V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing

Gaoning Pan
pgn@zju.edu.cn
Zhejiang University & Ant Group

Xingwei Lin
xwlin.roy@gmail.com
Ant Group

Xuhong Zhang
zhangxuhong@zju.edu.cn
Zhejiang University & Binjiang Institute of Zhejiang University

Yongkang Jia
kangel@zju.edu.cn
Zhejiang University

Shouling Ji*
sji@zju.edu.cn
Zhejiang University & Binjiang Institute of Zhejiang University

Chunming Wu*
wuchunming@zju.edu.cn
Zhejiang University

Xinlei Ying
xinlei.yxl@antgroup.com
Ant Group

Jiashui Wang
jiashui.wjs@antgroup.com
Ant Group

Yanjun Wu
yanjun@iscas.ac.cn
Institute of Software, The Chinese Academy of Sciences

ABSTRACT

With the wide application and deployment of cloud computing in enterprises, virtualization developers and security researchers are paying more attention to cloud computing security. The core component of cloud computing products is the hypervisor, which is also known as the virtual machine monitor (VMM) that can isolate multiple virtual machines in one host machine. However, compromising the hypervisor can lead to virtual machine escape and the elevation of privilege, allowing attackers to gain the permission of code execution in the host. Therefore, the security analysis and vulnerability detection of the hypervisor are critical for cloud computing enterprises. Importantly, virtual devices expose many interfaces to a guest user for communication, making virtual devices the most vulnerable part of a hypervisor. However, applying fuzzing to the virtual devices of a hypervisor is challenging because the data structures transferred by DMA are constructed in a nested form according to protocol specifications. Failure to understand the protocol of the virtual devices will make the fuzzing process stuck in the initial fuzzing stage, resulting in inefficient fuzzing.

In this paper, we propose a new framework called V-Shuttle to conduct hypervisor fuzzing, which performs scalable and semantics-aware hypervisor fuzzing. To address the above challenges, we first design a DMA redirection mechanism to significantly reduce the manual efforts to reconstruct virtual devices' protocol structures and make the fuzzing environment setup automated and scalable. Furthermore, we put forward a new fuzzing mutation scheduling mechanism called seedpool to make the virtual device fuzzing process semantics-aware and speed up the fuzzing process to achieve high coverage. Extensive evaluation on QEMU and VirtualBox, two of the most popular hypervisor platforms among the world, shows that V-Shuttle can efficiently reproduce existing vulnerabilities and find new vulnerabilities. We further carried out a long-term fuzzing campaign in QEMU/KVM and VirtualBox with V-Shuttle. In total, we discovered 35 new bugs with 17 CVEs assigned.

CCS CONCEPTS

• **Security and privacy** → **Virtualization and security**; *Software security engineering*;

KEYWORDS

Hypervisor; Virtual Device; Fuzzing; Vulnerability

ACM Reference Format:
Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. 2021. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *Proceedings of Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15–19, 2021 (CCS '21),* 17 pages.
https://doi.org/10.1145/3460120.3484811

1 INTRODUCTION

Cloud computing becomes quite prevalent nowadays, as organizations and individual users prefer to deploy their applications on top of the cloud computing infrastructure for its rapid and scalable deployment ability. Major cloud service providers, such as Amazon Web Services (AWS), Microsoft Azure, and Alibaba Cloud, continue to grow with the increasing demand for cloud computing resources. However, the popularity of cloud computing also leads to the security concerns of the cloud computing software and hardware stack. Famous PWN contests, such as Pwn2Own and Tianfu Cup [10, 11], have the virtualization category that targets hypervisors, including VMWare WorkStation/Esxi, QEMU/KVM, and VirtualBox. Virtual

[*] Chunming Wu and Shouling Ji are the co-corresponding authors.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484811

hardware devices used by guests are hardware peripherals emulated in modern hypervisors that provide a virtual machine with additional functionality. From the attacker's perspective, the virtual devices allow the attackers to write data to the host machine from the guest system. This feature makes the virtual devices become the most vulnerable attack surface of the hypervisor system architecture. In the past few years, almost all attacks on the hypervisor were launched from virtual devices [28, 46, 54]. Hence, it is critical to conduct security vetting of the virtual devices' code to avoid exposing vulnerabilities to attackers in advance. Toward this, we need efficient and scalable techniques to identify the potential vulnerabilities.

In practice, fuzzing has been proved an effective way to discover bugs and vulnerabilities in modern software [5, 14, 16, 19, 25, 32, 49]. However, our observation is that applying fuzzing to hypervisor is challenging, as the inherent hypervisor-specific challenges make fuzzing ineffective. Typically, a hypervisor is designed to expose interfaces to a guest system, such that guest users can drive the virtual device to emulate its behavior. Most devices follow the common operation model, where the device states are first initialized through MMIO, and then the complex data transfer process is completed through DMA, as shown in Figure 1. It is natural to write random data into these interfaces by using fuzzing techniques, but the data transferred by DMA is highly nested, which severely hinders traditional fuzzing from expanding code coverage. Specifically, the data structures defined in a device's specification are often constructed as a tree, where each node contains a link pointer to the next node. Certain DMA operations take a large size of the input from the guest space, and they use a nested structure in the input structures — i.e., a field member in one structure points to another structure. From the perspective of random fuzzing, such a nested structure is difficult to construct as it has to correctly guess the semantics of the overall organization (hierarchically nested pattern) and the internal semantics imposed in each node (i.e., a pointer field pointing to another).

Considering that the security of hypervisor is critical, various fuzzing tools have been proposed to detect bugs in a hypervisor [9, 13, 20, 22, 31, 34, 43, 50, 51]. The state-of-the-art methods include VDF [34], Hyper-Cube [50] and Nyx [51]. VDF is the first hypervisor fuzzing framwork, which utilizes AFL to implement a coverage-guided hypervisor fuzzing approach. Hyper-Cube designs a multi-dimensional, platform-independent fuzzing method based on a custom OS. Although Hyper-Cube does not apply the coverage-guided fuzzing technology into its fuzzing process, it still outperforms VDF due to its high-throughput design. However, both of them share the same idea: they write a bunch of random values to the basic interface (MMIO, DMA, etc.). Further, they have no knowledge of the protocol implementation of a virtual device - how the data structures transferred via DMA are organized. Nyx understands the protocol of the target device and builds structured fuzzing based on user-supplied specifications. However, it requires significant manual effort to create the template for a specification. For example, the authors of Nyx spent about two days on the most complex specification in their evaluation. Hence, Nyx does not scale across different device implementations, as it requires manual adaptation when customized for each new protocol. This is the common disadvantage of structured fuzzing [14, 48, 59], as its effectiveness heavily depends on the completeness of the nested form of structures, which is normally written manually based on the developers' understanding of the protocol specification. Typically, developers need to extract all types of basic data structures from the device protocol, including the connection relationship between basic structures, and the pointer offset in each data structure. Such a labor-intensive process to apply structured fuzzing to hypervisor is *time-consuming* and *error-prone*. As a result, existing fuzzing approaches cannot effectively test virtual devices.

In order to tackle this challenge, we propose V-Shuttle,[1] - a scalable and semantics-aware hypervisor fuzzing framework. Overall, we achieve a fully automatic fuzzing approach by decoupling the nested structures and enabling type awareness. In particular, we first intercept each access to a DMA object and redirect the access from a pointer to our controlled fuzzed input, eliminating the addressing of data structures by the hypervisor to make sure each DMA request will be supplied with the fuzzed data. Then, we perform fine-grained semantics-aware fuzzing by organizing the structures of different DMA object types into different categories and using *seedpool* to maintain the seed queues of these different categories. This method allows each DMA request to be supplied with semantics-valid fuzzed data, which further improves the efficiency of fuzzing.

We implemented V-Shuttle based on the well-known fuzzer AFL. We first evaluate our system by running experiments on 16 QEMU devices and obtain the code coverage. As the evaluation results show, V-Shuttle is truly scalable and automatic to explore deep code in a hypervisor, which eliminates the manual efforts to construct valid test cases according to specifications. Moreover, V-Shuttle even outperforms traditional structure-aware fuzzing, mainly because the process of manually understanding a specification is error-prone. Meanwhile, the semantics-aware fuzzing mode of V-Shuttle, also brings substantial improvements. Compared to state-of-the-art hypervisor fuzzers, V-Shuttle produces higher code coverage in most cases than VDF, Hyper-Cube, and Nyx. Regarding the capability of finding vulnerability, V-Shuttle identifies 35 previously unknown vulnerabilities in two popular hypervisors, out of which 17 new CVEs were assigned. We have reported the discovered vulnerabilities to the respective vendors and are working with them on fixing these vulnerabilities. Additionally, we have also successfully implemented V-Shuttle to Ant Group, a worldwide leading Internet company, which further demonstrates the scalability of our framework. We hope that our tool will aid developers in hardening the hypervisor, leading to better software security.

The main contributions of this work are as follows.

- **The Study on DMA**: We systematically analyze the driver-device interaction in virtual machine transaction, and study why we should focus on the DMA-related part of code. Additionally, we reveal that the data structures transferred via DMA have nested features which will reduce the efficiency of fuzzing.

[1] V-SHUTTLE stands for V model shuttlecraft that we aim to shuttle/escape from the guest virtual machine to host machine by fuzzing hypervisors.

- **A Fuzzing Framework**: We present the design and implementation of V-SHUTTLE, a scalable and semantics-aware hypervisor fuzzing framework, which can automatically decouple nested structures and guide fuzzing to explore hard-to-trigger code. To our knowledge, V-SHUTTLE is the first hypervisor fuzzing framework that has an automatic and deep understanding of the protocol implementations in devices.
- **Discovered Vulnerabilities**: As part of our evaluation, we discovered 35 previously unknown vulnerabilities with 17 CVEs assigned in QEMU and VirtualBox, two of the most widely used hypervisors. We responsibly disclosed the relevant details to the corresponding vendors.
- **An Open-source Tool**: We will open-source V-Shuttle[2], in order to facilitate further research on virtualization security.

The rest of the paper is organized as follows. Section 2 presents the background information with a motivating example. Section 3 describes V-Shuttle's design, and Section 4 describes the implementation details. We show the evaluation results of our approach in Section 5 and the deployment of V-Shuttle in Section 6. The related research and limitation of V-Shuttle are discussed in Section 7 and Section 8, respectively. Finally, we conclude in Section 9.

