HYPER-CUBE: High-Dimensional Hypervisor Fuzzing

Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner and Thorsten Holz
Ruhr-Universität Bochum

***Abstract*—Virtual machine monitors (VMMs, also called *hypervisors*) represent a very critical part of a modern software stack: compromising them could allow an attacker to take full control of the whole cloud infrastructure of any cloud provider. Hence their security is critical for many applications, especially in the context of Infrastructure-as-a-Service. In this paper, we present the design and implementation of HYPER-CUBE, a novel fuzzer that aims explicitly at testing hypervisors in an efficient, effective, and precise way. Our approach is based on a custom operating system that implements a custom bytecode interpreter. This high-throughput design for long-running, interactive targets allows us to fuzz a large number of both open source and proprietary hypervisors. In contrast to one-dimensional fuzzers such as AFL, HYPER-CUBE can interact with *any* number of interfaces in *any* order. Our evaluation results show that we can find more bugs (over 2×) and coverage (as much as 2×) than state-of-the-art hypervisor fuzzers. In most cases, we were even able to do so using multiple orders of magnitude less time than comparable fuzzers. HYPER-CUBE was also able to rediscover a set of well-known hypervisor vulnerabilities, such as VENOM, in less than five minutes. In total, we found 54 novel bugs, and so far obtained 43 CVEs. Our evaluation results demonstrate that next-generation coverage-guided fuzzers should incorporate a higher-throughput design for long-running targets such as hypervisors.**

I. INTRODUCTION

Since several years, cloud-based Infrastructure-as-a-Service (IaaS) is rapidly expanding in the IT business landscape. This paradigm is powered by cloud providers using multi-tenancy and economy of scale to provide computational resources significantly cheaper than an individual customer could. While this approach greatly increases cost efficiency, there are confidentiality, integrity, and availability risks for customers: lacking proper isolation might allow a malicious actor to compromise the infrastructure of a given customer, typically by exploiting a vulnerability in the software stack. To provide strong isolation between individual virtual machines (VMs), high-performance, hardware-supported virtual machine monitors (VMMs, also called *hypervisors*) are used by all major cloud providers. These hypervisors are hence part of an essential, trusted code base which the entire cloud infrastructure relies on. As a result, we need efficient and scaleable techniques to identify potential software vulnerabilities in hypervisor given their crucial role in modern software stacks.

In the past, fuzz testing ("fuzzing") has proven to be a very successful technique for uncovering novel vulnerabilities in complex applications [11], [17], [38], [40], [45], [53], [54]. Unfortunately, only a limited number of resources on fuzzing hypervisors is available at the moment. The reason behind this lack of support for fuzzing this type of software applications is the fact that there are several additional challenges for fuzzing this lower-level component: In contrast to current fuzzers that mostly support interaction with a single interface (e.g., stdin [54] or syscalls [5]), fuzzing hypervisors requires to interact with several different devices using a variety of interfaces and protocols. For example, the guest system can trigger arbitrary hypercalls, perform arbitrary I/O operations, and access and modify emulated memory-mapped I/O (MMIO) regions. Hypervisor fuzzing also introduces additional performance issues: Modern feedback-driven fuzzing usually restarts the target application for each fuzzing iteration. However, restarting a hypervisor is prohibitively expensive in terms of run-time. Furthermore, performance-increasing tricks such as a fork server [54] cannot be applied to hypervisors easily. To add to this complexity, there are multiple implementations of different hypervisor, some of which are proprietary and no source code is available. Therefore, existing methods are not trivial to adapt for hypervisor fuzzing.

As a consequence, there are few research projects on fuzzing hypervisors [6], [7], [25], [26], [30], [35], [46]. The state-of-the-art hypervisor fuzzer is VDF [30]. Based on a classic fuzzing design, VDF is a fork of the famous AFL fuzzer [54] that interacts with QEMU device emulators. As a result of this design, VDF is limited to low-dimensional fuzzing (i.e., only able to interact with memory and port mapped I/O). Another example of a hypervisor fuzzer, is IOFUZZ [35]. IOFUZZ is even more limited and only writes random values to random I/O ports, without any support for other common interfaces such as MMIO, DMA, or hypercalls. A similar tool was developed as part of Intel CHIPSEC suite [25]. Compared to the state-of-the-art approaches for fuzzing userspace programs (ring 3) or OS kernels (ring 0), fuzzing at the VMM level (ring -1) is still lacking.

In this paper, we tackle this challenge and present the design and implementation of a hypervisor-independent fuzzing approach based on a custom operating system. Our goal is to evaluate the security and stability of various open source and proprietary hypervisors. To this end, we suggest three goals to improve upon current designs. First, an efficient hypervisor fuzzer needs to provide a *high test case throughput*. Second, an effective hypervisor fuzzer should be able to simultaneously interact with *all available interfaces* to enable

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23096
www.ndss-symposium.org

a multi-dimensional fuzzing. Third, a precise fuzzer should be able to produce a *stable and deterministic* test cases for a diverse set of hypervisors. Considering these three goals, we designed HYPER-CUBE, a novel hypervisor fuzzer based on a fully custom and minimalist Operating System (OS) that implements a custom bytecode interpreter. Having a custom minimal OS allows us to take full control of our environment, yielding stable and deterministic test cases. Additionally, we can boot our OS on nearly any hypervisor, which effectively allows us to evaluate a diverse set of targets. Our custom bytecode interpreter enables a high test case throughput while accessing any number of interfaces simultaneously.

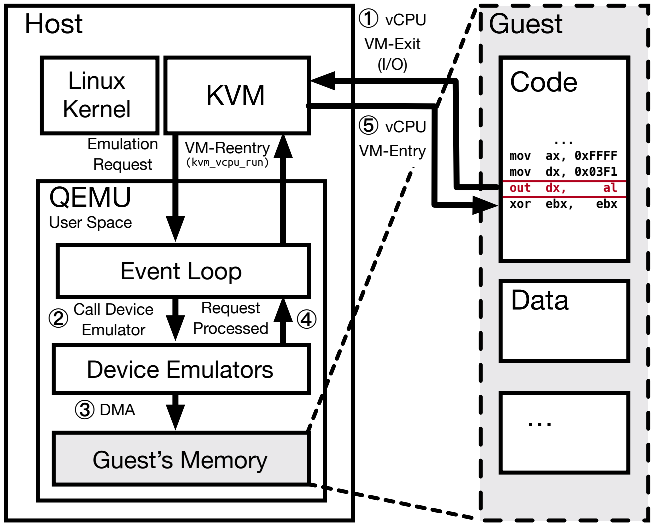
As our evaluation results show, this design and our prototype implementation of HYPER-CUBE allows us to achieve all three goals. Most importantly, we were able to uncover and report 54 bugs in six different hypervisors such as QEMU/KVM, VirtualBox, VMware Fusion, and Intel ACRN. So far, we obtained 43 CVEs, the remaining bugs are currently being investigated by the maintainers and CVEs are not yet assigned. We directly compare our performance on QEMU device emulators against VDF. HYPER-CUBE is able to find bugs in eight out of 14 emulators, while VDF only finds bugs in three. Additionally, HYPER-CUBE produces more test coverage—in many cases significantly—than VDF. Interestingly, we are orders of magnitudes faster: we were able to find both the bugs and the coverage in less than ten minutes, while VDF took nearly 60 days of fuzzing to achieve these results.

This result is somewhat surprising, since in contrast to VDF, HYPER-CUBE is not coverage guided. However, using a higher test case throughput, and a smart bytecode interpreter, we are able to drastically increase the efficiency of our fuzzer, while at the same time increasing its flexibility. Finally, in contrast to existing hypervisor fuzzing approaches that we are aware of, HYPER-CUBE is not limited to one specific or open source hypervisors. Because of our generic design, we can easily target even proprietary hypervisors as long as the hypervisor is able to boot commercial off-the-shelf (COTS) OSs. Based on these evaluation results, we conclude that our design is superior to existing state-of-the-art hypervisor fuzzers and significantly outperforms them in almost every aspect.

In summary, we make the following three contributions:

- We design a multi-dimensional, platform-independent fuzzing method that can test different interfaces and their interactions in an efficient and effective way.
- We describe a highly efficient method to perform fuzz testing against hypervisors. Our approach is independent of the hypervisor that should be tested.
- We implement our techniques in a custom operating system called HYPER-CUBE. Our experiments demonstrate that HYPER-CUBE is able to find security vulnerabilities in many real-world hypervisors.

To foster research on this topic, we release HYPER-CUBE at https://github.com/RUB-SysSec/hypercube.



```
...
mov  ax, 0xFFFF
mov  dx, 0x03F1
out  dx,     al
xor  ebx,   ebx
```

Fig. 1: Device emulation and its *trap and emulate* handling of privileged instructions in KVM and QEMU.

II. TECHNICAL BACKGROUND

Before presenting our approach to test hypervisors, we briefly recap various aspects of x86 virtualization and the way hypervisors are implemented. As our fuzzer is implemented using a custom OS, we also describe essential aspects of the boot process of an OS on the x86 architecture. Finally, we describe the interfaces that the hypervisor uses to interact with the OS running as a guest inside of the Virtual Machine (VM) because they are relevant targets for our fuzz testing. In principle, most design choices for a hypervisor fuzzer are independent of the CPU architecture and it is merely an engineering effort to implement similar approaches for other architectures. However, given that we implement a custom OS, we also need to consider several aspects related to the underlying processor architecture and its quirks. In this paper, we target hypervisors on Intel x86 CPUs and their backends for devices emulation. Thus, in the following, we focus on the Intel x86 architecture, which is—to the best of our knowledge—the most commonly used virtualization platform in practice.

