VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices

Andrew Henderson[1(✉)], Heng Yin[2], Guang Jin[1], Hao Han[1], and Hongmei Deng[1]

[1] Intelligent Automation, Inc., Rockville, MD 20855, USA
hendersa@icculus.org, {gjin,hhan,hdeng}@i-a-i.com
[2] University of California, Riverside, CA 92521, USA
heng@cs.ucr.edu

AQ1

Abstract. As cloud computing becomes more and more prevalent, there is increased interest in mitigating attacks that target hypervisors from within the virtualized guest environments that they host. We present VDF, a targeted evolutionary fuzzing framework for discovering bugs within the software-based virtual devices implemented as part of a hypervisor. To achieve this, VDF selectively instruments the code of a given virtual device, and performs record and replay of memory-mapped I/O (MMIO) activity specific to the virtual device. We evaluate VDF by performing cloud-based parallel fuzz testing of eighteen virtual devices implemented within the QEMU hypervisor, executing over two billion test cases and revealing over one thousand unique crashes or hangs in one third of the tested devices. Our custom test case minimization algorithm further reduces the erroneous test cases into only 18.57% of the original sizes on average.

Keywords: Virtualization · Fuzzing · Device testing · Security

1 Introduction

As cloud computing becomes more prevalent, the usage of virtualized guest systems for rapid and scalable deployment of computing resources is increasing. Major cloud service providers, such as Amazon Web Services (AWS), Microsoft Azure, and IBM SoftLayer, continue to grow as demand for cloud computing resources increases. Amazon, the current market leader in cloud computing, reported that AWS's net sales exceeded *7.88 billion USD* in 2015 [2], which demonstrates a strong market need for virtualization technology.

This popularity has led to an increased interest in mitigating attacks that target hypervisors from within the virtualized guest environments that they host.

This document has been approved for public release: 88ABW-2016-3973.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-3-319-66332-6_1) contains supplementary material, which is available to authorized users.

M. Dacier et al. (Eds.): RAID 2017, LNCS 10453, pp. 1–23, 2017.
DOI: 10.1007/978-3-319-66332-6_1

Unfortunately, hypervisors are complex pieces of software that are difficult to test under every possible set of guest runtime conditions. *Virtual hardware devices* used by guests, which are hardware peripherals emulated in software (rather than directly mapping to physical devices on the host system), are particularly complex and a source of numerous bugs [3–6]. This has led to the ongoing discovery of vulnerabilities that exploit these virtual devices to access the host.

Because virtual devices are so closely associated with the hypervisor, if not integrated directly into it, they execute at a higher level of privilege than any code executing within the guest environment. They are not part of the guest environment, per se, but they are privileged subsystems that the guest environment directly interacts with. Under no circumstances should activity originating from within the guest be able to attack and compromise the hypervisor, so effectively identifying potential vulnerabilities in these virtual devices is a difficult, but valuable, problem to consider. However, these virtual devices are written by a number of different authors, and the most complex virtual devices are implemented using thousands of lines of code. Therefore, it is desirable to discover an effective and efficient method to test these devices in a scalable and automated fashion without requiring expert knowledge of each virtual device's state machine and internal details.

Such issues have led to a strong interest in effectively testing virtual device code [9,28] to discover bugs or other behaviors that may lead to vulnerabilities. However, this is a non-trivial task as virtual devices are often tightly coupled to the hypervisor codebase and may need to pass through a number of device initialization states (i.e. BIOS and guest kernel initialization of the device) before representing the device's state within a running guest system.

Evolutionary fuzzing techniques (e.g., AFL [38]) has gained its popularity recently for its effectiveness in discovering crashes and hangs. It is widely used in industry, and most finalists in the DARPA Cyber Grand Challenge used it for vulnerability discovery. Several academic research papers soon appeared to further improve the effectiveness of evolutionary fuzzing, such as AFLFast [21], VUzzer [33], Driller [35], and DeepFuzz [22]. While these efforts greatly improve the state-of-the-art, they aim at finding defects within the entire user-level program, and cannot be directly applied to find bugs in virtual devices, for several reasons. First of all, the fuzz testing must be targeted at specific virtual device code, which is a rather small portion of the entire hypervisor code base. It must be *in-situ* as well, as virtual devices frequently interact with the rest of the hypervisor code. Last but not least, it must be stateful, since virtual devices need to be properly initialized and reach certain states to trigger defects.

To address these unique challenges, we propose Virtual Device Fuzzer (*VDF*), a novel fuzz testing framework that provides targeted fuzz testing of interesting subsystems (virtual devices) within complex programs. VDF enables the testing of virtual devices within the context of a running hypervisor. It utilizes *record and replay* of virtual device memory-mapped I/O (MMIO) activity to create fuzz testing seed inputs that are guaranteed to reach states of interest and initialize each virtual device to a known good state from which to begin each test. Providing proper seed test cases to the fuzzer is important for effective exploring

the branches of a program [25,34], as a good starting seed will focus the fuzzer's efforts in areas of interest within the program. VDF mutates these seed inputs to generate and replay fuzzed MMIO activity to exercise additional branches of interest.

As a proof of concept, we utilize VDF to test a representative set of eighteen virtual devices implemented within the QEMU whole-system emulator [19], a popular type-2 hypervisor that uses a virtualized device model. Whether QEMU completely emulates the guest CPU or uses another hypervisor, such as KVM [10] or Xen [18], to execute guest CPU instructions, hardware devices made available to the guest are software-based devices implemented within QEMU.

In summary, this paper makes the following contributions:

- We propose and develop a targeted, in-situ fuzz testing framework for virtual devices.
- We evaluate VDF by testing eighteen QEMU virtual devices, executing over 2.28 billion test cases in several parallel VDF instances within a cloud environment. This testing discovered a total of 348 crashes and 666 hangs within six of the tested virtual devices. Bug reports and CVEs have been reported to the QEMU maintainers where applicable.
- We devise a testcase minimization algorithm to reduce each crash/hang test case to a minimal test case that still reproduces the same bug. The average test case is reduced to only 18.57% of its original size, greatly simplifying the analysis of discovered bugs and discovering duplicate test cases that reproduce the same bug. We also automatically generate source code suitable for reproducing the activity of each test case to aid in the analysis of each discovered bug.
- We analyze the discovered bugs and organize them into four categories: excess host resource usage, invalid data transfers, debugging asserts, and multi-threaded race conditions.

2 Background

Within QEMU, virtual device code registers callback functions with QEMU's virtual memory management unit (MMU). These callback functions expose virtual device functionality to the guest environment and are called when specific memory addresses within the guest memory space are read or written. QEMU uses this mechanism to implement memory-mapped I/O (MMIO), mimicking the MMIO mechanism of physical hardware.

We have identified a model for guest activity that attempts to attack these virtual devices:

1. The virtual device is correctly instantiated by the hypervisor and made available to the guest environment.
2. The virtual device is correctly initialized via the guest's BIOS and OS kernel and is brought to a stable state during the guest boot process. Any needed guest kernel device drivers have been loaded and initialized.

3. Once the guest boots, the attacker acquires privileged access within the guest and attempts to attack the virtual devices via memory reads/writes to the MMIO address(es) belonging to these virtual devices.

