Truman: Constructing Device Behavior Models from OS Drivers to Fuzz Virtual Devices

Zheyu Ma*†¶, Qiang Liu†, Zheming Li*¶, Tingting Yin‡, Wende Tan§, Chao Zhang*‡¶✉, Mathias Payer†
*Institute for Network Sciences and Cyberspace (INSC), Tsinghua University ‡Zhongguancun Laboratory
†EPFL §Department of Computer Science and Technology, Tsinghua University
¶JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

***Abstract*—Virtual devices are a large attack surface of hypervisors. Vulnerabilities in virtual devices may enable attackers to jailbreak hypervisors or even endanger co-located virtual machines. While fuzzing has discovered vulnerabilities in virtual devices across both open-source and closed-source hypervisors, the efficiency of these virtual device fuzzers remains limited because they are unaware of the complex behaviors of virtual devices in general. We present `Truman`, a novel universal fuzzing engine that automatically infers dependencies from open-source OS drivers to construct device behavior models (DBMs) for virtual device fuzzing, regardless of whether target virtual devices are open-source or binaries. The DBM includes inter- and intra-message dependencies and fine-grained state dependency of virtual device messages. Based on the DBM, `Truman` generates and mutates quality seeds that satisfy the dependencies encoded in the DBM. We evaluate the prototype of `Truman` on the latest version of hypervisors. In terms of coverage, `Truman` outperformed start-of-the-art fuzzers for 19/29 QEMU devices and obtained a relative coverage boost of 34% compared to Morphuzz for virtio devices. Additionally, `Truman` discovered 54 new bugs in QEMU, VirtualBox, VMware Workstation Pro, and Parallels, with 6 CVEs assigned.**

I. Introduction

Hypervisors or Virtual Machine Monitors (VMMs) are the core software abstraction layer between Virtual Machines (VMs) and the underlying physical hardware. Hypervisors are essential to prevent attacks against co-located VMs or the underlying infrastructure from denial of service (DoS), information leak, control flow hijacking, and other security threats. Unfortunately, hypervisors, as with other software, are susceptible to vulnerabilities. In 2024, several vulnerabilities inside virtual USB controllers affected VMware products, allowing information leakage or VM escapes [1].

Fuzzing is an efficient technique for discovering vulnerabilities. The community has proposed various fuzzers to improve hypervisor security. Compared to CPU and memory virtualization [2], [3], [4], virtual devices capture greater attention [5], [6], [7], [8], [9], [10], [11], [12], [13], since they correspond to the main attack surface to hypervisors. The input for virtual devices consists of virtual device messages [11], which are commands and data interacting with interfaces of virtual devices, such as Port IO (PIO), Memory-mapped IO (MMIO), and Direct Memory Access (DMA). For example, an MMIO virtual device message may contain a read operation with an access size and a register address. Since virtual devices operate according to the specifications of their physical counterparts, virtual device messages are ordered and structured.

✉Corresponding author: chaoz@tsinghua.edu.cn

Network and Distributed System Security (NDSS) Symposium 2025
24 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240301
www.ndss-symposium.org

Specifically, virtual device messages are also subject to constraints imposed by *inter- and intra-message dependencies* [11]. Inter-message dependencies determine the order of virtual device messages, implying that one message must follow another. Intra-message dependencies encompass two constraints within a single message: constraints on a single field and relationships between fields. While existing research has explored virtual device fuzzing, several challenges remain.

First, a generic and automatic method capable of extracting a broad range of inter- and intra-message dependencies is missing. Existing work [11], [9], [8] has shown that both inter- and intra-message dependencies enhance code exploration and bug discovery capabilities, whereas random fuzzing [14] leads to shallow exploration of the code space. However, these dependencies are manually transcribed from a specification (Nyx-Spec [8]), which is not scalable; semi-automatically extracted from the virtual device source code (ViDeZZo [11]), which is limited by the availability of the source code; distilled from execution traces (MundoFuzz [9]), which is incomplete; or explored simply through a combination of heuristics and randomness (Morphuzz [10] and V-Shuttle [7]), which shows limited efficiency in discovering vulnerabilities in limited time.

> Challenge 1: Efficient hypervisor testing requires an automatic approach to extract a broad range of inter- and intra-message dependencies without manual help, source code of virtual devices, or random exploration.

Second, inter- and intra-message dependencies are insufficient to fuzz "bus-hidden" devices. The interfaces of bus-hidden devices are hidden by the corresponding bus, which prevents direct access to the device. Consequently, the guest must interact with the bus to access the device indirectly. These

devices are commonly found in virtualized environments, such as virtio devices on the virtio bus and USB devices on the USB bus. Despite inter- and intra-message dependencies, fuzzers struggle to explore the code space of bus-hidden devices due to the *lack of state dependency awareness*. A virtual device operates in multiple states throughout its lifecycle, and states are often not explicitly documented in the specifications. For example, a virtio device in the setup state only accepts configuration messages while the device processes data messages in the transmission state. Consequently, the device has different inter- and intra-message dependencies in different states. State dependency refers to the set of inter- and intra-message dependencies that are valid in a specific state and how the device transitions between states.

> Challenge 2: Exploring the code space of bus-hidden devices requires knowledge of the device's state dependency to guide the exploration.

We have two observations to address the above challenges.

First, the source code of an OS driver corresponding to a virtual device *implicitly* encodes inter- and intra-message dependencies. Even though virtual devices of hypervisors are closed-source, we can find open-source OS drivers for them. In practice, developers often implement a virtual device by referring to the corresponding OS driver [15]. Since the virtual device and its corresponding OS driver adhere to the same specification [16], we can infer the inter- and intra-message dependencies from the OS driver and overcome challenges such as analyzing closed-source hypervisors (e.g., VMware Workstation) or hypervisors written in diverse languages (e.g., QEMU is written in C, and VirtualBox is in C++).

Second, a comprehensive device behavior model guides the fuzzer to explore the code space of bus-hidden devices. A device behavior model (DBM) not only captures a virtual device's inter- and intra-message dependencies but also the state dependency. State dependency is essential for bus-hidden devices that are not accessible directly and operate within a two-layer structure: the bus and device layers. The bus layer exposes interfaces for communication, while the device layer interacts with the bus layer for specific operations. With the domain knowledge of the Linux kernel drivers [17], state dependency can be approximated by analyzing both the bus driver and the specific device driver simultaneously. Equipped with the device behavior model, the fuzzer guides the generation and mutation of virtual device messages toward unexplored or under-explored code.

We introduce `Truman`, named after `The Truman Show`, where the protagonist discovers his controlled reality in the virtual world. `Truman` is the first generic and automatic tool that extracts device behavior models (DBMs) from open-source OS drivers to fuzz virtual devices. First, `Truman` statically analyzes the OS driver to extract inter-, intra-, and state dependency graphs, forming a DBM (Section III-A). We develop a flow-, field-, and path-sensitive static analyzer leveraging Linux driver domain knowledge. While the Linux kernel consists of 30M lines of code (LoC) [18], static analysis of individual drivers remains feasible due to their relatively small size. For example, the virtio bus driver has only around 2K LoC [19]. Second, we develop a fuzzing engine that leverages the DBM to generate and mutate virtual device messages that satisfy dependencies (Section III-B). Specifically, `Truman` generates messages at different granularity, ranging from a single message level to the state level, directs the dependency-aware mutation to reduce the number of faulty messages, and executes messages on different hypervisors.

We have implemented a prototype of `Truman`, consisting of a static analyzer to generate the DBM and a fuzzing engine that parses the DBM, generates and mutates virtual device messages that satisfy dependencies. We compared `Truman` against two start-of-the-art (SOTA) approaches (Morphuzz and ViDeZZo) and a naive baseline (AFL++) for coverage evaluation. `Truman` outperforms the three fuzzers on 19/29 QEMU devices after 48-hour fuzzing. Regarding the known bug discovery capability, `Truman` discovered 10 bugs in the last major version of QEMU (v8.0.0) within 24 hours, outperforming Morphuzz and ViDeZZo, which found four bugs. Additionally, `Truman` discovered 54 new bugs across QEMU, VirtualBox, VMware Workstation Pro, and Parallels, with 31 bugs fixed and 6 CVEs assigned.

In summary, we make the following contributions:

- We first identify that the device's state dependency is required for fuzzing bus-hidden devices lacking direct access.
- We propose a generic, automatic, and novel approach to extract inter- and intra-message and state dependencies from open-source OS drivers to guide the fuzzing of virtual devices based on the observation that the OS driver implicitly encodes the specification of the virtual device.
- We evaluated the prototype of `Truman` and the results show that `Truman` surpasses two SOTA fuzzers in 19/29 QEMU devices and discovers 54 new bugs in recent hypervisors, with 6 CVEs assigned and 31 bugs fixed.
- `Truman` is open-sourced at (https://github.com/vul337/Truman).

II. Background and Motivating Example

A. Device Specification and Bus-hidden Devices

Virtual devices and the corresponding OS drivers adhere to a standard specification [16], a document that describes the protocol, ensuring seamless interaction between the guest OS and virtual devices. The OS driver is the software implementation of the device specification and is responsible for translating the requests of the guest OS into virtual device messages. Conversely, the virtual device is the hardware abstraction of the device specification responsible for handling virtual device messages. Although both OS drivers and virtual devices work together to implement the same specification, OS drivers encode high-level knowledge of the usage of virtual devices.

Bus-hidden devices are virtual devices that operate behind bus systems, such as the virtio bus or USB bus. Examples include devices like virtio-sound and USB storage. Specifically,

```c
void virtio_snd_set_config(VirtIODevice *vdev,
    const uint8_t *config) {
  VirtIOSound *s = VIRTIO_SND(vdev);
  const virtio_snd_config *sndconfig =
    (const virtio_snd_config *)config;

  memcpy(&s->snd_conf, sndconfig,
    sizeof(virtio_snd_config));
  le32_to_cpus(&s->snd_conf.jacks);
  le32_to_cpus(&s->snd_conf.streams); // overwrite
  le32_to_cpus(&s->snd_conf.chmaps);
}

void virtio_snd_handle_rx_xfer(VirtIODevice *vdev,
    VirtQueue *vq) {
  msg_sz = iov_to_buf(elem->out_sg, elem->out_num, 0,
    &hdr, sizeof(virtio_snd_pcm_xfer));
  if (msg_sz != sizeof(virtio_snd_pcm_xfer)) {
    goto rx_err;
  }
  stream_id = le32_to_cpu(hdr.stream_id);
  if (stream_id >= vsnd->snd_conf.streams // check
      || !vsnd->pcm->streams[stream_id]) {
    goto rx_err;
  }

  stream = vsnd->pcm->streams[stream_id]; // overflow
}
```

Fig. 1: A heap-buffer-overflow bug in the virtio-sound device.

para-virtualized devices, such as virtio devices, are important in cloud environments for their ability to enhance performance and efficiency by reducing latency and overhead. They are widely used in cloud platforms like AWS [20] and Google Cloud [21] to ensure scalability and high performance. Unlike other virtual devices, bus-hidden devices are not directly accessible through standard interfaces such as Port IO (PIO), Memory-mapped IO (MMIO), and Direct Memory Access (DMA). Bus-hidden devices exhibit a two-layer structure comprising the bus and device layers. The guest OS interacts with interfaces exposed by the bus layer, indirectly controlling the specific device. The two-layer structure introduces additional challenges for fuzzing. Consider the example of a virtio device. The virtio bus layer negotiates features and configures communication channels, while the device layer handles the virtual device's specific functions. Communication between the guest OS and the virtio device occurs through a structure named virtqueue, a shared memory region between the guest OS and the virtual device. In the setup state, the bus layer negotiates the device features and sets up the virtqueue. Once the virtqueue is established, the virtio device moves to the transmission state, where it transfers messages between the guest OS and the virtio device.