A. x86 Boot Process

One of the first programs which run on an x86 machine is the *Basic Input/Output System* (BIOS) or the more modern version of it, the *Unified Extensible Firmware Interface* (UEFI). In this paper, we call this program *firmware*. It is the firmware's job to test and initialize the hardware such as CPU and the main memory after turning on the machine. The next step of the x86 boot process is executing the OS bootloader. The bootloader is a small piece of code that launches and prepares everything needed to start the OS. One example of a well-known OS bootloader is GRUB (GRand Unified Bootloader). Finally, the bootloader calls the OS kernel code. The kernel then configures the remaining hardware, such as interrupt controllers, paging, or PCI devices.

TABLE I: Overview of hypervisor attack surfaces.

Interface	Component
Port I/O	Device Emulator / Core
MMIO	Device Emulator / Core
PCI DMA	Device Emulator
Hypercall Interfaces	Guest Tools Interface / Core
Instructions	Instruction Emulator

B. Input/Output on x86

An Intel x86 processor implements multiple mechanism and instructions to communicate with attached devices. To understand how the OS interacts with these devices, we provide in the following a brief overview of the different primitives of Input/Output (I/O) on x86. An overview of the available interfaces is presented in Table I.

a) Port I/O: The first mechanism to interface with external devices is the legacy port I/O address bus: besides the main memory, on x86, there is a second address space that can only be accessed using the `in` and `out` instructions.

b) Memory-Mapped I/O: The second mechanism to interface with external devices are *memory-mapped I/O* (MMIO) regions. Writing or reading these memory regions corresponds to a direct access to device registers or device memory. In modern hardware, this mode of communication is preferred over port I/O as it is more flexible and allows much higher bandwidths. Additionally, there are well-known ways to enumerate most available MMIO regions.

c) Direct Memory Access: The last mechanism to implement communication between the host and its devices is *Direct Memory Access* (DMA). PCI/PCI-Express based-DMA is implemented via bus mastering, which allows the PCI device to read from and write to the host's main memory directly. Note that this effectively swaps the roles that the CPU and the device play: in contrast to MMIO, the devices decide when and which memory to read and write. In the context of this paper, we ignore legacy ISA DMA since it is hardly used these days in practice.

C. Hypervisor

A hypervisor, also known as Virtual Machine Monitor (VMM), is a privileged software which provides physical hardware resources, such as memory and virtual CPUs (vCPU), in a controlled environment to their VMs. Generally speaking, the VMs have no control or access to physical hardware devices except for assigned ones. Instead, the hypervisor provides vCPUs and virtualized physical memory, and it also emulates all necessary devices such as interrupt controllers. Whenever the VM needs to perform a privileged operation, such as accessing the real hardware, it triggers a trap that causes a VM-exit. A VM-exit transfers the control back to the hypervisor which emulates the privileged operation on behalf of the VM, after ensuring that the operation is legitimate. This mechanism is called *trap and emulate*. By providing a fully virtualized environment, the hypervisor allows to simultaneously execute multiple COTS OSs on the same physical host.

1) CPU and Memory Virtualization: In the past, hypervisors relied on a technique called *binary translation* to implement full CPU and memory virtualization [9]. That means all critical instructions (e.g., memory access or branch instructions) are translated into another piece of code that enforces certain policies, while emulating the virtual behavior. For example, a memory access is replaced by code that emulates a virtual page table and adheres to the access policies defined by the hypervisor. By using binary translation, the hypervisor has complete control over all vCPUs and can emulate high-privileged instructions such as `MOV CR3`. Therefore, the hypervisor can ensure that the software is not able to escape from the virtualized context. The same applies to memory virtualization and all memory-accessing instruction.

However, implementing virtualization without hardware support introduces a significant performance overhead and several other challenges such as interrupt handling need to be addressed [8]. Therefore, all major CPU manufacturers such as Intel or AMD introduced hardware extensions that allow to increase the performance by using so-called hardware accelerated virtualization. Intel calls its technology "Intel VT-x", while AMD's technology is called "AMD-V". In the context of this paper, we only focus on Intel VT-x. However, it is worth mentioning that while the implementation details of these extensions vary, the overall idea is the same: a new execution mode is introduced in the CPU that has even higher privileges than the OS kernel. A hypervisor running in this higher privilege level has various methods to intercept different kinds of events, to restrict the guest's kernel from accessing actual hardware, and to provide direct access to physical memory dedicated to a VM without emulating certain instructions.

2) Device Emulation: To be able to boot a COTS OS, a hypervisor has to emulate standard hardware such as an interrupt controller, a timer, and various other peripheral devices. The two main mechanisms for interaction with hardware are MMIO and port I/O. To emulate the interaction with hardware, the hypervisor must first intercept the interaction with the hardware. It should be noted that in the case of DMA, the hypervisor does not need to intercept memory accesses, as it is only used by the emulated hardware rather than the guest OS. An example of the implementation of device emulation and its *trap and emulate* approach is given in Figure 1. In the following, we will outline the process of emulating a port I/O instruction based on KVM and QEMU. To capture MMIO or port I/O interactions, hypervisors typically use two different techniques. In both techniques, interacting with the emulated hardware causes a trap and a VM-exit ①, which the hypervisor handles by emulating the side effects. No matter what kind of trap the hypervisor receives, to obtain information on the actual interaction (such as which value was written), the hypervisor will typically have to disassemble and emulate the instruction itself.

In the first case, the hypervisor has to emulate MMIO. To trap access on MMIO pages, the hypervisor typically marks the corresponding pages as non-accessible. Accessing the MMIO page from within the virtual machine triggers a page fault. The hypervisor installs a handler for these page faults. Whenever a page fault occurs, the handler then checks whether the requested page is registered for MMIO by one of the emulated devices. Finally, the hypervisor passes the corresponding memory access to the device emulator. After the device emulator simulated all relevant side effects of the MMIO access, the hypervisor passes control back to the VM.

In the second case, the hypervisor has to emulate the port I/O. To trap on port I/O interactions, the hypervisor limits the execution of certain instructions such as `in` and `out` using the hardware virtualization extension. Executing such a restricted instruction will also cause a VM-exit. Similar to the previous case, the hypervisor passes the interactions to the corresponding device emulator ② and returns control back to the VM ⑤ after the device emulator finished ④. If the corresponding device emulator implements DMA capabilities, the emulator may read or write data from the guest's main memory ③.

Current hypervisors typically only support a limited set of standard hardware to be emulated. However, the code base used in device emulation is rather large, posing a significant attack surface. For example, QEMU in version 4.0 contains more than 400k lines of C code used to emulate devices.

3) Para-Virtualization: To reduce the amount of code needed and to improve the performance significantly, modern hypervisors implement so-called para-virtualization interfaces. In a para-virtualized environment, the OS is aware that it is running inside a VM and it does not try to use actual hardware. Instead, a specialized driver (typically referred to as *VirtIO driver*) uses a custom protocol specific to the hypervisor. Using a custom protocol increases the performance of the virtual machine and can reduce the attack surface in the hypervisor, as legacy corner cases no longer have to be simulated faithfully.

To facilitate the communication between the guest OS and the hypervisor (e.g., to implement para-virtualization protocol), modern hardware-accelerated virtualization introduces a new instruction that performs a so-called *hypercall*. From a conceptual perspective, hypercalls are closely related to system calls (*syscalls*), but instead of jumping from userland to ring 0, they jump directly to the hypervisor by triggering a *VM-exit*. To implement hypercalls, Intel VT-x has introduced a dedicated instruction called `vmcall`. However, to implement hypercall interfaces, some hypervisors use other privileged x86 instructions, which always or conditionally result in a VM-exit. For instance, the VMware hypervisor series uses the `in` and `out` instruction for implementing para-virtualized interfaces, which are used by their VMware guest tools. The Parallels hypervisor uses the `rdpmc` instruction for this purpose. It is common that upon encountering a hypercall, the hypervisor directly accesses the memory of the guest and reads data or provides results by directly writing them into the guest's memory. Similar to classic device emulation, this property makes hypercalls an interesting target for fuzzing.

D. Fuzzing Hypervisor

Fuzzing hypervisors has its own set of challenges. The higher number of interfaces that can be used interactively mandates a different setup than for ring 3 file format fuzzing. The guest system can trigger hypercalls, I/O operations as well as reading or writing to memory mapped I/O regions. Likewise, restarting the hypervisor is much more expensive than restarting an user-space process. Consequently, many current fuzzers target individual interfaces in isolation and try to avoid restarting the VMM. At the same time, the attacker model is very strong: typically the security model of hypervisors assumes that the whole guest OS is malicious and any interface available is also part of the attack surface. Consequently, any kind of memory corruption issues—and even some denial-of-service bugs—are highly security-relevant in the context of hypervisors. Especially memory corruptions might lead to virtual machine escapes, which is the worst-case scenario for the whole cloud hosting industry that relies on secure isolation.

So far, most research on the security of hypervisors was performed in industry [6], [7], [25], [26], [46]. For instance, Tang et al. [46] implemented an AFL extension with hypervisor fuzzing capabilities. It is implemented as a custom extension to QEMU's SeaBIOS. They extended the BIOS with an ability to consume inputs from the outside of the VM and to use them to interact with devices provided by the hypervisor. They also instrumented the code used for device emulation to provide coverage feedback to AFL. In the academic domain, to the best of our knowledge, there is only a single project (VDF) for fuzzing hypervisor related software [30]. The authors of VDF manually extracted individual devices from the QEMU codebase and fuzz them in AFL [54]. This approach has a significant drawback: manual work is needed for each device emulator and hypervisor to enable fuzzing.

