: Exploring Deep State Spaces via Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[2] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *Proceedings of the International Conference on Information Security (ISC)*, Samos, Greece, 2006.

[3] William Blair, Sajjad Arshad, Andrea Mambretti, Michael Weissbacher, Engin Kirda, William Robertson, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[4] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, 2015.

[5] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, 2013.

[6] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Dallas, TX, 2017.

[7] Cve-2017-7539. `https://www.cvedetails.com/cve/CVE-2017-7539/`, 2017.

[8] Cve-2019-15890. `https://www.cvedetails.com/cve/CVE-2019-15890/`, 2019.

[9] Cve-2019-6778. `https://www.cvedetails.com/cve/CVE-2019-16778/`, 2019.

[10] Cve-2019-9824. `https://www.cvedetails.com/cve/CVE-2019-9824/`, 2019.

[11] Cve-2021-20221. `https://bugs.launchpad.net/qemu/+bug/1914353`, 2021.

[12] Joeri De Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the USENIX Security Symposium*, Washington, DC, 2015.

[13] Bernhard Garn and Dimitris E Simos. Eris: A tool for combinatorial testing of the Linux system call interface. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Cleveland, OH, 2014.

[14] Jason Geffner. Virtualized environment neglected operations manipulation (VENOM). `https://www.crowdstrike.com/venom/`, 2015.

[15] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security*, 10(8):239, 2010.

[16] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the USENIX Security Symposium*, Washington, DC, 2013.

[17] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2019.

[18] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted evolutionary fuzz testing of virtual devices. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, Atlanta, GA, 2017.

[19] Jesse Hertz and Tim Newsham. Triforceafl. `https://github.com/nccgroup/TriforceAFL`, 2017.

[20] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. Via: Analyzing device interfaces of protected virtual machines. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[21] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the USENIX Security Symposium*, Bellevue, WA, 2012.

[22] How To Split A Fuzzer-Generated Input Into Several. `https://github.com/google/fuzzing/blob/master/docs/split-inputs.md`.

[23] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2019.

[24] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, Toronto, Canada, 2018.

[25] Su Yong Kim, Sungdeok Cha, and Doo-Hwan Bae. Automatic and lightweight grammar generation for fuzz testing. *Computers & Security*, 36:1–11, 2013.

[26] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. Cab-fuzz: Practical concolic testing techniques for COTS operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, 2017.

[27] Circumventing fuzzing roadblocks with compiler transformations. `https://lafintel.wordpress.com/`, 2016.

[28] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, 2015.

[29] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, Honolulu, HI, 2019.

[30] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 56:127–135, 2007.

[31] Notes for using ASAN with afl-fuzz. `https://github.com/google/AFL/blob/master/docs/notes_for_asan.txt`, 2020.

[32] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. `taviso.decsystem.org/virtsec.pdf`, 2007.

[33] Oss-fuzz. `https://google.github.io/oss-fuzz/`.

[34] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In *Proceedings of the Annual Computer Security Applications Conference*, 2021.

[35] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[36] Perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[37] Qemu : Security vulnerabilities. `https://www.cvedetails.com/vulnerability-list/vendor_id-7506/Qemu.html`, 2020.

[38] Qemu issue #1878054. `https://bugs.launchpad.net/qemu/+bug/1878054`, 2020.

[39] Qemu issue #1911075. `https://bugs.launchpad.net/qemu/+bug/1911075`, 2020.

[40] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[41] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, 2014.

[42] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *Proceedings of the USENIX Security Symposium*, Vancouver, BC, 2021.

[43] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Proceedings of the Network and Distributed Security Symposium (NDSS)*, San Diego, CA, 2020.

[44] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. Kafl: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium*, Vancouver, CA, 2017.

[45] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the Asia Conference on Computer and Communications Security*, pages 587–600, 2018.

[46] Kostya Serebryany. libFuzzer–a library for coverage-guided fuzz testing. `https://releases.llvm.org/10.0.0/docs/LibFuzzer.html`, 2015.

[47] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *Proceedings of the Network and Distributed Security Symposium (NDSS)*, San Diego, CA, 2019.

[48] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the USENIX Security Symposium*, Boston, MA, 2020.

[49] Jack Tang and Moony Li. When virtualization encounter AFL. *Black Hat Europe*, 2016.

[50] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Heraklion, Greece, 2016.

[51] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Proceedings of the USENIX Security Symposium*, pages 1–16, 2017.

[52] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Berlin, Germany, 2013.

[53] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 661–678. IEEE, 2018.

[54] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2019.

[55] Dingning Yang, Yuqing Zhang, and Qixu Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Liverpool, England, UK, 2012.

[56] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the USENIX Security Symposium*, Baltimore, MD, 2018.

[57] Michal Zalewski. American fuzzy lop. `https://lcamtuf.coredump.cx/afl/`, 2014.

[58] ZERODIUM - How to Sell Your 0day Exploit to ZERODIUM. `https://zerodium.com/program.html`.

[59] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018.

A Guest memory access APIs in various open source hypervisor implementations

Hypervisor	DMA-access API Functions
QEMU	`address_space_{ld,st,map}`
bhyve	`paddr_guest2host`
VirtualBox	`pfnPhys{Read,Write}`
Bochs	`DEV_MEM_{READ,WRITE}_PHYSICAL_DMA`

Table 2: Guest memory access APIs in various open source hypervisor implementations

B Throughput Discussion

First we ran MORPHUZZ in its default configuration, where each test-case is executed within an isolated, forked child.