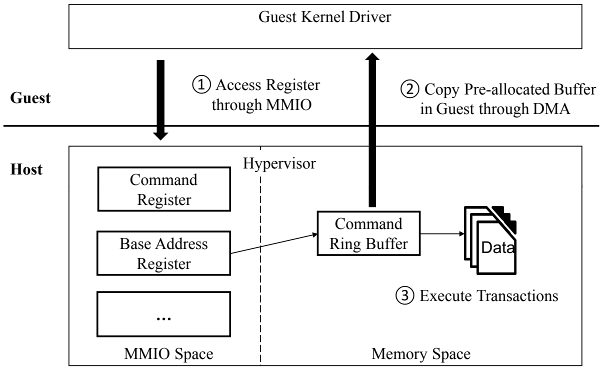


Figure 1: General workflow of the virtual machine transaction.

2 BACKGROUND AND MOTIVATION

We provide the necessary background information to understand what are virtual devices of hypervisors, and how the driver-device communication is handled. After that, we elaborate on the core challenge of hypervisor fuzzing.

2.1 Virtual Devices of Hypervisors

Virtual devices used by guest users are hardware peripherals emulated in modern hypervisors that provide a VM with additional functionality. A virtual device acts as real hardware in a guest VM, which means the drivers in a guest OS can drive a virtual device the same as they do for a physical device. Modern hypervisors virtualize nearly all the hardware such as graphics cards, storage devices, network cards, USB, etc. Each device's protocol specification defines a unique register-level hardware interface for communication between the device and the operating system. Generally, virtualization developers design virtual devices based on the specifications. Because of the virtual device's nature (virtual device emulation is at the host level, and the guest can access virtual devices with arbitrary data), they are typically the largest attack surface of hypervisors.

2.2 Driver-Device Interaction

Overall, a virtual device exposes three important interaction interfaces to guest adversaries: *memory-mapped* I/O (MMIO), *Port* I/O (PIO), and *direct memory access* (DMA). Figure 1 illustrates a general workflow of the virtual machine transaction. At the beginning of a device's execution, the guest driver usually writes data to MMIO or PIO regions to make the device do some initialization work, such as device state setup and address register initialization, which targets the pre-allocated buffer in the guest. After the initialization stage is done, the device turns to a state where the device is ready to process data. The device starts doing some device-specific work (i.e., transferring USB data and sending network packets). The primary interaction mechanism in this data processing stage is DMA, which allows the device to transfer large and complex data with the guest. Since the data processing part is the device's main functionality containing most code paths, this part is more likely to introduce security risks than other parts.

To demonstrate that virtual devices of a hypervisor widely use DMA, we did a statistic analysis on the percentage of devices in QEMU that support DMA communication. We selected the five most popular QEMU device categories (excluding some misc devices and back-end devices) used in virtualization scenarios. We manually analyze whether there is a DMA transmission mechanism in the device. The results are shown in Table 1, which shows that 72.5% of the devices support DMA. Except for the display devices, almost all devices have to use DMA to transfer complex data structures (especially those involving storage and network). Therefore, DMA is extensively used in a hypervisor, requiring us to pay more attention to DMA-related code when applying fuzzing to the hypervisor.

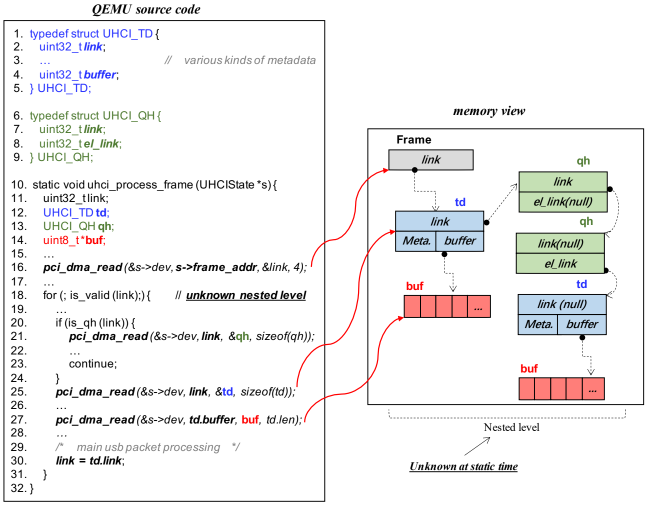
Table 1: Statistics of the number of devices supporting DMA and all devices in QEMU.

Category	Device (support DMA)	Number	Total
USB	uhci, ehci, ohci, xhci	4	4
Storage	esp, ahci, lsi53c810, megasas, mptsas, nvme, pvscsi, sdhci, virtio-blk, virtio-scsi, virtio-9p	11	12
Network	e1000, e1000e, eepro100, pcnet, rocker, rtl8139, tulip, vmxnet3, virtio-net	9	10
Display	(null)	0	7
Audio	ac97, cs4231a, es1370, intel-hda, sb16	5	7
Avg		**29(72.5%)**	40(100%)

2.3 Core Challenge-Nested Structures

The hypervisor is designed to transfer data from/to guest memory for device-driver communication. This transfer operation is always performed through specific APIs related to DMA mechanisms,

[2] https://github.com/hustdebug/v-shuttle



QEMU source code

```
typedef struct UHCI_TD {
  uint32_t link;
  ...                          //   various kinds of metadata
  uint32_t buffer;
} UHCI_TD;

typedef struct UHCI_QH {
  uint32_t link;
  uint32_t el_link;
} UHCI_QH;

static void uhci_process_frame (UHCIState *s) {
  uint32_t link;
  UHCI_TD td;
  UHCI_QH qh;
  uint8_t *buf;
  ...
  pci_dma_read (&s->dev, s->frame_addr, &link, 4);
  ...
  for (; is_valid (link);) {       // unknown nested level
      ...
      if (is_qh (link)) {
          pci_dma_read (&s->dev, link, &qh, sizeof(qh));
          ...
          continue;
      }
      pci_dma_read (&s->dev, link, &td, sizeof(td));
      ...
      pci_dma_read (&s->dev, td.buffer, buf, td.len);
      ...
      /*   main usb packet processing   */
      link = td.link;
  }
}
```

Figure 2: An example presenting nested structures in Universal Host Controller Interface (UHCI). Here td, qh, buf represent three different types of objects, respectively (td and qh are the structures containing the amounts of metadata, while buf is the raw data).

such as pci_dma_read and pci_dma_write in QEMU. Specifically, pci_dma_read copies a block of data from guest memory into a host buffer while pci_dma_write does the opposite procedure. By specifying address arguments, those DMA operations can target any location of the guest's physical memory.

We observed that data objects transferred via the DMA mechanism are often constructed as nested structures (i.e., structure A contains a pointer to structure B), where this nested feature is being supported by the above-mentioned pci_dma_read. More importantly, this nested feature could be multi-layer and multi-type, as the hypervisor organizes these structures in a hierarchical or tree structure starting with a root node. Specifically, this feature blocks fuzzing in exploring hypervisor code mainly due to the following two reasons:

1) Nested Form Construction. It is challenging for fuzzing techniques to construct a nested data object with multiple levels of data or sub-objects, which can be arbitrarily complex. (1) In terms of the overall organization, the devices' data structures can be represented as a hierarchy of nested nodes like a tree. The nodes are blocks of certain data, and pointers establish the links between nodes. Notice that the nested level of this tree could be rather deep, far more than one layer. Also, these tree-like structures can be viewed as recursive data structures because a tree may have other trees as elements. In the tree, a node includes the subtree with all its descendant nodes. Random fuzzing techniques struggle to come up with such a recursively defined data structure, which requires more domain knowledge about the device specifications. (2) At the node level, each node can be regarded as a combination of metadata and pointers. However, the offset of the pointer in a node is uncertain and varies according to the definition of different data structures. Given a node as mutation input, a coverage-guided fuzzer mutates the whole node and treats all fields equally, which results in a random pointer to an invalid or unmapped page. Unlike metadata, pointer values are usually fixed, pointing to meaningful content, and are not expected to be mutated. The lack of semantics at the node level also makes it difficult to construct nested structures. As a result, the fuzzer needs to understand the semantics of data organization (hierarchically nested pattern) and be aware of internal semantics imposed in every single node (which field is the pointer).

2) Node Type Awareness. Since the devices support various data types according to the specifications, fine-grained semantic knowledge about the nested nodes is desired. The nested structures are linked by different types of nodes. Each node has one or more pointers to different data types. The connection relationship of different nodes is regular and specified according to specifications (i.e., the packet descriptor points to the packet body), which requires us to establish a correct pointing relationship between nodes. Moreover, precise pointing relationships can only be known at runtime in many cases: Some fields are used to indicate the exact type of data structure the pointer refers to, since the same pointer can reference multiple types of data; Some fields are used to indicate whether the current node is a termination node. If the pointer has its termination bit set, it assumes there is no more work to complete for the current node and all its children. Hence, the random combination of arbitrary nodes generated by fuzzing does not satisfy the semantic requirements of the devices, which would be rejected at an early stage of processing and heavily limits the fuzzers to find deep bugs. At the node level, the fuzzer requires to extract pointers from given nodes and needs to be aware of the semantics of the pointer (referred node types).

To better illustrate how these nested structures are supported by the hypervisor, the following is the common case of how the hypervisor handles a nested structure: 1) Starting from the root node, the hypervisor first obtains a pointer (specified by an address register) pointing to a data structure A located in the guest memory; 2) the hypervisor dynamically allocates the buffer to hold the copy of A; 3) the hypervisor copies A from guest memory to this allocated buffer using pci_dma_read; 4) referring to a pointer field within A, which indicates its child node B, the hypervisor allocates another buffer to hold the copy of B; 5) the hypervisor performs another pci_dma_read to copy B from guest memory to its allocated buffer; 6) Following the pointer within structure B, the hypervisor performs next pci_dma_read again to copy the next structure C. As above, the hypervisor recursively traverses the tree and moves down until it reaches the termination node, at each node holding a copy of the user-supplied structure.

Without prior knowledge about such a complex nested form of structures, traditional fuzzing cannot properly fuzz the entire data structure as it hardly figures out complex data formats behind each object. Such nested structures are heavily used in hypervisor implementation, severely hindering traditional testing schemes from extending code coverage. We utilize the USB_UHCI protocol as an example to demonstrate the nested structures in the hypervisor.