Unfortunately, it is non-trivial to perform large-scale testing of virtual devices in a manner analogous to this model. The read/write activity would originate from within the guest environment, requiring the guest to completely boot and initialize prior to performing a test[1]. Because any read/write to a virtual device control register may change the internal state of the device, the device must be returned to a known good "just initialized" state prior to the start of each test.

While utilizing virtual machine (VM) state snapshots to save and restore the state of the guest is a potential solution, the time required to continually restore the state of the guest to a known good state makes this approach inefficient for large-scale testing. Consider the megabytes of system state data (guest RAM, CPU state, and device state and internal cache storage) required to restore a running VM to a known state. Even when ignoring the time required to retrieve such state information from secondary storage, megabytes of data within the snapshot must still be unserialized and placed into hypervisor data structures prior to the start of each test.

2.1 Understanding Guest Access of Virtual Devices

The flow of activity for virtual device access from within QEMU is shown in Fig. 1. This figure shows a KVM-accelerated QEMU hypervisor configuration. The guest environment executes within QEMU, and the virtual devices are provided to the guest by QEMU. CPU instruction execution and memory accesses, however, are serviced by the KVM hypervisor running within the host system's Linux kernel. A request is made from a guest process (a) and the guest kernel accesses the device on the process's behalf (b). This request is passed through QEMU's KVM interface to the KVM kernel module in the host's kernel. KVM then forwards the request to a QEMU virtual device (c). The virtual device responds (d) and the result is provided to the guest kernel (e). Finally, the guest process receives a response from the guest kernel (f).

Unlike the standard 0–3 ring-based protection scheme used by x86 platforms, virtualized systems contain two sets of rings: rings 0 through 3 on the host, and rings 0' through 3' on the guest. The rings within the guest are analogous to their counterparts on the host with one exception: the highest priority guest ring (ring 0') is at a lower priority than the lowest priority ring on the host (ring 3). While a guest environment may be compromised by malicious software, it is still safely contained within a virtualized environment. However, if malware were to compromise the hypervisor and gain host ring 3 privileges, it would effectively "break out" of the virtualization and gain the opportunity to attack the host.

[1] QEMU provides the *qtest* framework to perform arbitrary read/write activity without the guest. We discuss qtest, and its limitations when fuzz testing, in Sect. 3.

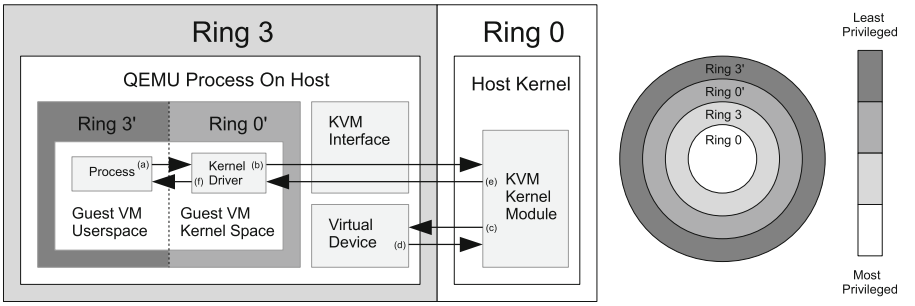


Fig. 1. Device access process for a device request originating from inside of a QEMU/KVM guest. Note that the highest level of privilege in the guest (ring 0') is still lower than that of the QEMU process (ring 3).

2.2 Understanding Memory Mapped I/O

Both physical and virtual peripherals provide one or more registers that control their behavior. By accessing these control registers, the hardware is instructed to perform tasks and provide information about the current state of the device. Each device's control registers are organized into one or more *register banks*. Each register bank is mapped to a contiguous range of guest physical memory locations that begin at a particular *base address*. To simplify interaction with these control registers, the registers are accessed via normal memory bus activity. From a software point of view, hardware control registers are accessed via reads and writes to specific physical memory addresses.

The x86 family of processors is unique because it also provides port I/O-specific memory (all memory addresses below `0x10000`) that cannot be accessed via standard memory reads and writes [29]. Instead, the x86 instruction set provides two special I/O-specific instructions, `IN` and `OUT`, to perform 1, 2, or 4 byte accesses to port I/O memory. Other common architectures, such as Alpha, ARM, MIPS, and SPARC, do not have this port I/O memory region and treat all control register accesses as regular memory-mapped I/O. For simplicity in our discussion, we use port-mapped I/O (PMIO) and memory-mapped I/O interchangeably throughout this paper.

Figure 2 shows where MMIO devices are mapped in guest physical memory on x86-based systems. PCI-based PMIO mappings occur in the addresses ranging from `0xC000` through `0xFFFF`, with ISA-based devices mapped into the sub-`0xC000` range. PCI devices may also expose control registers or banks of device RAM or ROM in the PCI "hole" memory range `0xE0000000-0xFFFFFFFF`.

While some ISA devices are historically mapped to specific addresses (for example, `0x3F8` for the COM1 serial port), other ISA devices can be configured to use one or more of a small set of selectable base addresses to avoid conflicts with other devices. PCI devices are far more flexible in the selection of their address mapping. At boot, the BIOS queries the PCI bus to enumerate all PCI devices connected to the bus. The number and sizes of the control register banks

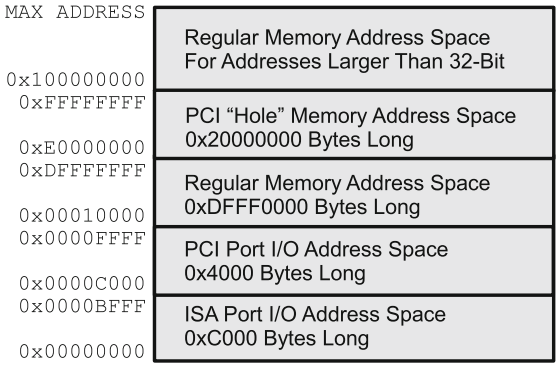


Fig. 2. The x86 address space layout for port- and memory-mapped I/O.

needed by each PCI device are reported to the BIOS. The BIOS then determines a memory-mapping for each register bank that satisfies the MMIO needs of all PCI devices without any overlap. Finally, the BIOS instructs the PCI bus to map specific base addresses to each device's register banks using the PCI base address registers (BARs) of each device.

However, PCI makes the task of virtual device testing more difficult. By default, the BARs for each device contain invalid addresses. Until the BARs are initialized by the BIOS, PCI devices are unusable. The PCI host controller provides two 32-bit registers in the ISA MMIO/PMIO address space for configuring each PCI device BAR[2]. Until the proper read/write sequence is made to these two registers, PCI devices remain unconfigured and inaccessible to the guest environment. Therefore, configuring a virtual PCI-based device involves initializing both the state of the PCI bus and the virtual device.

3 Fuzzing Virtual Devices

3.1 Evolutionary Fuzzing

Fuzzing mutates seed input to generate new test case inputs which execute new paths within a program. Simple fuzzers naively mutate seed inputs without any knowledge of the program under test, treating the program as a "black box". In comparison, evolutionary fuzzing, such as AFL [38] can insert compile-time instrumentation into the program under test. This instrumentation, placed at every branch and label within the instrumented program, tracks which branches have been taken when specific inputs are supplied. Such evolutionary fuzzing is much more effective at exploring new branches.