B. Motivating Example

We use a heap-buffer-overflow bug (shown in Figure 1) in the virtio-sound device to illustrate the challenges of fuzzing virtual devices. The bug occurs when the guest OS sends a crafted stream ID to the virtio-sound device, overwriting the configuration, and the `virtio_snd_set_config()` does not validate the configuration of streams correctly, leading to a heap-buffer-overflow bug. Specifically, the `virtio_snd_handle_rx_xfer()` transfers the audio buffer provided by the guest to the device. The function parses the virtual device message first and gets the stream ID from the message header (Line 21). Then, the function checks if the stream ID is within the expected range (Line 22). However, if a malicious guest overwrites the configuration of streams by `virtio_snd_set_config()`, the crafted stream ID will be in the "expected" range (Line 22), but the device does not create the stream yet, leading to heap-buffer-overflow (Line 27). The root cause of the bug is that the `virtio_snd_set_config()` does not validate the guest-provided configuration correctly. Triggering the bug requires three types of dependencies.

- *Inter-message Dependency:* To reach out to the buggy function `virtio_snd_handle_rx_xfer()`, the guest must send virtual device messages to the virtio core layer in the correct order during the setup state. Specifically, the guest must first send virtual device messages to set the correct queue size and address in the setup state. The correct order of virtual device messages ensures that the virtio device transfers to the transmission state, allowing proper communication between the guest and the virtual device in the transmission state.
- *Intra-message Dependency:* As Section II-A mentions, the virtio-sound device uses the virtqueue to communicate with the guest. The guest must set up the address of the transmission message to the virtqueue correctly to reach the buggy function, which requires the dependencies between the fields of the transmission message.
- *State Dependency:* The guest must be aware of the state of the virtio device to reach the buggy function. For example, sending a transfer message to the virtio device on the first attempt is useless since the virtio device is in the setup state and does not retrieve the buffer. Therefore, the guest must first send messages that satisfy inter- and intra-message dependencies in the setup state to the virtio device. Then, the guest sends messages that depend on initial messages during the transmission state. The message sequence ensures that the virtio device transitions correctly between states, triggering the buggy function.

In summary, triggering the bug must satisfy all three types of dependencies, which poses a challenge for a fuzzer that is unaware of these dependencies. To address the challenge, we extract these dependencies from the OS driver, as shown in Figure 2. For the inter-message dependency, since the OS driver encodes the messages that set the virtqueue size and virtqueue address (Lines 3 to 5), we extract these messages as the inter-message dependency. Similarly, we extract the intra-message dependency from the virtio driver, which initializes the buffer to set the correct address and length (Lines 10 to 12). The OS driver of the virtio device typically performs operations that set up the device in the probe function. Therefore, we collect and group these operations in the probe function as the dependency of the setup state (Lines 19 to 21). The extraction ensures that the fuzzer handles the dependencies appropriately, allowing the fuzzer to trigger the bug efficiently.

```
1  int vp_active_vq(struct virtqueue *vq, u16 msix_vec) {
2    // Inter-message dependency
3    vp_modern_set_queue_size(mdev, index, vring_size);
4    vp_modern_queue_address(mdev, index, desc_addr,
5                            avail_addr, used_addr);
6  }
7
8  static unsigned int virtqueue_add_desc_split(struct
       vring_desc *desc, unsigned int i, dma_addr_t addr,
       unsigned int len, u16 flags) {
9    // Intra-message dependency
10   desc[i].flags = flags;
11   desc[i].addr = addr;
12   desc[i].len = len;
13
14   return desc[i].next;
15 }
16
17 static int virtio_dev_probe(struct device *_d) {
18   // State dependency
19   virtio_add_status(dev, VIRTIO_CONFIG_S_DRIVER);
20   virtio_add_status(dev, VIRTIO_CONFIG_S_FEATURES_OK);
21   virtio_add_status(dev, VIRTIO_CONFIG_S_DRIVER_OK);
22 }
```

Fig. 2: The virtio OS driver encodes state, inter-message, and intra-message dependency.

C. Threat Model

We assume that the attacker has control over the guest OS in the cloud environment, which is consistent with related work in hypervisor security [5], [6], [7], [8], [9], [10], [11], [12], [13]. The attacker crafts arbitrary malicious PIO, MMIO, and DMA virtual device messages and sends them to virtual devices. Next, by leveraging vulnerabilities in virtual devices, the attacker can leak information, prevent the virtual devices from working, and even escape from the virtual machine, potentially compromising the entire cloud environment.

III. Design

`Truman` extracts device behavior models from open-source Linux kernel drivers to assist in virtual device fuzzing. Linux supports various peripherals, making it a suitable platform for extracting device behavior models [22]. Our key insight is that a virtual device and its corresponding OS driver are governed by a common specification [16], ensuring they function together. Therefore, we can leverage the knowledge implicitly encoded in OS drivers to fuzz virtual devices. Open-source OS drivers are promising candidates for inferring knowledge about virtual devices based on the following benefits. First, even if a virtual device is proprietary and closed-source, the OS drivers are generally open-source. For example, we found that every virtual device in VMware Workstation Pro has the corresponding open-source driver, even self-developed virtual devices such as Virtual Machine Communication Interface (VMCI) [23]. Second, the interface of virtual devices is generally unified regardless of how differently virtual devices are implemented across different hypervisors. Analyzing a driver provides insights applicable to various hypervisors. This makes it possible to develop a generalized framework that works across different hypervisors. Third, OS drivers serve as consumers of virtual devices and clearly show how to interact with devices. Therefore, device behavior models are more easily extracted from OS drivers than virtual devices based on the observation.

Figure 3 depicts `Truman`'s two main steps. First, `Truman` models the behavior of a virtual device by three types of dependencies: inter-, intra-message, and state dependencies, and extracts them from OS drivers via flow-sensitive, path-sensitive, and field-sensitive static analysis. `Truman` generates the device behavior models consisting of the above three kinds of dependencies as the output of the first step (Section III-A). Second, given the device behavior models, `Truman` assists in fuzzing via dependency-aware generation and mutation of seeds (Section III-B). Specifically, `Truman` develops a fuzzing engine that generates seeds at different granularity based on device behavior models to explore the code space of virtual devices effectively, especially for bus-hidden devices. Additionally, `Truman` mutates messages at the message level, sequence level, and state level guided by the device behavior models. The inter-message dependency keeps the order of messages, the intra-message dependency ensures the constraints of the fields, and state dependency restricts the dependencies in different states. Finally, `Truman` catches bugs with AddressSanitizer (ASAN) [24] for open-source virtual devices and with crash signals for closed-source targets.

A. Constructing Device Behavior Model

`Truman` extracts device behavior models from the open-source OS drivers. Device behavior models are crucial for improving the efficiency and effectiveness of fuzzing virtual devices. These models help accurately generate and mutate seeds by understanding the message dependencies and state transitions within the device, thereby reducing the time to cover the same code as random fuzzing and increasing the probability of discovering more vulnerabilities.

A device behavior model describes a virtual device using inter-, intra-message, and state dependencies. To extract these dependencies, `Truman` performs the following steps. First, to extract inter-message dependencies, `Truman` traces operations by traversing the OS driver using a new flow-sensitive, path-sensitive, and inter-procedural static program analyzer (Section III-A1). Second, `Truman` extracts two types of intra-message dependencies (Section III-A2). For constraints affecting a field in a message (e.g., the lower 8 bits of a 32-bit field must be equal to a constant), `Truman` builds an expression tree to represent the construction of the field. Additionally, for the dependency between fields of a message (e.g., the pointer relationship between DMA buffers), `Truman` leverages static taint analysis to infer the relationship between fields. Third, `Truman` utilizes the two-layer structure of bus-hidden devices to extract state dependency (Section III-A3). Specifically, `Truman` divides the state according to different types of functions in the bus driver first and further refines the state based on the message type extracted from the specific device driver. Combining the above three types of dependencies, `Truman` constructs the device behavior model for the virtual device. Figure 4 shows an example.

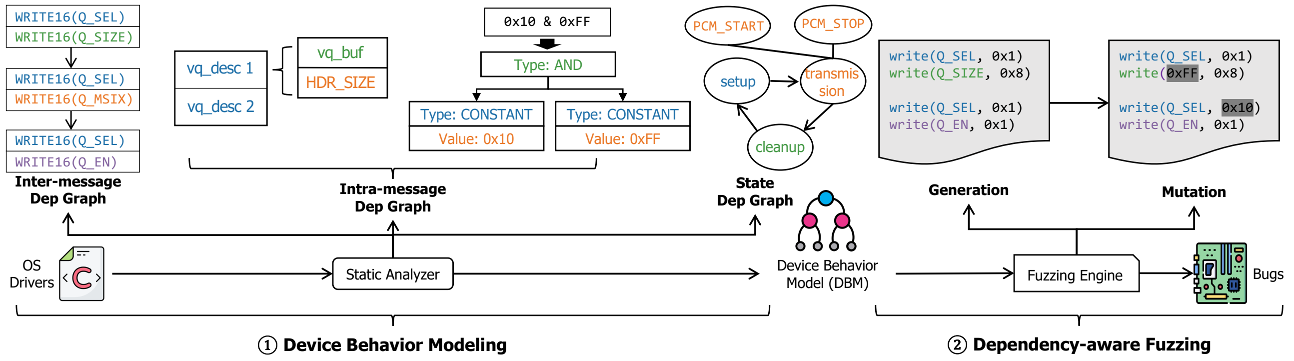
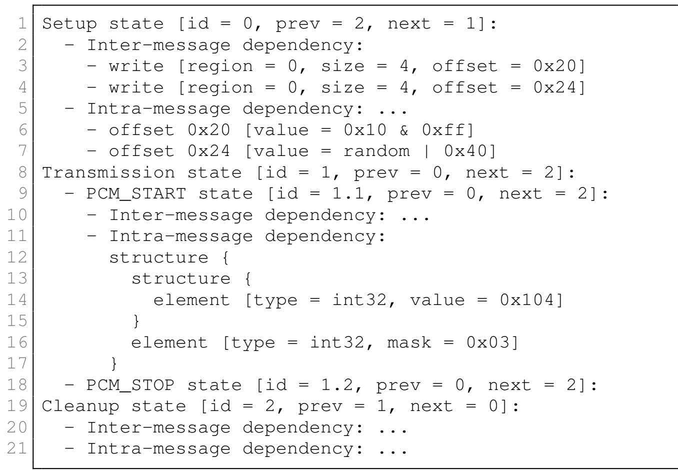


Fig. 3: Overview of information that Truman extracts from OS drivers and how Truman assists in virtual device fuzzing.



```
1  Setup state [id = 0, prev = 2, next = 1]:
2    - Inter-message dependency:
3      - write [region = 0, size = 4, offset = 0x20]
4      - write [region = 0, size = 4, offset = 0x24]
5    - Intra-message dependency: ...
6      - offset 0x20 [value = 0x10 & 0xff]
7      - offset 0x24 [value = random | 0x40]
8  Transmission state [id = 1, prev = 0, next = 2]:
9    - PCM_START state [id = 1.1, prev = 0, next = 2]:
10     - Inter-message dependency: ...
11     - Intra-message dependency:
12       structure {
13         structure {
14           element [type = int32, value = 0x104]
15         }
16         element [type = int32, mask = 0x03]
17       }
18   - PCM_STOP state [id = 1.2, prev = 0, next = 2]:
19 Cleanup state [id = 2, prev = 1, next = 0]:
20   - Inter-message dependency: ...
21   - Intra-message dependency: ...
```

Fig. 4: An example of the device behavior model.

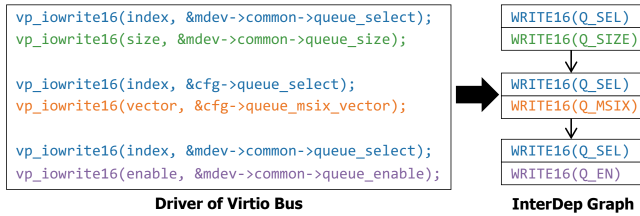


Fig. 5: Example of the inter-message dependency graph extracted from the virtio driver.