Example: Nested Structures in USB-UHCI. Universal Host Controller Interface (UHCI) is responsible for providing virtual USB devices to guests in modern hypervisors, which is Intel's spec for USB 1.0 [12]. Figure 2 presents a simplified function uhci_process_frame, which processes USB packet transmitted to USB endpoints. The function, scheduled in each cycle, requires a tree-structured memory buffer which is initially indicated by the device's address register (i.e., s->frame_addr). At line 16, the first entity is copied into the allocated hypervisor buffer link. A specific field in link determines the type of next referenced node, either TD or QH, which indirectly influences the control flow to different blocks. (1) If the indicated type is QH (Queue Head), the data structure pointed by link will be copied into allocated buffer qh (line 21). (2) If the indicated type is TD (Transfer Descriptor), then the data structure pointed by link will be copied into allocated buffer td (line 25). Next, a subsequent memory copy, pointed by a field member of previously copied buffer (i.e., td.buffer), occurs with a certain size (i.e., td.len) at line 27. After performing the USB transactions (line 29), the function continues to traverse the tree recursively during the execution of the entire loop (line 18). Noted here that without built-in knowledge of such a nested form of data structures, the hypervisor cannot be tested thoroughly in two aspects: (1) The execution likely stops due to invalid memory access where the hypervisor cannot fetch meaningful data, before reaching its main functionality (line 29). (2) It is hard to trigger the deep logic that handles recursively defined structures in the hypervisor. Without constructing recursive data structures, we cannot test the program's behavior completely since each recursion accumulates the program's state.

A straightforward way to handle nested structures is to use structure-aware fuzzing techniques. These techniques require developers to create a formal model that precisely captures the specification of devices. The model-based methods follow pre-defined rules to generate corresponding types of structures and concatenate them together. Based on the model, fuzzing techniques enumerate all possible nested forms of structures to verify the functionalities of hypervisors or find bugs. However, structure-aware fuzzing tools have significant drawbacks as they are time-consuming and error-prone processes, thus not scalable for hypervisor testing. Typically, protocol specifications contain hundreds of pages, requiring a non-trivial amount of manual effort to extract the definition of structure. Humans tend to make mistakes in such tedious work of understanding the specification. Besides, the implemented protocol may not entirely correspond to the specification since developers may add new functionalities. Thus, it is not plausible for the large-scale testing of the hypervisor. As a result, an automatic way to handle nested structures is desired. To the best of our knowledge, no prior work handles nested structures automatically, which is essential to apply an efficient and scalable fuzzing to the hypervisor.

3 V-SHUTTLE DESIGN

This section describes the design of V-Shuttle. At a high level, V-Shuttle is designed to be a scalable, semantics-aware, and lightweight hypervisor fuzzing framework, combining coverage-guided fuzzing and static analysis techniques. Moreover, in order to address the hypervisor-specific challenges in fuzzing, V-Shuttle designs two different approaches: 1) redirecting the DMA-related functions; 2) semantics-aware fuzzing via seedpools; We start by providing a threat model for hypervisor security. Based on this threat model, we describe our fuzzing approach.

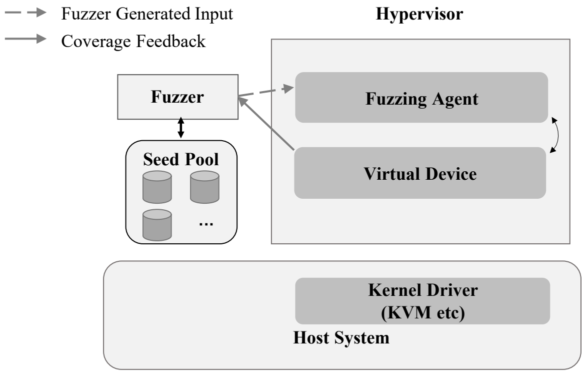


Figure 3: The overview of V-Shuttle.

3.1 Threat Model

We assume that the attacker is a privileged guest user with full memory access inside the virtual machine, which, in turn, can send arbitrary data to its device. This assumption is reasonable because each user has root privileges on his own virtual machine in the public cloud scenario. If the hypervisor does not take care of untrusted data from the guest user, security problems like denial-of-service (DoS), information leakage, or privilege escalation may happen. Once attackers exploit the vulnerability in the hypervisor to escape the VM, they could take over other VMs on the same host, enabling further access to sensitive data outside of the exploited VM.

3.2 System Overview

Fig 3 shows a high-level overview of V-Shuttle, which leverages a fuzzing agent integrated into the hypervisor to feed random input to the **virtual device** - the fuzzing target. The fuzzing agent runs in the hypervisor, persistently sending read/write requests to the tested virtual device. The following list summarizes the high-level functionalities of its main components.

The **fuzzer** is placed outside this hypervisor. Moreover, we leverage persistent mode to enable in-process fuzzing, which means we do not restart a new instance for each new input. This is mainly because: (1) Restarting a hypervisor process or reverting a snapshot is prohibitively expensive in terms of run-time. Even with the fork-server optimization, each new input still incurs the cost of fork(). (2) Hypervisor is an event-based system, which is designed to support long-running interaction. Empirically, most bugs are found by repeated fuzz testing of discovered branches. That is because deep states in hypervisor are less likely to be reached by one test case, relying on multiple interactions to build prior states. This technique not only enhances the overall fuzzing performance but also helps to explore deep interactive states.

The **fuzzing agent** is the core component of V-Shuttle placed inside the hypervisor, which (1) drives the fuzzing loop interacting with both the fuzzer as well as the virtual device, and (2) manages the DMA/MMIO allocation contexts. To adapt the traditional application-fuzzing way to hypervisors, we redirect all data interactions from the guest system to the fuzzed inputs. The fuzzing agent emulates an attacker-controlled malicious guest kernel driver in real-world scenarios, intercepting all DMA and IO read/write instructions from the device. Every read/write operation from the hypervisor device is dispatched to a registered function in the fuzzing agent implementation, which performs actions and returns fuzzer-generated data to the device. Note that the fuzzing agent is a general component integrated into the hypervisor, which can be adapted to almost all types of devices. When deploying the fuzzing process on the new device, no additional human labor is required.

3.3 DMA Redirection

As described in Section 2.3, heavy use of nested structures makes fuzzing ineffective, as it requires precise memory layout of complex nested structures, which is unfriendly to traditional fuzzing. Specifically, when the fuzzer generates random data structures, including the random pointer field, which indicates the following structure, the execution likely stops due to invalid memory access. However, providing syntactically valid structure info for mutators requires lots of manual effort and is not scalable because different virtual devices have different data structure specifications.

To this end, we design a generic *DMA redirection* approach to flatten the nested structures by intercepting the device's access to the guest's memory. Operating based on the source code of the hypervisor, V-Shuttle hooks into the hypervisor's DMA mechanism and converts DMA transferring to reading from fuzzed input. Specifically, we select the DMA-related APIs (e.g., `pci_dma_read` and its wrapped function) as patterns and insert macros into the target device's source code. Then all DMA-related APIs are replaced by macros with methods that read data from files. Thus all memory reads can be redirected to file-based fuzzed inputs during fuzzing. Listing 1 summarizes the simplified code of the DMA redirection. The reason why V-SHUTTLE focuses on the DMA-related functions is that they are responsible for delivering data between guest and host, constituting the key mechanism in constructing the nested data structures. In this way, the operation of DMA addressing is eliminated completely. Since we have full control over the DMA mechanism, any DMA request will be responded with a fuzzed input generated by the traditional coverage-guided fuzzer, no matter where the pointer points to, even address 0. Note that V-Shuttle only manipulates the data that the devices read from the guest's memory, but not the data they write, because the devices are more attackable by malformed guest input. When facing the DMA read request in the run time, the following procedure is performed: 1) V-Shuttle ensures that DMA read function call originates from the target device we are monitoring. This is because we're not interested in DMA transfer requests from other internal system components, which are not controlled by guest users. 2) Given the host process buffer and the buffer's size, V-Shuttle fetches appropriate data from the seed file generated by fuzzer directly, instead of reading from the guest's memory.

```
// before hooking
pci_dma_read(dev, buffer_addr, &buf, size);

// after hooking
if (fuzzing_mode)
    read_from_testcase(&buf, size);
```

Listing 1: Conversion to fuzzed inputs.

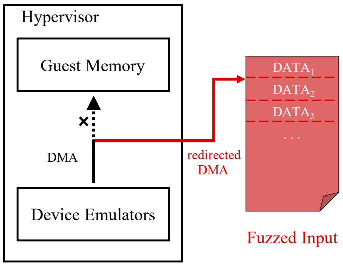


Figure 4: Flatten the nested structures into one-dimensional vectors through DMA redirection.

Figure 4 shows the flattened data structure graphically based on DMA redirection. Contrary to the traditional fuzzing method that generates well-structured inputs tediously in guest space, V-Shuttle directly provides a flattened sequence of DMA data for hypervisor fuzzing. This approach transforms all nested structures into one-dimensional vectors while maintaining the semantics of nested structures, such that all underlying code blocks are reachable through redirected DMA. For each fuzzing iteration, the fuzzer engine first generates the sequences of mutated DMA data. Then the device starts traversing the tree, in which each DMA request is redirected to the fuzzed input, taking a block of data from the DMA sequence in order. Thus, all code paths containing DMA requests are able to be covered smoothly, rather than stuck somewhere the device cannot fetch anything from the guest's memory.

DMA redirection is an approach that only flattens the nested data while leaving the semantics of the structure intact, which removes the need for higher-level tree structures like the one in Firgure 2. We eliminate the pointer in each node while still maintaining the pointing dependence information implied by each node, so this will not break the normal execution of the device. Benefiting from this design, V-SHUTTLE does not rely on pointers to address data but rather directly takes it from the fuzzed inputs. V-SHUTTLE treats all data structures as a bunch of metadata without caring about the pointers, just like userspace programs (ring 3). The content of each node in a higher-level tree structure is generated randomly, including the pointer field. As shown in Figure 2, the pointer to a buffer and the pointer `link` between nodes are both randomly generated. Each time a pointer is addressed, it will be redirected to the fuzzed inputs. This design choice makes the fuzzing process fully automatic to test the hard-to-trigger path requiring nested structures, and does not require any user assistance. As compared with state-of-the-art methods, our approach is fully scalable as well as domain knowledge-free and thus provides better extensibility.