If AFL generates a test case that covers new branches, that test case becomes a new seed input. As AFL continues to generate new seeds, more and more states of the program are exercised. Unfortunately, all branches are considered to be of

[2] `CONFIG_ADDRESS` at `0xCF8` and `CONFIG_DATA` at `0xCFC` [11].

equal priority during exploration, so uninteresting states are explored as readily as interesting states are. This leads to a large number of wasted testing cycles as uninteresting states are unnecessarily explored. Therefore, VDF modifies AFL to only instrument the portions of the hypervisor source code that belong to the virtual device currently being tested. This effectively makes AFL ignore the remainder of the hypervisor codebase when selectively mutating seed inputs.

AFL maintains a "fuzz bitmap", with each byte within the bitmap representing a count of the number of times a particular branch within the fuzzed program has been taken. AFL does not perform a one-to-one mapping between a particular branch and a byte within the bitmap. Instead, AFL's embedded instrumentation places a random two-byte constant ID into each branch. Whenever execution reaches an instrumented branch, AFL performs an `XOR` of the new branch's ID and the last branch ID seen prior to arriving at the new branch. This captures both the current branch and the unique path taken to reach it (such as when the same function is called from multiple locations in the code). AFL then applies a hashing function to the `XOR`'d value to determine which entry in the bitmap represents that branch combination. Whenever a particular branch combination is exercised, the appropriate byte is incremented within the bitmap.

VDF modifies AFL to use a much simpler block coverage mechanism that provides a one-to-one mapping between a particular instrumented branch and a single entry in the bitmap. Because VDF selectively instruments *only* the branches within a virtual device, the bitmap contains more than enough entries to dedicate an entry to each instrumented branch[3]. VDF's modifications do away with the `XOR`ing of IDs and AFL's hash function. Instead, IDs are assigned linearly, simplifiying the ground truth determination of whether a particular branch has been reached during testing while guaranteeing that no IDs are duplicated.

Thus, AFL takes a general purpose approach towards fuzzing/exploring all branches within a program. VDF's modified AFL takes a more focused approach that constrains fuzzing to only the branches of interest in a program. VDF's approach eliminates the possibility of ambiguous branch coverage, which is still possible to experience with an unmodified AFL.

3.2 VDF Workflow

Figure 3 shows the three-step flow used by VDF when testing a virtual device. In the first step, virtual device activity is recorded while the device is being exercised. This log of activity includes any initialization of PCI BARs for the virtual device via the PCI host controller (if needed), initialization of any internal device registers, and any MMIO activity that exercises the virtual device. This log is saved to disk and becomes the seed input for the fuzzer. This collection of seed input is described further in Sect. 3.3.

In the second step, the collected virtual device read/write activity is then provided as seed data to AFL. Multiple AFL instances can be launched in parallel,

[3] VDF still uses a two-byte branch ID, allowing for 65536 unique branches to be instrumented. In practice, this is more than adequate for virtual device testing.

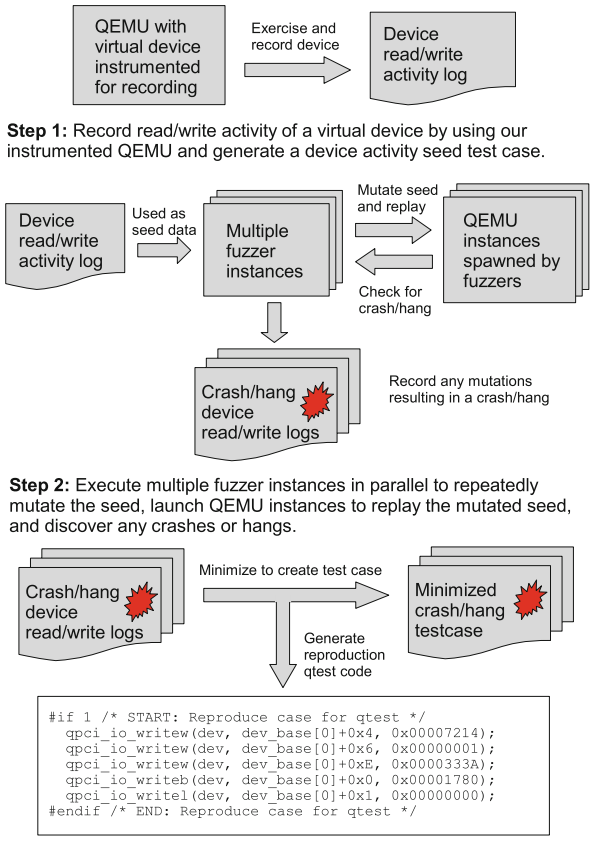


Fig. 3. VDF's process for performing fuzz testing of QEMU virtual devices.

with one required master instance and one or more optional slave instances. The primary difference between master and slave instances is that the master uses a series of sophisticated mutation strategies (bit/byte swapping, setting bytes to specific values like `0x00` and `0xFF`, etc.) to explore the program under test. Slave instances only perform random bit flips throughout the seed data.

Once the seed input has been mutated into a new test case, a new QEMU instance is spawned by AFL. VDF replays the test case in the new QEMU instance and observes whether the mutated data has caused QEMU to crash or hang. VDF does not blindly replay events, but rather performs strict filtering on the mutated seed input during replay. The filter discards malformed events, events describing a read/write outside the range of the current register bank, events referencing an invalid register bank, etc. This prevents mutated data from potentially exercising memory locations unrelated to the virtual device under test. If a test case causes a crash or hang, the test case is logged to disk.

Finally, in the third step, each of the collected crash and hang test cases is reduced to a minimal test case capable of reproducing the bug. Both a minimized

test case and source code to reproduce the bug are generated. The minimization of test cases is described further in Sect. 3.5.

3.3 Virtual Device Record and Replay

Fuzzing virtual devices is difficult because they are *stateful*. It is necessary to traverse an arbitrarily large number of states within both the virtual device and the remainder of the hypervisor prior to reaching a desired state within the virtual device. Because each virtual device must be initialized to a known good start state prior to each test, VDF uses *record and replay* of previous virtual device activity to prepare the device for test and then perform the test itself.

First, VDF records any guest reads or writes made to the virtual device's control registers when the device is initialized during guest OS boot[4]. This captures the setup performed by the BIOS (such as PCI BAR configuration), device driver initialization in the kernel, and any guest userspace process interaction with the device's kernel driver. Table 1 shows the different sources of initialization activity used by VDF when recording device activity during our testing.

Table 1. QEMU virtual devices seed data sources.

Device class	Device	Seed data source
Audio	AC97	Linux guest boot with ALSA [1] `speaker-test`
	CS4231a	
	ES1370	
	Intel-HDA	
	SoundBlaster 16	
Block	Floppy	qtest test case
Char	Parallel	Linux guest boot with directed console output
	Serial	
IDE	IDE Core	qtest test case
Network	EEPro100 (i82550)	Linux guest boot with `ping` of IP address
	E1000 (82544GC)	
	NE2000 (PCI)	
	PCNET (PCI)	
	RTL8139	qtest test case
SD Card	SD HCI	Linux guest boot with mounted SDHCI volume
TPM	TPM	Linux guest boot with TrouSerS test suite [16]
Watchdog	IB700	qtest test case
	16300ESB	Linux guest boot

[4] If only a minimal amount of recorded activity is required, VDF can capture initialization activity via executing a QEMU qtest test case.