1) Extracting Inter-message Dependency: Inter-message dependencies encode the order of virtual device messages, which is crucial for virtual devices to function correctly. Figure 5 shows an example of the inter-message dependency graph extracted by Truman from the virtio bus driver. For each virtqueue of the device, the driver first sets the size and the interrupt vector of the virtqueue, then enables the virtqueue. The fuzzer must follow this specific order to set up the device correctly. Otherwise, the virtio device will fail and fall into an error state.

Previous virtual device fuzzers [10] use general fuzzing engines (e.g., libFuzzer [25]) to learn these dependencies, while others encode this knowledge using a grammar [8] or initial seeds [7]. However, these existing methods require manual work, which is incomplete or inaccurate. Since OS driver implementation reflects the virtual device's behaviors, Truman automatically extracts meaningful inter-message dependencies from OS drivers. Intuitively, Truman identifies the order of messages by traversing the control flow graph (CFG) and call graph (CG) of the OS driver via inter-procedure static analysis. Specifically, Truman records the (W/R, size, offset) tuple of each message and the order of the messages. Eventually, Truman employs a three-step approach to build an inter-message dependency graph for each device from its corresponding OS driver.

First, Truman collects kernel functions that perform operations and maps these functions to virtual device messages. To achieve this, Truman gathers a list of standard functions for executing operations from the official Linux documentation (e.g., iowrite16() for an IO operation). These functions show the direction of the operation (read or write) and the size of the operation (e.g., 16-bit or 32-bit). Additionally, Truman defines heuristics to identify customized wrapper functions defined by a specific driver to complete the function list. For example, the virtio driver defines vp_iowrite16(), which wraps iowrite16(). Both vp_iowrite16() and iowrite16() represent a two-byte MMIO write. A list of recognized functions is shown in the Appendix Table V.

Second, based on the list of standard functions, Truman conducts flow-sensitive analysis to record all function call sites starting from entry functions [26] of the driver. The entry function is usually defined as a member of the xxx_ops structure according to the Linux kernel's documentation. Flow sensitivity is necessary to keep the virtual device messages in the same order as the interactions between drivers and devices.

Third, since the driver operates under different conditions, Truman performs path-sensitive analysis to record all possible paths of the driver. The path-sensitive analysis guarantees that Truman records all operations in all paths, significantly enriching the inter-message dependency. Truman is not impacted by path explosion since the code lines of the driver are limited (the virtio bus driver has only 2K LoC). Thus, the number of paths remains manageable. Additionally, Truman

Algorithm 1 Constructing Expression Tree for an IO Value

Input: IO value
Output: Expression tree root

1: root ← new ExpressionTree()
2: **if** value is constant **then**
3: root.type ← CONSTANT
4: root.value ← GetConstantValue(value)
5: **else**
6: operation ← GetOperation(value)
7: type = GetOperationType(operation)
8: **if** type is bitwise (OR, AND, XOR, SHIFT_LEFT, SHIFT_RIGHT) or conditional (PHI, SELECT) **then**
9: root.type ← type
10: **for** operand **in** GetOperands(operation) **do**
11: child ← ConstructExpressionTree(operand)
12: root.addChild(child)
13: **end for**
14: **else**
15: root.type ← RANDOM
16: **end if**
17: **end if**
18: **return** root

ignores the condition of branches when performing the path-sensitive analysis, further constraining the complexity of the analysis. In this way, `Truman` creates a directed graph of inter-message dependency where each node defines a message, and the edge defines the order of the message.

2) Extracting Intra-message Dependencies: Intra-message dependencies contain two types of constraints. The first is the constraints of single message fields, such as the requirement that the lower 8 bits of a 32-bit field be equal to a constant. The second is the dependency between multiple fields of a message, such as a pointer of a DMA buffer must point to another. `Truman` develops different approaches to extract these two types of dependencies.

Constraints Affecting a Single Field: Virtual device messages are usually restricted by constraints. Take the IO message as an example. An IO message follows the pattern `(address, size[, value])`. A 32-bit value of an IO message results from a bitwise OR operation of two 16-bit values with different meanings. `Truman` not only records the order of the operation but also the value of each operation during constructing the inter-message dependency graph. `Truman` conducts a backward dataflow analysis to recover the constraints of the value and generates an expression tree to represent the dependencies, as shown in Algorithm 1. The algorithm also holds for DMA messages, where a field in a DMA buffer has constraints. `Truman` utilizes the expression tree to generate and mutate the value of an operation with dependencies in the fuzzing stage. Figure 6 depicts the expression tree of a value field of an MMIO write message in the `sdhci` driver. The `sdhci` driver sends a command to the SD card, and the value is constructed by the bitwise OR operation of the command and flags. The expression tree describes the

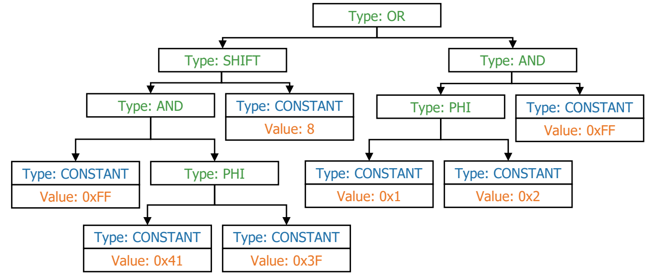


Fig. 6: An example of the expression tree for the IO value `MAKE_CMD(command, flag)` extracted from the `sdhci` device driver, where the macro `MAKE_CMD(c, f)` is defined as `(((c & 0xFF) << 8) | (f & 0xFF))`.

dependency of the values, and `Truman` generates the value of the IO message based on the expression tree.

Dependency Between Fields: A virtual device message contains multiple fields, and fields have dependencies between each other. The most important dependency is the pointer relationship, which is hard to satisfy in random fuzzing. For instance, the DMA buffer may have a pointer to another DMA buffer. Overall, `Truman` employs a two-step approach to extract the dependency between fields within a message. First, similar to the construction of inter-message dependency graph, `Truman` collects standard functions that allocate DMA buffers from Linux kernel (shown in the Appendix Table VI) and identifies the call sites of these functions in entry functions of the driver. Second, `Truman` uncovers the relationship between DMA buffers via a static taint analysis. To ensure the virtual devices work as expected, the correct pointer relationship between DMA buffers must be set. For example, in the Linux kernel, the primary DMA buffers are often coherent (ensuring that the device and the processor obtain the same data), which is used to synchronize the command of devices, and the nested DMA buffers are usually streaming buffers (commonly used for data transfers between device and memory for performance), which points to the primary DMA buffer. Specifically, `Truman` discovers the field dependencies via the static taint analysis defined as follows.

- *Taint Sources.* `Truman` marks the previously identified pointers of DMA buffers and their aliases with tainted flags. Then `Truman` leverages inter-procedural taint propagation, spreading taint sources to other pointers.
- *Taint Sinks.* `Truman` checks if a taint source is referenced by another DMA buffer, and thus `Truman` regards the address of the other DMA buffer as a sink. The sink pointer stores the address of the taint source, creating a nested DMA buffer. If `Truman` finds such a sink pointer, `Truman` will record the pointer between the taint source and the sink.
- *Propagation Rules.* `Truman` utilizes a summary-based taint flow construction approach to efficiently propagate the inter-procedural taint flows as shown in SUTURE [27].

In the intra-message dependency graph of dependencies between fields within a message, a node represents a field

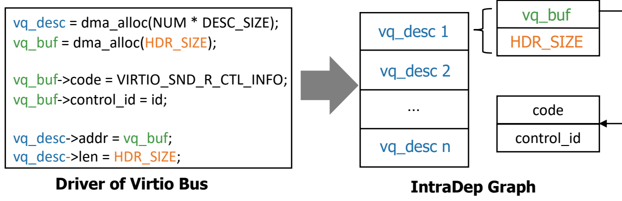


Fig. 7: Example of the intra-message dependency graph extracted from the virtio bus driver.

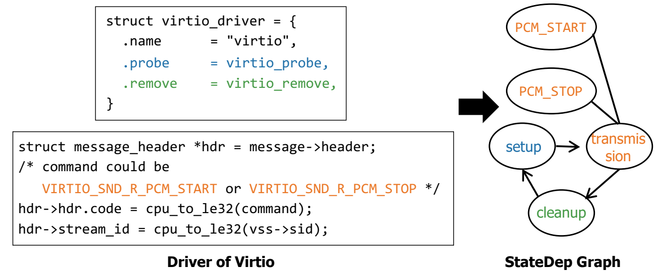


Fig. 8: Example of the state dependency graph extracted from the virtio driver.

in the message, and an edge between nodes represents a pointer from one field to another. Figure 7 shows an example of constructed intra-message dependency graph for the virtio sound device. The virtio sound driver first allocates a buffer for the `vq_desc` and then for `vq_buf`. Next, the driver writes the address of `vq_buf` to a field in `vq_desc`. `Truman` extracts the relationship between the `vq_desc` and `vq_buf` from the driver.

3) Extracting State Dependency: State dependency defines the set of inter- and intra-message dependencies that must be satisfied within a specific virtual device state and the transition of different states. State dependency is crucial for virtual device fuzzing, especially for bus-hidden devices (e.g., virtio devices). Each state has a different set of dependencies that must be satisfied. For example, the virtio device has multiple states, including setup, transmission, and error. In the setup state, the guest must set up the properties of the virtqueue, while in the transmission state, the driver must fill the message to the virtqueue. If a guest sends a message to the virtqueue first without setting up the virtqueue, the virtio device will report the error.

In previous research, FuzzUSB [28] identifies states of USB gadgets based on the input point in a coarse-grained manner, which cannot distinguish different states with the same input point in virtual devices (e.g., `pci_dma_read()` and `dma_memory_read()`). Other research divides the states based on state variables, which cannot be applied to bus-hidden devices because the devices have two layers of structure, and the solution cannot find the relationship between variables in the two layers. The above indicators for states are not suitable for virtual devices. We propose a new automatic approach to extract state dependency at fine-grained based on observing a two-layer structure of bus-hidden devices.

First, `Truman` divides the state based on different types of functions in the bus driver. OS drivers usually abstract the functionality of devices under the same bus by defining a set of standard functions. Take the virtio bus driver as an example. The virtio bus driver typically implements functions such as `probe()`, `resume()`, and `remove()`. As the name implies, the `probe()` function is used to set up the device, and the `resume()` function is used to resume the device. Therefore, `Truman` divides the state based on the special functions defined in the driver. Specifically, `Truman` groups the inter- and intra-message dependencies extracted in `probe()` as the setup state and the dependencies in the `remove()` as the cleanup state. The functions also imply transitions between different states, such as the `probe()` function, usually called before the `resume()` function, indicating the transition from the setup state to the resume state. Additionally, `Truman` groups the dependencies defined in entry functions for user-space the transmission state. `Truman` manually maps the states and the functions as shown in Appendix Table VII.

Second, `Truman` refines the state based on the message type extracted from the specific device driver. The transmission state can be divided into more fine-grained states depending on the message type requested. `Truman` conducts a backward analysis from the point of adding the message to the buffer. The analysis extracts the structure of the message and the possible value for the fields. Messages for virtio devices usually include the message type in the header, `Truman` extracts the message type based on heuristics, that is, variables of the message type are usually named as `type` or `code`. Then, `Truman` links the message type to the corresponding state. For example, `Truman` extracts the states *pcm_start* and *pcm_stop* of the virtio-sound device based on the request type of the message (`VIRTIO_SND_R_PCM_START` and `VIRTIO_SND_R_PCM_STOP`), as shown in Figure 8.