3.4 Semantics-Aware Fuzzing via SeedPools

Using the above-mentioned DMA redirection that flattens the nested structure, we successfully automate the fuzzing process to cover the primary execution paths. However, this intuitive method does not take the node type into account when organizing DMA sequences, which introduces a low-efficiency problem. As described in Section 2.3, the node types in nested structures are dynamically determined, which means the program's control flow changes dynamically depending on previously consumed DMA test cases. However, the above-proposed method only organizes DMA data into one-dimensional vectors in order, without classifying the type of each node. Since the combination sequence of DMA requests differs significantly in different fuzzing iterations, simply concatenating the node sequences leads to the loss of fuzzing semantics for each node. Suppose the hypervisor requests structure A first and then structure B in the current iteration. This renders coverage-guided fuzzer teaching itself to generate seed with semantics close to the combination sequence of A and B. However, if the hypervisor requests structure A first and then structure C in the next iteration, the execution flow will pass the path with A but fail in the path with C. This fuzzer-generated structure B would be rejected by the hypervisor, since structure B presents semantically invalid to the requested structure C. Such an indeterministic process without clear feedback guidance degenerates coverage-guided fuzzing into dumb fuzzing, as the fuzzer would be confused about which direction to evolve. Compared to the coverage-guided fuzzer, a semantics-blind dumb fuzzer will waste time in the mutation process, resulting in low test efficiency.

In order to address this issue, we propose a fine-grained semantics-aware fuzzing method. Fundamentally, our design aims at providing type awareness for the fuzzing engine, so that it can dynamically generate targeted test cases according to the requested data type of the program. This design choice allows us to leverage the advantages of coverage-guided fuzzing, favoring the input which exhibits a new coverage and guiding the fuzzer towards learning to generate semantically valid inputs to each type of data. Overall, with the help of type awareness, the fuzzing engine is capable of providing semantically correct node data when traversing the nested structures.

```
// before hooking
pci_dma_read(dev, buffer_addr, &buf, size);

// after hooking
if (fuzzing_mode)
    read_from_testcase(&buf, size, type_id);
```

Listing 2: Conversion to fuzzed inputs with type constraints.

To this end, V-Shuttle first decouples the nested structures into independent nodes by using an improved DMA redirection method. Next, it implements a seedpool-based fuzzing engine to maintain multiple seed queues, one for each type of decoupled node. Then with the type guidance, V-Shuttle performs semantics-aware fuzzing to the hypervisor. This design method is based on the basic knowledge: the semantics of each node is independent, and there is no dependency between them. Thus, this decoupling method does not destroy the semantics of the whole nested structure. Besides, this method trades off the semantic granularity and deployment cost well, since the number of data structures with different types in the hypervisor implementation is limited (no more than dozens). In the following, we describe each step in detail.

1) Static Analysis to Label DMA Objects. To gain awareness of node types, we retrieve type information indicated by each DMA operation at the code level. Typically, the object transferred by pci_dma_read is uncertain since one function call may serve for different types of objects (when wrapped into an internal function), which requires an accurate type indication for each DMA operation. Therefore, we define a DMA object as a host's structure that holds the copy of the guest's data through DMA. Each DMA object represents a unique type of node. Aiming to label all the DMA objects, V-Shuttle performs static analysis on the hypervisor's source code. In particular, V-Shuttle utilizes a live-variable analysis, which is a special type of data flow analysis. Considering the host's buffer field of DMA operations (e.g., pci_dma_read, and its wrapped function) as the source, we do the backward data flow analysis from the source to its declaration or definition (the DMA object). After collecting all the DMA objects, we assign unique IDs to each of them. These labeled objects help us identify the node type of each DMA request at runtime and ensure that each type of DMA object can be correctly grouped.

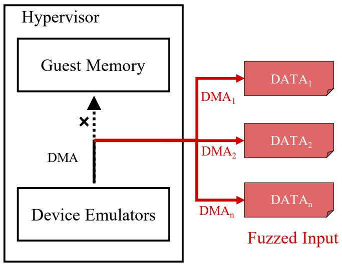


Figure 5: Decouple the nested structures into independent seed queues through DMA redirection with type constraints.

2) DMA Redirection with Type Constraints. Given the labeled objects, V-Shuttle now understands the specific type of node required when performing DMA transfer in the fuzzing iteration. Based on the previous DMA redirection, V-Shuttle introduces additional type constraints when converting the DMA transferring to reading from fuzzed input. Listing 2 summarizes the simplified code of the DMA redirection with type constraints. With additional type constraints, V-Shuttle ensures that each memory read will be constrained to fetch data from the specific seed queue according to the node type (rather than an ordered DMA sequence). In this manner, V-Shuttle decouples the nested structures into individual nodes, and clusters those into categories based on the node type, as shown in Figure 5.

3) Seedpool-Based Fuzzer Design. Aiming to handle multi-object inputs (more than a single file input), we extend the AFL to support multiple seed queues in parallel. We call these multiple parallel queues as *seedpool*. This allows the fuzzer to perform mutations on each seed queue individually for each type of program input. With coverage feedback, the fuzzing engine can quickly pick up on the device's structure and patterns, learning how to generate inputs tailored specifically to each type of object. Even if the program attempts to take different types of data as the input dynamically in the execution flow, it is feasible for the fuzzer to provide the semantically valid inputs from the corresponding seed queues.

This seedpool-based method reuses existing coverage-guided fuzzing algorithms and introduces parallelism. All basic seed queues are treated equally, independent of each other, and adopt the same mutation strategy (deterministic stage, havoc stage, etc.). Besides, all basic queues share a global coverage map, in which any interesting seed that exhibits a new branch will be added to its belonging seed queue. Based on this separate organization, each seed queue will finally evolve its own pattern that favors the input with the corresponding type, utilizing the self-learning ability of the coverage-guided fuzzer.

4) Semantics-aware Fuzzing Process. Combining runtime type awareness and seedpool-based fuzzing engine, V-SHUTTLE performs semantics-aware fuzzing to the hypervisor. The fuzzing process runs in a typical client-server model, where V-Shuttle (server) handles incoming DMA requests from the target hypervisor (client). The main fuzzing loop is as presented in algorithm 1 in Appendix, which has four main steps: (1) V-Shuttle establishes all the basic seed queue and initializes the global coverage map, as in lines 2-5 of algorithm 1. (2) V-Shuttle repeatedly blocks to wait for the DMA requests from the target hypervisor, which indicates the type of required data. (3) V-Shuttle selects seed from the corresponding seed queue and mutates it to generate a new candidate seed. (4) V-Shuttle feeds the target program with the new candidate seed and tracks the coverage information. If the candidate seed explores new coverage, it will be regarded as an interesting seed and pushed into its belonging seed queue, as presented in lines 9-12. This method renders each basic seed queue learning from scratch to generate its own type of interesting seeds. After convergence of this algorithm, we typically obtain semantics-valid input for each type of DMA object and thus enhance the overall efficiency of fuzzing. In the reproducing stage, V-Shuttle automatically recovers the connections between the seeds from different seed pools by keeping a reference from the currently accessed seed to the previously accessed seed, so as to produce reliable and reproducible crashes.

Example: Semantics-Aware Fuzzing in USB-UHCI. Figure 9 we present in Appendix shows the semantics-aware fuzzing in USB-UHCI in detail. As a first step, we list three DMA objects (`qh`, `td`, and `last_td`) we found by live-variable analysis. Specifically, since `qh` holds the user-supplied buffer through `pci_dma_read` in `uhci_process_frame`, we replace the `pci_dma_read(&qh,sizeof(qh))` that serves for the object `qh` with `pci_dma_read(&qh,sizeof(qh),1)`. In addition, since `td` and `last_td` hold the user-supplied buffer through `pci_dma_read` in `uhci_read_td` (`pci_dma_read` here serves for multiple objects in a wrapped function), we replace the function call `uhci_read_td(&td)` that serves for the object `td` with `uhci_read_td(&td,2)`, replace the function call `uhci_read_td(&td)` that serves for the object `last_td` with `uhci_read_td(&td,3)`, and replace `pci_dma_read(td,sizeof(*td))` with `pci_dma_read(td, sizeof(*td),id)`. In this way, these three kinds of objects represent nodes with different semantics in the nested structures. Then guided by the type information, V-Shuttle performs DMA redirection with ID constraints, and dynamically maintains three seed queues that target `qh`, `td` and `last_td`. Each time the hypervisor requests a kind of DMA object, the ID is sent to the fuzzer through the UNIX pipe as guidance information, with which V-Shuttle takes the corresponding seed from the fuzzed inputs. With the coverage feedback, each basic seed queue tends to produce semantics-valid inputs for each of the three DMA objects.

3.5 Lightweight Fuzzing Loop

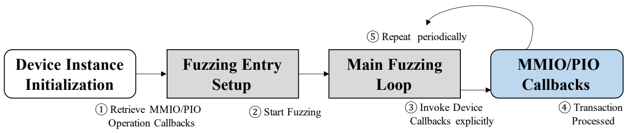


Figure 6: V-Shuttle's fuzzing loop.

Previous work on hypervisor fuzzing usually used some kind of agent running in the guest OS [13, 20, 31, 50, 58]. There are some limitations to this method. (1) This degrades performance due to the frequent use of VM-exit. Whenever the guest VM needs to access the hardware, it triggers a trap that causes a VM-exit in the host kernel. Then the VM-exit transfers the control back to the hypervisor, which emulates the privileged operation on behalf of the VM. This shows that every access request in the guest system results in a "heavy-weight exit" to the hypervisor. (2) This increases the complexity of implementation and communication instability, as it requires establishing a communication channel between host and guest for transmitting fuzzed instructions.

Environment Main Function Model. V-Shuttle utilizes a lightweight design to drive the fuzzing loop. Distinct from previous approaches, V-Shuttle integrates the fuzzing agent into the hypervisor instead of running it in guest OS (Section 3.2). The hypervisor is an event-based system. The control flow is driven by events from guest OS. Therefore, V-Shuttle constructs an environment main function to serve as a fuzzing entry point (fuzzing harness [17, 35]). The whole environment model is shown in Figure 6. V-Shuttle hooks hypervisor's API used to initialize MMIO/PIO regions. When the device is initialized during the VM booting, V-Shuttle retrieves MMIO/PIO operation(read/write) callbacks ①. We regard them as fuzzing entry points, as MMIO and PIO are the main entry for driving the interaction with hardware (as described in Section 2.2), with which V-SHUTTLE drives the fuzzing loop interacting between the fuzzer and the device ②. During the fuzzing loop, V-Shuttle explicitly invokes the I/O callbacks using fuzzer-generated data ③. Afterward, the device emulator processes the I/O requests and performs transactions ④. Finally, the fuzzing process loops back to the start, repeating the above steps ⑤. Hence, as both the fuzzer and the fuzzing agent run in one host system, sharing input files and coverage bitmap between them is straightforward. This design choice makes V-Shuttle lightweight, driver-free, and easily implemented. Because taking device operation callbacks as specific fuzzing entry points can avoid VM-exit, which lowers the performance cost. Meanwhile, when applying fuzzing to a new device, V-Shuttle would automatically set up the fuzzing requirements via the above methods without additional human resources.

