HyperPill: Fuzzing for Hypervisor-bugs by Leveraging the Hardware Virtualization Interface

Alexander Bulekov[*†§] *Qiang Liu* [*‡] *Manuel Egele*[†] *Mathias Payer*[*]
alxndr@bu.edu *qiang.liu@epfl.ch* *megele@bu.edu* *mathias.payer@nebelwelt.net*

[*]EPFL [†]Boston University [‡]Zhejiang University [§]Amazon

Abstract

The security guarantees of cloud computing depend on the isolation guarantees of the underlying hypervisors. Prior works have presented effective methods for automatically identifying vulnerabilities in hypervisors. However, these approaches are limited in scope. For instance, their implementation is typically hypervisor-specific and limited by requirements for detailed grammars, access to source-code, and assumptions about hypervisor behaviors. In practice, complex closed-source and recent open-source hypervisors are often not suitable for off-the-shelf fuzzing techniques.

HyperPill introduces a generic approach for fuzzing arbitrary hypervisors. HyperPill leverages the insight that although hypervisor implementations are diverse, all hypervisors rely on the identical underlying hardware-virtualization interface to manage virtual-machines. To take advantage of the hardware-virtualization interface, HyperPill makes a snapshot of the hypervisor, inspects the snapshotted hardware state to enumerate the hypervisor's input-spaces, and leverages feedback-guided snapshot-fuzzing within an emulated environment to identify vulnerabilities in arbitrary hypervisors. In our evaluation, we found that beyond being the first hypervisor-fuzzer capable of identifying vulnerabilities in arbitrary hypervisors across all major attack-surfaces (i.e., PIO/MMIO/Hypercalls/DMA), HyperPill also outperforms state-of-the-art approaches that rely on access to source-code, due to the granularity of feedback provided by HyperPill's emulation-based approach. In terms of coverage, HyperPill outperformed past fuzzers for 10/12 QEMU devices, without the API hooking or source-code instrumentation techniques required by prior works. HyperPill identified 26 new bugs in recent versions of QEMU, Hyper-V, and macOS Virtualization Framework across four device-categories.

All work was completed prior to author joining Amazon.

1 Introduction

Hypervisors provide the security foundations necessary for the cloud. They enable efficient use of hardware resources, by colocating workloads from multiple tenants on the same bare-metal machines, isolated in individual virtual-machines (VMs). As such, hypervisors ensure that code running in VMs cannot violate the virtualization boundary (e.g., by performing a VM escape attack) and compromise the workloads of the other tenants, or the hypervisor itself.

Unfortunately, VM escape attacks are a tangible reality. Hundreds of bugs have been identified in the complex hypervisor code. Due to the severity of these bugs, hypervisor compromises are awarded large bug bounties, similar to other high-value targets such as web browsers and mobile devices [53]. In parallel, fuzzing has emerged as one of the most powerful techniques for automatically uncovering vulnerabilities in a large range of software [4,10,14,18,20,21,27,28,39,45,49,52]. As such, a significant amount of academic research has focused on leveraging fuzzing to automatically identify bugs in hypervisor code, so that they can be promptly fixed, preventing malicious exploitation [6, 13, 26, 29, 32, 37, 38].

State-of-the-art approaches [6, 26] are capable of automatically finding complex bugs across most major attack-surfaces (i.e., PIO/MMIO/DMA). However, these approaches rely on access and manual modifications to hypervisor source-code to effectively fuzz virtual-devices. Even with access to source-code, porting current methods to new targets is a non-trivial process that requires considerable manual effort by an expert. Furthermore, most fuzzers do not handle the hypercall attack-surface as hypercalls are often implemented in a separate component from the core device-emulation (e.g., in the OS kernel), for performance reasons. Thus, even though major open-source targets such as QEMU and VirtualBox have been extensively fuzzed, closed-source targets such as Hyper-V and macOS Virtualization Framework have not been thoroughly fuzzed with state-of-the-art methods.

Due to the diversity of designs and runtimes (see Tab. 1), harnessing a new hypervisor requires a combination of knowl-

edge of advanced fuzzing techniques and expert knowledge about the operation of individual virtual-devices. While hypervisors offer similar functionality, there is currently no *fully automatic* hypervisor-fuzzer capable of exploring *all major interfaces* exposed by *any hypervisor*.

Efficient automated hypervisor fuzzing entails: 1. Input Space Enumeration 2. DMA Hooking .

Input Space Enumeration. Hypervisors expose vast memory spaces and port input ranges, that are sparsely populated by Memory-Mapped IO (MMIO) and Port IO (PIO) addresses actually associated with virtual-devices. To effectively fuzz these input-spaces, current fuzzers enumerate the port and memory input-spaces to identify the active PIO/MMIO addresses. V-Shuttle [32], ViDeZZo [26] and Morphuzz [6] require access and modifications to the hypervisor source-code to hook into APIs that create PIO/MMIO ranges. HyperCube [38] and Nyx [37] leverage heuristics to approximate the active PIO/MMIO ranges and hard-code legacy device addresses. Additionally, hypervisors provide access to hypercalls which feature hypervisor-specific function signatures. There are currently no fuzzers that automatically identify the parameters related to hypercalls.

DMA Hooking. Direct Memory Access (DMA) provides devices with the capability to directly read from memory, bypassing the CPU, enabling high-bandwidth applications. In the context of virtualization, the hypervisor can read DMA inputs from arbitrary locations in VM-memory at any time, without direct involvement from the virtual CPU (vCPU). The sizes, structures, and formats of these DMA inputs are diverse and device-specific. DMA is used ubiquitously by block-storage, networking, display, and USB controllers. Nyx [37] proposes device-specific grammars to effectively fuzz DMA-capable devices. However, such grammars require expert knowledge, are difficult to construct, and are prone to inaccuracies which decrease fuzzing performance. To avoid the need for difficult-to-obtain device-specific grammars, state-of-the-art fuzzers converged on hooking DMA-related APIs to facilitate fuzzing. To this end, V-Shuttle, ViDeZZo, and Morphuzz all rely on modifications to the hypervisor's DMA APIs [6, 26, 32]. Similarly, MundoFuzz [29] leverages recordings of device interactions from a kernel with instrumented PIO/MMIO/DMA APIs to automatically synthesize grammars. Even with source-code access, it remains challenging to identify all DMA-related APIs even in open-source software (e.g., QEMU alone features dozens of DMA-access APIs). Furthermore, developers commonly use pointer-based DMA APIs (e.g., `pci_dma_map()` in QEMU) which return a pointer to a large VM-memory region which can then be used to arbitrarily index into the guest-memory. Current fuzzers are not equipped to deal with DMA activity from such generic APIs.

	Open-Source	Type-1	Language	Execution Environment	PV Platform	Hypercall identifier	EPT MMIO Mechanism
QEMU-KVM	●	○	C	K+U	VIRTIO	RAX	V+M
FIRECRACKER	●	○	Rust	K+Upaper.tex	VIRTIO	RAX	V+M
VIRTUALBOX	●	○	C++	K+U	VIRTIO/VMBus	RAX/RCX	M
VMWARE (PC)	○	○	C,C++	K+U	VMWare	RDX	
VMWARE ESX	○	●	C	K	VMWare	RDX	M
MACOS	○	○	?	U	VIRTIO	RAX	V
HYPER-V	○	●	C++,?	VM+K+U	VMBus	RCX	V

Table 1: The diversity of hypervisors. In the *Execution Environment* column, the letters indicate that significant parts of the trap-and-emulate cycle are implemented in the Kernel (K), Userspace (U) and dedicated virtual-machines (VMs) that host parts of the hypervisor. The *MMIO Mechanism* column indicates that MMIO accesses are trapped using EPT Violations (V) and EPT Misconfigs (M).

Our Approach. Our approach leverages the fundamental insight that *while modern hypervisor internals feature a plethora of different designs, all implementations must rely on an architecture-specific hardware-virtualization interface to configure VMs*. As such all, hypervisors that run on the same CPU architecture leverage standardized CPU features to manage VMs. This interface encodes a wealth of information about the input-spaces available to VMs. We introduce HYPERPILL, a generic hypervisor-fuzzer that leverages this well-specified hardware-virtualization interface to automatically fuzz arbitrary hypervisors. Our design *transplants a single full-system snapshot of the hypervisor into an emulation-environment for fuzzing*. HYPERPILL can use the hardware-virtualization interface to inject arbitrary IO activity into snapshotted hypervisors. Furthermore, it leverages feedback from the standardized hardware-virtualization interface and the granular introspection capabilities enabled by emulation to automatically enumerate the PIO and MMIO input-spaces. HYPERPILL identifies all the memory associated with the virtualized guest and performs DMA hooking at the level of atomic memory accesses. During fuzzing, it leverages runtime feedback from the emulator to automatically identify the parameters needed to invoke hypervisor-specific hypercalls. Thus, HYPERPILL elides manual hooking of hypervisor-specific APIs, while providing finer-grained feedback than state-of-the-art approaches that rely on source-code instrumentation. Due to our unique emulation-based approach, HYPERPILL is the first fuzzer capable of automatically fuzzing all major input-spaces of arbitrarily-architected hypervisors with state-of-the-art techniques. Furthermore, HYPERPILL has the most fine-grained hypervisor introspection capabilities, even allowing it to outperform fuzzers that leverage source-code instrumentation.