III. Design

Since hypervisors perform a multitude of complex tasks, there are many ways one can approach testing the implementation of a given hypervisor. As a consequence, we begin by analyzing the challenges one faces when designing a fuzzing framework for hypervisors. We then formulate the goals that we want to achieve with our fuzzer. Lastly, we describe our design choices, which are all tailored towards addressing the identified challenges and achieving the goals discussed earlier.

A. Threat Model

Throughout the rest of this paper, we assume the following threat model that is also reflected in the design of our approach. First, an attacker has full control over the kernel running *inside* the virtual machine. Her goal is to either obtain control over *other* virtual machines hosted on the same physical machine or the host itself (i.e., the underlying system). This assumption is commonly valid in various cloud hosting scenarios, where many virtual machines with unknown tenants are co-located on the same server [43]. If an attack can escape her rented VM at any time, all data that is stored on the same physical server is at risk. Additionally, with the rise of private clouds, similar scenarios occur after an attacker compromised one service and now plans to escalate horizontally. Additionally, we are interested in Denial-of-Service (DoS) attacks, in which an attacker might prevent the operator from taking control of its VM or the attacker can circumvent load limiting mechanisms.

B. Challenges in Fuzzing Hypervisors

Current hypervisors have a large and diverse attack surface. As described in Section II, there are many ways in which the guest interacts with the hypervisor. To make matters worse,

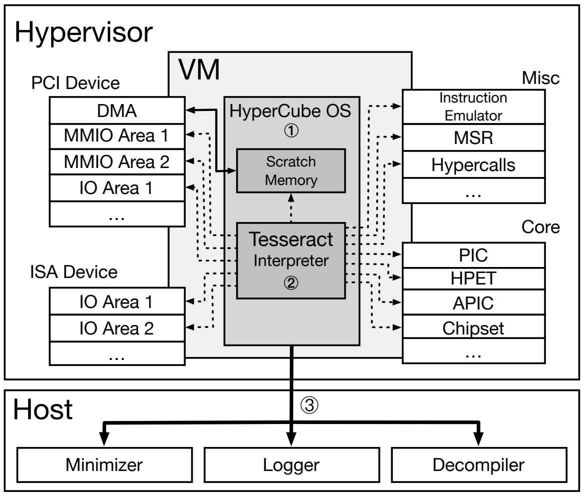


Fig. 2: High-level overview of the system architecture of Hyper-Cube

testing the various interfaces in isolation is not sufficient to evaluate the security of modern hypervisors: as our experience and evaluation results show, a comprehensive evaluation has to consider all interactions between different interfaces, typically a sequence of such interactions is needed to uncover a vulnerability. Additionally, it is often non-trivial to find documentation for different interfaces a given hypervisor exposes. Therefore, the fuzzer needs to be able to provide meaningful interactions with undocumented interfaces. As another challenge, we found that even for well-known interfaces or devices, the normal start-up routine of COTS OSs such as Linux or Windows might influence the bug finding process. For example, large parts of the emulator might be initialized during boot time. However, given that an attacker can reboot the machine, the initialization code also needs to be considered when studying the attack surface. Yet, testing the device from a COTS OS never exercises this code in an uninitialized state. The same boot process can also slow down hypervisors during fuzzing—restarting a hypervisor from scratch (e.g., after finding a crash) takes a significant amount of time that would slow down the fuzzing progress substantially. Lastly, depending on the OS running inside of the hypervisor, the behavior of hardware-accelerated hypervisors can be rather non-deterministic.

C. Architecture

In the following, we first provide an overview of our design goals and then present the architecture of our approach. Primarily, our design aims at providing high-performance fuzzing for a given hypervisor. Besides being as fast as possible, our approach should be generic and applicable to a wide range of both open source and proprietary hypervisors. Furthermore, our approach should have the ability to find bugs resulting from (complex) interactions between different communication channels. To this end, we improve upon the idea of *two-dimensional fuzzing*, as introduced by Xu et al. [52]. In two-dimensional fuzzing, two different interfaces are attacked in a fixed sequence. In our approach, we want to be able to attack an arbitrary number of interfaces in any order, we therefore aim at designing a "high-dimensional" fuzzer. Lastly, any fuzzer is only as good as its ability to reproduce the bugs found. Therefore, we require a very deterministic setup to obtain good stability during fuzzing, especially given that we interact with a low-level component of the software stack. In the following, we demonstrate how we can achieve these goals by describing the design of our fuzzer named Hyper-Cube.

1) High-Level Overview: Our fuzzer consists of three main components as illustrated in Figure 2.

1) The first component is a specialized OS called Hyper-Cube OS which boots inside the target hypervisor and enumerates existing hardware interfaces. A custom OS enables us full control over the complete process.
2) Afterwards, Hyper-Cube OS spawns a second component and passes the information about available interfaces to the spawned task. The task runs our bytecode interpreter named Tesseract. It can consume arbitrary byte strings and uses them to actually fuzz the hypervisor.
3) Additionally, there is a set of external, independent tools that are able to provide bytecode strings to the Tesseract interpreter, decompile executed bytecode programs, and observe the machine's behavior using channels such as a serial interface.

As we explain in the following, this architecture allows us to achieve our three aforementioned goals.

a) High Performance Fuzzing: Rebooting a COTS OS such as Linux requires a significant amount of time. If there are easy-to-find bugs within the hypervisor, the fuzzer spends a significant portion of its time on rebooting the COTS OS. Using a small custom OS such as Hyper-Cube OS can drastically decrease the boot time of the OS. Hyper-Cube OS does not interact with other interrupts, nor does it have any hardware initialization routine. Additionally, Hyper-Cube OS is very small, and uses only an absolute minimum amount of memory. During fuzzing, this design choice allows us to rapidly reload the OS after each crash and thus significantly improve the fuzzing performance.

Additionally, and more importantly, we took measures to improve the performance of the fuzzing process itself, since the process of compiling and loading a new program for each test case can be very time consuming. The ability to execute as many test cases as possible in any given time period still remains one of the most important performance criteria for fuzzers [27]. To increase the test throughput, we avoid compiling code. Instead, we use our specialized bytecode interpreter, named Tesseract, that runs in ring 0 of the Hyper-Cube OS.

The embedded interpreter Tesseract takes any stream of input bytes and interprets it as a sequence of interactions with the hypervisor. The custom nature of this interpreter allows us to design the bytecode in such a way that it is very "fuzzer-friendly". All instruction encodings are designed to maximize the likelihood of producing useful instructions. The encoding is chosen such that there are no invalid instructions. Additionally, memory addresses are not encoded as pointers, but as pair of memory region and an offset. Since Tesseract is aware of all interesting memory regions, it can avoid interacting with addresses that contain no relevant content, even though the fuzzer generates random data. To further increase the probability of producing reasonable opcodes, all arguments such as region IDs and offsets are interpreted modulo the available range.

b) Generic High-Dimensional Fuzzing: To the best of our knowledge, all previous hypervisor fuzzers target either one or two specific interfaces. For example, VDF is unable to fuzz interfaces such as DMA, MSRs, or hypercalls, and only focusses on MMIO. Since we have full control over the TESSERACT interpreter, we can integrate interactions with different interfaces. As a result, even a completely random stream of data produces a large number of valid interactions with all available interfaces. In particular, complex multi-channel interactions happen frequently and organically. For example, we observed that TESSERACT initially wrote data to a scratch buffer, passed a pointer to this data into an MMIO region, and finally an emulated device used this pointer in combination with PCI DMA and crashed after reading malformed data. Such complex sequences enable us to uncover bugs and we found that especially complex interaction sequences often trigger relevant behavior.

c) Stable and Deterministic Fuzzing: Previous work on hypervisor fuzzing usually used some kind of agent running in a COTS OS [6], [7], [26]. This reduces the determinism, as the kernel interferes with interrupt handling and introduces a variety of other tasks. Additionally, COTS kernels are fragile: re-configuring essential hardware such as the interrupt controller or PCI devices that overwrite memory most certainly break COTS kernels. Similar issues show up if the fuzzer interacts with disk interfaces such as IDE/SATA and corrupts the hard drive. By implementing our own OS HYPER-CUBE, we can significantly increase both the robustness and determinism during the testing phase. Most importantly, using a small OS around a custom interpreter allows us to have full control over the environment. In contrast to other OSs, there are no additional tasks that would interfere with our fuzzer. Also, as the page tables are fully under our control, we can perform mischievous but very useful operations, such as mapping fuzzer-controlled memory at address 0. All of this helps to produce both reliable and reproducible crashes. In fact, if we fix the payload, the entire interaction with the hypervisor is deterministic (except for minor variations in the interrupt timings). As a result, we are able to reproduce all bugs reliably. The only exception are some issues that only appear due randomization taking place in the hypervisor itself, which we cannot fully control.

In the following, we describe the design of each of the three components in detail.

2) HYPER-CUBE OS: The core of our fuzzer is our custom operating system HYPER-CUBE OS. It implements the multiboot 2 specification and therefore can be booted by common bootloaders such as GRUB. HYPER-CUBE OS, besides providing a general platform for our fuzzer, has two primary tasks: memory management and device enumeration.

HYPER-CUBE OS requires memory management for device enumeration and spawning TESSERACT. There are two different kinds of memory which need to be managed. First, operations such as device enumeration and spawning TESSERACT require to allocate memory. Second, certain interactions with available interfaces require access to physical memory. Our memory manager contains a simple heap that can be used to allocate memory when required. Overall, running HYPER-CUBE OS and TESSERACT only requires a very small amount of memory. Therefore, it is possible to use a simplistic implementation of a heap manager. To provide access to the physical memory, HYPER-CUBE OS creates multiple linear one-to-one mappings that make relevant physical address ranges available within the virtual address space.