4 IMPLEMENTATION

We implement live variables dataflow analysis (described in Section 3.4 - (1)) based on CodeQL [6] static analysis platform. We use American Fuzzy Lop (AFL) version 2.52b for fuzzing [5].

Hypervisor Instrumentation. To apply coverage-guided fuzzing, we selectively instrument device-related code in the hypervisor to gain feedback information, using AFL's edge coverage scheme. When the hypervisor starts, the instrumented code in the target hypervisor writes the coverage feedback to the bitmap, which is exported as a shared memory area accessible by the fuzzer component. Note that the instrumentation is limited to device-related code instead of the whole hypervisor for performance concern [58].

Initial Corpus Collection. To improve the fuzzing efficiency further, we collect initial seeds of valid test cases under standard full-system emulation, logging all accesses to the target device via DMA and MMIO/PIO. This step is optional: one could start from any arbitrary seed or craft test cases on their own.

5 EVALUATION

We evaluate V-Shuttle extensively on QEMU and VirtualBox, which are two popular hypervisor platforms among the world. Both QEMU and VirtualBox are the targets in the virtualization category of many PWN contests, such as Pwn2Own, Driven2Pwn and TianfuCup. We perform experiments to answer the following research questions (**RQ**):

> **RQ1:** Can V-Shuttle be scalable when fuzzing hypervisor virtual devices?
> **RQ2:** What is the performance of dumb fuzzing, structure-aware fuzzing, V-Shuttle and V-Shuttle semantics-aware mode?
> **RQ3:** What is the performance gain of V-Shuttle compared to the state-of-the-art hypervisor fuzzing tools such as Nyx, Hyper-Cube and VDF?
> **RQ4:** How is the vulnerability hunting capability of V-Shuttle?

Our experiments are run on a machine with 2.20 GHz, 48-core Xeon, and 256 GB RAM running Ubuntu 18.04 LTS. We targeted QEMU 5.1.0 and VirtualBox 6.1.14, and built them with *AddressSanitizer* [53] to expose memory corruption bugs. Each experiment is run for 24 hours and repeated for 10 times. We report their average statistical performance [38].

5.1 Scalability

We perform large-scale experiments to demonstrate the scalability of V-Shuttle (**RQ1**). We applied V-Shuttle(with semantics-aware mode enable) to a dozen QEMU virtual devices. The code coverage and performance overhead statistics data in Table 2 shows that V-Shuttle can be easily configured for various virtual devices fuzzing setup and efficiently promote the fuzzing process.

5.1.1 *Code Coverage.*

To examine the ability of V-Shuttle, we perform experiments in QEMU to discover code coverage in 24h fuzzing. We used *gcov* to measure branch coverage. We choose 16 popular QEMU devices for evaluation. They are chosen based on the following features: popularity in the community, development activeness, and diversity of categories. These devices are representative on the x86 platform (including audio, graphics, network, USB, and storage devices), which cover standard virtualization scenes such as cloud and virtual private server (VPS) hosting and desktop virtualization. Each device was fuzzed within a single hypervisor instance.

Table 2 presents some insightful statistics about the line, function, and branch coverages. For branch coverage, the smallest improvement in the percentage of branch coverage was seen in the `AHCI` virtual device (9.7% increase), and the largest improvement in branch coverage was seen in the `CS4231a` virtual device (82.8% increase). Also, the last line shows the final average coverage after applying V-Shuttle. On average, the initial seed test cases covered 40.98% of line coverage, 58.10% of functions coverage, and 25.03% of branch coverage. By fuzzing, V-Shuttle respectively increased their coverage to 87.95%, 89.58%, and 77.18%. V-Shuttle further covered 46.97%, 31.48%, and 52.15% of the code, because the hypervisor-specific solutions in V-Shuttle carry the fuzzing exploration towards the application execution stage.

The results show that our framework can be adapted to various types of devices, including USB, network, audio, storage, etc., which further confirms V-Shuttle's scalability. We emphasize that the whole process of implementing fuzzing to each device is automatic - no human intervention is required at any point. We attribute this feature to our DMA redirection solutions. By redirecting data interaction interfaces to fuzzing input, fuzzing hypervisor becomes the same as fuzzing application, which is naturally suitable for coverage-guided fuzzer, such as AFL, libfuzzer, etc.

However, there are still some code coverages not covered in Table 2, mainly due to two reasons: (1) some devices are not tested at all. (2) some code snippets can only be covered with specific emulated architectures (Arm/PPC/MIPS etc.) and startup configurations.

In summary, V-Shuttle can significantly improve the code coverage as well as scalability.

5.1.2 *Overhead Analysis.*

The last column in Table 2 presents the number of execution per second. As expected, V-Shuttle manages to achieve a throughput of 6110.23 executions per second on average, since we use a very lightweight design without `fork()` or restarting of the hypervisor. Besides, as we integrate the fuzzing agent into the hypervisor instead of running it in guest OS, the fuzzer spends negligible time on data interaction. This design choice makes V-Shuttle comparable to traditional application fuzzing. We demonstrate that our framework offers a significant advantage over other designs where the fuzzer runs in a kernel module. For comparison, we've also evaluated the throughput of dumb fuzzing as our baseline, which simply writes a bunch of random data into basic interfaces (MMIO, DMA) without the knowledge about the nested DMA structures. The results in Table 2 show our semantics-aware DMA redirection fuzzing approach is comparable to dumb fuzzing on the same machine and workload.

5.2 Effectiveness

To validate the effectiveness of V-Shuttle main framework and the semantics-aware fuzzing mode, we evaluate V-Shuttle-M (disabling the semantics-aware fuzzing mode) and V-Shuttle-S (enabling semantics-aware fuzzing mode) for comparison. We then implement a dumb fuzzing as our baseline, which has no knowledge of the DMA data and its deep nested characteristics (like VDF), and randomly mutates *k* bits of inputs to the basic interfaces (MMIO,

Table 2: The line, function and branch coverage of V-Shuttle as well as the performance results on the 16 QEMU devices (24 hours each). Initial coverage shows the percentage covered during device initialization states (i.e., BIOS and the guest kernel initialization of the device). Total coverage shows the percentage covered after 24 hours of fuzzing.

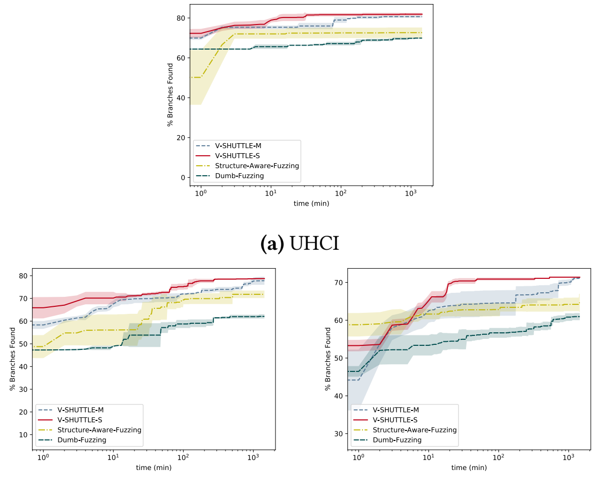
	Device	Line Coverage		Functions Coverage		Branches Coverage		Speed(exec/s)	
		Initial	Total	Initial	Total	Initial	Total	Dumb-Fuzzing	V-Shuttle
Audio	CS4231a	30.00%	**96.10%**	57.10%	**100.00%**	3.00%	**85.80%**	10918.21	**7632.70**
	Intel-HDA	68.30%	**95.00%**	78.60%	**95.20%**	42.10%	**78.30%**	9596.41	**8568.50**
	ES1370	54.20%	**99.62%**	73.70%	**100.00%**	33.80%	**91.91%**	8786.85	**6496.04**
	SoundBlaster	12.30%	**99.19%**	28.60%	**100.00%**	3.00%	**81.52%**	5123.76	**3242.22**
Graphics	ATI-VGA	27.40%	**86.00%**	66.70%	**80.00%**	15.30%	**79.40%**	10350.61	**10103.42**
Network	E1000	36.20%	**94.20%**	46.90%	**96.90%**	16.10%	**74.50%**	5532.90	**1186.92**
	NE2000	6.70%	**89.60%**	28.60%	**100.00%**	3.80%	**71.90%**	12213.31	**11392.45**
	PCNET	24.60%	**97.40%**	44.80%	**100.00%**	8.30%	**88.90%**	5880.21	**4833.35**
	RTL8139	28.10%	**97.60%**	59.10%	**97.70%**	12.30%	**88.40%**	6333.37	**5495.18**
USB	UHCI	81.30%	**89.10%**	86.10%	**88.90%**	68.90%	**82.30%**	10592.12	**9273.25**
	EHCI	40.70%	**82.70%**	53.40%	**89.00%**	32.70%	**71.90%**	3869.43	**2265.34**
	OHCI	46.90%	**83.70%**	65.10%	**86.00%**	33.30%	**79.20%**	7221.49	**5228.43**
Storage	NVME	38.60%	**72.40%**	47.30%	**76.40%**	22.80%	**65.10%**	10981.52	**7870.23**
	Lsi53c895a	26.90%	**79.00%**	46.70%	**71.10%**	9.30%	**75.70%**	6363.84	**4091.53**
	Megasas	58.10%	**63.80%**	68.30%	**70.00%**	43.90%	**58.50%**	5863.47	**4558.58**
	AHCI	75.30%	**81.80%**	78.60%	**82.10%**	51.90%	**61.60%**	5577.74	**5525.55**
Average		40.98%	**87.95%**	58.10%	**89.58%**	25.03%	**77.18%**	7844.64	**6110.23**

DMA). Also, we studied the Intel specifications and built structure-aware fuzzing (also refers to generation-based fuzzing) that targets these devices (**RQ2**). This structure-aware fuzzing is manually written according to various rules in the device specification, including manual construction of the nested structure and establishment of the relationship between different types of nodes. Our customized structure-aware fuzzing follows the common steps: (1) Setup device states, registers using I/O ports or mapped memory. (2) Generate random device data structures. (3) Issue commands for processing the data structures.