B. Performing Dependency-aware Fuzzing

`Truman` develops a fuzzing engine that leverages the device behavior models to generate, mutate, and execute virtual device messages with dependencies to explore the code space of the virtual device effectively. The fuzzing engine includes three main components: generator, mutator, and executor. The generator and mutator are independent of the specific hypervisor implementation. The analyst only needs to implement a specific executor to support a new hypervisor.

1) Generator: Based on the device behavior models extracted from OS drivers, `Truman` generates seeds at four levels of granularity, including the message level, basic block level, function level, and state level. The size of seeds increases from the message level to the state level. Each level of granularity is suitable for different scenarios; for example, the state-level seed is suitable for transitioning between different states. `Truman` employs a probability distribution to determine the level of seeds to generate.

Message Level: The message level seed is an IO message or a DMA message. For a single IO message with a tuple as

(address, size[, value]), the address and size fields are randomly generated, and value to be written is generated based on the intra-message dependency graph (expression tree). For a DMA message, Truman generates a DMA buffer and fills the buffer based on constraints of a field (expression tree) and the pointer relationship in the intra-message dependency graph.

Basic Block Level: At the basic block level, Truman generates seeds by creating each message within a basic block. This ensures that the generated messages align with the control flow within that specific basic block.

Function Level: The inter-message dependency graph extracted by Truman also records the inter-procedural control flow graph (ICFG) of the driver, Truman generates the function-level seed by randomly choosing a path in the ICFG and generating messages along the path. This approach ensures that the generated seeds reflect the sequence and dependencies of basic blocks within a function.

State Level: Truman generates a state-level seed by traversing the state dependency graph and mapping the state to the corresponding virtual device messages. A state-level seed encompasses messages in multiple functions, ensuring that the generated seed represents a more comprehensive semantic of the virtual device.

2) Mutator: The device behavior model not only facilitates seed generation but also enables seed mutation. Importantly, Truman is capable of incorporating dependency-aware seed mutation directly into the fuzzing engine. This approach leverages the understanding of inter- and intra-message dependencies and state dependency mapped out in the device behavior model. Specifically, Truman extends the message-level and sequence-level mutators defined in ViDeZZo [11] and introduces new state-level mutators.

Message-level mutators: These mutators focus on modifying the content of individual messages, including operations such as ChangeAddr, ChangeValue, etc. Truman enhances message-level mutators by incorporating the intra-message dependency in the device behavior model. For example, the ChangeValue mutator, which changes the write value of an IO message, previously generated a new value randomly. In contrast, Truman re-evaluates the expression tree of the value and generates a value that satisfies the dependency. For the DMA message, Truman maintains the structure of the DMA buffer but changes the value of a field in the buffer based on the intra-message dependency graph.

Sequence-level mutators: These mutators explore the combination of messages, including InsertMessages, RemoveMessages, ShuffleMessages, etc. Truman enhances the sequence-level mutators by incorporating the inter-message dependency from the device behavior model. For instance, the InsertMessages mutator inserts a new message into the message sequence, and Truman first chooses a message and follows the order in the inter-message dependency graph to generate new message sequences.

State-level mutators: These mutators restrict the message and sequence level mutators to the current state of the virtual device. By leveraging the state dependency in the device behavior model, Truman ensures that mutated seeds are valid within dependencies of the current state. For example, the value of an MMIO write message with the same offset may have different constraints in different states.

```
int init_device_behavior_model(const char* path) {
  DBM dbm = parse_dbm_from_file(path);
  return 0;
}

void add_interface(Type type, int addr, int size) {
  if (interface_exists(type, addr, size)) {
    return;
  }
  interfaces[interface_count].type = type;
  interfaces[interface_count].addr = addr;
  interfaces[interface_count].size = size;
  interface_count++;
}

int get_messages(Messages* messages) {
  Sequence generated_seq = generator();
  for (int i = 0; i < generated_seq.size; ++i) {
    message_t msg = generated_seq[i];
    add_message(messages, msg);
  }
  return generated_seq.size;
}
```

Fig. 9: APIs that are provided by the fuzzing engine.

3) Executor: The executor executes the messages generated and mutated by the fuzzing engine, so it depends on the specific hypervisor. The fuzzing engine in Truman provides APIs for the hypervisor-specific executor, as shown in Figure 9. These include init_device_behavior_model() to initialize the device behavior model, add_interface() to add a new interface if it does not already exist, and get_messages() to generate and add a sequence of messages to the provided messages object.

IV. Implementation

Truman extracts the device behavior models from Linux drivers, generates and mutates quality seeds to fuzz the virtual devices, and finally discovers bugs in different hypervisors. To build device behavior models, Truman first compiles Linux drivers to bitcode with Clang 13.0.0 (required by SVF), then leverages SVF [29] based on LLVM 13.0.0 [30] to conduct the static program analysis, and uses SUTURE [27] to perform the static taint analysis. After constructing the device behavior models, Truman utilizes the fuzzing engine to generate and mutate seeds and performs dependency-aware fuzzing for different virtual devices. Truman consists of around 3,000 lines of C++ for static analysis, 2,500 lines of C++ for the fuzzing engine, and 2,000 lines of Python scripts for setting up the environment, gluing the components of Truman, and analyzing the results.

A. Preparation of Static Analysis

1) Identify the Corresponding OS Driver of the Virtual Device: Truman pairs a virtual device under test and its corresponding OS driver by checking if they have the same

device identification. A driver typically defines an array of device IDs of supporting devices. The format of the structure of the device ID and the semantics for identifying device IDs are bus-specific. For a PCI device, the vendor ID field and the device ID field are used to match the driver with the device. With the pair of vendor ID and device ID, `Truman` can locate the exact OS driver corresponding to a virtual device, laying the foundation to extract the device behavior model from the OS driver and then fuzz the virtual device.

`Truman` first compiles the corresponding Linux driver with Clang and generates a bitcode file for further analysis (one bitcode file per driver). Specifically, we develop a Clang wrapper that finds the necessary files for the Linux driver, emits LLVM bitcode from these files, and merges all necessary bitcode files of a driver. Next, we extracted the device IDs and the driver's IDs. For a specific virtual device under the PCI bus, `Truman` gets the device ID after probing the device during running. As for the device driver side, the device IDs are always embedded in a special structure named `id_table` according to the document, `Truman` extracts the corresponding device IDs by static analysis. Generally, the driver supports multiple chipsets of hardware; thus, `Truman` searches the IDs in the driver that the driver supports to match the exact hardware. Finally, `Truman` records the pair of the virtual device and the corresponding driver to the database.

2) Locate Driver Entry Functions: `Truman` needs to identify entry functions as the start point of static analysis to extract the device behavior model from the OS driver. Entry functions are typically of two types: functional entry and interrupt handlers. `Truman` uses heuristics to locate the two types of entry functions: functional entry functions and interrupt handlers. First, we develop a set of regular expressions to match the entry functions of the drivers (usually named as `xxx_ops`). Similarly, `Truman` processes interrupt handlers of the driver by identifying the registration function from the documentation. According to the documentation, interrupt handlers are always registered using the function `request_irq()` and other similar functions. `Truman` recognizes the argument of register functions as interrupt handlers.

B. Static Analysis to Construct DBMs

1) Alias Analysis: Since addresses of PIO/MMIO regions and DMA buffers are pointers, it is important to precisely identify their aliases to extract the inter-message dependency graph and intra-message dependency graph. However, a general alias analysis cannot be aware of the relationship since these addresses are global variables, which are not be passed as function arguments, and even worse, driver entry functions often are not invoked by each other, which breaks the inter-procedural analysis based on the call graph. The general alias analysis fails to build a connection between the two regions and thus cannot determine if the regions are the same.

To address the above challenges, `Truman` develops a multi-layer object unfolding (MLOU) alias analysis. Based on the fact that these addresses are usually stored in a top-level global structure for a Linux driver, MLOU alias analysis calculates a pointer's offset within the global structure and regards a pointer as an alias of another pointer if two pointers' offsets are the same. The analysis starts from a PIO/MMIO region or DMA buffer address (pointer), iterates multiple structural layers, and calculates the pointer's offset in the top-level global structure. `Truman` conducts a few optimizations during the visit to multiple structural layers.

```
1 MMIO_WRITE(region_id, address, size, value)
2 MMIO_READ(region_id, address, size)
3 PIO_WRITE(region_id, address, size, value)
4 PIO_READ(region_id, address, size)
5 DMA_TRANSMISSION(data, data_length)
```

Fig. 10: Message types supported by the executor.

2) Device Register Identification: When building the inter-message dependency graph, precisely identifying the IO region the IO function reads from and writes to is crucial since many virtual devices have multiple regions. Specifically, for each IO call instruction, `Truman` first extracts the address operand of the instruction. Then, it separates the offset from the base address by analyzing the `GetElementPtr` instructions. Next, `Truman` looks for the corresponding alias of the base address in the IO region table. If not found, `Truman` applies the MLOU alias analysis mentioned above to decide which IO region the base address belongs to and then updates the IO region table. Finally, `Truman` records the constant offset.

C. The Fuzzing Engine

We develop a fuzzing engine to parse the device behavior model and generate or mutate dependency-aware seeds for the virtual device. The goal of the fuzzing engine is to minimize the effort of porting the fuzzer to different hypervisors. The key design of the engine is to hide the parsing process of the device behavior model and the generation or mutation of the seeds. To support a new hypervisor, developers only need to implement a hypervisor-specific executor and use the APIs provided by the fuzzing engine. The executor is message-driven and must dispatch messages shown in Figure 10. Specifically, we implement `Truman` based on two executors: QTest for QEMU and Hyper-Cube OS for other hypervisors. The fuzzing engine uses protocol buffers [31] to serialize and deserialize the seeds.

V. Evaluation

A. Evaluation Setup

To compare the performance of `Truman` with existing hypervisor fuzzers fairly, we conducted the evaluation with the following generic setup. The specific setup will be described in the corresponding subsection.

Fuzzers: We selected SOTA virtual device fuzzers Morphuzz (version 9.0.0) and ViDeZZo (commit adc2a22) as baseline. We also adapted a generic fuzzer AFL++ (commit 598a3c6b) to fuzz without any domain-specific knowledge.

Coverage and Sanitizers: To collect code coverage of QEMU, we instrument QEMU using Clang's source code

Virtual Device	Linux Driver (under `drivers`)	Inter-Dep Graph			Intra-Dep Graph			State-Dep Graph			Time(s)
		v5.15	v6.1	v6.6	v5.15	v6.1	v6.6	v5.15	v6.1	v6.6	
VirtualBox											
lsilogicsas	`scsi/mpt3sas/mpt3sas.c`	61	61	61	49/7	49/6	49/6	5	5	5	2.59
virtio-scsi	`scsi/virtio_scsi.c`	92	77	75	33/6	31/6	31/6	10	10	10	0.32
VMware											
svga	`gpu/drm/vmwgfx/vmwgfx.c`	139	153	140	69/1	79/1	71/1	6	6	6	5.01
pvscsi	`scsi/vmw_pvscsi.c`	15	15	15	9/5	9/5	9/5	3	3	3	0.11
QEMU Audio											
ac97	`sound/pci/intel8x0.c`	133	133	133	58/1	58/1	58/1	3	3	3	0.14
intel-hda	`sound/pci/hda/intel-hda.c`	32	32	32	10/0	10/0	10/0	12	12	12	0.15
QEMU Storage											
am53c974	`scsi/am53c974.c`	12	12	12	10/1	10/1	10/1	2	2	2	0.07
ahci	`ata/ahci.c`	35	35	35	13/1	13/1	13/1	7	7	7	0.11
nvme	`NVMe/host/NVMe.c`	28	28	28	9/19	9/19	9/19	7	7	7	0.18
QEMU Network											
igb	`net/ethernet/intel/igb/*`	874	898	873	192/6	202/7	188/7	8	8	8	4.6
e1000	`net/ethernet/intel/e1000/*`	563	563	562	172/10	172/10	171/10	5	5	5	1.28
virtio-net	`net/virtio_net.c`	92	77	75	33/11	31/11	31/11	15	15	15	0.32
QEMU Display											
ati	`video/fbdev/aty/aty128fb.c`	131	131	131	55/0	55/0	55/0	6	6	6	0.17
sm501	`mfd/sm501.c`	46	46	46	14/0	14/0	14/0	2	2	2	0.15
virtio-gpu	`gpu/drm/virtio/*`	92	77	75	33/14	31/14	31/14	18	18	18	0.32
QEMU USB											
ehci	`usb/host/ehci*`	174	180	180	33/6	36/6	36/6	11	11	11	0.77
xhci	`usb/host/xhci*`	272	272	283	80/6	80/6	85/6	11	11	12	2.35

TABLE I: The corresponding OS drivers for virtual devices, the statistics of extracted device behavior models, and the average analysis time for different long-term kernel versions. Numbers in the table are the count of nodes in the dependency graph.

coverage profiling [32]. We capture a snapshot of the corpus of fuzzing every 10 minutes and replay the corpus afterward to observe the trends in coverage increase. For bug detection, we also instrument QEMU with ASAN to capture memory corruption. We rely on VM crashes to identify bugs for closed-source hypervisors such as VMware Workstation Pro and Parallels Desktop.