The largest part of HYPER-CUBE OS is concerned with the enumerating the different interfaces that are available for fuzzing (see Figure 2). This requires to enumerate address ranges used for MMIO or port I/O. In addition, it also requires to interact with basic hardware such as the PIC or APIC. Enumerating these devices is performed using a variety of techniques. Some information (such as the set of reserved memory regions that are used for MMIO) is passed on from the BIOS/UEFI by the bootloader, while other information, such as the set of I/O ports or PCI devices, are manually enumerated.

3) TESSERACT: After HYPER-CUBE OS was successfully booted, it calls into its only functionality: performing random interactions with the hypervisor. We use a custom bytecode interpreter (TESSERACT) to describe those interactions. TESSERACT is a relatively straightforward, yet complex instruction set interpreter. The bytecode is either given from the outside of the VM or produced by a pseudo random number generator inside of the interpreter. This design allows us to interleave a variety of different types of operations to perform multi-dimensional fuzzing. The bytecode is designed in such a way that any arbitrary byte string is also a valid program.

First, we decode the instruction at the current instruction pointer. All values are interpreted modulo the sensible range of values. For example, we use the extensive scanning, which HYPER-CUBE OS performs during boot, to identify the set of MMIO regions. Writing to MMIO is then performed by writing to a `(region-id, offset)` pair. Both the `region-id` and the `offset` are computed modulo the available length.

After decoding the instruction, we call the corresponding opcode handler function. Since we use handler functions for each opcode, we can perform rather complicated operations in a single step. For example, this can be used to implement opcodes that overwrite a given range with an increasing sequence of bytes or repeatedly write increasing values to the same address. Both the complex instruction set and the liberal instruction decoding allow the interpreter to be very effective, even if a random program is provided.

The interpreter also maintains a set of scratch region. Each scratch region is a single 4KB page and is regularly overwritten with random data. The interpreter can write pointers to offsets within the scratch region to I/O ports or MMIO regions. This allows TESSERACT to create custom data structures and pass pointers to them to the hypervisor.

4) External Tools: The last component of our fuzzer is a set of three independent helper tools running on the host. These tools are mainly used after the fuzzing process. The same interfaces could also be used to implement more advanced fuzzers with coverage feedback or symbolic execution capabilities. The following three components are available:

1) During fuzzing, the logger intercepts serial communication performed by the virtual machine and stores it for later analysis.
2) To perform initial fuzzing, we generate a random byte stream from a given seed using the internal pseudo random number generator. However, we often execute millions of TESSERACT instructions before we find a bug. Therefore, after finding the bug, we need to trim the stream of inputs to a more manageable size. To minimize the crashing programs found, we regenerate the instruction stream from the same seed. Then the minimization tool iteratively removes a random segment from the stream and observes whether the resulting program still crashes. After a convergence of this algorithm, we typically obtain programs containing in the order of tens of instructions. For debugging purposes, it is also possible to run TESSERACT as a standalone ring 3 application.
3) To understand the resulting bytecode and to analyze the bugs found, we designed a decompiler that turns the given (minimized) bytecode into an equivalent C program. This C program could be compiled into a module for HYPER-CUBE OS, inserted into a kernel driver for any COTS OS for debugging purposes, or inspected manually to understand the behavior of the input.

IV. IMPLEMENTATION DETAILS

We implemented HYPER-CUBE OS in C and x86 assembly. The bytecode interpreter TESSERACT was implemented in such a way that it is easy to embed in HYPER-CUBE and to test as a standalone application. In this section, we describe the implementation details related to HYPER-CUBE OS and TESSERACT as well as the external tools used in our fuzzer.

A. HYPER-CUBE OS

To understand how and which information HYPER-CUBE OS gathers and provides to TESSERACT, we have to describe the boot process and memory management of our OS.

1) Boot Process: During early boot time, the firmware loads programs from different peripherals (which are called Option ROMs) for basic I/O initialization. The firmware then runs Option ROMs belonging to detected hardware and generates x86 specific data structures such as an e820 memory map. These data structures provide a way for the OS kernel, or any other code running afterwards, to interact with the detected peripherals or the firmware itself. The layout of these data structures depends on whether the computer is using a BIOS or a UEFI firmware. After preparing the relevant data structures and initializing basic devices, the firmware launches the bootloader.

Different OS bootloaders use different routines to load an OS kernel. To standardize this process, Ford et al. introduced the multiboot specification [19]. To add support for UEFI, the multiboot specification was later extended to version 2 [20]. Using a multiboot 2 compliant bootloader allows developers to ignore almost all differences between BIOS and UEFI. Note that different bootloaders require different binary formats for loading the OS kernel. However, by adhering to the multiboot 2 specification, the developers can build the kernel as an ELF file using existing tool support. The initial boot interface of HYPER-CUBE OS is based on the multiboot 2 specification and we use the GRUB 2 bootloader. GRUB can be used to boot HYPER-CUBE OS via both legacy BIOS or a UEFI firmware. An additional benefit of this design is that the GRUB bootloader enters the entry point of our kernel in 32-bit protected mode, which is the primary operating mode of HYPER-CUBE OS. This way, we do not have to implement a routine to mode switch from real to protected mode.

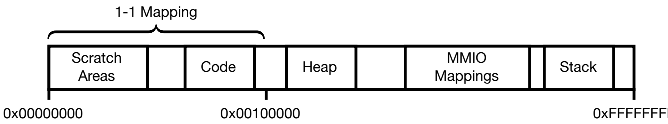


Fig. 3: Virtual memory layout of HYPER-CUBE

a) Initializing Interrupts: To synchronize interaction with external hardware, the Programmable Interrupt Controller (PIC) and its successors, the Advanced Programmable Interrupt Controller (APIC), are used. The PIC is configured via port IO and relies on the Interrupt Descriptor Table (IDT). The interrupt table contains a list of function pointers to functions that are called when an interrupt or exception occurs. Interrupts are triggered by external hardware, whereas exceptions are generated by the CPU itself. During the startup routine, HYPER-CUBE OS configures either the PIC or the APIC and installs all essential interrupt and exception handlers. To ensure that the fuzzing process will not be interrupted by external interrupts initiated by the hypervisor at any point during the fuzzing processes, HYPER-CUBE OS prevents the delivery of all external interrupts by masking all interrupt registers within the OS. Note that the interrupts are still produced by the emulators.

2) Memory Management: Our custom OS requires a custom memory management. In particular, various tasks need to allocate memory and access physical memory (such as MMIO regions), even when paging is enabled. We implemented a custom heap allocator, which allows tasks to allocate a whole page at a time. This simple design drastically reduces the implementation effort and the amount of fragile metadata stored on the heap. Additionally, it increases the robustness of the OS against failures introduced by memory corruptions resulting from abusing devices that directly write to memory using DMA. On the other hand, if many small allocations are required, it drastically increases the memory overhead. HYPER-CUBE OS uses only a very small number of allocations during device enumeration and for its interpreter TESSERACT. Therefore, this additional memory overhead is negligible for our use case. We typically observed fewer than 150 allocations during a typical fuzzing campaign. However, the precise number depends on the number of emulated devices provided by the hypervisor.

Usually, an OS enables paging to manage its memory in a virtual address space. If paging is enabled, the CPU performs a lookup to map virtual pages to physical page frames using the current page table. This allows a much more flexible handling of memory including access permissions and remapping fragmented memory ranges into continuous regions. However, some tasks still need access to physical addresses. For example, the page table itself needs to be described in terms of physical memory. To enable access to various MMIO regions and the page tables, which reside in physical memory, our OS maintains some regions in which the virtual addresses map directly to physical addresses. In the lowest megabyte of memory (addresses 0x0 to 0x100000), we create such a 1-to-1 mapping of the physical memory to the virtual memory. This region contains the code, the kernel stack, and additional scratch memory that HYPER-CUBE OS provides to TESSERACT for internal use. During boot time, the BIOS/UEFI provides GRUB a memory map that contains available and reserved ranges in physical memory. This memory map is later passed to HYPER-CUBE OS. It uses the available memory to create a kernel heap that is maintained by the allocator mentioned previously. The kernel heap makes full use of the virtual address lookup and can contain any number of physical pages in any order. Above the remapped area, HYPER-CUBE OS creates a region in which all of the available memory is used to create a heap. Lastly, in the uppermost regions, we create another remap that contains various kinds of MMIO regions, as detected by the enumeration procedure. For example, memory ranges marked as reserved by the BIOS or UEFI firmware mostly contain MMIO regions of emulated devices and are remapped to this area during device enumeration. A visualization of the memory layout is given in Figure 3.