We choose 3 USB controllers (UHCI, OHCI, and EHCI) for evaluation, mainly for the following reasons: 1) The USB controllers use DMA more frequently and are more representative. Since our work mainly focuses on DMA, the performance will be good as long as the devices use DMA. Also, as described in Section 2.2, most devices of hypervisors use DMA. 2) The security of USB is particularly important. USB is widely used and deployed in the public cloud, and is usually mounted by default. In recent years, there are many virtual machine escaping cases on USB devices [1–3]. Hence it is crucial to test USB. 3) Building the structure-aware fuzzing for a device involves massive human effort, since it requires us to understand the specification. Therefore, we limit our comparison to these three devices.

We evaluate V-Shuttle-M, V-Shuttle-S, structure-aware fuzzing, and dumb fuzzing on QEMU for 24 hours, 10 runs. We ran the target hypervisor with *gcov*, and called *__gcov_flush*() every second to dump the coverage found over time. Figures 7 shows the branch coverage results in log scale.

5.2.1 V-Shuttle Main Framework. We can learn from Figure 7 that V-Shuttle-M can greatly outperform dumb fuzzing. That is because dumb fuzzing has no prior knowledge of data structure defined by specifications, thus stuck in the initial fuzzing stage. V-Shuttle-M also discovers more branches than structure-aware fuzzing in almost every case. For instance, on EHCI, structure-aware fuzzing discovers about 64.6% of branches in 24 hours, while V-Shuttle-M discovers 71.4%. According to our analysis, it is mainly because (1) V-Shuttle-M's feature could cover hard-to-take branches caused by frequent use of `pci_dma_read` function, where its argument `address` is hardly predictable, which points to an unknown type structure with multi-layered feature (Section 3.3).



(a) UHCI

(b) OHCI

(c) EHCI

Figure 7: QEMU's branch coverage results of V-Shuttle during a 24-hours run. The line indicates the averages while the shaded area represents the 95% confidence intervals across 10 runs.

(2) Our structure-aware fuzzing is still in progress and not well-crafted enough. This result tells, unlike the automatic and accurate nature of V-Shuttle-M, writing manual rules by human effort is subject to error-prone as well as time-consuming tasks.

5.2.2 Semantics-Aware Fuzzing Mode. Comparing V-Shuttle-S with V-Shuttle-M, we can learn that the final coverage results are nearly identical for all the cases, but V-Shuttle-S reaches the peak point faster than V-Shuttle-M. Due to the semantics-aware feature of the seedpool-based seed generation, V-Shuttle-S quickly learned how to generate semantically valid data structures through different contexts, which led to the deep execution of the hypervisor. We believe this provides lots of insight to accelerate hypervisor fuzzing.

In summary, both the V-Shuttle main framework and semantics-aware fuzzing mode present outstanding coverage improvement (better than loosely written structure-aware templates) without ongoing manual efforts. Better performance can be achieved if integrating the semantics-aware fuzzing mode, which could further accelerate the convergence speed of V-Shuttle.

Table 3: The branch coverage found by VDF, Hyper-Cube, Nyx and V-Shuttle on the 8 QEMU devices. Std denotes the standard error over 10 runs. △ denotes the difference in percentage points between V-Shuttle and Nyx.

	VDF	Hyper-Cube	Nyx	V-Shuttle		
Device	Cov	Cov	Cov	Cov	Std	△
CS4231a	56.00%	74.76%	74.76%	**85.80%**	1.07	11.04%
Intel-HDA	58.60%	79.17%	78.33%	78.30%	0.55	-0.03%
ES1370	72.70%	91.38%	91.38%	91.91%	1.21	0.54%
SoundBlaster	81.00%	83.80%	81.34%	81.52%	0.42	0.18%
E1000	81.60%	66.08%	54.55%	**74.50%**	0.90	19.95%
NE2000	71.70%	71.89%	71.89%	71.90%	0.92	0.01%
PCNET	36.10%	78.71%	89.49%	88.90%	1.35	-0.59%
RTL8139	63.00%	74.68%	79.28%	**88.40%**	0.64	8.72%

5.3 Comparison with State-of-the-Art Fuzzers

To answer **RQ3**, we compare V-Shuttle (enabling semantics-aware fuzzing mode) against state-of-the-arts fuzzers (Nyx, Hyper-Cube and VDF). Unfortunately, Nyx and Hyper-Cube were not openly available at the time we performed the experiments. Therefore, we only compare them with the devices we have already tested (Table 2), and we compare them against the numbers published in their paper. In the comparison, we carefully make the settings to ensure fairness. 1) There is little difference in different versions of QEMU, as the amount of code changes is tiny. 2) The evaluation time is the same. All of them are the results of 24-hour fuzzing. 3) The machine configuration is also comparable. Most importantly, according to our observation, the fuzzing speed is not the key factor affecting the final coverages. Almost all the coverages of the tested devices by Nyx and Hyper-Cube reach the peak at a very early stage of the 24 hours.

We display the overall results in Table 3. As can be seen, our approach achieves significantly better than VDF in all (but one) scenarios. The difference is due to the fact that the code changed since VDF performed their experiments, which does not represent a real difference in performance. Compared to Nyx, there are only 3 out of 8 cases are improved, mainly due to the following two reasons. First, compared with other cases, these three cases use DMA more intensively. Therefore, the performance gain benefits from our DMA redirection approach. Second, the amount of the code of the other five devices is very small (from 701 to 1753 LoC), which means that most of the code paths can be triggered in simple MMIO operations. Thus, there is little space for improvement. The same results can be seen when compared to blind fuzzer Hyper-Cube.

However, on the complex devices, the advantages of V-Shuttle begin to show. For `E1000` and `RTL8139`, we are able to achieve 19.95% and 8.72% better than Nyx. After manual analysis, we found that nested structures are intensively used when the device encapsulates the network packets. We attribute the better performance to our DMA redirection approach, which unrolls the nested structures and helps to discover deeper code paths. This demonstrates that our approach significantly improves the coverage finding capability.

5.4 Vulnerability Hunting

We demonstrate the vulnerability hunting capability of V-Shuttle(**RQ4**) from two aspects: ❶ Is V-Shuttle able to uncover new bugs and vulnerabilities in different hypervisors? ❷ Can V-Shuttle reproduce previously known vulnerabilities found by other hypervisor fuzzers?

Table 4: Overview of the vulnerabilities found by V-Shuttle in our targets.

Hypervisor	Type	#Bugs
QEMU	Use-After-Free	2
	Heap-based Buffer Overflow	4
	Stack Overflow	1
	Infinite Loop	3
	Segmentation Fault	6
	Null Pointer Derefence	4
	Assertion	6
VirtualBox	Heap-based Buffer Overflow	4
	Divide by Zero	2
	Segmentation Fault	3

5.4.1 Uncover New Vulnerabilities Ability. An overview of the types of crashes found is shown in Table 4. A full list of the vulnerabilities with more details on the exploitability can be found in Appendix. V-Shuttle has successfully detected 35 previously unknown bugs, including 26 bugs from QEMU and 9 from VirtualBox. We have responsibly reported all the bugs to corresponding hypervisor developers and have received their positive feedback. At the time of paper writing, 24 of all the bugs have been fixed. 17 of them got CVE numbers due to the severe security consequences.

Bug Diversity. The 35 bugs in Table 6 cover almost all common types of memory errors and almost all common device types of hypervisor, showing that V-Shuttle can improve hypervisor security from a variety of aspects. In particular, buffer overflows, and use-after-free bugs are commonly believed to be exploitable, whereas V-Shuttle found 12 bugs and 1 bug, respectively. V-Shuttle also detected 5 assertion failures from QEMU, which indicate that the executions reach unexpected states.

Case Study 1: QEMU OHCI Out-of-bounds Access (CVE-2020-25624) V-Shuttle uncovered an out-of-bounds (OOB) read/write access vulnerability in QEMU's `USB OHCI` controller emulator. The issue occurs while servicing the isochronous transfer descriptors (ITD), which describes the isochronous endpoint's data packets and is linked into the endpoint list. OHCI controller derives variables `start_addr`, `end_addr` from `iso_td` supplied by the guest user via DMA transfer. The device calculates the length of the transmission according to the `start_addr` and `end_addr`. The problem here is that the device does not check the negative length when `end_addr` is less than `start_addr`, which could cause OOB read and write due to integer overflow vulnerability. A guest user using this flaw could crash the QEMU process resulting in a denial of service.

It is difficult to trigger this bug by traditional fuzzing, as it requires prior knowledge about the layout of the endpoint linked list to avoid invalid memory access due to randomly generated pointer values. However, V-Shuttle was able to trigger this vulnerability by intercepting the device's DMA read operations and supplying fuzzed structure `iso_td` to the device regardless of the pointer. In this way, `len` can be fuzzed enough to cause overflow; otherwise, this field may remain unfuzzed.

Case Study 2: VirtualBox `BusLogic` Heap-based Buffer Overflow (CVE-2021-2074) V-Shuttle identified a heap-based buffer overwrite vulnerability in VirtualBox's `BusLogic SCSI` emulator, which has 8.2 CVSS Score according to *CVE Details* [7]. Successful attacks of this vulnerability can result in the takeover of Oracle VM VirtualBox. The `BusLogic` device parses the command buffer and processes the command parameters from the guest. When initializing a new command, the device gets the number of bytes `cbCommandParametersLeft` for the command code parameters. `cbCommandParametersLeft` is subtracted by one each time while filling the buffer with parameters. Then the device starts execution of command if there are no parameters left. However, `cbCommandParametersLeft` is not checked against 0 at the start. This allows an attacker to first set the number to 0 and then issue a command initialization, causing it to underflow. This will lead to an arbitrary heap out-of-bounds write up to the size of the `uint8_t`, which can be exploited to escape virtual machine.

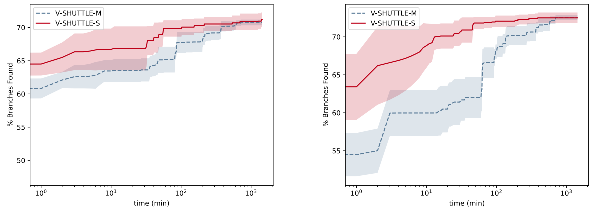
In the fuzzing process, V-Shuttle continuously generated I/O operations that could let the execution run towards the command process function and finally trigger the vulnerability.