Fuzzing Resources: We evaluated `Truman` on a server with 16 physical cores of Xeon Gold 5218 and 64GB RAM installed with Ubuntu 22.04. The server was exclusively used for fuzzing to avoid interference from other processes. We fuzzed every virtual device on one CPU core and bound the fuzzing process to the CPU core for 48 hours to perform a coverage comparison. For other evaluations, we limited the fuzzing duration to 24 hours. We repeated the evaluation five times to ensure the stability of the results.

B. Device Behavior Modeling

To evaluate the effectiveness of `Truman` in extracting device behavior models from OS drivers, we conducted the static analysis on the OS drivers of different long-term maintained versions (v6.6, v6.1, and v5.15) of the Linux kernel. To ensure the diversity of hypervisors, we evaluated both open-source hypervisors (VirtualBox and QEMU) and closed-source hypervisors (VMware Workstation Pro). We also chose representative virtual devices from different categories: storage, network, display, and USB. Table I shows the statistics of the device behavior models extracted by `Truman` from different Linux kernel versions. We illustrate the results as follows:

The Pair of Virtual Devices and OS Drivers: `Truman` successfully found the corresponding OS drivers for all 17 virtual devices, including virtual devices in VMware Workstation Pro, which are closed-source. The corresponding vendor ID and device ID of devices are shown in Table VIII in the Appendix. Since the vendor ID and device ID are used to identify the driver, they remain the same since the drivers are created and the IDs are not modified in different OS versions.

Inter-Message Dependency Graph: The number of nodes in the inter-message dependency graph indicates the number of unique operations the device performs. The results differ across different versions of the Linux kernel because of the updates or cleanup of the OS drivers. The `igb` driver in Linux kernel v6.1 has the highest number of nodes in the inter-message dependency graph. In contrast, the `am53c974` driver has the fewest nodes, indicating different characteristics of various categories of devices. For the network devices and USB devices, the number of nodes in the inter-message dependency graph is relatively high, indicating that the devices have more complex interactions with the guests. From the aspect of different versions of the Linux kernel, the results are stable, 11 out of 17 devices remain results across different kernel versions. Other devices only have a slight change, which is expected because the device driver may have some updates or fixes across different versions. The results show that `Truman` is suitable for different Linux kernel versions.

Intra-Message Dependency Graph: The values in the intra-message dependency graph represent the number of two types of intra-message dependencies: constraints affecting a field and relationships between fields. The results for the constraints affecting a field show similar patterns to the inter-message dependency graph since most of these constraints are related to the IO messages. The results indicate that the number of relationships between fields is relatively small since these

Device	AFL++	p	$\hat{A}_{12}$	Morphuzz	p	$\hat{A}_{12}$	ViDeZZo	p	$\hat{A}_{12}$	`Truman`
Block										
ahci	45.75%	0.012	1	58.88%	0.012	0	**64.91%**	0.012	0	50.25%
fdc	33.73%	0.012	1	39.09%	1	0.5	38.06%	0.007	1	**39.14%**
nvme	10.68%	0.008	1	30.56%	0.056	0.88	11.43%	0.008	1	**34.25%**
sdhci	54.29%	0.008	1	63.89%	0.016	0.96	70.53%	0.753	0.58	**71.13%**
virtio-blk	17.34% (21.96%)	0.009	1	30.48% (32.40%)	0.011	1	39.53% (31.52%)	0.650	0.4	**41.11%** (29.84%)
virtio-scsi	11.28% (21.44%)	0.008	1	37.04% (32.38%)	0.008	1	13.81% (27.36%)	0.010	1	**57.97%** (29.70%)
Audio										
ac97	83.02%	0.007	1	**95.33%**	1	0.5	**95.33%**	1	0.5	**95.33%**
cs4231a	84.86%	0.011	1	91.57%	0.699	0.42	**91.71%**	0.403	0.34	91.43%
es1370	64.17%	0.007	1	67.74%	0.007	1	**69.13%**	1	0.5	**69.13%**
sb16	77.14%	0.010	1	85.19%	0.016	0.96	**86.43%**	0.006	0	85.66%
virtio-sound	24.22% (18.56%)	0.011	1	50.42% (29.25%)	0.139	0.8	20.18% (17.34%)	0.011	1	**57.24%** (21.12%)
Display										
ati	81.78%	0.046	0.9	82.41%	0.452	0.66	77.71%	0.012	1	**82.57%**
cirrus	86.11%	0.011	1	87.58%	0.011	1	**92.87%**	0.011	0	90.30%
virtio-gpu	4.51% (15.14%)	0.011	1	26.65% (26.62%)	0.672	0.4	3.05% (14.43%)	0.007	1	**28.39%** (16.89%)
Network										
eepro100	79.44%	0.548	0.64	84.49%	0.346	0.3	**88.20%**	0.010	0	82.41%
e1000	34.51%	0.012	1	63.31%	0.046	0.9	38.59%	0.008	1	**68.10%**
e1000e	58.36%	0.600	0.38	**64.38%**	0.691	0.4	50.49%	0.056	0.88	59.63%
igb	31.45%	0.203	0.76	34.07%	0.833	0.44	29.09%	0.020	0.96	**35.42%**
pcnet	52.09%	0.008	1	**94.36%**	0.116	0.18	64.21%	0.012	1	92.27%
rtl8139	60.18%	0.008	1	76.88%	0.059	0.88	75.80%	0.008	1	**83.11%**
vmxnet3	18.91%	0.011	1	23.97%	0.012	1	32.96%	0.007	1	**53.41%**
virtio-net	12.96% (20.41%)	0.012	1	18.12% (33.42%)	0.012	1	24.99% (35.12%)	1	0.48	**25.65%** (31.87%)
USB										
ehci	52.97%	0.011	0.28	75.02%	1	0.52	**75.81%**	0.290	0.28	75.20%
ohci	54.91%	0.008	1	76.23%	1	0.5	5.05%	0.008	1	**77.12%**
xhci	29.56%	0.011	1	44.32%	0.526	0.64	29.78%	0.011	1	**47.94%**
Misc										
virtio-balloon	12.24% (16.67%)	0.009	1	20.19% (25.85%)	0.009	1	16.63% (23.23%)	0.010	1	**30.78%** (24.66%)
virtio-crypto	8.16% (19.56%)	0.011	1	31.56% (30.23%)	0.008	1	9.04% (20.98%)	0.011	1	**62.99%** (21.00%)
virtio-iommu	15.11% (13.01%)	0.007	1	39.16% (25.68%)	0.290	0.7	14.48% (13.82%)	0.007	1	**42.31%** (16.89%)
virtio-mem	12.26% (11.85%)	0.007	1	22.07% (24.91%)	0.008	1	14.66% (13.78%)	0.004	1	**28.57%** (17.12%)
Virtio.Mean	12.97%			28.47%			19.09%			**38.10%** (+34%)
Geo.Mean	37.77%			51.62%			41.93%			**55.90%**

TABLE II: The branch coverage that `Truman` achieved compared to the state-of-the-art fuzzer, with statistical significance (p-values) [33] and effect size ($\hat{A}_{12}$) [34], and geometric mean coverage metrics. For the virtio devices, the coverage in the braces is the coverage of the virtio core layer.

dependencies are typically shown between two different DMA buffers, and the number of DMA buffers is typically limited. The results show that the `nvme` device, network devices, and USB devices have more DMA buffers than others, indicating that these devices use DMA buffers extensively. The results across different versions of the Linux kernel are stable, with 15 out of 17 devices remaining. We manually checked the differences between the results in the device `lsilogicsas` and `igb`, and we found that the differences are caused by the updates of the device driver across different versions of the Linux kernel.

State-Dependency Graph: The results show that virtio devices have more states than other devices since they have fine-grained states for different request types. The results across different Linux kernel versions are stable, with 16 out of 17 devices remaining constant, which matches our expectations. The only exception is the device `xhci`. In the newer version of the Linux kernel, the device driver introduces a function named `hcd_pci_poweroff_late()`, allowing more fine-grained control of the power-off process of the device, leading to the change in the state dependency graph.

Time Consumption: The average analysis time for all devices is less than 5 seconds, showing the efficiency of `Truman` since the OS drivers are relatively small.

C. Code Coverage Comparison

To evaluate different fuzzers by coverage across virtual devices, we chose the latest stable QEMU version, 9.0.0, as the benchmark for the comparison since most SOTA fuzzers only support fuzzing open-sourced hypervisors. Truman, equipped with QEMU's executor QTest, was compared with Morphuzz, ViDeZZo, and AFL++ (in the default config without CmpLog). We selected 29 virtual devices across various categories, i.e., audio, storage, network, display, and USB, ensuring a broad representation. The devices not only include well-known devices (e.g., `e1000`, `ac97`), which were thoroughly fuzzed in previous research, but also include devices that were not well explored in previous studies (e.g., `NVMe`, `vmxnet3`). Additionally, we chose to evaluate virtio devices, which are widely used in cloud production environments due to their efficiency and high throughput.