3) Device Enumeration: Hardware devices, such as PCI devices or core components like the APIC or the *High Precision Event Timer* (HPET), can map internal registers to physical memory. Accessing these MMIO regions allows to directly influence the state of the device. After booting, a major task of HYPER-CUBE OS is to enumerate the different interfaces that are available for fuzzing. As mentioned earlier, this requires to enumerate all MMIO and port I/O address ranges used by emulated devices and peripherals.

a) Core Components: The MMIO regions provided by core components such as the APIC or the HPET are described by so called *Advanced Configuration and Power Interface* (ACPI) tables. The base addresses of HPET and APIC MMIO regions are stored in their corresponding ACPI tables. To enumerate those components, HYPER-CUBE OS parses all relevant ACPI tables. A pointer to the array of ACPI tables is provided by the multiboot bootloader. Pointers to both APIC and I/O APIC MMIO regions are located in the `APIC` ACPI table. Lastly, if an HPET is present, the base address of the HPET can be found in the `HPET` ACPI table. All base pointers stored in the ACPI tables are registered in TESSERACT as potential fuzz target areas for later use. In addition, the base pointers of the APIC and I/O APIC are required for the configuration of all interrupts. The HPET pointer is not used further except for fuzzing.

b) PCI-/PCIe-Enumeration: To find all emulated PCI and PCI-Express devices, HYPER-CUBE OS either relies on the legacy PCI configuration I/O ports or more modern PCI configuration spaces based on the MMIO region located at the *Enhanced Configuration Mechanism* (ECAM) base pointer. Similar to the base pointers of HPET and ACPI, the base pointer of the ECAM area is located in an ACPI table named `MCFG`. Either way, HYPER-CUBE OS scans all PCI buses for devices. Once a PCI or PCI-Express device is found, all Base Address Registers (BARs) are parsed, and the port I/O or MMIO regions are registered as fuzzing target areas in the TESSERACT interpreter. To enable DMA for all PCI devices, HYPER-CUBE OS sets the PCI bus mastering bit in the command register of each PCI device. This way, PCI devices are allowed to read and write the guest's physical memory. HYPER-CUBE OS enumerates all MMIO regions provided by the hypervisor. Theoretically, a hypervisor could create regions that are *not* registered in the ACPI table, but they would be invisible to a COTS OS without hypervisor-specific drivers. We did not encounter this in practice during our evaluation.

c) ISA-Enumeration / I/O Port Probing: Various other devices use the legacy port I/O address bus to expose internal registers to the host. This includes old ISA devices such as VGA graphic cards, sound devices, or core components like the PIC. While there is a list of well-known I/O ports used for common devices, unfortunately, there is no proper, systematic way of enumerating and detecting certain emulated ISA devices. To enumerate available emulated devices, we perform an active scan of all 2^{16} possible I/O ports. To find devices, we try to read from and write to all ports. If, after writing, the value in the I/O port changes, we consider the port as a possible interface to an emulated device. Unfortunately, this technique does not work for all emulated devices because not all I/O ports are writable. Using this method, we create a dictionary of interesting I/O ports that is later used more often during the fuzzing run. Our evaluation shows that this method—in combination with a small list of well known ports—is typically able to find more than 90% of all provided device I/O ports. The list of well-known ports contains 11 ranges, containing ports of emulated devices such as the legacy interrupt-controller, serial, and VGA. Another uncommon method to enumerate ISA devices is described in different legacy ISA Plug & Play specifications. Unfortunately, to the best of our knowledge, no x86 hypervisor provides support for this enumeration method.

B. TESSERACT

TESSERACT is a complex instruction set interpreter. We implemented specific opcodes targeting common interactions with hypervisors. Most opcodes are available in multiple variants: opcodes such as read and write are implemented for each interface. Additionally, there are variants using user-provided dictionaries of data values and offsets. We also implemented opcodes, which use "repeat string operation"-prefix based instruction such as `REP/REPE/REPZ/REPNE/REPNZ` to improve performance by limiting the number of emulated instructions and the coverage by using additional x86 opcodes. This also includes instructions accessing MMIO as well as port I/O regions. Overall, TESSERACT implements the following opcode handler:

- **write_mmio(region_id, offset, data)** writes a single word `data` to the address given by `region_id+offset`. Available for 8, 16 and 32-bit words.

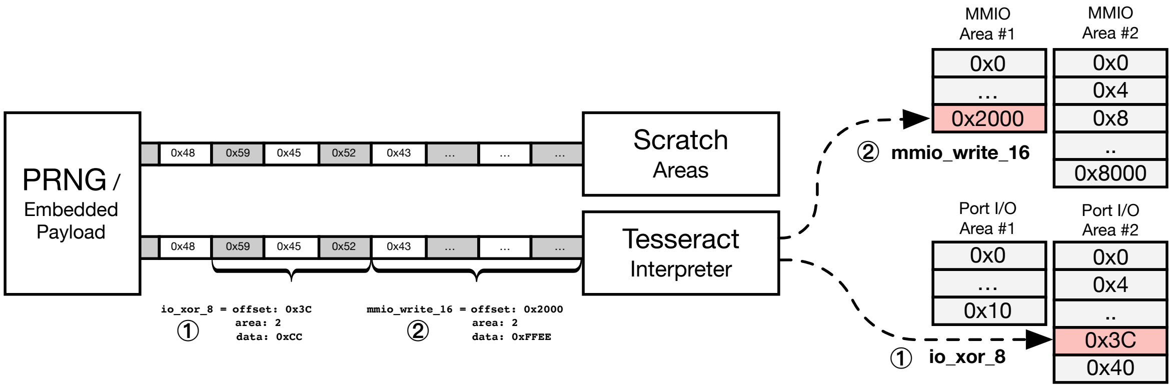


Fig. 4: TESSERACT consuming a byte string provided either by a PRNG or an embedded payload. Upon receiving the byte string, TESSERACT decodes it into opcodes such as ① and ②. It then calls the handler that actually performs I/O operations.

- **read_mmio(region_id, offset)** reads a single word from the address given by `region_id+offset`. Available for 8, 16 and 32-bit words.
- **xor_mmio(region_id, offset, mask)** reads a single word from the given address, and writes it back after applying the given XOR mask. Available for 8, 16 and 32-bit words.
- **bruteforce_mmio(region_id, offset, data, num)** writes `num` consecutive `data` words to the given address. Available for 8, 16 and 32-bit words.
- **memset_mmio(region_id, offset, data, num)** writes the word `data` to `num` consecutive addresses, beginning at the given address. Available for 8, 16 and 32-bit words.
- **writes_mmio(region_id, offset, data, num)** same as **memset_mmio**, however it uses a `rep` prefixed instruction to perform the task, testing instruction emulation.
- **reads_mmio(region_id, offset, num)** same as **writes_mmio**, but instead of writing data, it reads it.
- **mmio_write_scratch_ptr(region_id, offset, scratch-id, scratch-offset)** writes a pointer to the given offset in the scratch area to the address in the given MMIO region.
- ***_io()** all opcodes accessing MMIO regions are implemented for I/O ports as well.
- **write_msr(msr_num, mask)** writes to a *Machine Specific Registers* (MSR). This operation is limited to a list of ≈ 240 well-known MSRs. The mask is xored into the selected MSR.
- **hypercall(eax, ebx, ecx, edx, esi)** executes arbitrary hypercalls using the given registers as arguments. This instruction is specific to KVM.
- **vmport(ecx, ebx)** executes arbitrary vmport hypercalls with the registers set to the arguments to the hypervisor. This instruction is specific to VMware and other hypervisors which implement this hypercall interface.

The TESSERACT interpreter uses a textbook-style bytecode interpreter design that we briefly describe in the following. The bytecode stream is interpreted in a loop. In each iteration, the current bytecode instruction is decoded. The first byte encodes the opcode. Once the interpreter knows the opcode type, the corresponding handler is called. The opcode handler parses the arguments for this opcode as described above. Then, the handler performs the basic operation (such as accessing a given address). Lastly, the instruction pointer is increased by the size of the current opcode and the next iteration of the interpreter loop is executed.

There are two ways to provide bytecode instructions to our fuzzer. If no bytecode string is supplied from the outside, TESSERACT uses a Pseudorandom Number Generator (PRNG) to generate random instructions. We use a fast and small non-cryptographic PRNG with a large average cycle length ($> 2^{126}$) called `jsf32` [1]. Since our fuzzer uses the PRNG as the source of an infinite stream of instructions, changing the initial state of the PRNG allows us to run different deterministic fuzzing campaigns. An illustration of this process is given in Figure 4. The alternative way is to embed a fixed bytecode string into the OS image. This mechanism is used by the minimization script and can be used to generate proof-of-concept images of HYPER-CUBE to reproduce findings.

Using the ability to inject a static bytestring into the fuzzing image, we build a test case minimization script. After the fuzzer found an initial PRNG state that leads to a crash, we regenerate the same bytestring using the same state. This bytestring is often very large (millions to tens of millions of instructions). The minimization script removes a random subslice of instructions with a length up to 50% of the input. The resulting bytestring is embedded into an HYPER-CUBE OS image, and the image is booted in the VM. If the resulting image still produces a crash, the script repeats this process, otherwise the modification is reverted, and the process is repeated. After obtaining a minimal bytestring, the decompiler can be used to translate it into a human readable form. The decompiler is implemented as a modified version of the interpreter, that emits C code for each opcode handler call.

V. Evaluation

We use our prototype implementation of HYPER-CUBE to evaluate the results of our design choices. In particular, we aim to answer the following four research questions:

- **RQ 1.** Are we able to uncover new bugs and vulnerabilities in different hypervisors using HYPER-CUBE?
- **RQ 2.** Are we able to rediscover previously known vulnerabilities, such as the QEMU vulnerability VENOM (CVE-2015-3456) or vulnerabilities found by other hypervisor fuzzers?
- **RQ 3.** How does HYPER-CUBE perform in terms of test coverage compared to other hypervisor fuzzing approaches?
- **RQ 4.** What is the performance impact of our design choice compared to other hypervisor fuzzing setups?

To test whether HYPER-CUBE is able to uncover new bugs and to answer **RQ 1.**, we used HYPER-CUBE to evaluate six different open and closed source hypervisor implementations. We found bugs in all of them. In total, HYPER-CUBE found over 50 bugs that we all manually analyzed. At the time of submission, we reported and got confirmation for 49 bugs, report and confirmation are still in process for the remaining 5 bugs. Until now, 43 CVEs were assigned.

Similarly, to answer **RQ 2.**, we picked a set of previously known, critical security vulnerabilities in QEMU and tested if HYPER-CUBE is able to rediscover them in a reasonable time frame. Specifically, we picked four well-known CVEs, and we also reproduced the experiments performed by the authors of VDF. Overall, we show that HYPER-CUBE is able to rediscover all bugs and even finds bugs that VDF missed.