Evaluation. We leveraged HYPERPILL to find bugs in recent QEMU, Hyper-V, and macOS Virtualization Framework hypervisors. For coverage, we compared HYPERPILL against

two state-of-the-art approaches (i.e., Morphuzz and ViDeZZo) that require source code access. HyperPill outperforms the two approaches for 10/12 QEMU devices. HyperPill successfully identified 26 new bugs in all hypervisors across four device-categories, of which 11 are bugs in QEMU, which has been fuzzed extensively since 2020. Our throughput evaluation shows that HyperPill is slower than ViDeZZo but faster than Morphuzz. The performance penalty introduced by emulation is offset by HyperPill's ability to surpass state-of-the-art approaches through high-quality inputs.

In summary, we make the following contributions:

- We present a novel method for fuzzing arbitrary hypervisors that are essential to modern-day use by leveraging the core hardware-virtualization interface. Due to the rich feedback available from this interface, HyperPill can automatically enumerate and fuzz all major hypervisor attack-surfaces (i.e., PIO/MMIO/Hypercalls/DMA) hypervisors without any target-specific harnesses work.
- We implement our hypervisor-fuzzer, HyperPill, which leverages the hardware-virtualization interface to fuzz arbitrary hypervisors. HyperPill takes a full-system snapshot of the hypervisor target and transplants the snapshotted target into an emulation environment, where it probes the input-spaces and fuzzes the target.
- We evaluate HyperPill's capabilities. We report on 26 new bugs discovered by HyperPill, of which 9 are in virtual-devices covered by state-of-the-art hypervisor-fuzzers. In addition to demonstrating HyperPill's ability to find bugs, we show that HyperPill surpasses the coverage of previous target-specific harnesses which require access to and modification of source-code.
- To enable further research into grammarless, generic hypervisor harnessing and fuzzing, we will open-source all of HyperPill's components.

2 Virtualization Background

Hypervisors. Hypervisors, or virtual machine monitors (VMMs) are software that create and execute virtual machines (VMs). In the mid-2000s, x86 virtualization changed substantially, as Intel and AMD introduced hardware-backed virtualization support. The new virtualization interfaces allowed for high-performance low-complexity offloading of CPU and memory virtualization tasks that were traditionally handled in software using complex high-overhead approaches such as binary translation and MMU-shadowing [36]. These advances drastically changed the computing landscape and accelerated the proliferation of virtualization-based cloud technology. However, while hardware-accelerated CPU and memory virtualization greatly increased performance and usability, Hypervisor code is still largely responsible for providing VMs with access to virtual-hardware resources, such as networking, display, and storage. For example, when a VM attempts to access the network, the hypervisor must trap the VM's access and emulate the network access by running virtual network device code. Such virtual-devices are generally implemented in hundreds of thousands of lines of C/C++ code. The critical nature of this code has led recent hypervisors, such as Firecracker, to leverage "safer" languages (e.g., Rust) to implement virtual-devices. As such, even with hardware acceleration, hypervisor code remains a prime target for VM-escape attacks. Before exploring the attack-surface further, we detail the hardware-accelerated virtualization interface.

Hardware-Accelerated Virtualization Extensions. Commodity CPU architectures have support for hardware-accelerated virtualization (VT-x for Intel, AMD-V for AMD, Virtualization Extensions for ARM, RISC-V H, Virtualization Extensions for PowerPC). We focus on prevalent x86 hypervisors leveraging the Intel VT-x extensions. However, note that semantically, hardware-accelerated virtualization extensions feature broadly similar functionality. The privileged hypervisor uses an architecture-specified API to configure a virtual-machine, and dedicated instructions (e.g., `vmresume` on Intel) to "enter" into the guest, which is called "VM-Entry". When an external event occurs (e.g., a timer interrupt), or the guest accesses a privileged resource (e.g., a network-card MMIO address), control returns to the hypervisor, which is tasked with handling this "VM-Exit". The CPU informs the hypervisor of the reason for the "VM-Exit" through an architecture-specific mechanism (e.g., a data-structure in memory).

This advanced virtualization functionality requires the hypervisor to perform comprehensive configuration of VM-related control settings. On Intel CPUs, the `vmptrld` instructs the CPU to use an area in memory to maintain the control information for a new VM. This area is called a VM Control Structure, or *VMCS*. The VMCS contains hundreds of fields (called *encodings*) that specify the VM's configuration state. For example, the VMCS contains the VM's instruction pointer, page-table base register or segmentation configurations. For a VM-Entry, the `vmlaunch/vmresume` instructions are used to activate the VM specified by the VMCS and run the guest code. When the guest triggers a VM-Exit, the CPU modifies the VMCS to provide the hypervisor with the necessary information to handle the VM-Exit. Namely, the CPU places an exit identifier in the `EXIT_REASON` encoding, which specifies the reason for the exit (e.g., PIO, MMIO or hypercall). Further information is placed in the `EXIT_QUALIFICATION` encoding. The type of information depends on the `EXIT_REASON`. For example, for faulting IO accesses, the `EXIT_QUALIFICATION` contains the address, size, and type (read/write) of the access. The hypervisor inspects the VMCS to learn the exit reason and exit qualification. Following the ubiquitous trap-and-emulate virtualization model, the hypervisor is responsible for emulating the behavior that triggered the VM-Exit. The way in which VM-Exits are handled is hypervisor-specific. However, the mechanisms used to configure VMs are not.

All hypervisors that wish to configure hardware-accelerated VMs, rely on an identical and well-specified architectural interface to encode crucial information about the VMs. We name this interface the hardware-virtualization interface.

Modern MMU Virtualization. Traditionally, MMU virtualization was a performance obstacle. An MMU converts virtual-addresses to physical-addresses using a translation layer (usually encoded in hierarchical page-tables). However, virtualization requires a second layer of translation. First, the VM's Guest Virtual Address (GVA) must be converted to a Guest Physical Address (GPA) using the guest's page-table. Then, the Guest Physical Address must be converted to a Host Physical Address (HPA). Since the hypervisor runs in the host's virtual-address space, and it needs access to the guest's physical memory (e.g., to perform DMA), the guest's memory is mapped into the host's virtual memory. With the advent of hardware-accelerated virtualization, CPUs began providing an accelerated method for guest-paging called *Second Level Address Translation (SLAT)*. On Intel CPUs, SLAT is implemented as Extended Page Tables (EPT). The guest continues to maintain its page-table (PT), which translates GVAs to GPAs. However, the hypervisor maintains a second page-table, which translates GPAs to HPAs. When executing the VM, the hardware translates addresses through both the PT and the EPT table to convert GVAs to HPAs. Notably, the EPT mechanism is also responsible for trapping MMIO accesses. Since MMIO ranges are ranges of GPA that require emulation, they are configured within the EPT to trap into the hypervisor, rather than map to HPAs.

In summary, the EPT contains information required to identify physical memory frames that are allocated to the guest and to determine which guest-address accesses trap into the hypervisor.

Interacting with the Hypervisor. VMs interact with the hypervisor through a combination of Port/Memory-Mapped IO, Hypercalls, and Directed Memory Access.

Memory-Mapped IO dedicates certain parts of the guest's physical memory for device IO. Unlike regular read/write access to physical memory, operations to memory-mapped regions cause VM-Exits which the hypervisor emulates by executing virtual-device code. Transferring data to and from the device via MMIO is a blocking/synchronous operation for the vCPU. *Port IO* follows the same trap-and-emulate virtualization model as MMIO, however, it targets a separate 16-bit address-space. PIO is x86-specific and accessed via `in/out` instructions. PIO is usually reserved for low-bandwidth communication with devices, such as timers and serial ports.

Hypercalls are a virtualization-specific mode of communication with hypervisors. Hypercalls rely on a special instruction that can be executed within the guest to trigger a direct VM-Exit into the hypervisor. There is no unified hypercall calling convention. Each hypervisor can implement a hypercall interface that relies on parameters passed through any number of registers (which may refer to memory addresses). While hypercalls follow the same trap-and-emulate virtualization model, the flexibility of the interface often that they can be decoded and handled more rapidly than PIO/MMIO. For PIO/MMIO, the hypervisor has to locate and disassemble the faulting instruction to identify the source and destination operands (discussed further in § 3.2.1). Conversely, the semantics of hypercalls are hard-coded.

Direct Memory Access (Unlike PIO/MMIO and hypercalls) is performed without the direct involvement of the CPU (there is no "DMA" instruction). Usually, when the VM wishes to signal to the hypervisor that some data is available for DMA transfer, it communicates the location of the data in physical memory to the device using PIO/MMIO/Hypercalls. The hypervisor can then, independently, access the data in the guest's memory. From the perspective of the vCPU, it simply provides the virtual-device with a pointer to some data, by writing to a PIO/MMIO register or via a hypercall.

Notably, DMA access patterns can be arbitrarily complex. For example, DMA ring buffers are a common paradigm, where the VM sets up a "ring" data-structure that stores metadata and pointers to individual buffers. Virtual-devices access both the ring and the buffers using DMA. To address this complexity, hypervisor-fuzzers have relied on grammars and, more-recently, API-hooking to address the problem of the complex DMA input-space. By hooking DMA APIs, fuzzers gain information about DMA-access patterns which can be used to automatically generate grammars (ViDeZZo [26] and MundoFuzz [29]) or to directly fuzz the data returned through DMA APIs (Morphuzz [6] and V-Shuttle [32]).