Second, the recorded startup activity is partitioned into two sets: an *init* set and a *seed* set. The init set contains any seed input required to initialize the device for testing, such as PCI BAR setup, and the activity in this set will never be mutated by the fuzzer. VDF plays back the init set at the start of each test to return the device to a known, repeatable state. The seed set contains the seed input that will be mutated by the fuzzer. It can be any read/write sequence that exercises the device, and it usually originates from user space activity that exercises the device (playing an audio file, pinging an IP address, etc.).

Even with no guest OS booted or present, a replay of these two sets returns the virtual device to the same state that it was in immediately after the register activity was originally recorded. While the data in the sets could include timestamps to ensure that the replay occurs at the correct time intervals, VDF does not do this. Instead, VDF takes the simpler approach of advancing the virtual clock one microsecond for each read or write performed. The difficulty with including timestamps within the seed input is that the value of the timestamp is too easily mutated into very long virtual delays between events. While it is true that some virtual device branches may only be reachable when a larger virtual time interval has passed (such as interrupts that are raised when a device has completed performing some physical event), our observation is that performing a fixed increment of virtual time on each read or write is a reasonable approach.

Event Record Format. VDF event records contain three fields: a header field, base offset field, and data written field. This format captures all data needed to replay an MMIO event and represents this information in a compact format requiring only 3–8 bytes per event. The compactness of each record is an important factor because using a smaller record size decreases the number of bits that can potentially be mutated.

The header is a single byte that captures whether the event is a read or write event, the size of the event (1, 2, or 4 bytes), and which virtual device register bank the event takes place in. The base offset field is one to three bytes in size and holds the offset from the base address. The size of this field will vary from device to device, as some devices have small register bank ranges (requiring only one byte to represent an offset into the register bank) and other devices map much larger register banks and device RAM address ranges (requiring two or three bytes to specify an offset). The data field is one or four bytes in size and holds the data written to a memory location when the header field specifies a write operation. Some devices, such as the floppy disk controller and the serial port, only accept single byte writes. Most devices accept writes of 1, 2, or 4 bytes, requiring a 4 byte field for those devices to represent the data. For read operations, the data field is ignored.

While VDF's record and replay of MMIO activity captures the interaction of the guest environment with virtual devices, some devices may make use of interrupts and DMA. However, we argue that such hardware events are not necessary to recreate the behavior of most devices for fuzz testing. Interrupts are typically *produced* by a virtual device, rather than *consumed*, to alert the guest environment that some hardware event has completed. Typically, another

read or write event would be initiated by the guest in reaction to an interrupt, but since we record all this read/write activity, the guest's response to the interrupt is captured without explicitly capturing the interrupt.

DMA events copy data between guest and device RAM. DMA copies typically occur when buffers of data must be copied and the CPU isn't needed to copy this data byte-by-byte. Our observation is that if we are only copying data to be processed, it is not actually necessary to place legitimate data at the correct location within guest RAM and then copy it into the virtual device. It is enough to say that the data has been copied and then move onto the next event. While the size of data and alignment of the data may have some impact on the behavior of the virtual device, such details are outside the scope of this paper.

Recording Virtual Device Activity. Almost every interaction between the guest environment and virtual devices occurs via virtual device callback functions. These functions are registered with QEMU's MMU and are triggered by MMIO activity from the guest. Such callback functions are an ideal location to record the virtual device's activity. Rather than attempt to capture the usage of each device by reconstructing the semantics of the guest's kernel and memory space, we capture device activity at the point of the hardware interface that is provided to software. In fact, we have no immediate need to understand the details of the guest environment as the virtual devices execute at a level above that of even the guest's BIOS or kernel. By placing recording logic in these callback functions, VDF is able to instrument each virtual device by manually adding only 3–5 LOC of recording logic to each MMIO callback function.

Playback of Virtual Device Activity. Once VDF has recorded a stream of read/write events for a virtual device, it must replay these events within the context of a running QEMU. Because QEMU traverses a large number of branches before all virtual devices are instantiated and testing can proceed, it isn't possible to provide the event data to QEMU via the command line. The events must originate from within the guest environment in the form of memory read/write activity. Therefore, QEMU must be initialized before performing event replay.

QEMU provides *qtest*, which is a lightweight framework for testing virtual devices. qtest is a QEMU *accelerator*, or type of execution engine. Common accelerators for QEMU are *TCG* (for the usage of QEMU TCG IR) and *KVM* (for using the host kernel's KVM for hardware accelerated execution of guest CPU instructions). The qtest framework works by using a test driver process to spawn a separate QEMU process which uses the qtest accelerator. The qtest accelerator within QEMU communicates with the test driver process via IPC. The test driver remotely controls QEMU's qtest accelerator to perform guest memory read/write instructions to virtual devices exposed via MMIO. Once the test is complete, the test driver terminates the QEMU process.

While the qtest accelerator is convenient, it is inadequate for fuzz testing for two reasons. First, the throughput and timing of the test is slowed because of QEMU start-up and the serialization, deserialization, and transfer time of the

IPC protocol. Commands are sent between the test driver and QEMU as plain-text messages, requiring time to parse each string. While this is not a concern for the virtual clock of QEMU, wall clock-related issues (such as thread race conditions) are less likely to be exposed.

Second, qtest does not provide control over QEMU beyond spawning the new QEMU instance and sending control messages. It is unable to determine exactly where a hung QEMU process has become stuck. A hung QEMU also hangs the qtest test driver process, as the test driver will continue to wait for input from the non-responsive QEMU. If QEMU crashes, qtest will respond with the feedback that the test failed. Reproducing the test which triggers the crash may repeat the crash, but the analyst still has to attach a debugger to the spawned QEMU instance prior to the crash to understand the crash.

VDF seeks to automate the discovery of any combination of virtual device MMIO activity that triggers a hang or crash in either the virtual device or some portion of the hypervisor. qtest excels at running known-good, hard-coded tests on QEMU virtual devices for repeatable regression testing. But, it becomes less useful when searching for unknown vulnerabilities, which requires automatically generating new test cases that cover as many execution paths as possible.

To address these shortcomings, we have developed a new *fuzzer* QEMU accelerator, based upon qtest, for VDF's event playback. This new accelerator adds approximately 850 LOC to the QEMU codebase. It combines the functionality of the qtest test driver process and the qtest accelerator within QEMU, eliminating the need for a separate test driver process and the IPC between QEMU and the test driver. More importantly, it allows VDF to directly replay read/write events as if the event came directly from within a complete guest environment.

3.4 Selective Branch Instrumentation

Fuzz testing must explore as many branches of interest as possible, so determining the *coverage* of those branches during testing is a metric for measuring the thoroughness of each testing session. While the code within any branch may host a particular bug, execution of the branch must be performed to trigger the bug. Thus, reaching more branches of interest increases the chances that a bug will be discovered. However, if the fuzzer attempts to explore *every* branch it discovers, it can potentially waste millions of tests exploring uninteresting branches.