We also measure the code coverage achieved by HYPER-CUBE when fuzzing various QEMU modules (**RQ 3.**). We compare our measurements with the results reported by the authors of VDF. We are able to produce more code coverage on nearly all devices. Additionally, we achieve this code coverage in significantly less time. More specifically, our experiment only took ten minutes to outperform VDF in terms of bug finding capability as well as code coverage. In contrast, VDF used 60 days of computation for their experiments.

To measure the performance impact of our individual design choices and to answer **RQ 4.**, we perform three different experiments. First, we analyze the throughput of TESSERACT when compared to compiler-based approaches. Second, we compared the time it takes to re-initiate the fuzzing process after a crash of the targeted hypervisor. This time is especially relevant in case of successful bug discovery or when side effects might corrupt essential data structures resulting in full system reboots. Last, we measure the memory usage of our implementation.

A. Evaluation Setup

All experiments were performed on desktop machines with an Intel i7-6700 processor (4 cores) and 24 GB of RAM. Unless stated otherwise, all experiments were performed on host systems running Ubuntu 16-04 LTS. No seed inputs were used in any experiment, as our fuzzer generates the stream of bytecode instructions using a PRNG. Since our design is independent of the hypervisor under test, we can test a wide variety of different hypervisor implementations. In particular, we evaluated against the following six hypervisors: QEMU/KVM (`4.0.1-rc4`), Bhyve (`12.0-RELEASE`), ACRN (`29360 Build`), VirtualBox (`5.1.37_Ubuntu r122592`), VMware Fusion (`11.0.3`) and Parallels Desktop (`14.1.3`).

While the design of HYPER-CUBE is quite flexible, it still has some limitations in the implementation. We require 32-bit UEFI or GRUB assisted booting. Since Hyper-V and its type 2 virtual machines require a 64-bit UEFI bootloader, we do not support this hypervisor in our current implementation. This could be changed by extending the bootloader or the OS boot routine. Additionally, we have limited support for para-virtualized virtual devices and therefore we excluded para-virtualization hypervisors such as Xen, VMware ESXi and Hyper-V from our evaluation set. Note that this does not pose a limitation to the techniques presented: support for para-virtualization could be added by extending the boot procedure and the instruction set in TESSERACT.

B. Finding New Vulnerabilities

To demonstrate that HYPER-CUBE is versatile and effectively able to uncover bugs in different hypervisors, we tested current versions of different hypervisor implementations. As noted earlier, we were able to find bugs in all tested hypervisors and obtained 43 CVEs so far. An overview of all bugs found is given in Table II. Overall, we reported 54 bugs, a more specific breakdown is as follows: in total, we found eight memory corruption bugs. Additionally, in one case, we were able to crash the entire host kernel (Bhyve - the standard hypervisor used in FreeBSD). Three of the bugs caused a deadlock of the hypervisor itself, effectively making it useless. The remaining bugs cause different kinds of crashes in the hypervisor. Notably, 26 bugs were triggering assertion failures. While these bugs crashed the VMM and CVEs were assigned by MITRE, Red Hat Product Security does not consider these as security issues. In total, 12 of the 43 CVEs assigned are due to such assertion failures.

The high number of bugs found in all tested hypervisors demonstrates that HYPER-CUBE is able to uncover large quantities of novel bugs (**RQ 1.**). To better understand the kind of bugs HYPER-CUBE is able to uncover, we provide more detailed descriptions for three of them.

1) Case Study: Bhyve `rep movs` emulation failure: HYPER-CUBE found a bug in the instruction emulator of Bhyve and the way some instructions are handled if the devices emulation is performed by the hypervisor. During fuzzing, TESSERACT executed a `rep movs` instruction, which was targeting the APIC MMIO region. Unlike all other MMIO regions, Bhyve failed to emulate the `rep movs` instruction, which targets the APIC controller. These instructions were emulated in the host kernel instead of the ring 3 emulator. This bug can be exploited to crash the host kernel.

2) Case Study: QEMU `qxl` arbitrary read access: HYPER-CUBE uncovered a vulnerability in the `qxl` VGA emulation, which leads to control of a pointer to the VGA mode array and to an arbitrary out-of-bounds read access. To

TABLE II: Reported bugs found by Hyper-Cube.

Hypervisor	Asserts	Null-Pointer	FP Exception	Memory Corruptions	Deadlocks	Kernel DoS	Other[a]	Total / CVEs
QEMU / KVM	12	4	-	5	2	-	-	23 / 23
Bhyve	6	4	1	-	1	1	-	13 / 13
VirtualBox	3	1	4	3	-	-	-	11 / 5
VMware Fusion	3	-	-	-	-	-	1	4 / -
Parallels Desktop	1	1	-	-	-	-	-	2 / 1
ACRN	1	-	-	-	-	-	-	1 / 1
Total	**26**	**10**	**5**	**8**	**3**	**1**	**1**	**54 / 43**

[a] Not yet classified.

trigger this bug, it is necessary to use two different interfaces. `qxl` provides a PIO write region to select a VGA mode within the VGA `modes` array. The VGA `modes` array is stored in a presumably read-only PCI-MMIO area named `QXL_ROM` region. However, due to an implementation mistake, this area is mapped writable to the guest. This allows us to control the metadata. Finally, when the window is resized, the size is read from a pointer in the `modes` array. By changing it, we can control the size of the buffer. This vulnerability allows an out-of-bounds read access in a $4GB + 8KB$ range relative to the allocation.

3) Case Study: VirtualBox `intel-hda` heap corruption: In the `intel-hda` sound card emulator implementation of Virtualbox, Hyper-Cube uncovered a heap corruption issue. The `command output ring buffer` (`CORB`) size can be set to either two entries (8 bytes), 16 entries (64 bytes) or 256 entries (1KiB, default) with an MMIO write to the `CORBSIZE` register. The `CORBSIZE` MMIO write handler sets the selected size and allocates a buffer with the given size. The `intel-hda` device can then be reset by an MMIO write to the `global control` (`GCTL`) register. This handler resets the buffer size to the default of 256 entries (1KiB) and clears the buffer with a memset of size 1KiB. Clearing the buffer can lead to a heap out-of-bound write of up to 1016 bytes when the CORB size was previously set to 2 entries, and the buffer only has a size of 8 bytes. The vulnerability was present in the Ubuntu version of VirtualBox and has now been fixed.

C. Rediscovering Known Vulnerabilities

It is a well-known fact that fuzzers are typically able to find bugs in poorly tested software. We therefore ensure that our fuzzer is also able to find bugs previously found by other methods.

1) Old CVEs: Since there is very limited related work on hypervisor fuzzing, we picked some well-known CVEs for this experiment. In a first set, we chose some high-impact bugs with a CVSS $>$ 7.0. Additionally, we tried to rediscover all bugs found by VDF. In total we analyzed four known bugs, displayed in Table III. We were able to find all bugs in a matter of seconds. Since all the bugs are inside of QEMU device emulators, we evaluated our fuzzer both with QEMU running TCG mode as well as KVM based virtualization. Even though TCG is typically considered slower at emulation than hardware supported virtualization techniques such as KVM, fuzzing QEMU TCG was drastically faster. This is due to the fact that we produce a workload that mostly consists of interactions with the hardware. In KVM, each interaction requires multiple expensive context switches, while no such context switch takes place in QEMU TCG mode.

Unfortunately, most fuzzers developed by practitioners in the industry are neither released nor properly evaluated by the scientific community. However, Tang et al. explicitly mention that they needed one and a half hours to rediscover the VENOM vulnerability [46]. Hyper-Cube typically requires less than ten seconds to discover it (as illustrated in Table III).

2) Hyper-Cube vs VDF: In addition to reproducing well-known CVEs, we compare Hyper-Cube against VDF by reproducing the results reported in their paper. Unfortunately, the authors of VDF did not provide detailed descriptions of the crashes found and we were not able to obtain a copy of VDF. However, we used the same version of QEMU and performed similar experiments (displayed in Table IV). It should be noted that we excluded their experiment against the Trusted Platform Module (TPM), as it was manually added and is not part of the default configuration of QEMU. Furthermore, VDF uses extensive traces obtained from real OS boot processes, while we did not provide any seed inputs or dictionaries to Hyper-Cube. Similar to the authors of VDF, in this experiment we limited fuzzing to the target device by ignoring all other devices during device enumeration. We compare our results with the values reported by the authors of VDF. We were able to find crashes in all three devices VDF crashed. Beyond merely reproducing all of the bugs found by VDF, we even managed to find crashes in five additional devices that VDF missed.

The fact that we managed to find bugs in all the devices that VDF broke is surprising, since VDF uses coverage feedback and is able to improve the performance drastically by targeting individual components in isolation. On the other hand, our approach has to deal with the whole system and is unable to make use of coverage feedback.

Note that we achieved these results after testing the same targets for only 10 minutes, while VDF fuzzed the same targets for 60 days. Like the authors of VDF, we limited the fuzzer to test only the port I/O and MMIO areas related to the targeted emulator in this experiment. We have three observations on why we are able to outperform VDF with such a margin. First of all, VDF only interacts with MMIO and port I/O to test the target hypervisor. Our approach is able to trigger more code by other means such as DMA. Second, VDF has to reset the device state after each test case to obtain coverage measurements that are undisturbed by previous tests. This has two downsides: (i) resetting the state of the target

TABLE III: Previously known vulnerabilities in QEMU / KVM found by HYPER-CUBE (average time in seconds over 20 runs each ± standard deviation).