Secondly, we ran MORPHUZZ built with AddressSanitizer, which carries performance overheads, but uncovers additional classes of bugs (e.g., non-crashing memory corruptions). Finally, we ran QMORPHUZZ without resetting any state, only restarting the fuzzer after a crash brings down the process.

With ASAN enabled, this number drops to 1,610 *qtest* instructions per second. ASAN's overheads are exacerbated by the fact that, for a 64-bit executable, ASAN maps 20 terabytes of virtual memory as part of its detection capability. Since QMORPHUZZ relies on a fork server, the kernel must make a copy of the large page-tables resulting from ASAN's virtual-memory use. Using perf [36], we found that close to 40% of the execution time was spent in the kernel, during the `fork()` system-call. Finally, without resetting any state, QMORPHUZZ executes 5,850 instructions per second. Though, this is a sizable (i.e., 40%) gain over the first configuration, it comes at the cost of input stability and MORPHUZZ's reproducible-crash guarantee. Additionally, performance is hurt by the fact that the entire process must be restarted, each time a crash is triggered.

Note that QMORPHUZZ's `pattern` instruction and on-demand DMA fulfillment , can produce inputs that populate megabytes of data just-in-time (e.g., filling entire ring-buffers in DMA). Though we configured libFuzzer to limit inputs to 2,048 bytes, MORPHUZZ's use of DMA patterns results in much longer *qtest* sequences. Thus, there can be drastic differences in execution times for each *qtest* command. In summary, MORPHUZZ trades performance for input-stability. In addition to enabling MORPHUZZ to reproduce crashes with no post-processing, this stability allows MORPHUZZ to achieve higher coverage and bug-finding performance for complex devices, than prior works (**RQ4**).

C List of Bugs Reported by MORPHUZZ

QEMU
Stack-overflow in ahci_cond_start_engines
Stack-overflow in _eth_get_rss_ex_dst_addr
Stack-overflow in rtlNUMBER_transmit_one
Stack-overflow in pcnet_poll_timer
Stack-overflow in e1000_receive_iov
Stack-overflow in flatview_do_translate through e1000
Stack-overflow in intel_hda_corb_run
Stack-overflow in xhci_pci_intr_raise
Buffer-underflow in xhci_runtime_write
Global-buffer-overflow in mode_sense_page
Heap-buffer-overflow in sdhci_write_dataport
Heap-buffer-overflow in sdhci_data_transfer
Heap-buffer-overflow in sd_erase
Heap-buffer-overflow in msix_table_mmio_write
Heap-buffer-overflow in pcnet_receive
Heap-use-after-free in e1000e_write_packet_to_guest
Heap use-after-free in e1000e_write_to_rx_buffers
Heap-use-after-free in ehci_flush_qh
Heap-use-after-free in usb_packet_copy
Heap-use-after-free in usb_packet_unmap
Heap-use-after-free in virtio_gpu_ctrl_response
Heap-use-after-free through double-fetch in ehci
Memcpy-param-overlap in flatview_write_continue
Memcpy param-overlap in ip_stripoptions
Memcpy param-overlap through e1000e_write_to_rx_buffers
Memory Exhaustion in vmxnet3_activate_device
Memory Exhaustion in hpet_timer
Segfault in virtio_write_config
Segfault in address_space_to_flatview through ide
Segfault in blk_bs
Segfault in megasas_command_complete
Segfault in megasas_handle_frame
Segfault in tcg_handle_interrupt
Segfault in usb_bus_from_device
Infinite Loop in sdhci_data_transfer
Floating-point exception in ide_set_sector
Assertion-failure in audio_bug
Assertion-failure in mch_update_pciexbar
Assertion-failure in vmxnet3_validate_interrupt_idx
Assertion-failure in vmxnet3_validate_queues
Assertion-failure in address_space_stw_le_cached through virtio-net
Assertion-failure in address_space_stw_le_cached through virtio-blk
Assertion-failure in address_space_cache_invalidate through virtio-gpu
Assertion-failure in address_space_unmap through ahci_map_clb_address
Assertion-failure in address_space_unmap through virtio-blk
Assertion-failure in virtio_blk_reset
Assertion-failure in bdrv_aio_cancel
Assertion-failure in bmdma_active_if
Assertion-failure in e1000e_write_lgcy_rx_descr
Assertion-failure in e1000e_write_rx_descr
Assertion-failure in e1000e_write_to_rx_buffers
Assertion-failure in e1000e_intrmgr_on_throttling_timer
Assertion-failure in e1000e_intmgr_collect_delayed_causes
Assertion-failure in eth_get_gso_type through e1000e
Assertion-failure in iov_from_buf_full through e1000e
Assertion-failure in net_tx_pkt_add_raw_fragment through vmxnet3
Assertion-failure in net_tx_pkt_reset through vmxnet3
Assertion-failure in pci_bus_get_irq_level
Assertion-failure in scsi_dma_complete, with megasas
Assertion-failure in usb_detach
Assertion-failure in ati_reg_read_offs and ati_reg_write_offs
bhyve
Segfault in vq_getchain
Assertion in modify_bar_registration
Assertion in unregister_mem
Assertion in pci_vtnet_proctx
Assertion in pci_vtnet_cfgwrite
QEMU (ARM)
Heap-use-after-free in gic_dist_writeb
Segfault in smc91c111_writeb
Assertion-failure in gic_clear_pending_sgi
Assertion-failure in bcm2835_thermal_read
Assertion-failure in dwc2_hsotg_write

Table 3: New bugs found by MORPHUZZ