Table II shows the coverage that `Truman` achieved compared to the state-of-the-art fuzzers. We found that `Truman`

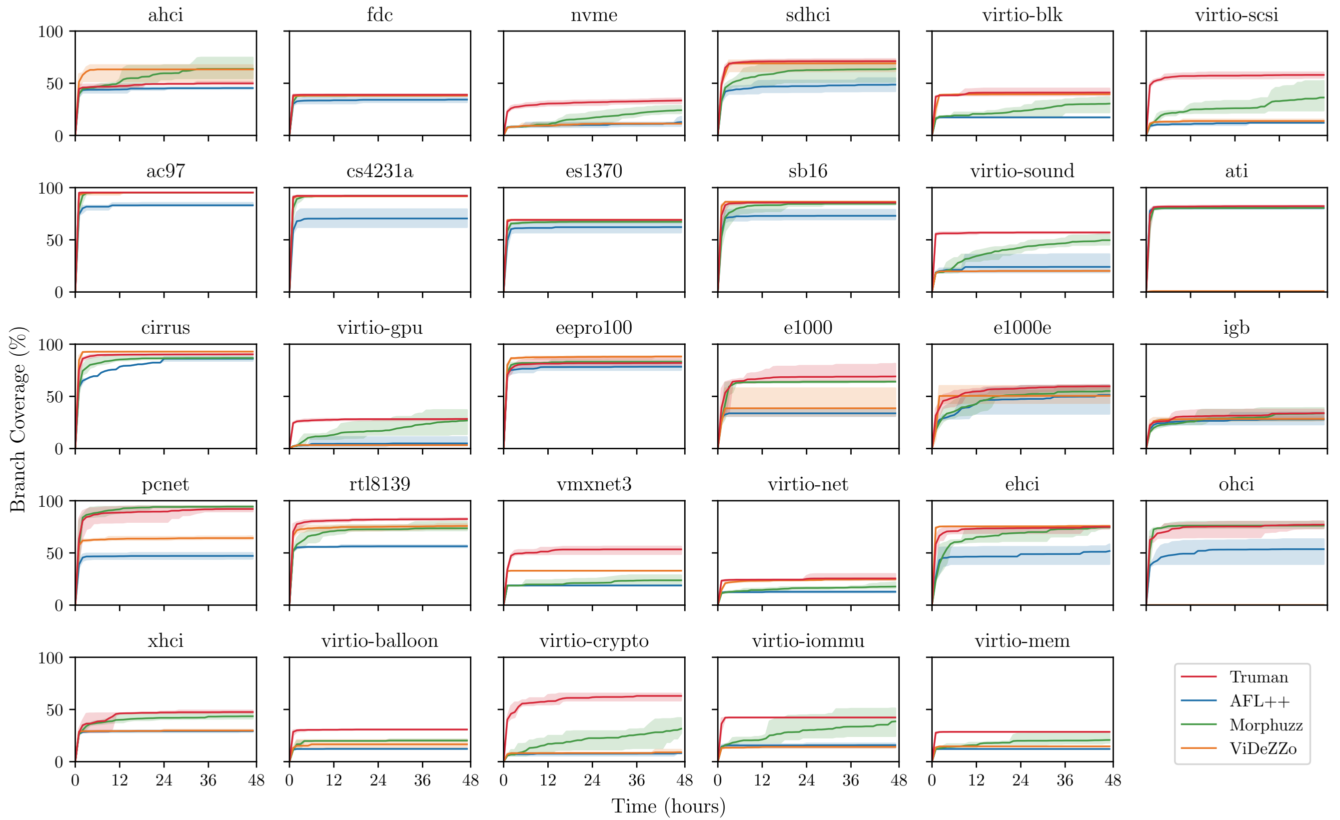


Fig. 11: Branch coverage over 48 hours fuzzing 29 devices. The shaded regions represent the maximum and minimum coverage achieved by fuzzers across five runs.

achieves higher coverage for 19 of 29 virtual devices and outperforms AFL++, Morphuzz, and ViDeZZo. To statistically validate the improvements, we conducted the Mann-Whitney U test, as indicated by the best practices for fuzzing experiments [35]. Specifically, we use the general form of the Mann-Whitney U test, which assesses whether the coverage values from `Truman` are generally higher than those from other fuzzers. The Mann-Whitney U test is appropriate for our context as a non-parametric test comparing two independent samples without the assumption of normality, which suits our coverage data. Our coverage measurements are ordinal, independently obtained across fuzzers and runs, fulfilling the test's requirements of independent samples and comparability in ranks between groups. If these assumptions did not hold, the test could produce incorrect results, and our findings might not be statistically significant. The Mann-Whitney U test produces low p-values (< 0.05), indicating statistically significant improvements achieved by `Truman`. However, there are some exceptions such as the comparison between `Truman` and ViDeZZo for the `sdhci` device, where the p-value is 0.753. The high p-value suggests that the coverage difference between `Truman` and ViDeZZo is not statistically significant. To further evaluate the practical significance of these improvements, we apply the Vargha-Delaney $\hat{A}_{12}$ effect size metric. The metric quantifies the probability that `Truman` consistently outperforms other fuzzers, offering an intuitive measure of performance improvement that does not rely on normality assumptions. The $\hat{A}_{12}$ metric requires an ordinal scale for measurements and assumes independent observations between groups, both of which are met in our setup. High $\hat{A}_{12}$ values (i.e., > 0.5) suggest that `Truman` frequently achieves better coverage. The AFL++ fuzzer behaves poorly in the coverage comparison, which is expected since AFL++ is a generic fuzzer without domain-specific knowledge, such as the nested DMA buffers. For devices which are thoroughly fuzzed in previous research (e.g., `ac97`, `rtl8139`, and `pcnet`), `Truman` obtains a similar coverage to Morphuzz and ViDeZZo because of the existing high coverage. `Truman` achieves the best coverage for all para-virtualized virtio devices and obtains a boost of 34%. We also evaluate the coverage for two layers, one for the specific device and another for the virtio core layer. `Truman` rarely reaches higher coverage in the virtio core layer because `Truman` focuses on the devices under the virtio bus.

Figure 11 shows the coverage increase over 48 hours of fuzzing. The result shows that `Truman` achieves a relatively high coverage for most devices in the first few hours of fuzzing. We contribute the rapid coverage increase to the device behavior model, which generates and mutates the seeds that satisfy the expectation of the device, leading to exploring the code space more efficiently. Note that some runs of

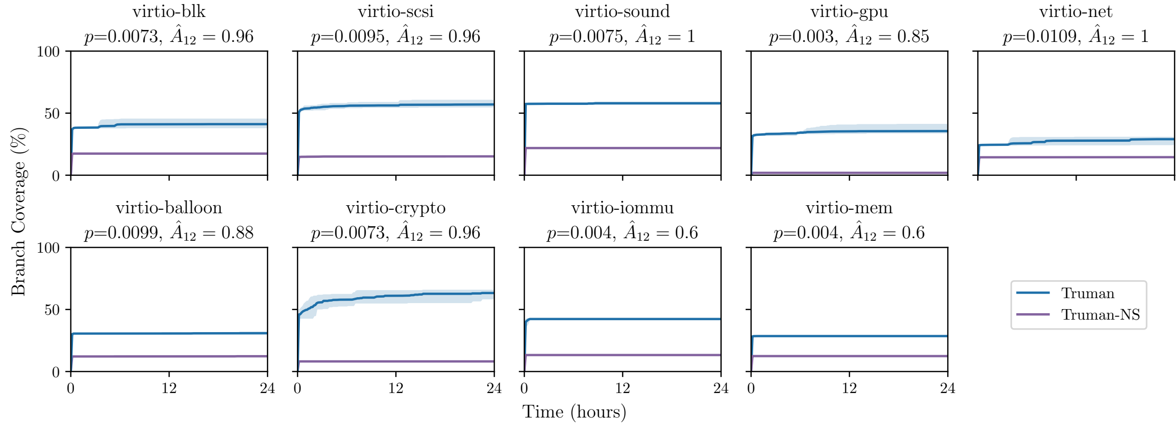


Fig. 12: Branch coverage over 24 hours fuzzing bus-hidden devices across five runs. "NS" represents the coverage achieved by `Truman` without state dependency.

Morphuzz outperform `Truman`, especially for the devices `ahci`, `virtio-gpu`, `virtio-iommu`, where Morphuzz demonstrates a higher spread and maximum coverage. This suggests that `Truman` 's reliance on the device behavior model accelerates early exploration but it may plateau more quickly, potentially limiting the depth of exploration due to potential limitations of our device behavior model. We also found that there are inconsistencies between our evaluation results and the results in the ViDeZZo paper. The inconsistencies have two causes: First, we used a simpler but different configuration than ViDeZZo used in their evaluation for `fdc` device. Second, we found that ViDeZZo does not reset the devices between inputs, causing a shallow bug that could cause the fuzzer to stop early, which restricts its ability to explore the code space.

From the results, we found that `Truman` improved coverage significantly for some devices (e.g., `virtio-sound`, `virtio-scsi` and `virtio-crypto`), but not for all devices (e.g., `e1000e` and `igb`). We have analyzed the existing coverage for these virtual devices and concluded that the relatively low coverage is mainly caused by the following reasons: First, the configuration of virtual devices is restricted. `Truman` achieved similar coverage for devices such as `fdc`, `sdhci`, and `ehci`. The fixed configuration of these devices limits the maximum coverage that fuzzers can achieve, preventing fuzzers from exploring additional code beyond the current configuration. Second, virtual devices also need inputs from external environments. For example, the network devices also need the network packets from external to trigger the network operations of devices such as `receive()`. Fuzzers are unlikely to explore additional code without external inputs.

D. Effectiveness of State Dependency

We evaluated the effectiveness of state dependency on coverage by calculating both the Mann-Whitney U test and the Vargha-Delaney $\hat{A}_{12}$ effect size to compare `Truman`'s performance with and without state dependency. The evaluation setup is the same as the code coverage comparison, but we only focus on the bus-hidden devices, which are the devices that are not well explored by the previous research. Figure 12 shows the coverage over 24 hours of fuzzing for these devices. The results show that inter- and intra-message dependency alone cannot achieve high coverage for the bus-hidden devices. The p-values indicate statistically significant improvements in coverage for `Truman` compared to `Truman-NS`, with all p-values less than 0.05. Additionally, the $\hat{A}_{12}$ values further demonstrate that `Truman` consistently outperforms `Truman-NS` across all devices. We contribute the high coverage of `Truman` to the fine-grained state dependency, which helps the fuzzer be aware of the request type, leading to exploring the code space more efficiently.

Device	Type	Morphuzz	VideZZO	Truman
ac97	audio	0	0	0
intel-hda	audio	1	1	1
am53c974	storage	0	1	1
ide-hd	storage	1	0	2
nvme	storage	0	0	2
igb	network	0	0	0
e1000	network	0	0	0
virtio-net	network	1	1	1
virtio-gpu	display	0	0	1
sm501	display	1	1	2
ati-vga	display	0	0	0
ehci	USB	0	0	0
xhci	USB	0	0	0
Total		**4**	**4**	**10**

TABLE III: Known bug discovery ability comparison between Morphuzz, ViDeZZo, and `Truman` in QEMU v8.0.0.

E. Known Bug Discovery Comparison

We evaluated the bug discovery ability of `Truman` by comparing it with the state-of-the-art fuzzers Morphuzz and ViDeZZo. We chose QEMU version 8.0.0 (the previous major version) as the target hypervisor for fair comparison. This version contains many old bugs that were fixed in the latest version, which is a suitable benchmark for assessing the bug discovery ability of different fuzzers. For virtual devices, we

selected the same devices in the static analysis evaluation and fuzzed each device for 24 hours, repeating the evaluation five times to ensure the stability of the result. Due to the randomness inherent in fuzzing, a bug was counted if it was triggered in one of five runs. The results are shown in Table III. From the results, Truman covered all bugs found by Morphuzz and ViDeZZo, found 10 bugs for 13 virtual devices, while Morphuzz and ViDeZZo found 4 bugs.

F. Discovery of New Bugs

Truman conducted a long-term evaluation of the latest hypervisors to discover new bugs. We chose the most prevalent virtualization framework, including QEMU, VirtualBox, VMware Workstation Pro, and Parallels. We equipped Truman with the QTest for QEMU and enhanced HyperCube OS for other hypervisors. To discover bugs, we compile QEMU and VirtualBox with ASAN and keep all assertions. For closed-source hypervisors, we rely on VM crashes to capture the bugs. Among these targets, Truman found 54 new bugs, including various types (assert, null-pointer-dereference, memory leak, or stack overflow), as shown in Table IV. Some virtual devices in QEMU have been fuzzed extensively since 2020 in OSS-Fuzz (such as ide), but Truman also discovered new bugs. As for the responsibility of bug disclosure, we have reported all bugs to the corresponding developers and received some active feedback from them. We also proposed patches for these bugs, and 31 bugs have been fixed in the latest version.

G. Case Study

We select some representative bugs found by Truman to illustrate the unique advantages of Truman's ability to extract the device behavior models of virtual devices.

QEMU: Memory-leak in NVMe: NVMe is an attractive target for fuzzing because of its high throughput and low latency. However, the previous research cannot cover more code space of the NVMe devices because the main functionalities of NVMe are encoded in the DMA structure. If the guest wants to trigger a command in the NVMe, it must prepare the correct DMA buffer and fill the command field and the corresponding buffer. Morphuzz [10] cannot recognize the command field and its relationship with the buffer, relying on the generic algorithm of fuzzing to test the device randomly. However, the NVMe commands are magic numbers, so it is difficult for fuzzers to generate randomly, but Truman extracts them from the corresponding device drivers automatically. ViDeZZo [11] manually extracts dependencies, and the authors do not apply them to the NVMe device. This example shows the ability to infer the intra-message of the device of Truman.