To address this issue, VDF leverages the instrumentation capabilities of AFL to selectively place instrumentation in only the branches of interest (those belonging to a virtual device). By default, the compiler toolchain supplied with AFL instruments programs built using it. VDF modifies AFL to selectively instrument only code of interest within the target program. A special compile-time option has been added to AFL's toolchain, and only branches in source files compiled with this flag are instrumented. Other files will have uninstrumented branches that are ignored by the fuzzer as they are seen as (very long) basic blocks of instructions that occur between instrumented branches.

Prior to the start of each testing session, VDF dumps and examines all function and label symbols found in the instrumented hypervisor. If a symbol is

found that maps to an instrumented branch belonging to the current virtual device under test, the name, address, and AFL branch ID (embedded in the symbol name) of the symbol are stored and mapped to the symbol's location in the fuzz bitmap. At any point during testing, the AFL fuzz bitmap can be dumped using VDF to provide ground truth of exactly which branches have been covered.

```
static void voice_set_active (AC97LinkState *s, int bm_index, int on) {
    switch (bm_index) {
    case PI_INDEX:
        AUD_set_active_in (s->voice_pi, on);
        break;
    case PO_INDEX:
        AUD_set_active_out (s->voice_po, on);
        break;
    case MC_INDEX:
        AUD_set_active_in (s->voice_mc, on);
        break;
    default:
        AUD_log ("ac97",
            "invalid bm_index(%d) in voice_set_active",
            bm_index);
        break;
    }
}
ID: COVERED: ADDRESS: SYMBOL:                          LINE:
--- -------- -------- -------                          -----
00c COVER    002e92e0 voice_set_active                 296
00d COVER    002e9324 REF_LABEL__tmp_ccBGk9PX_s__27_39 296
00e COVER    002e9368 REF_LABEL__tmp_ccBGk9PX_s__28_40 296
00f UNCOVER  002e93a4 REF_LABEL__tmp_ccBGk9PX_s__29_41 296
```

Fig. 4. A sample of the branch coverage data for the AC97 virtual device.

Figure 4 shows an example of the coverage information report that VDF provides. This example shows both the original source code for a function in the AC97 audio virtual device (top) and the generated branch coverage report for that function (bottom). The report provides two pieces of important information. The first is the ground truth of which branches are instrumented, including their address within the binary, the symbol associated with the branch (inserted by the modified AFL), and the original source file line number where the branch's code is located. The second is whether a particular branch has been visited yet.

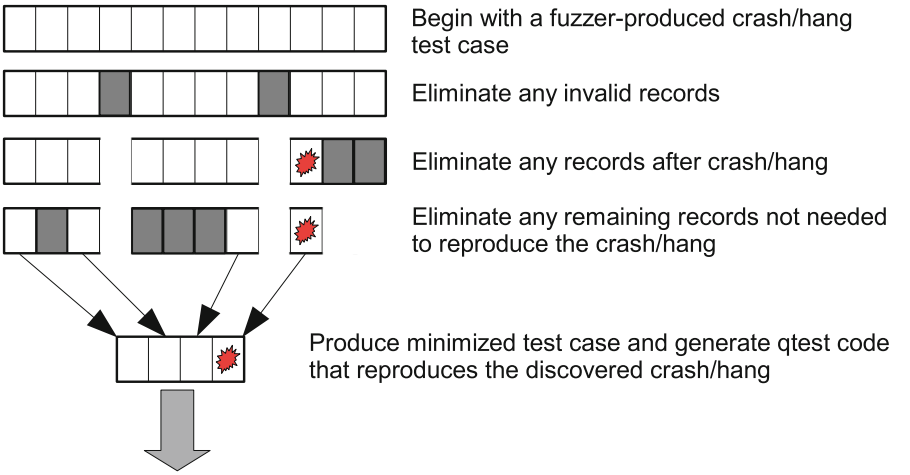
The four branches listed in the report are associated with the four cases in the switch statement of the `voice_set_active()` function, which is located on line 296 in the source file. An analyst familiar with the internals of the AC97 virtual device could review this report and then devise new seed inputs to trigger any unexplored branches. Thus, such reports are useful for not only an understanding of *which* branches have been reached, but they also providing insight into *how* unexplored virtual device branches might be reached.

3.5 Creation of Minimal Test Cases

Once VDF detects either a crash or a hang in a virtual device, the test case that produced the issue is saved for later examination. This test case may contain a

large amount of test data that is not needed to reproduce the discovered issue, so it is desirable to reduce this test case to the absolute minimum number of records needed to still trigger the bug. Such a minimal test case simplifies the job of the analyst when using the test case to debug the underlying cause.

AFL provides a test case minimization utility called `afl-tmin`. `afl-tmin` seeks to make the test case input smaller while still following the same path of execution through the binary. Unfortunately, this will not be useful for reducing the test cases recorded by VDF, which is only interested in reaching the state in which a crash/hang occurs. It has no interest in reaching every state in the test case, but only the states necessary to reach the crash/hang state. Therefore, VDF performs a three-step test case post-processing, seen in Fig. 5, to produce a minimal test case which passes through a minimimal number of states from any test case shown to reproduce an issue.




```
qpci_io_writew(dev, dev_base[0]+0x4, 0x00007214);
qpci_io_writew(dev, dev_base[0]+0x6, 0x00000001);
qpci_io_writew(dev, dev_base[0]+0xE, 0x0000333A);
qpci_io_writel(dev, dev_base[0]+0x1, 0x00DFFF00);
```

Fig. 5. The test case minimization process.

First, the test case file is read into memory and any valid test records in the test case are placed into an ordered dataset in the order in which they appear within the test case. Because the fuzzer lacks semantic understanding of the fields within these records, it produces many records via mutation that contain invalid garbage data. Such invalid records may contain an invalid header field, describe a base offset to a register outside of the register bank for the device, or simply be a truncated record at the end of the test case. After this filtering step, only valid test records remain.

Second, VDF eliminates all records in the dataset that are located after the point in the test case where the issue is triggered. To do this, it generates a new test case using all but the last record of the dataset and then attempts to trigger the issue using this truncated test case. If the issue is still triggered, the last record is then removed from the dataset and another new truncated test case is generated in the same fashion. This process is repeated until a truncated test case is created that no longer triggers the issue, indicating that all dataset records located after the issue being triggered are now removed.

Third, VDF eliminates any remaining records in the dataset that are not necessary to trigger the issue. Beginning with the first record in the dataset, VDF iterates through each dataset record, generating a new test case using all but the current record. It then attempts to trigger the issue using this generated test case. If the issue is still triggered, the current record is not needed to trigger the issue and is removed from the dataset. Once each dataset record has been visited and the unnecessary records removed, the dataset is written out to disk as the final, minimized test case. In addition, source code is generated that is suitable for reproducing the minimized dataset as a qtest testcase.

While simple, VDF's test case minimization is very effective. The 1014 crash and hang test cases produced by the fuzzer during our testing have an average size of 2563.5 bytes each. After reducing these test cases to a minimal state, the average test case size becomes only 476 bytes, a mere 18.57% of the original test case size. On average, each minimal test case is able to trigger an issue by performing approximately 13 read/write operations. This average is misleadingly high due to some outliers, however, as over 92.3% of the minimized test cases perform fewer than six MMIO read/write operations.