CVE (Bug)	CVSS	TCG (sec)	KVM (sec)
2015-3456 (VENOM)	7.7	5.8 ± 6.8	$49.7 \pm 50, 6$
2017-2615 (Cirrus VGA)	9.1	5.1 ± 3.0	86.7 ± 70.4
2017-2620 (Cirrus VGA)	9.9	19.6 ± 14.9	331.32 ± 303.06
2015-8743 (NE2000)	7.1	47.7 ± 42.0	203.5 ± 341.7

is a slow process itself, and (ii) most inputs produced by VDF are small. For example, the crashes found are only 2500 bytes large on average. As a consequence, VDF incurs an expensive reset after every few hundred operations. HYPER-CUBE, on the other hand, is able to perform many billions of operations without such overhead. We assume that HYPER-CUBE is able to execute orders of magnitude more operations per second compared to VDF. However, a precise comparison is not possible, since the authors of VDF did not report any numbers. More importantly, VDF also replays slightly mutated versions of the same test cases over and over again, reducing the variance in test scenarios. Normally, feedback-driven fuzzers are expected to use this as an advantage to learn what test cases provide novel behavior and which ones do not. However, due to the fact that TESSERACT already encodes a significant amount of prior information and is able to guess interesting inputs with high probability, it seems that in this case the performance losses do not make up for the incremental learning. Last, VDF fuzzes emulated devices in isolation, depriving them from a set of real-world interactions such as interrupts. Our approach attacks the target device in a full environment featuring all interactions.

As a consequence of both experiments, we conclude that HYPER-CUBE is able to significantly outperform state-of-the art fuzzers (**RQ 2.**). This also demonstrates the advantage that a custom interpreter provides by focusing on useful interactions. This finding is inline with recent research on grammar-based fuzzing [10], [12], [29], [36], [39].

D. Performance

After demonstrating HYPER-CUBE's ability to find novel and well-known bugs, we also evaluate other aspects of its performance. Besides the crashes found, the authors of VDF also reported branch coverage achieved on various devices. Similarly, we measure the coverage that was achieved and compare it against the values reported by VDF. To give a general feeling for the performance criteria, we also evaluate metrics such as throughput, boot time, and memory usage.

1) Coverage: To estimate the quality of our test results and to compare it against VDF, we measured the coverage produced by our approach. As mentioned earlier, we were not able to obtain a copy of VDF and thus can only compare our results with the numbers published by the authors of VDF. It should be noted that the authors of VDF chose to perform their experiments on 6 cores for a total of roughly 60 days each. We choose to run our experiments on a single core while the `gcov` profiler was actively measuring the coverage produced in the device emulator. Overall, we tested each device for ten minutes. Measuring the coverage introduces a rather heavy performance penalty, however we err on the side of caution by ignoring this fact and thus underestimating the performance of HYPER-CUBE. Additionally, the authors of VDF created seed traces for all MMIO interactions from real OS interactions or test cases that already provide some good initial coverage. In contrast to VDF, we have no tracing utility and therefore started with a blank state without any seed files. Even though we start from this disadvantaged situation, we drastically outperform VDF in all but one case. Unfortunately, we are unaware of the exact reasons why we are able to outperform VDF with such a margin.

TABLE IV: Branch coverage and bugs found by HYPER-CUBE and VDF

	VDF			HYPER-CUBE		
Device	Cov	Bug	Time	Cov	Bug	Time
AC97	53.0%	✓	59 days	**80.75%**	✓	10 min
CS4231a	56.0%		65 days	**76,95%**		10 min
ES1370	72.7%		69 days	**94,77%**		10 min
Intel-HDA [a]	58.6%	✓	59 days	**66.37%**	✓	10 min
SoundBlaster	81.0%		58 days	**85.89%**		10 min
Floppy	70.5%		57 days	**83.38%**	✓	10 min
Parallel	**42.9%**		25 days	38.61%		10 min
Serial	44.6%		62 days	**79.90%**		10 min
IDE Core	27.5%		65 days	**68.81%**	✓	10 min
EEPro100	75.4%		62 days	**83.94%**	✓	10 min
E1000 [b]	81.6%	✓	61 days	**81,69%**	✓	10 min
NE2000 (PCI)	71.7%		58 days	**72.78%**	✓	10 min
PCNET (PCI)	36.1%		58 days	**83,08%**		10 min
RTL8139	63.0%		58 days	**73.73%**	✓	10 min
SDHCI [c]	**90.5%**	✓	62 days	77.04%	✓	10 min

[a] We fixed a bug in this device emulator to perform this experiment properly. Otherwise, HYPER-CUBE will find this crash within seconds.

[b] Due to a deadlock, which was found by HYPER-CUBE within seconds, this experiment was performed on the latest version of QEMU (`4.0.1-rc1`) instead. Even in this version, HYPER-CUBE was able to uncover a new deadlock bug within minutes.

[c] Due to the high amount of memory corruption bugs and deadlocks, which were found by HYPER-CUBE within seconds, we launched QEMU multiple times to perform a proper coverage measurement.

However, we investigated the coverage that we found, and noticed that many interactions require that we provide the device with a pointer to our memory that contains certain data, or perform interactions with interrupts neither of which is done by VDF. Using our multi-dimensional fuzzing approach, we can interleave these operations easily. Together with the observations on performance made in Section V-C2, this might explain the significant difference in coverage.

2) Interpreter Throughput: To further substantiate the performance impact of our design, we compare our interpreter-based approach against a compiler based approach using GCC. Similar compiler-based approaches are used by multiple system-level fuzzers such as CFAFT [18].

For this comparison we measure the time that GCC uses to compile C code equivalent to 16MB of opcodes. We generate this C code via our own opcode decompiler. Since fuzzers like CFAFT usually use smaller test cases, and to prevent GCC from using all memory available (since the 16MB opcode yield to a significantly large chunk of C code, around 1.8 million lines of code), we generate separate C code files from every ten thousands of opcodes and measure the overall time for compiling these files. We omitted to include the time it takes to actually load the resulting kernel module and to execute it, as the minimal Linux image does not contain a compiler

TABLE V: Throughput of TESSERACT vs. GCC (with -O0)

Approach	Runtime
TESSERACT	0.028 sec
GCC each slice (10k)	0.356 sec
GCC Total (1.8 million)	64.522 sec

TABLE VI: Boot time comparison using QEMU 4.0.1-rc4 ASAN (average over 20 runs each ± standard deviation).

Mode	Linux-4.15	HYPER-CUBE Probing	HYPER-CUBE
TCG	5.856 ± 0.027 sec	0.487 ± 0.038 sec	0.401 ± 0.007 sec
KVM	2.337 ± 0.053 sec	0.785 ± 0.079 sec	0.219 ± 0.011 sec

environment. We therefore overestimate the throughput of compiler-based approaches - a conservative choice. We then measure the time it takes for our HYPER-CUBE to generate, load and execute the same 16MB of opcode. The results are displayed in Table V. Note that the overall time for compiling 1.8 million lines of code for GCC is measured by the sum of the time it takes for every ten thousands opcodes C file to be compiled. Overall, our approach has a throughput that is roughly 2,300 times higher than compiler-based approaches. It should be noted that this number is calculated conservatively, as it does not yet contain the cost for generating large amounts of C code, nor for actually loading or executing the compiled executable.

E. Restart After Crash

Another important performance metric for fuzzer is how fast they can recover from crashes. Since we use a very lightweight custom OS, we can boot a lot faster than COTS OSs. This offers a significant advantage over other designs where the fuzzer runs in a kernel module or driver as part of a COTS OS. We compared the boot time of HYPER-CUBE against a minimal Linux BusyBox image. We booted 20 times, and measured the average time until the fuzzer started. HYPER-CUBE OS can be used in two modes: in an initial boot process, HYPER-CUBE OS uses its I/O enumeration abilities to generate a map of available ranges. After this first boot process, future reboots can reuse this mapping to save time. The results are displayed in Table VI. Overall it took Linux an average of 2.3 seconds to boot, while HYPER-CUBE OS is able to boot in only 0.22 seconds, resulting in a 10 times speedup. Depending on the fuzzing workload, the actual performance gains might be much lower, particularly in cases where the fuzzer does not find many crashes and therefore reboots are very uncommon events.

F. Memory Usage

Given that typically fuzzing is performed in parallel, the memory footprint of the target is rather important. However, a barebone Linux only uses some 30-50 MB of memory, much less than a typical hypervisor. HYPER-CUBE OS requires even less memory (around 10MB). However, since neither 10MB nor 50MB contributes significantly to the overall memory usage, the effect of saving memory on the fuzzing performance is negligible.

TABLE VII: Automatic emulator detection rate of HYPER-CUBE. #scanning indicates the number of interfaces identified by our scanning. #well-known denotes the number of ports that the scanning did not find, but were contained in our list of well-known ports. #baseline is the number of interfaces, as reported by the hypervisor.

Hypervisor	#scanning	#well-known	#baseline	%found
VirtualBox MMIO Scan	12	0	12	100.00%
QEMU / KVM MMIO Scan	5	0	5	100.00%
VirtualBox Port I/O Scan	185	29	203	91.13%
QEMU / KVM Port I/O Scan	352	7	357	98.60%
Total				97.43%

G. Emulator Detection

During the device enumeration process, HYPER-CUBE can enumerate all present PCI and PCI-Express devices as well as their I/O ports and MMIO regions. Unfortunately, there is no such mechanism to systematically enumerate ISA devices. HYPER-CUBE contains an approximate solution to find interesting I/O ports. We were able to obtain the ground truth by using internal tools for both QEMU and VirtualBox. Using this ground truth, we were able to measure the effective detection rate of our approach. As it is illustrated in Table VII, our method is highly effective at uncovering relevant I/O addresses. As expected, all MMIO regions were correctly identified. It should be noted that HYPER-CUBE also offers the ability to add additional ranges to the target list if a manual check or documentation indicates the need.