QEMU: Null-pointer-dereference in virtio-sound: When the guest transfers a mismatched message to the rx buffer, the virtio-sound drops the message and enters the error handling. The bug exists in the error handling code of the virtio-sound device. The root cause of the bug is that the device uses some resources that are not initialized correctly in the error handling code, leading to a null-pointer-dereference bug. To trigger this bug in the error handling code, the fuzzer must follow the correct sequence of the messages to the virtio core layer and then send the message to the virtio-sound device. Such a sequence of searching space is extremely unlikely for other

Device	Bug Type and Location
QEMU	
adb	assert in adb_request
am53c974	assert in scsi_unmap_complete_noio
aspeed_i2c	assert in aspeed_i2c_bus_new_write
bcm2835	assert in bcm2835_thermal_read
dwc2	assert in dwc2_hsotg_read
exynos4210	assert in exynos4210_rng_read
igb	assert in igb_setup_tx_offloads*
stm32l4x5	assert in rcc_update_cfgr_register
virtio-crypto	assert in cryptodev_builtin_close_session
virtio-iommu	assert in virtio_iommu_handle_command
virito-sound	assert in virtio_snd_get_qemu_format
xlnx-ospi	assert in fifo8_pop
xilinx-spips	assert in fifo8_pop
ide	division by zero in ide_set_sector
pl011	division by zero in pl011_get_baudrate
aspeed_gpio	global-buffer-overflow in aspeed_gpio_read
sm501	global-buffer-overflow in sm501_2d_operation
arm_gic	heap-overflow in gic_extract_lr_info
m25p80	heap-overflow in flash_erase
sifieve_plic	heap-overflow in sifive_plic_read
ufs	heap-overflow in ufs_dma_read_req_upiu
virtio-sound	heap-overflow in virtio_snd_handle_rx_xfer
virtio-sound	heap-overflow in virtio_snd_pcm_in_cb
applesmc	memory leak in qdev_applesmc_isa_reset
musb	memory leak in musb_reset
nvme	memory leak in nvme_dsm
virtio-blk	memory leak in virtio_blk_zone_report
virtio-crypto	memory leak in cryptodev_builtin_operation
virtio-crypto	memory leak in virtio_crypto_create_asym_session
a9gtimer	NPD in a9_gtimer_get_current_cpu
arm_gic	NPD in gic_update_internal
lan9118	NPD in lan9118_receive
tcx	NPD in tcx_blit_writel
ufs	NPD in ufs_mcq_process_db
virito-sound	NPD in virtio_snd_handle_rx_xfer
goldfish_tty	stack overflow in goldfish_tty_cmd
lsi53c895a	stack overflow in lsi_reg_writeb*
tulip	stack overflow in tulip_xmit_list_update*
VirtualBox	
ata	assert in ataR3AsyncIOThread
vmm	assert in PDMCritSectLeave
buslogic	division by zero in buslogicR3OutgoingMailboxAdvance*
virtio-scsi	NPD in virtioCoreR3VirtqUsedBufPut
lsilogic	stack overflow in lsilogicR3ProcessSCSIIORequest*
ehci	stack overflow in ehciR3SubmitITD*
VMware	
buslogic	assert in buslogic.c
e1000	assert in e1000.c
hdaudio	assert in alc885.c
pcnet	assert in iospace_shared.h
piix4	assert in piix4SM.c
pvscsi	assert in pvscsi_monitor.c
sb16	sound device FIFO overflow
Parallels	
ide	assert in Ide.cpp
hba	assert in hba.cpp
virito-net	assert in PciVirtIO.cpp
Total	54

TABLE IV: New bugs discovered by Truman in different hypervisors. NPD stands for null-pointer-dereference. The bugs with * are assigned with CVE numbers.

fuzzers without dependency inference. However, `Truman` extracts such inter-message dependencies from the driver, and the fuzzer schedules the sequences of these operations.

VirtualBox: Null-pointer-dereference in virtio-scsi: The virtual device virtio-scsi is widely used in the cloud environment to achieve high throughput, so it is sincerely security-sensitive. The bug in the virtio-scsi comes from the invalid receive buffer provided by the guest. The guest first interacts with the core layer of virtio and then the virtio-scsi device. During the interaction with the virtio-scsi virtual device, the guest sends an invalid receive buffer to the virtio-scsi device, and the device does not check the pointer to the buffer, causing a null-pointer-dereference bug. To trigger this deep bug in the virtio-scsi device error handling code, `Truman` must interact correctly with the virtio core layer according to the state dependency, such as setting up the device, configuring the virtio queue, kicking the device to fetch the queue, finally triggering the error state of the device. This demonstrates the ability of `Truman` to traverse the state graph of the virtio devices and try to find the possible bugs.

VirtualBox: Stack-buffer-overflow in the USB controller EHCI: The Enhanced Host Controller Interface (EHCI) is an import USB controller attached to the PCI bus, and the device uses DMA extensively. The bug exists in processing the `itd` entry; the virtual device reads an entry from the DMA buffer of the devices and uses it as an index of another buffer. However, the user-provided buffer index can be large, causing an overflow bug. `Truman` triggers this bug in less than one second due to the awareness of the nested DMA structure of the virtual device, showing the effectiveness of the Morphuzz will fill DMA pages when the virtual device triggers the read operation of the buffer. Still, it is not aware of the real structure of the DMA, leading to hiding this legitimate bug.

VI. Discussion

A. Support of More Hypervisors and Virtual Devices

By applying extracted device behavior models to the executor of Hyper-Cube [6], we have discovered vulnerabilities in the closed-source hypervisors such as VMware Workstation Pro and Parallels Desktop. However, Hyper-Cube only supports 32-bit guest VMs, which is not compatible with hypervisors such as Hyper-V [36] and the macOS Virtualization Framework [37]. Upgrading Hyper-Cube to support 64-bit guest VMs is a challenging task. We plan to utilize the device behavior models with the executor of Hyperpill to discover bugs in more hypervisors.

Although software-based virtual devices are the most attractive targets of recent virtual device fuzzers, an increasing number of cloud providers are adopting a combined approach of hardware and software, i.e., Virtual Function I/O (VFIO) [38], Single Root I/O Virtualization (SR-IOV) [39], and Amazon Nitro Cards [40], allowing a VM to access the actual physical devices directly. However, discovering vulnerabilities in such an environment is challenging, especially when collecting coverage from hardware and detecting crashes in hardware.

B. Incompleteness of Semantics in OS Drivers

While OS drivers and virtual devices are designed to conform to the same specifications, allowing `Truman` to infer virtual device behavior and construct corresponding device behavior models, the semantics of OS drivers often exhibit incompleteness for serval reasons. First, OS drivers may not implement all the features outlined in the specification. This is typically evidenced by "todo" comments in the code to mark sections that require further development. The comments explicitly indicate acknowledged gaps in the current implementation. Secondly, virtual devices may incorporate features or optimizations not thoroughly documented in the specification, leading to a semantic mismatch between the virtual device and the OS driver. However, `Truman` combines the extracted device behavior models with coverage-guided fuzzing to mitigate the aforementioned issues. Another potential issue is the absence of a corresponding driver for a virtual device, but this scenario is rare in practice and was not observed in the hypervisor we tested.

C. Application in Fuzzing OS Drivers

OS drivers suffer from attacks on two sides: the syscalls and the device input. Therefore, fuzzing OS drivers requires inputs from both malicious users (via syscalls) and malicious devices (via PIO/MMIO/DMA/Interrupt) [41], [42], [43], [44], [45], [46], [47]. Syzkaller is a tool that discovers bugs in OS drivers via syscall sequences. However, existing driver fuzzers are restricted to the number of virtual devices, which limits the variety of drivers that can be tested, leading to significant gaps in code coverage and the number of bugs. Previous research [41], [42], [43] shows that automatically generating virtual devices for fuzzing OS drivers is challenging. With the help of device behavior model, the automatic generation of complex virtual devices is possible. This would expand the range of drivers that can be tested, uncover more bugs, and reduce both manual effort and costs, thereby enhancing the efficiency and effectiveness of OS driver fuzzing.

VII. Related Work

A. Static Analysis

Static analysis has become an essential tool in software development, enabling a thorough review of software without the need for execution. By utilizing methods that cover both data and control flow analyses, it helps reveal potential vulnerabilities that are hidden within complex software systems, such as the Linux kernel [48], [49], [50], [51], [52]. Researchers have developed various static analysis tools to assist developers in identifying bugs in the code. Among them, LLVM IR is one of the most well-known tools, offering developers a universal way to develop their static analysis algorithms. `Truman` thus adapts the existing static analysis infrastructure to analyze open-source device drivers and is the first to extract knowledge from them to fuzz the virtual devices.

B. Fuzzing

Over the years, fuzzing has emerged as an efficient technique in vulnerability detection. Its initial versions, largely characterized as black-box approaches, faced limitations due to inefficient guidance. The introduction of AFL [53] in 2013 marked a progressive improvement, proposing the principles of genetic algorithms, which signifies the rise of gray-box fuzzing, collecting code coverage during run-time and guiding the fuzzer to retain the inputs that cover new code. Researchers then have extensively studied fuzzing and found thousands of vulnerabilities via numerous improvements to fuzzing algorithms and many useful tools [14], [54], no longer limited to user-space programs [55], [56], [57], [58], [59], but also covering operating system kernels [60], [61], [62], [63], [64], [43], [51], [52], [65], hypervisors [11], [13], [10], IoT devices [66], [67], [68], [17], [69], trusted execution environment [70], [71]. `Truman` leverages these advancements in fuzzing, particularly in grey-box fuzzing and its application, to develop a fuzzing approach for virtual devices.

C. Hypervisor Fuzzing

Several notable hypervisor fuzzers have been proposed in recent years. Among them, VDF [5] was the first to introduce fuzzing into the hypervisor, using record and replay techniques to test the PIO/MMIO interfaces of virtual devices but does not support the DMA interfaces. Solutions such as Hyper-Cube [6] and NYX [8] implemented customized operating systems for hypervisor fuzzing, effectively covering multiple interfaces. These approaches, however, do not engage deeply with semantic information, limiting their effectiveness in detecting more nuanced vulnerabilities.

Recent works, such as V-Shuttle [7] and Morphuzz [10], focused on the nested DMA problem. They introduced the concept of hooking DMA address translation functions, enabling translation on demand without needing to understand the specific structure of the nested DMA. MundoFuzz [9] and ViDeZZo [11] went further at the semantic level. MundoFuzz inferred register types and dependency information from execution trace information. ViDeZZo, on the other hand, determined intra-message dependencies through static analysis of the virtual device code and inter-message dependencies through defined mutators. VD-GUARD [12] introduces a hybrid approach, blending static control flow analysis with DMA-guided fuzzing to uncover vulnerabilities in virtual devices. HyperPill [13] fuzzes both open-source and closed-source hypervisors, leveraging the fact that hypervisors are built on top of the same hardware virtualization specification (Intel VT-x). `Truman` can enhance HyperPill in the sense of dependency-aware in the future.

While these previous studies have made significant strides in code coverage increase and vulnerability detection of hypervisors, they also reveal gaps in reliance on source code and dependencies of virtual devices, which `Truman` seeks to address and improve upon.