4 Evaluation

The configuration used for all evaluations is a cloud-based 8-core 2.0 GHz Intel Xeon E5-2650 CPU instance with 8 GB of RAM. Each instance uses a minimal server installation of Ubuntu 14.04 Linux as its OS. Eight cloud instances were utilized in parallel. Each device was fuzzed within a single cloud instance, with one master fuzzer process and five slave fuzzer processes performing the testing. A similar configuration was used for test case minimization: each cloud instance ran six minimizer processes in parallel to reduce each crash/hang test case.

We selected a set of eighteen virtual devices, shown in Table 2, for our evaluation of VDF. These virtual devices utilize a wide variety of hardware features, such as timers, interrupts, and DMA. Each of these devices provides one or more MMIO interfaces to their control registers, which VDF's fuzzing accelerator interacts with. All devices were evaluated using QEMU v2.5.0[5], with the exception of the TPM device. The TPM was evaluated using QEMU v2.2.50 with an applied patchset that provides a libtpms emulation [20] of the TPM

[5] US government approval for the engineering and public release of the research shown in this paper required a time frame of approximately one year. The versions of QEMU identified for this study were originally selected at the start of that process.

Table 2. QEMU virtual devices tested with VDF.

Device class	Device	Branches of interest	Initial coverage	Final coverage	Crashes found	Hangs found	Tests per instance	Test duration
Audio	AC97	164	43.9%	53.0%	87	0	24.0 M	59d 18 h
	CS4231a	109	5.5%	56.0%	0	0	29.3 M	65d 12 h
	ES1370	165	50.9%	72.7%	0	0	30.8 M	69d 18 h
	Intel-HDA	273	43.6%	58.6%	238	0	23.1 M	59d 12 h
	SoundBlaster 16	311	26.7%	81.0%	0	0	26.7 M	58d 13 h
Block	Floppy	370	44.9%	70.5%	0	0	21.0 M	57d 15 h
Char	Parallel	91	30.8%	42.9%	0	0	14.6 M	25d 12 h
	Serial	213	2.3%	44.6%	0	0	33.0 M	62d 12 h
IDE	IDE Core	524	13.9%	27.5%	0	0	24.9 M	65d 6 h
Network	EEPro100 (i82550)	240	15.8%	75.4%	0	0	25.7 M	62d 12 h
	E1000 (82544GC)	332	13.9%	81.6%	0	384	23.9 M	61d
	NE2000 (PCI)	145	39.3%	71.7%	0	0	25.2 M	58d 13 h
	PCNET (PCI)	487	11.5%	36.1%	0	0	25.0 M	58d 13h
	RTL8139	349	12.9%	63.0%	0	6	24.2 M	58d 12 h
SD Card	SD HCI	486	18.3%	90.5%	14	265	24.0 M	62d
TPM	TPM	238	26.1%	67.3%	9	11	2.1M	36d 12 h
Watchdog	IB700	16	87.5%	100.0%	0	0	0.3 M	8 h
	I6300ESB	76	43.4%	68.4%	0	0	2.1 M	26 h

hardware device [23]. Fewer than 1000 LOC were added to each of these two QEMU codebases to implement both the fuzzer accelerator and any recording instrumentation necessary within each tested virtual device.

VDF discovered *noteworthy bugs in six virtual devices* within the evaluation set, including a known denial-of-service CVE [7] and a new, previously undiscovered denial-of-service CVE [8]. Additional bugs were discovered relating to memory management and thread-race conditions, underscoring VDF's ability to discover bugs of a variety of natures utilizing the same techniques and principles.

4.1 Virtual Device Coverage and Bug Discovery

During our testing with VDF, we collected four metrics to aid in our understanding of both the speed and magnitude of VDF's coverage. These metrics are (1) the number of branches covered by the initial seed test case; (2) the total number of branches in the virtual device; (3) the current total number of branches covered (updated at one minute intervals); and (4) the percentage of total bugs discovered during each cumulative day of testing. Taken together, these metrics describe not only the total amount of coverage provided by VDF, but also the speed at which coverage improves via fuzzing and how quickly it discovers crash/hangs.

Figure 6 shows the average percentage of covered branches over cumulative testing time. Of the eighteen tested virtual devices, 30.15% of the total branches were covered by the initial seed test cases. After nine cumulative days of testing (36 h of parallel testing with one master and five slave fuzzing instances),

62.32% of the total branches were covered. The largest increase in average coverage was seen during the first six cumulative hours of testing, where coverage increased from the initial 30.15% to 52.84%. After 2.25 days of cumulative testing, average coverage slows considerably and only 0.43% more of the total branches are discovered during the next 6.75 cumulative days of testing. While eleven of the eighteen tested devices stopped discovering new branches after only one day of cumulative testing, six of the seven remaining devices continued to discover additional branches until 6.5 cumulative days had elapsed. Only in the serial device were additional branches discovered after nine cumulative days.

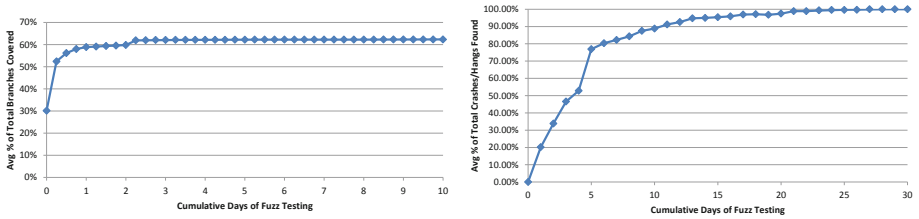


Fig. 6. Average percentage of branches covered (left) and average percentage of total bugs discovered (right) over time during fuzz testing.

Table 2 presents some insightful statistics about coverage. The smallest improvement in the percentage of coverage was seen in the AC97 virtual device (9.1% increase), and the largest improvement in coverage was seen in the SDHCI virtual device (72.2% increase). The smallest percentage of coverage for any virtual device with discovered crashes/hangs was 53.0% (AC97), but eight others had a greater level of coverage than 53.0% with no discovered crashes/hangs.

Figure 6 also shows the average percentage of discovered hangs/crashes over cumulative testing time. As shown in Table 2, a total of 1014 crashes and hangs were discovered in six virtual devices. These 1014 test cases were all discovered within 27 days of cumulative testing for each device, with no additional test cases being discovered after that point. Approximately 50% of all test cases were discovered after four days of cumulative testing, with approximately 80% of all test cases discovered after five days of cumulative testing.

One interesting insight is that even though the number of branches covered is very close to its maximum after approximately 2.5 cumulative days of testing, only approximately 25% of all crash/hang test cases were discovered at that point in time. This shows that it is not necessarily an *increase in branch coverage* that leads to the discovery of bugs, but rather the *repeated fuzz testing of those discovered branches*.

4.2 Classification of All Discovered Virtual Device Bugs

While it is straightforward to count the number of discovered crash/hang test cases generated by VDF, it is non-trivial to map these test cases to their underlying cause without a full understanding of the virtual device under test.

Our proposed test case minimization greatly simplifies this process, as many unique bugs identified by VDF minimize to the same set of read/write operations. The ordering of these operations may differ, but the final read/write that triggers the bug remains the same. Each discovered virtual device bug falls into one of four categories: Excess resource usage (AC97), invalid data transfers (E1000, RTL8139, SDHCI), debugging asserts (Intel-HDA), and thread race conditions (TPM).