3 HyperPill Approach

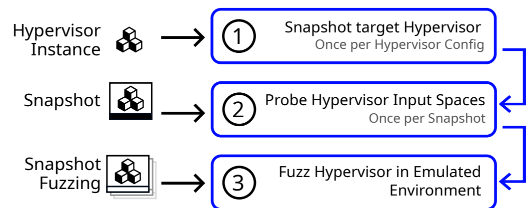

Figure 1: High-level overview of HyperPill's approach.

Our HyperPill design enables effective fuzzing of arbitrary hypervisors without any target-specific configuration. HyperPill's approach centers around (1) collecting a snapshot of a hypervisor, (2) inspecting the snapshot to identify the fuzzing input-spaces, and, finally, (3) fuzzing the hypervisor. Our approach does not require any manual engineering for individual hypervisors or composition of device-specific

grammars. Instead, HYPERPILL leverages the fundamental hardware-virtualization interface, which must be used by all hypervisors that take advantage of hardware-accelerated virtualization. HYPERPILL can fuzz both type-1 (ones that run on bare-metal) and type-2 (ones that run as applications managed by a general-purpose OS) hypervisors. In summary, our fuzzer is the first *fully automatic gray-box* hypervisor-fuzzer that leverages architectural-state instead of grammars or configurations to fully explore *all major interfaces exposed to guest VMs* by *any hypervisor*. In this section, we describe HYPERPILL's approach. § 4 explains how we bring together these innovations in our implementation of HYPERPILL.

Threat Model. In HYPERPILL's threat-model, an attacker has complete control over the code running in a guest managed by a potentially vulnerable hypervisor. The attacker seeks to compromise the isolation guarantees provided by the hypervisor to gain control over other VMs on the system, or the hypervisor itself. To this end, the attacker can initiate arbitrary PIO/MMIO/Hypercalls that are handled by the hypervisor. Furthermore, the attacker fully controls the data the hypervisor reads from guest-memory e.g., via DMA. Additionally, the attacker can seek to perform Denial-of-Service attacks, by disrupting the operation of the hypervisor and subsequently the neighboring VMs. In this work, we exclude side-channels from the attack-surface.

① Making a Snapshot of the Hypervisor

As HYPERPILL is designed to target arbitrary hypervisors, and hypervisors are known to be particularly stateful targets, HYPERPILL requires a robust and generic mechanism for resetting state between inputs. While prior approaches such as Morphuzz and ViDeZZo relied on implementation-specific solutions to add instrumentation and state-resetting to hypervisor source-code, HYPERPILL does not have access to such source-code-dependent features [6, 26]. Furthermore, HYPERPILL targets complex hypervisor implementations, such as Hyper-V where the hypervisor consists of components spread over the host kernel, a guest kernel, and a process running within the guest. To this end, HYPERPILL relies on full-system memory and register snapshots to reliably capture the state of the entire environment hosting the hypervisor. This approach is independent of any hypervisor implementation details such as type-1/type-2, host OS, source availability, etc. By collecting the snapshot in a separate offline stage, we ensure that input-space enumeration (Stage ②) only needs to be performed once, and that fuzzer instances are not delayed by costly hypervisor-initialization (which is even more costly in an emulated environment).

3.1.1 Timing the Snapshot

As described in § 2, hypervisors initialize the necessary data-structures to configure a guest, before invoking `vmenter` to launch the guest. Eventually, a guest or hardware-triggered event causes execution control to be returned to the hypervisor. If the event was guest-triggered (e.g., a virtual MMIO read or hypercall), the hypervisor must emulate the corresponding functionality (e.g., by running virtual-device code). Software vulnerabilities in the hypervisor are most likely to manifest themselves during this emulation. As such, HYPERPILL makes a snapshot of the hypervisor just as the guest traps into the host and passes execution control to the hypervisor for emulation of the VM-Exit. Specifically, HYPERPILL invokes a custom hypercall from the guest to make a snapshot. As such the next instruction executed when HYPERPILL loads/resumes the snapshot is the first instruction of the hypervisor's trap-and-emulate code. In § 4.1 we detail HYPERPILL's implementation of this snapshotting technique.

② Probing the Hypervisor's Input-Space

In this section, we explain how HYPERPILL uses static and dynamic techniques to precisely enumerate the PIO/MMIO and DMA input-spaces. Note that due to lack of a unified hypercall API, HYPERPILL does not perform any offline hypercall enumeration. Instead, we tailored the fuzzing-process, itself, to effectively explore the hypercall input-space (described further in § 3.2.4).

3.2.1 Injecting IO into the Hypervisor

To perform active probing of PIO/MMIO regions, HYPERPILL needs the ability to inject PIO/MMIO operations into the snapshotted hypervisor. Leveraging the hypervisor snapshot, HYPERPILL can inject arbitrary VM-Exits into the hypervisor, by performing slight modifications to the guest-memory. To this end, HYPERPILL modifies the fields within the VMCS that the CPU uses to communicate information about the VM-Exit to the hypervisor. The address of the currently configured VMCS structure (set using the `vmxset` instruction), is included in the CPU state. As the format of the VMCS is well-documented in architecture manuals, HYPERPILL can dereference the address to probe details about the active VM. Thus, even though the snapshotted VMCS represents a VM-Exit for HYPERPILL's custom hypercall, HYPERPILL can adjust the snapshot to reflect any other VM-Exit, such as one related to PIO or MMIO, or a different hypercall. Specifically, since hypervisors often disassemble the faulting instruction, when HYPERPILL injects an MMIO or PIO instruction, HYPERPILL overwrites the customized hypercall instruction, with one corresponding to the VM-Exit (a `mov` for MMIO or an `in/out` for PIO).

With the capability to inject arbitrary VM-Exits, HYPERPILL can generically explore the PIO/MMIO/Hypercall interfaces exposed by any hypervisor.

3.2.2 Enumerating MMIO and PIO

Leveraging the ability to inject arbitrary VM-Exits, HYPERPILL relies on probing to identify MMIO/PIO regions. Semantically, PIO accesses are similar to MMIO accesses, however, they leverage a relatively small 16-bit address-space (independent of guest-memory) and PIO-specific x86 instructions, i.e., `in/out`. However, MMIO ranges are adjacent to standard RAM addresses in the guest's vast (e.g., 48-bit) physical address-space. Fortunately, HYPERPILL can quickly eliminate the vast-majority of physical-addresses from its list of MMIO candidates, by inspecting the guest's EPT. As described in § 2, the EPT maps guest-physical-addresses (GPAs) to host-physical-addresses (HPAs). There are two mechanisms (EPT Violations and Misconfigurations) available to hypervisors for trapping MMIO accesses. Both mechanisms rely on specially configured pages in the guest's EPT. HYPERPILL reads the EPT pointer field from the VMCS and walks the EPT. HYPERPILL records all guest-physical-address ranges that trigger EPT Violation and Misconfiguration VM-Exits upon access and treats these ranges as potential MMIO ranges. These ranges do not necessarily all trigger behavior associated with MMIO emulation. For example, some EPT Violation ranges may simply be gaps in the guest's physical address space that are not associated with any RAM or MMIO devices (e.g., the PCI hole). To handle access to such ranges, the hypervisor generally ignores the access (or injects a page-fault into the guest). Thus, after identifying a list of candidate regions, HYPERPILL enters an active probing stage.

To enumerate PIO addresses and identify actual MMIO ranges from the list of candidate MMIO ranges, HYPERPILL relies on a multifaceted introspection approach. We identified that the majority of candidate regions are not associated with any virtual-device code. Accessing such ranges results in virtually identical hypervisor behavior (e.g., a rapid re-entry into the guest, after setting up a physical-page mapping or injecting a page-fault exception). As such, we identify key execution characteristics that identify active PIO/MMIO ranges. HYPERPILL iterates over potential PIO and MMIO addresses, injects the corresponding VM-Exits and tracks these characteristics to determine whether each address is associated with a virtual-device. Specifically, HYPERPILL tracks:

- *Instruction Count*. The number of instructions executed by the hypervisor to handle the access is a reliable indicator of PIO/MMIO ranges. Non-interesting accesses typically have similar and small instruction counts. However, MMIO/PIO accesses require executing device-specific code and have large instruction counts due to executing device emulation code.
- *Unique Program Counter Values*. PIO/MMIO virtual-devices feature device-specific code to emulate accesses. As such, accesses to different device-ranges result in new code executed (previously-unseen program-counter values). HYPERPILL marks ranges that execute unique program counters (not executed by accesses to any other range) to identify whether a range is associated with a virtual-device.
- *Entry to User-Space*. Hypervisors such as KVM and Hyper-V execute their PIO/MMIO virtual-device code in user-space processes. As such, while all VM-Exits are initially handled in privileged ring 0, accesses to actual PIO/MMIO regions cause the hypervisor to exit into the virtual-device worker process (unprivileged ring 3). For each PIO/MMIO candidate, HYPERPILL tracks the execution of the hypervisor to determine whether it executes the `sysret` instruction to exit into user-space.

Combining the lists of regions sensitive to these feedbacks, HYPERPILL constructs a final list of active PIO and MMIO regions which can be used to fuzz these interfaces.

