VIDEZZO: Dependency-aware Virtual Device Fuzzing

Qiang Liu*†, Flavio Toffalini†, Yajin Zhou*, Mathias Payer†
* *Zhejiang University,* † *EPFL*

***Abstract*—A virtual machine interacts with its host environment through virtual devices, driven by virtual device messages, e.g., I/O operations. By issuing crafted messages, an adversary can exploit a vulnerability in a virtual device to escape the virtual machine, gaining host access. Even though hundreds of bugs in virtual devices have been discovered, coverage-based virtual device fuzzers hardly consider intra-message dependencies (a field in a virtual device message may be dependent on another field) and inter-message dependencies (a message may depend on a previously issued message), thus resulting in limited scalability or efficiency.**

VIDEZZO, our new dependency-aware fuzzing framework for virtual devices, overcomes the limitations of existing virtual device fuzzers by annotating intra-message dependencies with a lightweight grammar, and by self-learning inter-message dependencies with new mutation rules. Specifically, VIDEZZO annotates message dependencies and applies three categories of message mutators. This approach avoids heavy manual effort to analyze specifications and speeds up the slow exploration by satisfying dependencies, resulting in a scalable and efficient fuzzer that boosts bug discovery in virtual devices.

In our evaluation, VIDEZZO covers two hypervisors, four architectures, five device categories, and 28 virtual devices, and reaches competitive coverage faster. Moreover, VIDEZZO successfully finds 24 existing and 28 new bugs across diverse bug types. We are actively engaging with the community with 7 of our submitted patches already accepted.

1. Introduction

Hypervisors (virtual machine monitors—VMMs) are widely deployed in cloud infrastructure. They transfer data and instructions from guest operating systems to the host environment through virtual devices that are driven by I/O operations (Port I/O—PIO or Memory-Mapped I/O—MMIO). These I/O operations follow specific protocols and thus are called *virtual device messages*.

Virtual devices are the most prominent attack surface in hypervisors. Hypervisors isolate an untrusted guest from the hypervisor and all other virtual machines. A key security property is that a guest cannot escape from its virtual machine. However, hypervisors are complex pieces of software and researchers have discovered ways to escape them (e.g., QEMU, VirtualBox, and VMWare), with 41.5% (22/53) of these escapes due to bugs in virtual devices [1]. According to our CVE survey [2], 57.4% (252/439) of the vulnerabilities in QEMU were found in virtual devices.

The security of virtual devices has been under heavy scrutiny [3], [4], [5], [6], [7], [8], [9], [10]. Since VFD [7] in 2017, fuzzing has become the dominant approach as it implicitly abstracts device complexity through concrete executions, outperforming symbolic approaches. Later, a platform-independent black-box hypervisor fuzzer [8] discovered multiple bugs due to its high throughput and multi-dimension inputs. Fuzzing of virtual devices advanced further when started considering coverage feedback [9] and guest-provided data through DMA channels [9], [10], [11].

Despite hundreds of bugs in virtual devices, existing solutions are limited due to two, so far, overlooked challenges.

Intra-Message Dependency: a field in a virtual device message may be dependent on another field. Guests communicate with virtual devices through virtual device messages. Each virtual device message follows a given *message structure* and encodes *message fields* that have different semantics, e.g., a four-byte scalar or a pointer. Particularly, a field may be dependent on another field. For example, a bit in a data field may tell a virtual device the type of pointer field. Virtual device fuzzers unaware of the dependencies are slower in reaching certain code or may even miss critical functionalities.

Inter-Message Dependency: a message may depend on a previously issued message. A virtual device message may modify the internal state of a virtual device and can be chained to form a sequence of complex interactions. In a virtual device, two messages might go through different paths but are entangled by the device-internal state, which implies an ordered sequence of messages. Mutators unaware of these dependencies may violate order constraints, which wastes time and hardware resources.

Existing solutions have shown the importance of virtual device fuzzing but were unaware of these two challenges and suffered from either limited scalability or efficiency.