Excess Host Resource Usage. Host system resources must be allocated to QEMU to represent the resources belonging to the guest environment. Such resources include RAM to represent the physical RAM present on the guest, CPU cores and cycles to perform CPU and virtual device emulation, and disk space to hold the guest's secondary storage. Additional resources may be allocated by QEMU at runtime to meet the data needs of virtual devices, which presents a potential opportunity for a malicious guest to trick QEMU into allocating large amounts of unnecessary resources.

VDF discovered a crash while testing the AC97 audio virtual device, caused by QEMU allocating approximately 500 MB of additional host memory when the control register for `AC97_MIC_ADC_Rate` is set to an invalid, non-zero value. An important observation on this type of resource bug is that it will easily remain hidden unless the resource usage of the QEMU process is strictly monitored and enforced. For example, using the Linux `ulimit` command to place a limit on the virtual memory allocated to QEMU will discover this bug when the specified memory limit is exceeded. VDF enforces such a limitation during its testing, restricting the amount of virtual memory allocated to each QEMU instance. Once this limit is exceeded, a `SIGTRAP` signal is raised and a crash occurs.

Allocating excessive resources for a single guest instance is typically not a concern, but the potential impact increases greatly when considering a scenario with large numbers of instances deployed within a cloud environment. Discovering and correcting such bugs can have a measurable impact on the resource usage of hosts implementing cloud environments. Cloud service providers must allocate some amount of host hardware RAM and secondary storage to each VM hosted on that hardware. Thus, each VM must have a resource quota that is determined by the service provider and enforced by the host and hypervisor. However, if this quota does not take into account the resources used by the hypervisor itself, an excess host resource usage bug can potentially consume considerable host resources. Therefore, we reported this as a bug to the QEMU maintainers.

Invalid Data Transfers. Many virtual devices transfer blocks of data. Such transfers are used to move data to and from secondary storage and guest physical memory via DMA. However, invalid data transfers can cause virtual devices to hang in an infinite loop. This type of bug can be difficult to deal with in production systems as the QEMU process is still running while the guest's virtual clock is in a "paused" state. If queried, the QEMU process appears to be running

and responsive. The guest remains frozen, causing a denial of service of any processes running inside of the guest.

VDF discovered test cases that trigger invalid data transfer bugs in the E1000 and RTL8139 virtual network devices and the SDHCI virtual block device. In each case, a transfer was initiated with either a block size of zero or an invalid transfer size, leaving each device in a loop that either never terminates or executes for an arbitrarily long period of time.

For the E1000 virtual device, the guest sets the device's `E1000_TDH` and `E1000_TDT` registers (TX descriptor head and tail, respectively) with offsets into guest memory that designate the current position into a buffer containing transfer operation descriptors. The guest then initiates a transfer using the `E1000_TCTL` register (TX control). However, if the values placed into the `E1000_TDH/TDL` registers are too large, then the transfer logic enters an infinite loop. A review of reported CVEs has shown that this issue was already discovered in January 2016 [7] and patched [14].

For the RTL8139 virtual device, the guest resets the device via the `ChipCmd` (chip control) register. Then, the `TxAddr0` (transfer address), `CpCmd` ("C+" mode command), and `TxPoll` (check transfer descriptors) registers are set to initiate a DMA transfer in the RTL8139's "C+" mode. However, if an invalid address is supplied to the `TxAddr0` register, QEMU becomes trapped in an endless loop of DMA lookups. This was an undiscovered bug, which has been patched and assigned CVE-2016-8910 [8] as a denial of service exploit.

For the SDHCI virtual device, the guest sets the device's `SDHC_CMDREG` register bit for "data is present" and sets the block size to transfer to zero in the `SDHC_BLKSIZE` register. The `switch` case for `SDHC_BLKSIZE` in the `sdhci_write()` MMIO callback function in `hw/sd/sdhci.c` performs a check to determine whether the block size exceeds the maximum allowable block size, but it does not perform a check for a block size of zero. Once the transfer begins, the device becomes stuck in a loop, and the guest environment becomes unresponsive. Luckily, fixes for this issue were integrated into mainline QEMU [12] in December 2015.

Debugging Asserts. While using an `assert` is a commonly-used debugging technique in mature software codebases, `assert`s are used to catch a particular case that should "never happen". If that impossible case actually *can* happen as a result of untrusted input, proper error-handling logic should be added to the code to address it. Within the Intel-HDA audio device, the `intel_hda_reg_write()` function in `hw/audio/intel-hda.c` uses an `assert` call to trigger a `SIGABRT` when a write is made to an address offset of 0 from the MMIO register base address. VDF was able to trigger this assert, which we have reported as a bug to the QEMU maintainers.

Thread Race Conditions. The virtual TPM in mainline QEMU is a pass-through device to the host's hardware TPM device. It is possible to implement a TPM emulated in software using libtpms [20] and then have QEMU pass TPM activity through to the emulated hardware. QEMU interacts with the separate

process implementing the TPM via RPC. However, it is also possible to integrate libtpms directly into QEMU by applying a patchset provided by IBM [23]. This allows each QEMU instance to "own" its own TPM instance and directly control the start-up and shutdown of the TPM via a TPM backend in QEMU.

VDF discovered a hang that is the result of the TPM backend thread pool shutdown occurring before the tasks allocated to the thread pool have all been completed. Without an adequately long call to `sleep()` or `usleep()` prior to the thread pool shutdown to force a context switch and allow the thread pool worker threads to complete, the thread pool will hang on shutdown. Because the shutdown of the TPM backend is registered to be called at `exit()` via an `atexit()` call, any premature `exit()` prior to the necessary `sleep()` or `usleep()` call will trigger this issue. QEMU's signal handlers are never unregistered, so using a `SIGTERM` signal to kill QEMU is unsuccessful.

Note that this thread pool is part of the TPM backend design in QEMU, and is not part of the libtpms library that implements the actual TPM emulator. Most likely this design decision was made to avoid any noticeable slowdown in QEMU's execution by making the TPM virtual device run in an asynchronous manner to avoid any performance impact caused by performing expensive operations in the software TPM. Other newer TPM pass-through options, such as the Character in User Space (CUSE) device interface to a stand-alone TPM emulator using libtpms [13], should not experience this particular issue.

5 Related Work

Fuzzing has been a well-explored research topic for a number of years. The original fuzzing paper [32] used random program inputs as seed data for testing Unix utilities. Later studies on the selection of proper fuzzing seeds [25,34] and the use of concolic fuzzing to discover software vulnerabilities [17] have both been used to improve the coverage and discovery of bugs in programs undergoing fuzz testing. By relying on the record and replay of virtual device activity, VDF provides proper seed input that is known to execute branches of interest.

Frameworks for testing virtual devices are a fairly recent development. qtest [9] was the first framework to approach the idea of flexible low-level testing of virtual devices. VDF leverages qtest, but has improved on the approach to better improve test case throughput and test automation. Tang and Li proposed an approach [36] using a custom BIOS within the guest environment that listened on a virtual serial port to drive testing. VDF's approach relies upon no software executing within the guest environment (BIOS, kernel, etc.), and performs device-specific BIOS-level initialization as part of its init set.