3.2.3 Enumerating Guest-Memory (DMA)

HYPERPILL's memory snapshot contains all the hypervisor's physical memory, including the physical frames that the hypervisor allocates for itself, for the VM, and unallocated memory. While handling MMIO/PIO/Hypercall VM-Exits, the hypervisor can access data in guest-memory (e.g., DMA). As such, the guest's memory is an essential component of the virtualization attack-surface. HYPERPILL statically classifies potential hypervisor guest-memory-accesses into two categories.

- *Direct Memory Access (DMA).* Accesses that are used for bulk transfers of data by virtual-devices (e.g. reads from ring-buffers kept in the guest's memory to rapidly transfer data to devices such as network-cards).
- *Instruction Emulation (IE).* Accesses that are used to disassemble an instruction that caused a VM-Exit. For example, when a guest triggers an MMIO read, the hypervisor must locate and disassemble the corresponding faulting instruction to determine the destination register and size of the read. The GVA of the PIO/MMIO instruction and the guest-page-tables used to identify and disassemble the instruction should be set up accordingly.

With this information, HYPERPILL can differentiate memory-accesses between ones that can potentially contain attacker-controlled data (DMA), and ones that should reflect the guest's faulting instruction. To do this, HYPERPILL identifies and categorizes the frames in the snapshot that belong to the guest and can contain attacker-controlled data. HYPERPILL performs a page-table walk over the EPT to identify all HPAs that are mapped to the guest. In practice, HYPERPILL performs the DMA-categorization and enumeration of candidate MMIO regions during a single walk of the EPT. Additionally, HYPERPILL walks the guest's page-tables to determine

which guest-physical-pages contain the last-executed VM instruction and the page-table entries required to locate it.

As a result of these page-table walks, HYPERPILL classifies all memory frames in the hypervisor snapshot as:

1. Frames that are not mapped in the guest.
2. Frames that are dedicated to the guest, and are not part of the guest's page-tables.
3. Frames that are dedicated to the guest and are part of the guest's page-tables.

The IE surface is largely independent of the rest of the trap-and-emulate cycle. Thus, to increase fuzzing performance, HYPERPILL fuzzes DMA independent of the IE space. When the hypervisor reads from memory, HYPERPILL first identifies whether the memory belongs to the guest. If the read targets guest-memory, HYPERPILL further determines whether the read targets the guest's page-table structures, or the frame that contains the last-executed guest instruction. HYPERPILL only treats the read as a virtual DMA access and fills the corresponding location in memory with fuzzed-data if the access is not classified as IE. *With this capability,* HYPERPILL *can fully explore the DMA interface exposed by any hypervisor.*

③ Fuzzing the Hypervisor

In the prior sections, we described how HYPERPILL collects and analyzes a hypervisor snapshot to identify the active PIO/MMIO/DMA ranges. After completing this offline learning stage, HYPERPILL leverages the snapshot to fuzz the hypervisor. To this end, HYPERPILL "restores" the snapshot (i.e., loads the saved memory and register state, and prepares to resume the execution). Then, based on the fuzzing input HYPERPILL injects a PIO/MMIO or Hypercall VM-Exit, as described in § 3.2.1. HYPERPILL resumes the hypervisor. Unless there is a crash, the hypervisor handles the VM-Exit and resumes the VM, using the `vmresume` instruction. However, HYPERPILL intercepts the `vmresume` and immediately injects another VM-Exit based on the fuzzer-input, instead of executing any guest-code. Thus, fuzzing-time is fully dedicated to injecting new VM-Exits into the hypervisor and running the hypervisor's emulation code on fuzz-inputs. HYPERPILL repeats this process until the fuzzer-input has been fully executed (no more bytes remain in the input). Hypervisors are stateful targets and often require multiple IO operations to execute certain code. To reflect this statefulness, HYPERPILL's inputs are composed of sequences of IO operations executed from a "clean" snapshot state. This ensures that the fuzzer can rapidly receive and react to runtime feedback to uncover new parts of the hypervisor's code. Then, HYPERPILL resets the hypervisor's state from the original snapshot, before executing the next fuzzer input.

3.2.4 Fuzzing Hypercalls

As described in § 2, modern CPUs support hypercalls which enable fast and flexible requests of services from the hypervisor by the guest. Hypercalls are commonly used to provide an interface to high-performance paravirtual-devices. As such, they are a critical part of the virtualization input-space. Semantically, hypercalls behave similarly to PIO/MMIO accesses: the guest invokes a `vmcall` instruction that immediately traps into the host. However, unlike PIO/MMIO, there is no general hypercall ABI that is shared by all hypervisors. For example, KVM uses the contents of guest register RAX to identify the service requested by the hypercall, while Hyper-V relies on RCX for the same purpose. Furthermore, since there is no defined interface, hypervisors can use arbitrary registers to communicate additional hypercall parameters. Depending on the service requested, Hyper-V can interpret the contents of registers RDX, R8, XMM0-5 as guest-arguments. More generally, essentially any register can be used to pass data through a hypercall. This creates a challenge, as fuzzing the contents of *all* registers for each hypercall would result in an unreasonably large input state space, requiring hundreds of input bytes for each call and, worse, a majority of these bytes would be wasted on unused registers that the fuzzer would spend time mutating. Most prior fuzzers do not cover the hypercall input-space. Furthermore, the fuzzers that aspire to fuzz hypercalls (e.g. Hyper-Cube [38]) handle them by encoding hypervisor-specific expert knowledge about hypercall-ids and the corresponding registers used to pass arguments. These approaches require manual work and constant upkeep, since hypervisors continue to add new hypercalls.

Instead, HYPERPILL leverages powerful introspection capabilities to efficiently fuzz arbitrary hypercall implementations. Unlike, the PIO/MMIO/DMA spaces which are enumerated offline in stage ②, HYPERPILL dynamically constructs valid hypercalls by leveraging runtime feedback from fuzzing. HYPERPILL's approach is based on the observation that hypercall argument registers impact the hypervisor's execution path. As such, the contents of these registers appear in numeric comparisons during the trap-and-emulate process (e.g. bounds-checks and hypercall-handler selections). By default, HYPERPILL fills all guest-registers with randomly generated data from a pseudo-random-number generator (pRNG), rather than the fuzzing-input. As the hypervisor handles the `vmcall`, HYPERPILL inspects the operands of all comparison instructions. If any of the operands is found within the randomly generated register data, HYPERPILL records that subsequent mutations of the input should set the contents of the corresponding register directly from the fuzzer-input (in § 4.3.1 we explain how this functionality is implemented). HYPERPILL automatically identifies registers that affect hypercall execution, without any prior knowledge of the hypervisor-specific hypercall-API and without wasting fuzzer-bytes on unused registers (wasting potentially hundreds of bytes to fill

unused registers, severely hampers fuzzing efficiency) Our novel runtime-feedback-based approach is inspired by AFL++ CmpLog and RedQueen [2, 11]. However, instead of only solving obstacles, such as magic-bytes, HYPERPILL uses the feedback to transparently infer hypercall calling-conventions.

4 Implementation

As described in § 3, HYPERPILL collects a snapshot of a hypervisor, inspects the snapshot, and finally fuzzes the hypervisor. We implemented these three stages as independent components of HYPERPILL. HYPERPILL-Snap collects a precise snapshot of a hypervisor at the exact moment it traps a guest-access for emulation. HYPERPILL-Inspect locates the guest's memory within the snapshot and enumerates all the active PIO and MMIO ranges. HYPERPILL-Fuzz leverages the results of the prior stages to fuzz the hypervisor.

4.1 HYPERPILL-Snap

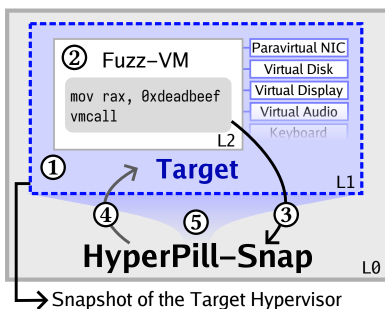

Figure 2: HYPERPILL-Snap snapshots the target hypervisor.

HYPERPILL relies on a full-system snapshot of the hypervisor (the target), hosting a guest (the attacker-controlled VM, in our threat-model). To collect this snapshot, HYPERPILL relies on nested-virtualization. HYPERPILL-Snap is, itself, a hypervisor (a customized version of KVM), designed to run as *L0* (*L0* or Layer 0 is the standard name for the outermost hypervisor, L1 refers to the VM executed by L0 and L2 refers to a nested VM managed by L1). ① HYPERPILL-Snap runs the target hypervisor, in a nested configuration, as an *L1* guest. ② The analyst launches a guest within the target hypervisor (configured with desired virtual-devices), creating the *L2* Fuzz-VM. The x86 ISA supports nested virtualization and dictates that all L2 VM-Exits first pass through L0, before being injected into L1. As such, HYPERPILL-Snap can inspect L2 exit-reasons before injecting them into the target. In the default case, HYPERPILL-Snap adheres to KVM's default behavior and simply forwards all VM-Exits into L1. ③ When HYPERPILL-Snap encounters a hypercall with a special identifier (the value `0xdeadbeef` stored in register RAX), ④ it likewise injects it into the target by modifying the $VMCS_2^1$ (the VMCS configured by L1 for the L2 Fuzz-VM). However instead of resuming the target to allow it to handle the VM-Exit, HYPERPILL-Snap pauses execution of the target hypervisor and ⑤ collects a snapshot of L1's entire memory and CPU state. Thus, to trigger the snapshot, the analyst simply needs to invoke the `0xdeadbeef` hypercall from the L2 guest. We take the snapshot once the L2 OS (Linux) has had the chance to fully boot and initialize all drivers and virtual-devices. The resulting snapshot includes the L1 (i.e., target hypervisor's) physical memory (which includes L2's physical memory), the register state, and internal CPU state, such as the $VMCS_2^1$ pointer.