VIII. Conclusion

Virtual devices in hypervisors are vulnerable and valuable targets for attackers. However, fuzzing virtual devices needs a new generic and automatic tool to extract the device behavior model including not only inter- and intra-message dependencies but also state dependency. To fill this gap, we propose `Truman` that extracts the device behavior model from a virtual device's corresponding OS driver and assist in fuzzing. The approach proposed by `Truman` is feasible because both the virtual device and its OS drivers adhere to the same specification. Additionally, `Truman` also develops a fuzzing engine to support the generation and mutation of virtual device messages that satisfy the dependencies encoded in the device behavior model. With the help of the device behavior model, `Truman` outperforms 19 out of 29 QEMU devices for code coverage compared to state-of-the-art fuzzers AFL++, Morphuzz and ViDeZZo during 48-hour fuzzing and discovers 54 bugs in the well-known hypervisor including QEMU, VirtualBox, VMware Workstation Pro, and Parallels, with 31 bugs fixed and 6 CVEs assigned.

Acknowledgment

We thank all anonymous reviewers for their valuable feedback to improve this paper. We also recognize the unprecedented involvement of the shepherd who went above and beyond by carefully validating our LaTeX source code of our paper. While the tight shepherding deadline prohibited us from submitting our artifact for evaluation, we encourage the community to build on our prototype.

This work was supported, in part, by the National Natural Science Foundation of China (U24A20337), National Key R&D Program of China (2021YFB2701000), Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd., and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868) and SNSF PCEGP2 186974.

References

[1] VMware, "VMSA-2024-0006," https://support.broadcom.com/web/ecx/support-content-notification/-/external/content/SecurityAdvisories/0/24266, 2024, accessed: 2024-09-20.

[2] X. Ge, B. Niu, R. Brotzman, Y. Chen, H. Han, P. Godefroid, and W. Cui, "HYPERFUZZER: An efficient hybrid fuzzer for virtual cpus," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[3] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo, "Virtual cpu validation," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[4] P. Fonseca, X. Wang, and A. Krishnamurthy, "Multinyx: a multi-level abstraction framework for systematic analysis of hypervisors," in *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[5] A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng, "Vdf: Targeted evolutionary fuzz testing of virtual devices," in *Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.

[6] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "HYPER-CUBE: High-Dimensional Hypervisor Fuzzing." in *Network and Distributed System Security (NDSS) Symposium*, 2020.

[7] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[8] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *USENIX Security Symposium (USENIX Security)*, 2021.

[9] C. Myung, G. Lee, and B. Lee, "MundoFuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference," in *USENIX Security Symposium (USENIX Security)*, 2022.

[10] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "MORPHUZZ: Bending (input) space to fuzz virtual devices," in *USENIX Security Symposium (USENIX Security)*, 2022.

[11] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer, "VIDEZZO: Dependency-aware Virtual Device Fuzzing," in *IEEE Symposium on Security and Privacy (SP)*, 2023.

[12] L. Yuwei, C. Siqi, X. Yuchong, W. Yanhao, C. Libo, W. Bin, Z. Yingming, X. Zhi, and S. Purui, "VD-Guard: DMA Guided Fuzzing for Hypervisor Virtual Device," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[13] A. Bulekov, Q. Liu, M. Egele, and M. Payer, "HYPERPILL: Fuzzing for Hypervisor-bugs by Leveraging the Hardware Virtualization Interface," in *USENIX Security Symposium (USENIX Security)*, 2024.

[14] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[15] D. H. Barboza, "sungem: Add WOL MMIO," https://lore.kernel.org/all/20230707113108.7145-9-danielhb413@gmail.com/, 2024, accessed: 2024-09-20.

[16] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[17] Q. Liu, C. Zhang, L. Ma, M. Jiang, Y. Zhou, L. Wu, W. Shen, X. Luo, Y. Liu, and K. Ren, "Firmguide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[18] L. Torvalds, "Linux kernel source tree," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/, 2024, accessed: 2024-09-20.

[19] Linux, "virtio driver," https://github.com/torvalds/linux/tree/12cc5240f41a90b7fabc075c92c04846670c6932/drivers/virtio, accessed: 2024-09-20.

[20] Amazon, "Amazon Web Services," https://aws.amazon.com/, 2006, accessed: 2024-09-20.

[21] Google, "Google Cloud," https://cloud.google.com/, 2008, accessed: 2024-09-20.

[22] A. Kadav and M. M. Swift, "Understanding modern device drivers," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 87–98, 2012.

[23] VMware, "open-vm-tools," https://github.com/vmware/open-vm-tools/tree/master, 2024, accessed: 2024-09-20.

[24] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[25] LLVM, "libFuzzer - a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2021, accessed: 2024-09-20.

[26] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[27] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in os kernels," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[28] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. Butler, A. Bianchi, and D. J. Tian, "Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks," in *IEEE Symposium on Security and Privacy (SP)*, 2022.

[29] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *International Conference on Compiler Construction (CC)*, 2016.

[30] LLVM, "The LLVM Compiler Infrastructure," https://llvm.org/, 2003, accessed: 2024-09-20.

[31] Google, "protobuf," https://github.com/protocolbuffers/protobuf, accessed: 2024-09-20.

[32] L. Team *et al.*, "Source-based Code Coverage," https://clang.llvm.org/docs/SourceBasedCodeCoverage.html, 2024, accessed: 2024-09-20.

[33] L. Sachs, *Applied statistics: a handbook of techniques*. Springer Science & Business Media, 2012.

[34] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[35] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024.

[36] A. Velte and T. Velte, *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[37] A. Inc., "Create virtual machines and run macOS and Linux-based operating systems." https://developer.apple.com/documentation/virtualization, accessed: 2024-09-20.

[38] A. Williamson, "VFIO: A user's perspective," in *KVM Forum*, 2012.

[39] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.

[40] Amazon, "AWS Nitro System: A combination of dedicated hardware and lightweight hypervisor enabling faster innovation and enhanced security," https://aws.amazon.com/ec2/nitro/, 2024, accessed: 2024-09-20.

[41] Y. Wu, T. Zhang, C. Jung, and D. Lee, "DEVFUZZ: automatic device model-guided device driver fuzzing," in *IEEE Symposium on Security and Privacy (SP)*, 2023.

[42] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting bugs in device drivers from golden seeds," in *USENIX Security Symposium (USENIX Security)*, 2022.

[43] Z. Ma, B. Zhao, L. Ren, Z. Li, S. Ma, X. Luo, and C. Zhang, "Printfuzz: Fuzzing linux drivers via automated virtual device simulation," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[44] H. Peng and M. Payer, "USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation," in *USENIX Security Symposium (USENIX Security)*, 2020.

[45] F. Hetzelt, M. Radev, R. Buhren, M. Morbitzer, and J.-P. Seifert, "VIA: Analyzing Device Interfaces of Protected Virtual Machines," in *Annual Computer Security Applications Conference (ACSAC)*, 2021.

[46] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[47] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *Network and Distributed System Security (NDSS) Symposium*, 2019.

[48] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR.CHECKER: A soundy analysis for linux kernel drivers," in *USENIX Security Symposium (USENIX Security)*, 2017.

[49] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in linux device drivers," in *2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[50] J. Choi, K. Kim, D. Lee, and S. K. Cha, "NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis," in *IEEE Symposium on Security and Privacy (SP)*, 2021.

[51] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "StateFuzz: System Call-BasedState-Aware linux driver fuzzing," in *USENIX Security Symposium (USENIX Security)*, 2022.

[52] M. Yuan, B. Zhao, P. Li, J. Liang, X. Han, X. Luo, and C. Zhang, "DDRace: finding concurrency UAF vulnerabilities in Linux drivers with directed fuzzing," in *USENIX Security Symposium (USENIX Security)*, 2023.

[53] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2014, accessed: 2024-09-20.

[54] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[55] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[56] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *USENIX Security Symposium (USENIX Security)*, 2020.

[57] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-Net: Network Fuzzing with Incremental Snapshots," in *European Conference on Computer Systems (EuroSys)*, 2021.

[58] V. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: A Greybox Fuzzer for Network Protocols," in *IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track (ICST)*, 2020.

[59] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets," in *USENIX Security Symposium (USENIX Security)*, 2023.

[60] Y. Lan, D. Jin, Z. Wang, W. Tan, Z. Ma, and C. Zhang, "Thunderkaller: Profiling and improving the performance of syzkaller," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023.

[61] T. Yin, Z. Gao, Z. Xiao, Z. Ma, M. Zheng, and C. Zhang, "Kextfuzz: Fuzzing macos kernel extensions on apple silicon via exploiting mitigations," in *32nd USENIX Security Symposium (USENIX Security)*, 2023.

[62] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for OS kernels," in *USENIX Security Symposium (USENIX Security)*, 2017.

[63] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid Fuzzing on the Linux Kernel." in *Network and Distributed System Security (NDSS) Symposium*, 2020.

[64] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *IEEE Symposium on Security and Privacy (SP)*, 2020.

[65] G. Lee, D. Xu, S. Salimi, B. Lee, and M. Payer, "Syzrisk: A change-pattern-based continuous kernel regression fuzzer," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (Asia CCS)*, 2024.

[66] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing." in *Network and Distributed System Security (NDSS) Symposium*, 2018.

[67] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[68] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL:High-Throughput greybox fuzzing of IoT firmwar19e via augmented process emulation," in *USENIX Security Symposium (USENIX Security)*, 2019.

[69] Q. Wang, B. Chang, S. Ji, Y. Tian, X. Zhang, B. Zhao, G. Pan, C. Lyu, M. Payer, W. Wang *et al.*, "SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices," in *IEEE Symposium on Security and Privacy (SP)*, 2023.

[70] L. Chen, Z. Li, Z. Ma, Y. Li, B. Chen, and C. Zhang, "EnclaveFuzz: Finding Vulnerabilities in SGX Applications," in *Network and Distributed System Security (NDSS) Symposium*, 2024.

[71] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer, "TEEZZ: Fuzzing trusted applications on cots android devices," in *IEEE Symposium on Security and Privacy (SP)*, 2023.

APPENDIX

Function	State
`probe`	Setup
`remove`	Cleanup
`suspend`	Suspend
`resume`	Resume
`Shutdown`	Save
`xxx_ops`	Transmission

TABLE VII: A mapping of PCI driver functions to states.

PIO/MMIO Write	PIO/MMIO Read
iowrite64	ioread64
iowrite32_rep	ioread32_rep
iowrite16_rep	ioread16_rep
iowrite8_rep	ioread8_rep
iowrite32	ioread32
iowrite16	ioread16
iowrite8	ioread8
writeq	readq
writel	readl
writew	readw
writeb	readb
outl	readl
outw	readw
outb	readb
pci_write_config_dword	pci_read_config_dword
pci_write_config_word	pci_read_config_word
pci_write_config_byte	pci_read_config_byte
pcie_write_config_dword	pcie_read_config_dword
pcie_write_config_word	pcie_read_config_word

TABLE V: IO Functions in the Linux kernel.

Coherent DMA	Streaming DMA
dma_alloc_coherent	dma_map_single
dma_pool_create	dma_map_page
dma_pool_zalloc	dma_map_resource
dma_pool_alloc	dma_map_sg
	dma_map_single_attrs
	dma_map_sg_attrs

TABLE VI: DMA allocation functions in the Linux kernel.

Virtual Device	Vendor ID	Device ID
VirtualBox		
lsilogicsas	0x1000	0x0054
virtio-scsi	0x1AF4	0x1004
VMware		
svga	0x15AD	0x0710
pvscsi	0x15AD	0x07c0
QEMU Audio		
ac97	0x8086	0x2415
intel-hda	0x8086	0x0403
QEMU Storage		
am53c974	0x1022	0x2020
ahci	0x8086	0x2415
nvme	0x8086	0x5845
QEMU Network		
igb	0x8086	0x10C9
31000	0x8086	0x100E
virtio-net	0x1AF4	0x1000
QEMU Display		
ati	0x1002	0x5046
sm501	0x126F	0x0501
virtio-gpu	0x1AF4	0x1050
QEMU USB		
ehci	0x8086	0x24CD
xhci	0x1B36	0x000D

TABLE VIII: The Vendor ID and Device ID of virtual devices.