A number of tools utilize record and replay. ReVirt [31] records system events to replay the activity of compromised guest systems to better analyze the nature of the attack. Aftersight [27] records selected system events and then offloads those events to another system for replay and analysis. Its primary contribution of decoupled analysis demonstrates that record and replay facilitates repeated heavyweight analysis after the moment that the event of interest originally occurred. PANDA [30], a much more recent work in this area, uses a modified

QEMU to record non-deterministic guest events that occur system-wide. These events are then replayed through increasingly heavier-weight analysis plugins to reverse engineer the purpose and behavior of arbitrary portions of the guest.

Symbolic execution of complex programs is also a common technique to calculate the path predicates and conditionals needed to exercise branches of interest. KLEE [24] performs symbolic execution at the process level. Selective Symbolic Execution (S2E) [26] executes a complete guest environment under QEMU and performs symbolic execution at the whole-system level. The approach proposed by Cong et al. [28] attempts to extract the code for five network virtual devices from QEMU, stub out key QEMU datatypes, and then perform symbolic execution on the resulting code. VDF is capable of performing its testing and analysis of a much larger set of virtual devices within the context of QEMU. However, the techniques laid out in [28] can complement VDF by generating new seed test cases designed to augment VDF's ability to reach new branches of interest.

Driller [35] uses both white box fuzzing and symbolic execution to discover vulnerabilities within programs. Unlike VDF, which is interested in exploring only branches of interest, Driller seeks to explore all branches within a program. It switches between symbolic execution and fuzzing when fuzzing gets "stuck" and can no longer discover data values that explore new branches. VDF focuses on executing large numbers of fuzzing test cases without using expensive symbolic execution to create new seeds.

The discovery of vulnerable code is a difficult and ongoing process, and there is interest in research work orthogonal to our effort that seeks to protect the host system and harden hypervisors. DeHype [37] reduces the privileged attack surface of KVM by deprivileging 93.2% of the KVM hypervisor code from kernel space to user space on the host. The Qubes OS project [15] compartmentalizes software into a variety of VMs, allowing the isolation of trusted activities from trusted ones within the OS. Qubes relies upon the bare-metal Xen hypervisor, which is much harder to exploit than a hypervisor executing under the host OS.

6 Conclusion

In this paper, we presented VDF, a system for performing fuzz testing on virtual devices, within the context of a running hypervisor, using record/replay of memory-mapped I/O events. We used VDF to fuzz test eighteen virtual devices, generating 1014 crash or hang test cases that reveal bugs in six of the tested devices. Over 80% of the crashes and hangs were discovered within the first day of testing. VDF covered an average of 62.32% of virtual device branches during testing, and the average test case was minimized to 18.57% of its original size.

Acknowledgment. The authors would like to thank the staff of the Griffiss Institute in Rome, New York for generously allowing the use of their cloud computing resources. This material is based upon research sponsored by the Air Force Research Lab, Rome Research Site under agreement number FA8750-15-C-0190.

References

1. Advanced Linux Sound Architecture (ALSA). http://www.alsa-project.org
2. Amazon.com, Inc., Form 10-K 2015. http://www.sec.gov/edgar.shtml
3. CVE-2014-2894: Off-by-one error in the cmd start function in smart self test in IDE core. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2894
4. CVE-2015-3456: Floppy disk controller (FDC) allows guest users to cause denial of service. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456
5. CVE-2015-5279: Heap-based buffer overflow in NE2000 virtual device. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5279
6. CVE-2015-6855: IDE core does not properly restrict commands. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6855
7. CVE-2016-1981: Reserved. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1981
8. CVE-2016-8910: Qemu: net: rtl8139: infinite loop while transmit in C+ mode. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8910
9. Features/QTest. http://wiki.qemu.org/Features/QTest
10. Kernel-Based Virtual Machine. http://www.linux-kvm.org/
11. PCI - OSDev Wiki. http://wiki.osdev.org/PCI
12. [Qemu-devel] [PATCH 1/2] hw/sd: implement CMD23 (SET_BLOCK_COUNT) for MMC compatibility. https://lists.gnu.org/archive/html/qemu-devel/2015-12/msg00948.html
13. [Qemu-devel] [PATCH 1/5] Provide support for the CUSE TPM. https://lists.nongnu.org/archive/html/qemu-devel/2015-04/msg01792.html
14. [Qemu-devel] [PATCH] e1000: eliminate infinite loops on out-of-bounds transfer start. https://lists.gnu.org/archive/html/qemu-devel/2016-01/msg03454.html
15. Qubes OS Project. https://www.qubes-os.org/
16. TrouSerS - The open-source TCG software stack. http://trousers.sourceforge.net
17. Avgerinos, T., Cha, S.K., Lim, B., Hao, T., Brumley, D.: AEG: automatic exploit generation. In: Proceedings of Network and Distributed System Security Symposium (NDSS) (2011)
18. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. ACM SIGOPS Operating Syst. Rev. **37**(5), 164 (2003)
19. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, Freenix Track, pp. 41–46 (2005)
20. Berger, S.: libtpms library. https://github.com/stefanberger/libtpms
21. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016 (2016)
22. Böttinger, K., Eckert, C.: Deepfuzz: triggering vulnerabilities deeply hidden in binaries. In: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA 2016 (2016)
23. Bryant, C.: [1/4] tpm: Add TPM NVRAM Implementation (2013). https://patchwork.ozlabs.org/patch/288936/
24. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th Symposium on Operating Systems Design and Implementation, pp. 209–224. USENIX Association (2008)

25. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy, pp. 380–394. IEEE, May 2012
26. Chipounov, V., Georgescu, V., Zamfir, C., Candea, G.: Selective symbolic execution. In: Proceedings of Fifth Workshop on Hot Topics in System Dependability, June, Lisbon, Portugal (2009)
27. Chow, J., Garfinkel, T., Chen, P.M.: Decoupling dynamic program analysis from execution in virtual environments. In: USENIX Annual Technical Conference, pp. 1–14 (2008)
28. Cong, K., Xie, F., Lei, L.: Symbolic execution of virtual devices. In: 2013 13th International Conference on Quality Software, pp. 1–10. IEEE, July 2013
29. Corbet, J., Rubini, A., Kroah-Hartman, G.: Linux Device Drivers, 3rd edn. O' Reilly Media Inc., Sebastopol (2005)
30. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable Reverse Engineering for the Greater Good with PANDA. Technical report, Columbia University, MIT Lincoln Laboratory, TR CUCS-023-14 (2014)
31. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: ReVirt: enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Operating Syst. Rev. **36**(SI), 211–224 (2002)
32. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990)
33. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer: application-aware evolutionary fuzzing. In: NDSS, February 2017
34. Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D.: Optimizing seed selection for fuzzing. In: 23rd USENIX Security Symposium (2014)
35. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of NDSS 2016, February 2016
36. Tang, J., Li, M.: When virtualization encounter AFL. In: Black Hat Europe (2016)
37. Wu, C., Wang, Z., Jiang, X.: Taming hosted hypervisors with (mostly) deprivileged execution. In: Network and Distributed System Security Symposium (2013)
38. Zalewski, M.: American Fuzzy Lop Fuzzer. http://lcamtuf.coredump.cx/afl/