Resuming the snapshot will result in L1 receiving the `0xdeadbeef` hypercall VM-Exit, and attempting to handle it. Thus HYPERPILL-Snap achieves our goal of collecting a snapshot of the target hypervisor at the exact moment it is about to handle a VM-Exit.

4.2 HYPERPILL-Inspect

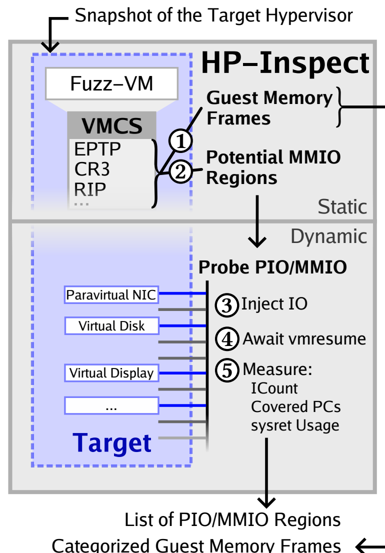

Figure 3: HYPERPILL-Inspect analyzes the hypervisor's PIO/MMIO/Memory input-space.

As described in § 3, HYPERPILL's snapshot analysis features both static and dynamic components. HYPERPILL-Inspect performs an initial static analysis of the snapshot to identify potential L2 guest's MMIO ranges and categorize each memory page contained in the snapshot. Specifically, HYPERPILL-Inspect dereferences the VMCS pointer (part of the snapshotted CPU state), and loads the Extended Page Table (EPT) pointer from the VMCS. HYPERPILL-Inspect walks the EPT to identify the memory frames that are allocated to the L2 guest (L2-guest-memory frames). Additionally, HYPERPILL-Inspect identifies candidate MMIO pages, which are configured within the EPT to cause Misconfig or Violation VM-Exits. HYPERPILL-Inspect then loads the value of the

L2 guest's CR3 register from the $VMCS_2^1$, which stores the location of the L2 guest's page-table (see § 3.2.3). By walking the L2 guest's page-table, HYPERPILL-Inspect further categorizes allocated L2-guest-memory frames as either part or not part of the L2 guest's page-table.

For PIO/MMIO enumeration, HYPERPILL-Inspect relies on dynamic feedback from the hypervisor (e.g., instruction-counts or covered PCs as described in § 3.2.2). HYPERPILL-Inspect relies on our custom execution environment, called MELTER, to execute the snapshot and collect dynamic feedback. MELTER is a full-system emulation-based fuzzer, based on Bochs [5]. Bochs is an open-source x86-64 emulator. Bochs has comprehensive support for VT-x, which is crucial, since HYPERPILL is designed to fuzz hypervisors, that rely extensively on VT-x. We use this execution environment in HYPERPILL-Inspect (and reuse it in HYPERPILL-Fuzz). Upon startup, MELTER loads the target hypervisor memory and register snapshot taken in stage (1) into the emulation context. MELTER features two modes: *enumeration* and *fuzzing*. HYPERPILL-Inspect only relies on the *enumeration* mode.

4.2.1 Enumerating PIO and MMIO Ranges

HYPERPILL-Inspect relies on dynamic probing to identify active PIO/MMIO ranges. As mentioned in § 3.2.1, HYPERPILL can inject arbitrary IO operations, by modifying the VMCS and guest memory, before resuming the snapshot. To identify active PIO/MMIO regions, we observe the property that accesses to active PIO/MMIO regions cause the hypervisor to execute code that implements virtual-device functionality. Since HYPERPILL-Inspect has instruction-level insight into the code executed by the hypervisor, it can easily identify port and memory accesses that execute unique code. For PIO, HYPERPILL-Inspect iterates over each address in the 16-bit port-address space and injects a PIO write (simulating an `outb` instruction). For MMIO, HYPERPILL-Inspect iterates over each page within the candidate MMIO ranges identified during the EPT walk and injects an MMIO write (simulating a `mov` instruction with the candidate page as the destination). After resuming the snapshot, HYPERPILL-Inspect waits for the hypervisor to handle the injected VM-Exit and resume the L2 guest using the `vmresume` instruction. In between each injected probe, MELTER restores the state of the hypervisor from the initial snapshot. For each candidate PIO/MMIO address, HYPERPILL-Inspect records (1) The number of instructions executed before the L2 guest was resumed (2) The executed program-counters (3) Whether the hypervisor entered user-space. HYPERPILL-Inspect merges this information to output the final list of active PIO/MMIO regions. Note that since MELTER is based on Bochs, it is straightforward to add the necessary instrumentation to the emulator to collect this information. Our probing technique guarantees that HYPERPILL only fuzzes subregions of the PIO/MMIO input-space that cause variations in the hypervisor's behavior. Crucially, unlike past approaches, HYPERPILL-Inspect does not rely on any instrumentation of APIs responsible for mapping PIO/MMIO regions.

HYPERPILL-Inspect only needs to run once per snapshot collected by HYPERPILL-Snap (i.e. once per target hypervisor configuration). It outputs a list of identified and categorized guest memory frames and active PIO/MMIO regions.

4.3 HYPERPILL-Fuzz

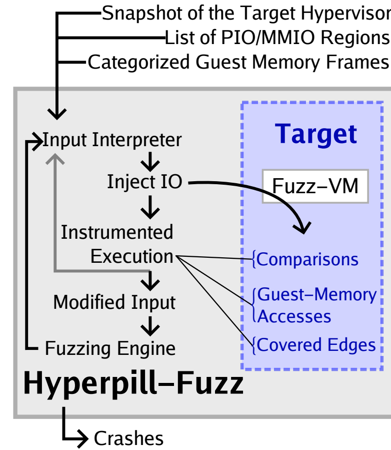

Figure 4: HYPERPILL-Snap fuzzes the target hypervisor.

HYPERPILL-Fuzz leverages the snapshot collected by HYPERPILL-Snap and the enumerated PIO/MMIO/DMA input-spaces from HYPERPILL-Inspect to efficiently fuzz the hypervisor using the *fuzzing* mode of MELTER. HYPERPILL-Fuzz relies on an input-interpreter, which splits a single fuzzer-input into a sequence of IO operations. The input-interpreter features five core operations.

`in[b,w,l](addr)` perform a PIO read
`out[b,w,l](addr, val)` perform a PIO write
`read[b,w,l,q](addr)` perform an MMIO read
`write[b,w,l,q](addr, val)` perform an MMIO write
`hypercall(reg_bitmap, *vals)` perform a hypercall

For these operations, the **b**`yte`, **w**`ord`, **l**`ong`, **q**`uad` suffixes specify the size of the access.

For each of these operations, HYPERPILL-Fuzz reads the address/value (if needed) and injects the corresponding type of VM-Exit into the L2 guest (following the procedure outlined in § 3.2.1). HYPERPILL-Fuzz resumes the snapshot and waits for the hypervisor to handle the injected IO operation. As soon as the hypervisor attempts to re-enter the guest by using the `vmresume` instruction, HYPERPILL-Fuzz injects the next operation in the input. Once HYPERPILL-Fuzz has executed the entire input, it restores the hypervisor's initial

memory and register state from the snapshot. Thus, there is no state-leakage between individual fuzzer inputs.

Two types of IO: *Hypercalls* and *DMA* require functionality beyond the simple opcode interpreter. We cover each of these special cases individually.

4.3.1 Hypercalls

As discussed in § 3.2.4, the `vmcall` instruction does not specify any fixed ABI to pass data to the hypervisor. Furthermore, different hypercalls can feature different APIs. A hypervisor can use any number of implementation-specific registers to receive inputs from the guest (e.g., one register can contain the hypercall-id and another one can contain a DMA pointer to a structure containing hypercall parameters). Initially, HYPERPILL does not use any fuzzer-provided values to fill registers, when executing hypercalls. Instead, HYPERPILL fills all registers with random data from a pRNG. During the emulation of a hypercall, if any of the random-values occur as arguments of `cmp` instructions, HYPERPILL-Fuzz identifies the register that contained the random-value as significant. HYPERPILL-Fuzz *dynamically modifies the* `vmcall` *operation within the input testcase* to explicitly specify that the corresponding register-value should be specified by the fuzzer, and *overwrites* the input-interpreter's "hypercall" operation to reflect the pRNG-generated value. More specifically, the "hypercall" operation in fuzzer inputs, is followed by a 32-bit register-bitmap which specifies whether any of the general-purpose and 16 XMM registers should be filled with fuzzer-data. The bitmap is followed by the values of the fuzzer-specified registers. If new significant registers are identified, the fuzzing input is modified. If the input achieves new coverage, the *modified* input is stored in the fuzzer's corpus, for future mutation. To support this functionality, we modified the fuzzing backend (libFuzzer) to support (1) Modifying fuzzer inputs during execution (so the hypercall operation can be modified if a significant register is identified) (2) Preventing fuzzer mutations from affecting parts of the input (so the mutations do not overwrite the 32-bit register-bitmap, which should only be modified if new significant registers are found). This process is highlighted in Fig. 4: after executing a testcase, *a modified input* is returned to the fuzzing-engine, for further mutation.