Table 5: Eifficiency of V-Shuttle in finding previously known vulnerabilities. We measured the total number of executions and the time required. V-Shuttle found all of the known vulnerabilities within a reasonable amount of time.

Bug	Description	Exec	Time	Found
CVE-2020-25625	OHCI infinite loop	40.5M	2h16m50s	✓
CVE-2020-25085	SDHCI Heap buffer overflow	8.88M	26m19s	✓
CVE-2021-20257	E1000 inifinite loop	235k	40s	✓
CVE-2020-25084	EHCI use-after-free	79.4M	4h37m22s	✓
CVE-2020-11869	ATI-VGA integer overflow	35.6M	2h22m40s	✓

5.4.2 Rediscover Old Vulnerabilities Ability. To demonstrate practicality, we also present some of the publicly known vulnerabilities we could find using our framework. We picked a set of previously known security vulnerabilities on a vulnerable QEMU (version 5.0.0). Additionally, we tried to rediscover three high-impact CVEs found by Nyx (CVE-2020-25085, CVE-2020-25084, and CVE-2021-20257). In total, we analyzed five known bugs, and we measured the number of executions required to discover these bugs. As shown in Table 14, V-Shuttle found all of these previously known bugs within a reasonable number of executions (i.e., from 235 K to 79.4 M) as well as within a reasonable amount of time (i.e., from 40 sec to around 4 hrs).

6 DEPLOYMENT AND APPLICATION OF V-SHUTTLE



(a) UHCI

(b) EHCI

Figure 8: Ant Group-QEMU's branch coverage results found by V-Shuttle during a 24-hours run. The line indicates averages while the shaded area represents the 95% confidence intervals across 10 runs.

Now there are many open-source hypervisor solutions. Many enterprises customize their cloud services based on open source hypervisors. This fact introduces additional risk, complexity, and costs for fuzz testing and bug fixing. Consequently, cloud computing development presents us strongly with a need to establish a general hypervisor fuzzing platform. By collaborating with the worldwide leading Internet company Ant Group, we have an opportunity to deploy and examine V-Shuttle on its commercial platform.

We choose two USB devices, UHCI and EHCI, which are deployed in the production environment and perform the experiments to discover branch coverage. The results are shown in Figure 8. As expected, V-Shuttle-S achieves higher coverage than V-Shuttle-M in the early time and converges to almost the same coverage in the long term. This experiment shows similar performance as in QEMU as described in Section 5.2. By experimenting with Ant Group's QEMU, we also show that our framework can be ported to a variety of hypervisors with little effort. Specifically, when deploying V-Shuttle to a new hypervisor, what a professional needs to do is to perform a static analysis on the target device to collect DMA objects, integrate the fuzzing agent into the hypervisor by some simple configurations, instrument and compile the source code. This process is very lightweight, e.g., it only takes about an hour for a professional in related fields to implement V-Shuttle into a new hypervisor. V-Shuttle's scalability allows it able to be quickly ported to various hypervisor implementations. The deployment and application of V-Shuttle on Ant Group's commercial platform further demonstrates that V-Shuttle is not just an academic tool but also a meaningful tool in the real world.

7 RELATED WORK

Fuzzing Techniques. In the past few years, fuzzing has proven to be a very successful technique for discovering software vulnerabilities [14, 16, 19, 25, 32, 49]. AFL is one of the most well-known fuzzers [5]. Later, many advanced fuzzers were developed based on AFL [15, 29, 41]. Some research combined fuzzing with other bug detection technologies [18, 44, 62]. Other approaches focused on improving the scheduling algorithms [21, 40] and feedback mechanisms [61]. Recently, hybrid fuzzing methods have been researched extensively [24, 57, 60, 65, 66]. Additionally, some research focused on evaluating fuzzers [38, 39]. Manès et al.'s survey [42] provides an in-depth discussion in fuzzing.

Hypervisor Fuzzing. Most previous research on fuzzing hypervisors used blind fuzzing [9, 13, 20, 22, 31, 43]. Later, some security researchers try to combine the hypervisor fuzzing with coverage guidance [4, 8, 58]. In academia, VDF [34] is the first hypervisor fuzzing platform that applies coverage-guided fuzzing to hypervisors, which utilizes AFL toolchain to instrument QEMU source code to collect the coverage information. However, VDF does not take device protocol into account and only generates rough seed input, which limits its performance. Hyper-Cube [50] leverages a custom OS to provide multi-dimensional fuzzing, which is high-throughput. However, Hyper-Cube struggles to explore complex devices due to its black-box design. Nyx [51] proposes to fuzz hypervisors by using fast snapshots and coverage guidance, which increases the ability to test interesting behavior. However, its nested virtualization design makes it more complex to setup the environment, as the target hypervisor needs to run inside KVM-PT. Additionally, Nyx still needs a lot of manual work put into specific generators, not automatically enough.

Other Fuzzing Techniques. There are some similar fuzzing tools focusing on device-driver interactions. PeriScope [55] hooks into the kernel's page fault handling mechanism to apply coverage-guided fuzzing on WiFi drivers. Agamotto [56] accelerates the kernel driver fuzzing with virtual machine checkpoints. USBFuzz [47] uses an emulated device in the VM to fuzz USB drivers. Different from hypervisor fuzzing, they target at the kernel driver side. Additionally, more and more researchers adopt fuzzing to a wider set of targets ranging from kernel [27, 30, 33, 36, 37, 45, 52, 63, 64] to IoT [23, 26, 67].

8 DISCUSSION

PoC Reconstruction. V-Shuttle requires some human resources to help reconstruct PoC. Because our core fuzzing engine is integrated into the hypervisor's host process instead of running in the guest system, we need to recover the PoC from the seed sequences. If the hypervisor crashes when fuzzing the target device, we will restart fuzzing another instrumented hypervisor which enables recording all the MMIO/PIO and DMA access logs. Given the crash backtrace and all access logs, then we construct the PoC driver manually. We intend to find a more automatic way as future work.

Supporting Closed-Source Hypervisors. V-Shuttle requires hypervisor modifications in order to redirect a device's data requests to the guest system via the MMIO/PIO and DMA interface. A few components of V-Shuttle are deployed before the compilation phase. Meanwhile, we use AFL's compile-time instrumentation to obtain the coverage information. For this reason, for now, V-Shuttle does not support closed-source hypervisors such as VMware Workstation. We believe this could be overcome by adopting a sort of binary patching and dynamic binary instrumentation techniques with extra effort in the future.

Hypervisor Internal States. Due to the heavy costs involved in hypervisor restarts, V-Shuttle continuously fuzzes the hypervisor without restarting it between fuzzing iterations. This can limit the effectiveness of fuzzing because the internal states of the target system persist across iterations. Changing the target device's internal states can also lead to instability in the coverage guidance, as the same input can exercise different code paths depending on the hypervisor state. Worse, when changes to the persisting states accumulate, the device may eventually lock itself up.

9 CONCLUSION

The virtual device represents an attack surface, through which software vulnerabilities in the hypervisor can be exploited. However, existing hypervisor fuzzers are inefficient (e.g., VDF), and meanwhile not scalable or automatically extendable (e.g., Nyx). To address the limitations of existing hypervisor fuzzing techniques, in this paper, we propose V-Shuttle, a scalable and semantics-aware framework to fuzz virtual devices in hypervisors. V-Shuttle is portable to fuzz devices on different hypervisors, leveraging coverage-guided fuzzing. Furthermore, V-Shuttle can effectively enable a broad fuzzing, which can target a wide range of devices. To examine the performance of V-Shuttle, we apply it on QEMU and VirtualBox, two of the most popular hypervisor platforms. Via extensive evaluation, we show V-Shuttle is very effective and efficient. It discovers 26 new memory bugs in QEMU and 9 new bugs in VirtualBox, with 17 bugs received official CVEs. Furthermore, by collaborating with a worldwide leading Internet company, we also deploy V-Shuttle on its commercial platform. The results again demonstrate the superiority of V-Shuttle. To facilitate future related research, we will open source V-Shuttle at https://github.com/hustdebug/v-shuttle.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments on our work. This work is partly supported by the National Key R&D Program of China (2020YFB1804705), NSFC under No. U1936215, the Key R&D Program of Zhejiang Province (2021C01036, 2020C01021), the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform: ZJUNGICS2021021), State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCHA202001, and the Major Scientific Project of Zhejiang Lab (2018FD0ZX01). This work is also partly supported by the Ant Group through the Ant Research Intern Program.

REFERENCES

[1] 2020. CVE-2020-14364 QEMU: usb: out-of-bounds r/w access issue while processing usb packets, 2020. https://bugzilla.redhat.com/show_bug.cgi?id=1869201.

[2] 2020. Pwn2Own 2020: Oracle VirtualBox Escape, 2020. https://starlabs.sg/blog/2020/09/pwn2own-2020-oracle-virtualbox-escape/.

[3] 2020. Pwning VMware, Part 2: ZDI-19-421, a UHCI bug, 2020. https://nafod.net/blog/2020/02/29/zdi-19-421-uhci.html.

[4] 2021. Adventures in Hypervisor: Oracle VirtualBox Research. https://starlabs.sg/blog/2020/04/adventures-in-hypervisor-oracle-virtualbox-research/.

[5] 2021. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[6] 2021. CodeQL for research. https://securitylab.github.com/tools/codeql/.

[7] 2021. CVE details. https://www.cvedetails.com/.

[8] 2021. Fuzzing QEMU Device Emulation. https://qemu.readthedocs.io/en/latest/devel/fuzzing.html.

[9] 2021. 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/.

[10] 2021. PWN2OWN. https://www.zerodayinitiative.com/blog/2021/4/2/pwn2own-2021-schedule-and-live-results.

[11] 2021. Tianfu Cup International Cybersecurity Contest. http://tianfucup.com/.

[12] 2021. Universal Host Controller Interface. https://wiki.osdev.org/Universal_Host_Controller_Interface.

[13] 2021. Viridian Fuzzer. https://github.com/mwrlabs/ViridianFuzzer.

[14] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.

[15] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1597–1612.

[16] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[17] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 975–985.

[18] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2020. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. *arXiv preprint arXiv:2002.03416* (2020).

[19] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1985–2002.