Based on the performance evaluation, we can say that HYPER-CUBE drastically outperforms existing, state-of-the-art fuzzers such as VDF and the fuzzer designed by Tang et al. [46] (**RQ 3.**). Our design based on a custom operating system and a custom bytecode interpreter providing multi-dimensional fuzzing and a high throughput interpreter are the main reason for this. The boot time and memory overhead are far less impactful on our performance gains. This answers research question **RQ 4.**.

VI. RELATED WORK

In the last five years, fuzzing has been a very active field of research. Triggered by the publication and widespread success of AFL, a myriad of research projects aimed to strengthen the bug finding ability of fuzzers in various scenarios. In most cases, the algorithms used for scheduling [13]–[15], [41], [49], feedback [2], [21], [31], [33], and mutations [10], [11], [29], [36], [39] were improved. In other projects, techniques based on concolic execution [22]–[24], [28], [34], [45], [48], [53], [55] or taint tracking [17], [40] were combined with fuzzing to solve ”fuzzing roadblocks” such as magic bytes. Lastly, the primitives used to run the fuzzer such as the mechanism of spawning processes [51] or obtaining coverage [4], [44] were redesigned to provide more performance. Additionally, fuzzing was adopted to a wide range of targets: while AFL only targets ring 3 applications, researchers quickly adapted the design to kernel fuzzing [3], [5], [44], hypervisor fuzzing [30] and even used fuzzing to test neural networks [37], [50].

A custom OS was used to test hypervisors by previous projects. For example CrashOS [32] is a suite of handwritten test cases used to find regressions and well-known bugs, running as an operating system. As it does not provide any way to uncover new bugs, it is orthogonal to our work. Intel CHIPSEC suite provides different components to fuzz emulated devices [25]. Similarly, Ormandy created a fuzzer to create random I/O accesses within different hypervisors [35]. Henderson et al. [30] introduced an approach to fuzz specific device emulators by modifying an open source hypervisor and providing AFL support. Tang et al. [46] introduced a similar approach to provide interoperability between AFL and QEMU by implementing an interfaces on top of QEMU's SeaBIOS. Finally, Amardeep Chana from MWR Labs and Microsoft Security Research and Defense, introduced fuzzers for fuzzing the Hyper-V hypercall interface (VMBus) [16], [42].

VII. Discussion

In this paper, we introduced an architecture for a non-coverage-guided hypervisor fuzzer. Surprisingly, we found that even though our approach makes far fewer assumptions than state-of-the-art coverage-guided fuzzers, it is still able to outperform them in all relevant metrics (coverage found, bugs found, coverage found per time, and bugs found per time). This goes to show that for smaller codebases such as individual device emulators, a very high fuzzing throughput and a reasonable choice of input format, are still the most important properties of a fuzzer. Yet, on the other hand, our evaluation (Table IV) also shows that for one of the most complex devices (QEMU's `sdhci` emulator) with deeply nested conditions, coverage guidance was actually improving the coverage.

A. Coverage-Guided Hypervisor Fuzzing

It would be very interesting to see how a combination of Hyper-Cube and coverage-guided fuzzing would perform. In particular, it might be relevant to try to combine the high-throughput fuzzing of our current approach with the incremental learning provided by coverage-guided fuzzing. We believe that the basic problem with VDF is that it introduces a very high overhead to try a single new operation. To uncover new coverage or new interesting behavior, the state in the hypervisor needs to be constructed one operation at a time. Since the space of possible operations is very large, it seems that letting the state grow until something interesting (in this case, a crash) happens drastically increases the throughput. We can execute thousands of opcodes on the current state in the time it takes to reset the device state once. As a consequence, the probability of finding an operation that triggers new behavior in any given time frame increases drastically.

B. Hyper-V

Recently, much focus was put on fuzzing Hyper-V and its VMBus as well as all synthetic devices [47]. Currently, Hyper-Cube cannot boot in a Hyper-V generation 2 virtual machine because it does not support 64-bit UEFI mode. Unfortunately, to get access to fuzz the implementation of synthetic devices, Hyper-V requires the OS to boot in this mode. This is by no means a limitation of the design, but merely of the implementation. Currently, Hyper-Cube has only limited support for para-virtualization: while Hyper-Cube is able to access all interface used in para-virtualization, it is not very efficient at doing so. As each hypervisor implements its own specific communication channel, we currently only implement generic operations that will be unlikely to trigger interesting coverage on their own. Both feedback-driven fuzzing or specific opcodes would greatly increase the ability to deal with that kind of situation.

VIII. Conclusion

In this paper, we presented a generic and comprehensive technique to find bugs in hypervisors and device emulators. Hyper-Cube discovered 54 different bugs in proprietary and open source hypervisors. However, the most notable result from this paper is not that fuzzers are able to discover bugs in hypervisors. Instead, we found it more interesting that our black-box design is able to outperform at least two coverage-guided hypervisor fuzzers. We believe that our results indicate that in situations where the target consumes a continuous stream of inputs, the high-throughput design outweighs the precision offered by a coverage-guidance. Future fuzzer designs should incorporate this insight to create more efficient coverage-guided fuzzers.

Acknowledgments

This work was supported by the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States – EXC 2092 CaSa – 39078197. In addition, this work was supported by the European Union's Horizon 2020 Research and Innovation Programme (ERC Starting Grant No. 640110 (BASTION) and 786669 (REACT)) and the German Federal Ministry of Education and Research (BMBF, project HWSec – 16KIS0592K). The content of this document reflects the views only of their authors. The European Commission/Research Executive Agency are not responsible for any use that may be made of the information it contains.

References

[1] A small noncryptographic PRNG. http://www.burtleburtle.net/bob/rand/smallprng.html. Accessed: January 20, 2020.

[2] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/. Accessed: January 20, 2020.

[3] Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Accessed: January 20, 2020.

[4] Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: January 20, 2020.

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

[6] Viridian Fuzzer. https://github.com/mwrlabs/ViridianFuzzer. Accessed: January 20, 2020.

[7] XenFuzz. https://www.openfoo.org/blog/xen-fuzz.html. Accessed: January 20, 2020.

[8] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. *SIGOPS Oper. Syst. Rev.*, 40(5), October 2006.

[9] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[10] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[11] 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)*, 2019.

[12] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: Synthesizing structure while fuzzing. In *USENIX Security Symposium*, 2019.

[13] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[15] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[16] Amardeep Chana. Mwr-labs: Ventures into hyper-v - fuzzing hypercalls. `https://labs.mwrinfosecurity.com/blog/ventures-into-hyper-v-part-1-fuzzing-hypercalls/`. Accessed: January 20, 2020.

[17] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[18] YoungHan Choi, HyoungChun Kim, HyungGeun Oh, and Dohoon Lee. Call-flow aware api fuzz testing for security of windows systems. In *2008 International Conference on Computational Sciences and Its Applications*, 2008.

[19] Bryan Ford and Erich Stefan Boleyn. Multiboot specification. `https://www.gnu.org/software/grub/manual/multiboot/multiboot.html`, 1995.

[20] Bryan Ford and Erich Stefan Boleyn. Multiboot specification version 2.0. `https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html`, 2016.

[21] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[22] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[24] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[25] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, and Yuriy Bulygin. Attacking hypervisors via firmware and hardware. *Black Hat USA*, 2015.

[26] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, and Yuriy Bulygin. Attacking hypervisors via firmware and hardware. *Black Hat USA*, 2015.

[27] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. Antifuzz: Impeding fuzzing audits of binary executables. In *USENIX Security Symposium*, 2019.

[28] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, 2013.

[29] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[30] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. Vdf: Targeted evolutionary fuzz testing of virtual devices. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2017.

[31] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[32] Airbus Security Lab. Crashos: A tool dedicated to the research of vulnerabilities in hypervisors by creating unusual system configurations. `https://github.com/airbus-seclab/crashos`. Accessed: January 20, 2020.

[33] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[34] David Molnar, Xue Cong Li, and David Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[35] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. In *CanSecWest 2007*, 2007.

[36] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Zest: Validity fuzzing and parametric generators for effective random testing. *arXiv preprint arXiv:1812.00078*, 2018.

[37] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. deepxplore: Automated whitebox testing of deep learning systems.

[38] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[39] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *arXiv preprint arXiv:1811.09447*, 2018.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

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

[42] Microsoft Security Research and Defense. Fuzzing para-virtualized devices in hyper-v. `https://blogs.technet.microsoft.com/srd/2019/01/28/fuzzing-para-virtualized-devices-in-hyper-v/`. Accessed: January 20, 2020.

[43] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[44] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security Symposium*, 2017.

[45] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[46] Jack Tang and Moony Li. When Virtualization Encounters AFL. `https://www.blackhat.com/docs/eu-16/materials/eu-16-Li-When-Virtualization-Encounters-AFL-A-Portable-Virtual-Device-Fuzzing-Framework-With-AFL-wp.pdf`. Accessed: January 20, 2020.

[47] Microsoft Virtualization Security Team. Fuzzing para-virtualized devices in hyper-v. `https://blogs.technet.microsoft.com/srd/2019/01/28/fuzzing-para-virtualized-devices-in-hyper-v/`. Accessed: January 20, 2020.

[48] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

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

[50] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Hongxu Chen, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. Coverage-guided fuzzing for deep neural networks. *arXiv preprint arXiv:1809.01266*, 2018.

[51] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[52] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *IEEE Symposium on Security and Privacy*, 2019.

[53] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[54] Michał Zalewski. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. Accessed: January 20, 2020.

[55] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.