4.3.2 DMA

Unlike PIO, MMIO, and Hypercalls, DMA is a reaction to other IO operations, without the virtual CPU's direct involvement in the data transfer. In the context of virtualization, the hypervisor can initiate DMA at any point during its execution, by accessing some of the guest's physical memory. Thus, HYPERPILL-Fuzz cannot implement DMA as a simple opcode, since there is no way to directly "inject" it through a VM-Exit. Instead, DMA occurs implicitly as a side effect of the other IO operations which inform the hypervisor that it should initiate a DMA transfer. Since HYPERPILL does not feature any device-grammars, it is not aware, a priori, which IO operations will cause DMA. State-of-the-art open-source hypervisor-fuzzers such as Morphuzz and ViDeZZo identified that the DMA input-space can be effectively fuzzed by adding strategic fuzzing hooks to the hypervisor source-code, targeting the DMA access APIs [6, 26]. When the hypervisor initiates a DMA access, the hook fills the corresponding region of guest-memory with fuzzer-provided data *just-in-time*. This strategy elides the need for prior information about which IO operations initiate DMA accesses and the size/format of the DMA data.

Unlike these prior approaches, HYPERPILL cannot directly modify DMA access APIs. However, since HYPERPILL-Fuzz runs the hypervisor in an emulator, and has precise information about which memory frames are dedicated to the guest's memory, it can intercept the hypervisor's DMA accesses. After injecting a PIO/MMIO/Hypercall into the hypervisor, HYPERPILL tracks all the hypervisor's memory accesses. If the hypervisor reads from memory marked as DMA by HYPERPILL-Inspect, HYPERPILL intercepts the access and fills it with data from the current fuzzer input. This DMA-access tracking is completely oblivious to the inner workings of the hypervisor and the structure of its DMA APIs. Furthermore, unlike prior approaches which require manual modifications to hypervisor source-code, HYPERPILL does not risk missing DMA accesses due to incomplete API instrumentation – DMA is tracked at the level of (emulated) physical memory. In § 5, we will see that this enables HYPERPILL to outperform fuzzers even when they have access to the DMA API source-code.

5 Evaluation

We evaluate HYPERPILL's fuzzing capabilities to answer the following research questions:

- **RQ1** Does HYPERPILL trigger complex coverage, and does it compare favorably against state-of-the-art fuzzers that target open-source hypervisors?
- **RQ2** Can HYPERPILL discover new bugs in a wide range of hypervisors without hypervisor-specific configuration or modifications?
- **RQ3** Can HYPERPILL find new bugs even in virtual-devices that have already been fuzzed extensively?
- **RQ4** Can HYPERPILL rediscover previously known bugs in hypervisors?
- **RQ5** How does HYPERPILL's emulation-based execution performance compare to that of other fuzzers?

5.1 Experimental Setup

We conducted our experiments for QEMU version 8.0.0 (released April 2023), Hyper-V shipped with Windows 10, and macOS Virtualization Framework shipped with macOS Ventura (version 13). We selected these three hypervisors as

they are the default hypervisors that ship with the three most-popular consumer operating-systems. As these hypvervisors run the gamut from open-source (KVM) to closed-source (macOS, Hyper-V) , different programming languages (C: KVM, C++: Hyper-V, Unknown: macOS), and availability of debugging symbols (KVM: full, Hyper-V: some, macOS: none) this selection showcases the broad applicability of HYPERPILL.

We performed a coverage comparison with the two most recently updated fuzzers (i.e., Morphuzz and ViDeZZo) that target QEMU and have their source-code available [6, 26]. For Morphuzz, we use the upstream version maintained in the QEMU repository. For ViDeZZo, we adjusted the code to support QEMU v8.0.0. We conducted our evaluation over 12 devices (i.e., four block devices, three graphics adapters, three network adapters, and two USB controllers). For each category, we selected the most complex devices estimated by lines of code in the QEMU repository. Nine of these devices overlap with the ones evaluated in prior works [6, 26]. We fuzzed each QEMU device for 24 hours on 8 cores (8 fuzzing workers per device) and repeated these experiments five times. Similarly, we fuzzed our Hyper-V and macOS snapshots for 24 hours on 8 cores. We performed all experiments on university cluster servers equipped with 2x Intel Xeon Gold 6132 @ 2.60 GHz with memory ranging from 192 GB to 512 GB (memory was not a limiting factor for any fuzzer instance).

5.2 Coverage

We compared HYPERPILL's coverage performance to Morphuzz and ViDeZZo on QEMU, a hypervisor well suited to comparing fuzzers due to its popularity and broad suite of virtual-devices [6, 17, 26, 29, 37, 38]. For all fuzzers we instrumented the qemu-binary with LLVM source-coverage [43]. Since HYPERPILL is based on a snapshot-fuzzing approach, we added instrumentation to MELTER to prevent the fuzzer from resetting LLVM's coverage-bitmap after each input. Tab. 2 presents our coverage results to answer **RQ1**. For 10 out of 12 devices, HYPERPILL outperformed Morphuzz and ViDeZZo. HYPERPILL achieved the best coverage across all paravirtual VIRTIO devices, which are used in security-sensitive cloud environments. Furthermore, HYPERPILL was the only fuzzer capable of fuzzing the QXL device, which no prior fuzzer targets due to QXL's reliance on the multi-threaded external libspice library. HYPERPILL performed well both for devices that require no DMA interactions (cirrus) and for devices that rely on them extensively (e.g. xHCI—the most advanced USB controller implemented in QEMU).

		Morphuzz		ViDeZZo		HYPERPILL		
		12 Cores 24 Hours						
Device	⊡	Branch Coverage (Executions/Second)						Bug
Block								
ahci	✓	42.43%	(25.68)	30.42%	(562.24)	**45.90%**	(26.18)	✓
nvme		29.12%	(23.82)			**36.44%**	(14.45)	✓
sdhci	✓	69.81%	(22.98)	**72.37%**	(107.22)	66.85%	(32.34)	
virtio-scsi	✓	27.96%	(23.83)	11.73%	(217.28)	**48.83%**	(51.68)	
Display								
cirrus		88.10%	(19.06)	83.42%	(138.78)	**88.67%**	(32.18)	✓
qxl	✓					**59.68%**	(26.96)	✓
virtio-gpu	✓	24.37%	(26.21)	2.77%	(222.42)	**45.52%**	(36.53)	✓
Networking								
e1000e	✓	50.27%	(24.83)	41.52%	(53.04)	**55.99%**	(42.22)	✓
igb	✓	29.73%	(25.63)			**35.93%**	(60.85)	✓
vmxnet	✓	50.75%	(27.01)	19.64%	(145.73)	**56.89%**	(48.14)	
USB								
ehci	✓	73.76%	(24.58)	**74.38%**	(177.08)	73.32%	(10.46)	
xhci	✓	55.54%	(28.83)	29.25%	(1061.36)	**76.64%**	(69.26)	✓
Geo. Mean		45.20%	(24.65)	28.00%	(203.07)	55.45%	(33.20)	

Table 2: HYPERPILL 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 execution per second. The ⊡ column indicates whether the device accesses guest-controlled memory via coarse-grained DMA APIs that return indexable pointers to guest-memory. The final column indicates whether we found bugs, for each device.

For 8 of the 10 devices that rely on coarse-grained DMA APIs that return pointers to large regions of guest-memory (see ⊡ column in Tab. 2), HYPERPILL achieves the highest coverage. We attribute this to the fact that, unlike prior approaches, HYPERPILL can precisely hook atomic DMA-accesses, rather than having to pre-fill large DMA-pointer regions.

We identified two problems resulting in low ViDeZZo coverage for some evaluated devices. First, ViDeZZo's intra-message annotation is not well-supported, for certain devices (virtio-scsi/virtio-gpu) and is missing for igb and nvme. Additionally, since ViDeZZo does not reset state between inputs, ViDeZZo requires discovered bugs to be fixed, so that the fuzzer can continue making progress.

Note that no fuzzer achieves 100% coverage for any device. This is due to the fact that devices contain code that is unreachable by fuzzing, e.g., initialization code. Furthermore, devices contain options which essentially create dead-code regions which are only accessible with a particular device configuration (e.g., virtio-gpu provides 44 configuration options, alone). In our experiments, all three fuzzers were started with identical configuration options.

RQ1: HYPERPILL outperforms the coverage of previous fuzzers across a wide range of complex virtual-devices.