[20] Sören Bleikertz. 2021. XenFuzz. https://www.openfoo.org/blog/xen-fuzz.html.

[21] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[22] Amardeep Chana. 2021. MWR-Labs: Ventures into Hyper-V - Fuzzing hypercalls. https://labs.mwrinfosecurity.com/blog/ventures-into-hyper-v-part-1-fuzzing-hypercalls/.

[23] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.

[24] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. {MEUZZ}: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*. 77–92.

[25] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1967–1983.

[26] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1201–1218.

[27] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[28] Nelson Elhage. 2011. Virtunoid: A kvm guest! host privilege escalation exploit. *Black Hat USA* 2011 (2011).

[29] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[30] Google. 2021. syzkaller - kernel fuzzer, 2019. https://github.com/google/syzkaller..

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

[32] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. In *Annual Computer Security Applications Conference*. 360–372.

[33] HyungSeok Han and Sang Kil Cha. 2017. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2345–2358.

[34] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. 2017. Vdf: Targeted evolutionary fuzz testing of virtual devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 3–25.

[35] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. Fuzzgen: Automatic fuzzer generation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2271–2287.

[36] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[37] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In *NDSS*.

[38] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[39] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21). USENIX Association*.

[40] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 533–544.

[41] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1949–1966.

[42] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

[43] Tavis Ormandy. 2007. An empirical study into the security exposure to hosts of hostile virtualized environments.

[44] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Parmesan: Sanitizer-guided greybox fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2289–2306.

[45] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 729–743.

[46] Gaoning Pan, Xingwei Lin, Xinlei Ying, Jiashui Wang, and Chunming Wu. 2021. Scavenger: Misuse Error Handling Leading To QEMU/KVM Escape. *Black Hat Asia* (2021).

[47] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing {USB} Drivers by Device Emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2559–2575.

[48] Aaron Portnoy and Pedram Amini. 2021. Sulley. https://github.com/OpenRCE/sulley.

[49] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[50] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS*. 23–26.

[51] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[52] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kafl: Hardware-assisted feedback fuzzing for {OS} kernels. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 167–182.

[53] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 309–318.

[54] Zhijian Shao, Jian Weng, and Yue Zhang. 2020. A Guest-to-Host Escape on QEMU/KVM Virtio Device. *Black Hat Asia* (2020).

[55] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*.

[56] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2541–2557.

[57] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[58] Jack Tang and Moony Li. 2016. When virtualization encounter AFL. *Black Hat Europe* (2016).

[59] Peach Tech. 2021. Peach. http://www.peachfuzzer.com/.

[60] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

[61] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. NDSS.

[62] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 765–777.

[63] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1643–1660.

[64] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 818–834.

[65] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 745–761.

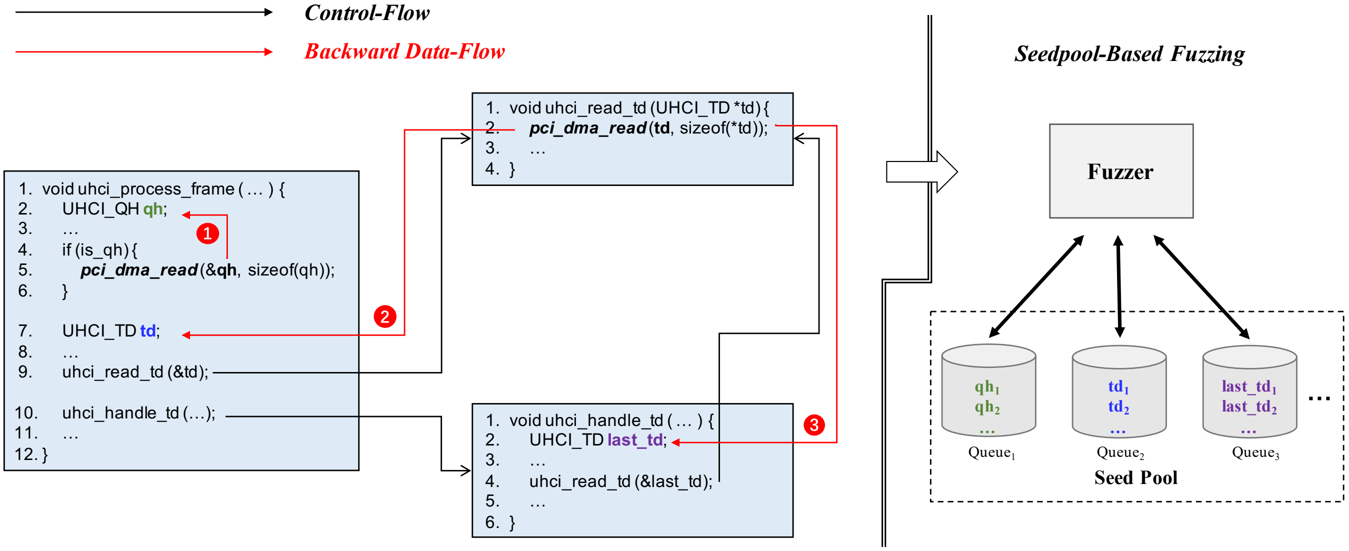
[66] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing.. In *NDSS*.

[67] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1099–1114.

10 APPENDIX

Algorithm 1 Main semantics-aware fuzzing loop of V-Shuttle

```
Input: Initial seeds queues Seedpool[], Target Hypervisor H
// setup each basic seed queues and global information ;
for all queue of the Seedpool[] do
    queue.setup();
end for
GlobalMap.init();
repeat
    id = H.request()
    seed = Mutate(Seedpool[id]);
    Cover = H.feed(seed);
    if Cover.haveNewCoverage() then
        Seedpool[id].push(seed)
    end if
until timeout or abort-signal;
Output: Crashing seeds crashes
```



```
void uhci_process_frame ( … ) {
    UHCI_QH qh;
    …
    if (is_qh) {
        pci_dma_read (&qh, sizeof(qh));
    }

    UHCI_TD td;
    …
    uhci_read_td (&td);

    uhci_handle_td (…);
    …
}
```

```
void uhci_read_td (UHCI_TD *td) {
    pci_dma_read (td, sizeof(*td));
    …
}
```

```
void uhci_handle_td ( … ) {
    UHCI_TD last_td;
    …
    uhci_read_td (&last_td);
    …
}
```

Figure 9: An example presenting the semantics-aware fuzzing via seedpools. Here qh, td, last_td refer to three different DMA objects, respectively (td and last_td are the same type of data structure, but in different contexts. We still treat them as different DMA objects.). Each of them will be organized into an independent seed queue.

Table 6: List of 35 previously unknown vulnerabilities in QEMU and VirtualBox discovered by V-Shuttle. The remaining issues marked as *requested* are still under investigation. Issue-ID indicates the assertion failures we reported through `launchpad.net`.

Hypervisor	Description	Device Type	CVE/Issue-ID	CVSS Score	Impact
QEMU	Heap buffer overflow (write) in `ohci_copy_iso_td`	USB	**CVE-2020-25624**	5.0	DoS
	Stack buffer overflow (read) in `ohci_service_iso_td`	USB	confirmed	-	DoS
	Heap buffer overflow (read) in `ohci_service_td`	USB	confirmed	-	DoS
	Infinite loop in `e1000e_write_packet_to_guest`	Network	CVE-2020-25707	2.5	DoS
	OOB access in `ati_2d_blt`	Graphics	CVE-2020-27616	2.8	DoS
	Reachable assert failure via `eth_get_gso_type`	Network	CVE-2020-27617	3.8	DoS
	Divide by zero in `dwc2_handle_packet`	USB	CVE-2020-27661	3.8	DoS
	Integer Overflow in `sm501_2d_operation`	Graphics	requested	-	DoS
	Infinite loop in `xhci_ring_chain_length`	USB	CVE-2020-14394	3.2	DoS
	Heap-use-after-free in `nic_reset`	Network	requested	-	Exploitable
	Heap buffer overflow (write) in `dp8393x_do_transmit_packets`	Network	confirmed	-	DoS
	Failed malloc in `omap_rfbi_transfer_start`	Graphics	requested	-	DoS
	Infinite loop in `allwinner_sun8i_emac_get_desc`	Network	confirmed	-	DoS
	Divide by zero in `exynos4210_ltick_cnt_get_cnto`	Timer	confirmed	-	DoS
	Divide by zero in `zynq_slcr_compute_pll`	Misc	confirmed	-	DoS
	Failed malloc in `vmxnet3_activate_device`	Network	CVE-2021-20203	3.2	DoS
	NULL pointer derefence in `fdctrl_read`	Storage	CVE-2021-20196	3.2	DoS
	Heap-use-after-free in `ehci_flush_qh`	USB	requested	-	Exploitable
	NULL pointer derefence in `lsi53c895a`	Storage	requested	-	DoS
	NULL pointer derefence in `vmport_ioport_read`	Core	requested	-	DoS
	NULL pointer derefence in `a9_gtimer_get_current_cpu`	Timer	requested	-	DoS
	Assertion in `usb_msd_send_status`	USB	#1901981	-	DoS
	Assertion in `usb_ep_get`	USB	#1907042	-	DoS
	Assertion in `ohci_frame_boundary`	USB	#1917216	-	DoS
	Assertion in `vmxnet3_io_bar1_write`	Network	#1913923	-	DoS
	Assertion in `lsi_do_dma`	Storage	#1905521	-	DoS
VirtualBox	Heap buffer overflow (write) in `xhciR3WriteEvent`	USB	CVE-2020-2905	8.2	Exploitable
	Heap buffer overflow (write) in `xhciR3WriteEvent`	USB	CVE-2020-14872	8.2	Exploitable
	OOB Read in `ehciR3ServiceQHD`	USB	CVE-2020-14889	6.0	Info leak
	Divide by zero in `e1kTxDLoadMore`	Network	CVE-2020-14892	5.5	DoS
	Integer overflow in `e1kGetTxLen`	Network	CVE-2021-2073	4.4	DoS
	Heap buffer overflow (write) in `buslogicRegisterWrite`	Storage	**CVE-2021-2074**	8.2	Exploitable
	Divide by zero in `ataR3SetSector`	Storage	CVE-2021-2086	6.0	DoS
	NULL pointer derefence in `blk_read`	Storage	CVE-2021-2130	4.4	DoS
	Unintialized stack object in `LsiLogicSCSI`	Storage	CVE-2021-2123	3.2	Info leak