5.3 Bug-Finding

In total, HYPERPILL identified 26 new bugs (11 in QEMU, 9 in Hyper-V, and 6 in macOS Hypervisor-Framework, detailed in Appendix Tab. 3) in all the hypervisors it fuzzed and across all device-categories, which answers **RQ2**. Of these 11 are bugs found in QEMU, which has been fuzzed extensively on OSS-Fuzz (with Morphuzz) since 2020, which answers **RQ3**. All bugs were identified during the 24-hour fuzzing campaigns. Here, we detail four bugs found by HYPERPILL since they are representative of HYPERPILL's focus on gener-

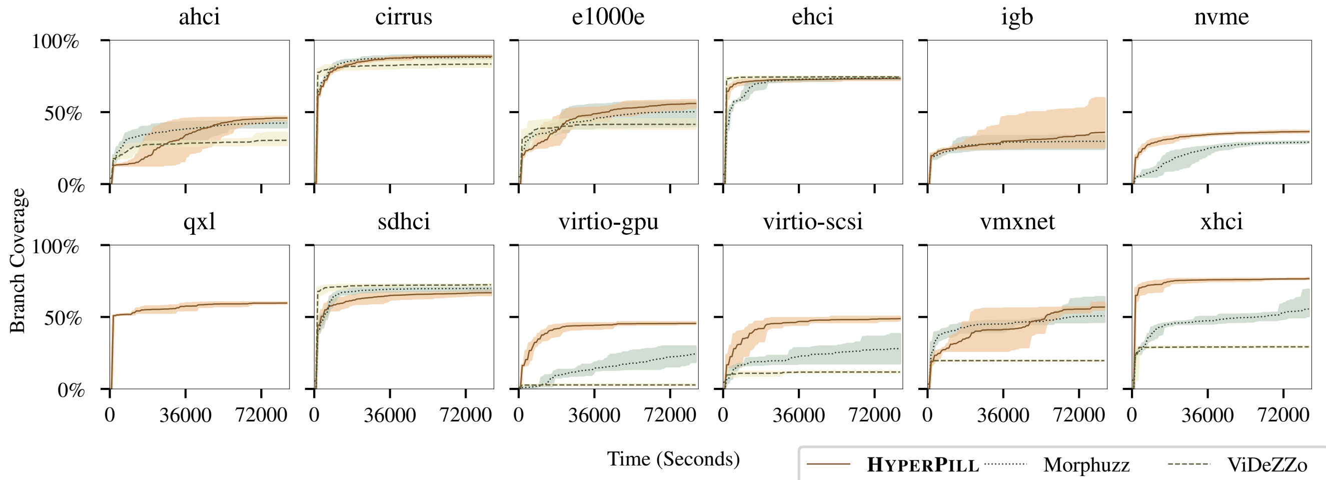

Figure 5: Branch coverage over 24 hours fuzzing 12 devices. The shaded regions represent the maximum/minimum coverage achieved by the corresponding fuzzer, across five experiments.

QEMU
- Arbitrary memory-access in e1000e_start_xmit
- Heap-overflow in usb_mouse_poll
- Heap-overflow in virtqueue_alloc_element
- Heap-overflow in qxl_cookie_new
- Heap-overflow in igb_tx_pkt_switch
- Out-of-bounds memory access in nvme_process_sq
- Out-of-bounds memory access in nvme_io_mgmt_send
- DoS via arbitrary-sized allocation in qxl
- DoS via arbitrary-sized allocation in virtio_gpu
- DoS in process_ncq_command
- DoS in icmp_input

Hyper-V
- Heap-corruption in EthernetCard::HandleTransmitSetupFrame
- Abort in EthernetCard::PollForTransmitDataTimer
- Abort after IdeChannel::EnlightenedHddCommand
- EthernetCard::SetupEthernetCardModeFromRegisters
- Out-of-bounds write in GuestStateAccess::SetDeviceInfo
- Abort after PitDevice::NotifyIoPortRead
- Abort in I8042Device::HandleCommand
- Abort after HvCallDetachDevice
- Abort after HvCallGetGpaPagesAccessState

macOS Virtualization Framework
- Memory-privilege violation in xHCI
- Out-of-bounds write in virtio-gpu
- Out-of-bounds write in virtio-audio
- Out-of-bounds access in virtio-block
- Out-of-bounds access in virtio-console
- Out-of-bounds access in virtio-net

Table 3: New bugs found by HYPERPILL

ically fuzzing arbitrary hypervisors across the entire spectrum of input spaces, while producing complex inputs that require intricate interactions with individual hypervisor components.

QEMU: Heap-Buffer-Overflow in virtio-scsi. QEMU's VIRTIO devices are security-sensitive paravirtual-devices that are actively used for cloud-applications. QEMU's virito-scsi device has been fuzzed on OSS-Fuzz since 2020. HYPERPILL discovered a heap-buffer-overflow in virtio-scsi's call to the `virtqueue_alloc_element()` function. This bug was neither discovered by ViDeZZo nor Morphuzz [6, 26]. HYPERPILL hooked 12 individual DMA accesses (3 of which were performed through lossy DMA-pointer APIs) and populated a total of 278 bytes. Source-code level hooking of the DMA-pointer API would require filling several thousand bytes (most of which are not touched). As such, API-hooking approaches (e.g., Morphuzz and ViDeZZo) waste effort mutating input bytes used to populate unused memory. However, HYPERPILL does not have the same problem since HYPERPILL performs DMA hooking at the level of atomic accesses.

QEMU: Heap-Overflow in QXL. QEMU's high-performance QXL (Spice) display device is the preferred display driver for cloud applications and thin-client solutions. However, due to QXL's multithreaded design, past fuzzers have been unable to fuzz it. Threading is incompatible with Morphuzz's fork-server and since ViDeZZo does not reset state in between inputs, ViDeZZo is unable to produce deterministic inputs [6, 26]. Furthermore, QXL uses QEMU's `memory_region` API to allocate a VRAM buffer that it accesses via a pointer. Since accesses to this buffer do not pass through the DMA APIs, Morphuzz and ViDeZZo do not receive feedback when the QXL device accesses guest-controlled data within this buffer. Additionally, QXL relies extensively on an external library (i.e., `libspice`), which is not instrumented for coverage-collection, by default. However, HYPERPILL discovered a heap-overflow in QXL's `qxl_cookie_new()` function due to the introspection capabilities enabled by emulation. The testcase reproducing this bug required interaction with QXL via PIO and DMA.

Hyper-V: Denial-of-Service via Hypercall. One of the 9 bugs HYPERPILL identified in Hyper-V is a denial-of-service bug via Hyper-V hypercall `0x83` (HvCallDetachDevice). This bug showcases HYPERPILL's ability to explore the complex hypercall input space. HYPERPILL successfully used com-

parison feedback to identify that Hyper-V uses register RCX to communicate the hypercall-id. Leveraging feedback from comparisons, HYPERPILL learned that `0x83` is an "interesting" hypercall-id. Furthermore, HYPERPILL used the runtime feedback to identify registers RAX and R8 as registers that contain argument values used by the hypercall's handler. After injecting the hypercall, HYPERPILL hooked accesses to five distinct DMA buffers, filling with a total of 60 bytes, to trigger the bug. This example demonstratrs that HYPERPILL's ability to simultaneously fuzz all input-spaces, while leveraging runtime feedback to continually refine inputs, enabling HYPERPILL to produce complex test-cases.

macOS: Out-of-Bounds in virtio-net. One of the 6 bugs HYPERPILL identified in macOS Virtualization-Framework is an out-of-bounds memory access in the virtio-net device. The crashing input performed three MMIO accesses to the device's MMIO regions. HYPERPILL detected accesses to four distinct DMA buffers (populating 47 bytes), to trigger the crash. To the best of our knowledge, there are no symbols publically available for macOS Virtualization Framework. As such, manually hooking macOS' DMA access APIs would require a considerable amount of work by a virtualization expert. However, HYPERPILL's use of feedback from the hardware-virtualization interface allows it to transparently fuzz complex attack-surfaces (e.g., DMA).

RQ2: HYPERPILL effectively finds bugs across a wide range of devices and hypervisors with diverse designs.
RQ3: Furthermore, HYPERPILL finds new bugs in code that has already been extensively fuzzed.

QEMU: Unique AHCI Coverage. In all of our runs, HYPERPILL was the only fuzzer that reached the AHCI device's `execute_ncq_command` function. Upon inspection, we attributed this to the fact that the caller (`process_ncq_command`), relies on a coarse-grained DMA API to map an entire guest page containing a table of AHCI slot IDs. As HYPERPILL does not rely on API-hooking, it can detect individual accesses to the table. In combination with the virtio-scsi and QXL bug case-studies, this demonstrates the value of instruction-level hooking of DMA accesses over API hooking approaches as proposed by Morphuzz, ViDeZZo and V-Shuttle.

5.3.1 Rediscovering Previously-Known Bugs

We backported the 5 CVEs used in the ViDeZZo evaluation to QEMU v8.0.0 and used each fuzzer to fuzz each affected device (5 total) for 24 hours. Our results averaged over 5 runs are presented in Table 4.

HYPERPILL identified all of the bugs. ViDeZZo identified 4 out of the 5 bugs faster than HYPERPILL because it does not perform any state resetting between inputs. However, as shown in our coverage and new-bug evaluation (§ 5.2 and § 5.3), this has a long-term effect of limiting total coverage achieved, for complex devices.

CVE	Description	Morphuzz	ViDeZZo	**HYPERPILL**
CVE-2020-11869	ATI Integer-overflow	0:37	0:11	0:13
CVE-2020-25084	EHCI UAF	4:31	0:43	1:11
CVE-2020-25085	SDHCI Heap-Overflow	5:34	2:36	6:24
CVE-2020-25625	OHCI DoS	0:14	0:04	0:06
CVE-2021-20257	E1000 DoS	3:34	Timeout	1:29

Table 4: Mean times to find known bugs (h:mm)

RQ4: HYPERPILL consistently finds previously-known bugs in hypervisors and outperforms related works for complex devices.

5.4 Throughput

HYPERPILL fuzzes hypervisors by leveraging snapshot-fuzzing in an emulated environment, allowing it to perform rapid state-resets and achieve hypervisor-introspection capabilities that do not exist natively. However, due to the nature of emulation, HYPERPILL has to pay an intrinsic performance penalty. To measure the penalty, we compared the execution-rate of test-cases executed by HYPERPILL with Morphuzz and ViDeZZo. The results are presented in Tab. 2. ViDeZZo executes QEMU code natively and does not reset state in-between inputs. As such it has the highest execution rate, outperforming Morphuzz and HYPERPILL by a factor of 9. Morphuzz relies on a fork-server, incurring a high-overhead due to the costly system-calls invoked between inputs.

Orthogonally, HYPERPILL's full-system snapshot resetting occurs entirely within the emulator process and does not require any system-calls. To measure the difference between Morphuzz's and HYPERPILL's state-resetting, we configured both fuzzers to fuzz all virtual-devices for 24 hours and measured the amount of time each fuzzer spends resetting state. We found that Morphuzz spends 71% of the time resetting state, while HYPERPILL only spends 9%.

As such, HYPERPILL outperforms Morphuzz for 8 target devices, even though it relies on emulation. Note that both ViDeZZo and Morphuzz provide inputs to QEMU through the internal "QTest" framework, skipping the costs associated with executing KVM's VM-Exit processing code.

RQ5: The lower execution rate of HYPERPILL is more than compensated by the detailed feedback and its generic applicability to any (even closed-source) hypervisor.

6 Discussion

Despite HYPERPILL's positive results, we briefly discuss avenues for future improvements.

Detecting Crashes. Similar to other full-system snapshot-fuzzers, HYPERPILL features target-specific methods for detecting crashes. For QEMU (an open-source target), HYPERPILL hooks into common failure-cases such as `assert()`, `abort()`, and `asan_stack_trace()`. For closed-source targets, HYPERPILL relies on coarser-grained methods, such as detecting `hlt` instructions, debugging interrupts, and protection-violating page-faults. In the future, HYPERPILL's crash-detection can be improved by integrating binary sanitizers and adding hooks to crash-detection mechanisms such as Windows' page-heap [31] and macOS' KASAN.

Improving Performance. Since HYPERPILL runs the hypervisor in an emulator, some operations (such as large memory copies), are significantly slower than counterparts running on native hardware. HYPERPILL's performance could be improved by leveraging multiple snapshots (e.g., ones taken after expensive operations), caching results and side effects of identical function-calls, and adding "accelerated" implementations of common expensive functions such as `memcpy()`. Furthermore, though HYPERPILL is the first hypervisor-fuzzer tailored toward fuzzing hypervisors across multiple threads or execution environments, HYPERPILL currently does not optimize the L1 OS scheduler to ensure that only hypervisor-related threads are executed. In the future, HYPERPILL could configure the OS to only schedule hypervisor-related tasks, or to skip execution of unrelated tasks by re-invoking the scheduler until a hypervisor task is selected.

Automatically Generating Standalone Reproducers. Currently, we manually convert HYPERPILL crashes into standalone crash-reproducers that can be used without the snapshot or emulator. In the future, these crashes can automatically be converted into reproducers using a custom kernel that can be booted within a hypervisor to replay HYPERPILL's crash.

Extensions to Other Architectures. HYPERPILL targets x86-64 hypervisors that leverage Intel VT-x. However HYPERPILL's techniques similarily apply to other popular architectures, such as ARM and POWER which feature similar virtualization extensions. Similar to Bochs for x86, there are mature emulators for these architectures (QEMU). Furthermore, for ease of use, HYPERPILL's stages can be modified, without affecting the core design. For example, a hardware debugging interface (JTAG) can be used to collect a full-system snapshot, rather than nested-virtualization.

Additional Attack Surfaces. In this work, we focus primarily on bugs in virtual-device implementations and hypercall handlers. However, hypervisors also feature code that parses guest instructions and walks guest page tables. With additional consideration for the semantics of these attack-surfaces (e.g. ensuring parsed instructions match the instructions that trigger VM exits), HYPERPILL can be adapted to fuzz these surfaces. We leave these extensions to HYPERPILL as future work.

7 Related Work

Fuzzing has gained academic momentum with the introduction of the American Fuzzy Lop (AFL) [52]. AFL's coverage-guided fuzzing approach significantly influenced research, leading to improvements in various aspects of fuzzing performance. Scholars have enhanced input scheduling [21, 35, 47], mutation algorithms [7, 28, 34], and input feedback mechanisms [1, 15, 54]. Some researchers have explored concolic execution techniques [22, 23, 51] to overcome challenges such as comparisons against "magic constants" and checksums [33]. Fuzzers like AFL with laf-intel [24], CmpLog [11], RedQueen [2], and libFuzzer [41] have applied source-code instrumentation to navigate comparisons against magic bytes.

The scope of fuzzing extends to diverse targets, including code interpreters [16, 19, 45, 50], compilers [8, 25, 27], and network protocols [3, 10, 14]. Operating system kernels have been a focal point, with specialized systems addressing kernel drivers [9], kernel race conditions [20], file systems [49], and the system-call interface [12, 18, 39]. Periscope [42] examines MMIO and DMA communication to identify vulnerabilities in a kernel exposed to a compromised device. Other studies apply static [46, 48] and dynamic [40] techniques to detect and analyze double-fetch issues in software.

In the realm most pertinent to our work, researchers have recognized that virtual-devices present similar challenges to fuzzers as kernel system calls and drivers. IOFuzz [30] identifies bugs in virtual-devices by writing random values to PIO. VDF [17] collects MMIO and PIO traces as seeds for coverage-guided fuzzing, but it does not fuzz DMA or reset state between input executions. Hyper-Cube [38] and Nyx [37], are virtual-device fuzzers, that rely on a custom-built guest operating system. Hyper-Cube is a black-box fuzzer that faces limitations when fuzzing virtual-devices reliant on magic constants and DMA. Hyper-Cube does not support Hyper-V. Nyx uses full-system snapshotting and hardware-assisted coverage but requires manually-written specifications for complex DMA devices.

Simultaneously, V-Shuttle [32] introduces a targeted method to fuzz the DMA input-space by replacing DMA access calls with reads from a file generated by AFL. Morphuzz, ViDeZZo, and MundoFuzz leverage source-code analysis and hooking to alleviate the need for manual specifications. HyperFuzzer is a closed-source fuzzer that targets Hyper-V's

instruction emulation code (a surface largely independent of the virtual-devices we examine in this work) [13]. In an industrial context, there is previous work on fuzzing virtual-devices [44] using a minimal OS connected to AFL.

8 Conclusion

HYPERPILL is the first generic fuzzer capable of automatically fuzzing arbitrary hypervisors across all major input-spaces (i.e., PIO/MMIO/Hypercalls/DMA). HYPERPILL takes advantage of the standard hardware-virtualization interface to enumerate the input-spaces. During fuzzing, HYPERPILL leverages the fine-grained feedback afforded by emulation, to create complex inputs and identify crashes in hypervisors. In our evaluation, we found that HYPERPILL can outperform state-of-the-art techniques that leverage hypervisor modifications and API hooking, achieving the highest coverage for 10/12 QEMU devices. Our system identified 26 new bugs, which we are responsibly disclosing to the vendors. HYPERPILL is available at https://github.com/HexHive/HyperPill

9 Acknowledgements

We thank the anonymous reviewers and our deeply involved shepherd for their feedback on the paper. We would like to thank Jeremy Lai for his help making HYPERPILL a reality. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_186974, DARPA HR001119S0089-AMP-FP-034, NSF CNS-1942793, and Red Hat Collaboratory grant 2024-01-RH05.

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, 2020.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, 2019.

[3] 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.

[4] 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, 2020.

[5] Bochs: the open source ia-32 emulation project. https://bochs.sourceforge.io/.

[6] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. Morphuzz: Bending (input) space to fuzz virtual devices. In *Proceedings of the USENIX Security Symposium*, 2022.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. HYPERFUZZER: An efficient hybrid fuzzer for virtual cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

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

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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 (AISec)*, Toronto, Canada, 2018.

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

[23] 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.

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

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

[26] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. Videzzo: Dependency-aware virtual device fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, 2023.

[27] 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.

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

[29] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. Mundofuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference. In *Proceedings of the USENIX Security Symposium*, 2022.

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

[31] Gflags and pageheap. https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap.

[32] 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 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[33] 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, 2018.

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

[35] 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.

[36] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5), 2005.

[37] 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.

[38] 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.

[39] 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.

[40] 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 (AsiaCCS)*, 2018.

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

[42] 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.

[43] source-based code coverage — clang 18.0.0. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.

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

[45] 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.

[46] 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*, 2017.

[47] 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.

[48] 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)*. IEEE, 2018.

[49] 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.

[50] 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.

[51] 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.

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

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

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