Scalability. A scalable virtual device fuzzer requires low manual effort and a flexible system design to support different hypervisors, architectures, and virtual device categories. NYX [9] introduces two configurations based on the same frontend: NYX-LEGACY and NYX-SPEC (If an argument applies to both NYX-LEGACY and NYX-SPEC, we refer to NYX). Compared to the former that only involves regular PIO/MMIO operations, the latter adds a manually encoded data structure to support object manipulation, e.g., adding allocation of linked lists for XHCI. Specifically, when handling guest-provided data through DMA channels, NYX-SPEC requires human effort to encode the specification of a given virtual device, achieving high code coverage quickly

for supported targets but involving significant human effort for new virtual devices.

Efficiency. An efficient virtual device fuzzer must quickly explore code coverage. Besides considering specifications to speed up code coverage exploration, NYX, V-SHUTTLE [10], and MORPHUZZ [11] miss other opportunities because they suffer from either too large or too small mutation granularity. In particular, NYX uses a generic graph that encodes multiple virtual device messages in each node. Since it avoids crossover mutation among nodes, this approach limits NYX in exploring more interleaved behaviors. V-SHUTTLE and MORPHUZZ, instead, mutate bytes randomly, thus violating the semantics of two consecutive messages since they ignore their structural representation.

Our goal is to overcome the two challenges and achieve both *scalability and efficiency* in fuzzing virtual devices based on two observations. First, we notice that source code encodes message semantics, serving as a reference for message structures. Widely-used hypervisors (QEMU and VirtualBox) are open-source, encoding abundant information about how to interact with virtual devices. Moreover, source code is amenable to automatic analysis and inherently less labor-intensive to validate than complex specifications. Second, well-formed messages exercise more coverage and provide better feedback to the fuzzer for future mutations.

Our Approach. We introduce a new dependency-aware virtual device fuzzing framework VIDEZZO (Virtual Device Fuzzer), which considers both intra-message and inter-message dependencies.

Lightweight Intra-Message Annotation. To support intra-message dependencies, we design a novel and lightweight descriptive grammar (Section 3.1). When reviewing the source code, a security analyst of a virtual device may record intra-message annotation with our descriptive grammar to allow a fuzzer to know how to handle intra-message dependencies. We argue that our lightweight grammar is a good trade-off between the full grammar implementation from the hardware specification used in NYX-SPEC and the heuristic-based approach used in V-SHUTTLE and MORPHUZZ. We semi-automate the annotation extraction. Low manual effort here supports the scalability.

Novel Inter-Message Mutators. To handle inter-message dependencies, we design three new categories of mutators based on a virtual device message as a mutation atom. These mutators create a single message or form message sequences leveraging the genetic nature of fuzzers to provide consistency (message-level), diversity (sequence-level), and semantics (group-level) (Section 3.2). These message-aware mutators not only self-learn the inter-message dependencies but also keep the advantages of different mutation granularity.

Based on the above two techniques, we present the design of VIDEZZO in Section 4. VIDEZZO has two parts: VIDEZZO-CORE and VIDEZZO-VMM bindings. The former manages fuzzing input, parses it into virtual device messages, and processes these messages according to our design. The latter, VIDEZZO-VMM, registers targeted virtual devices, initializes the guest VMM without running any operating system, and dispatches VMM-specific messages. VIDEZZO-CORE is VMM-agnostic, while VIDEZZO-VMM requires customization for each new VMM. The flexible system design enables the scalability of VIDEZZO.

Importantly, VIDEZZO-CORE enables persistent mode, avoiding a heavy fork server to improve performance. We leverage reflective delta-debugging to address side effects due to the accumulated internal state. Specifically, VIDEZZO stores all intermediate test cases and supports delta debugging [12] to reduce the collected seeds to a minimal stable Proof of Concept (PoC).

Compared to previous work, VIDEZZO is both scalable and efficient. VIDEZZO currently supports two hypervisors, i.e., QEMU and VirtualBox, four architectures, i.e., i386, x86_64, AArch32, AArch64, 28 virtual devices in five device categories, i.e., USB, net, display, audio, and storage, and reaches competitive coverage faster. VIDEZZO is also effective in finding bugs. We successfully *reproduced 24 existing bugs and found 28 new bugs* across diverse bug types with 1 CVE assigned so far. We have been actively engaging with the QEMU and VirtualBox communities and provided 7 accepted patches.

Contributions. VIDEZZO's main contributions are:

- Design of a new scalable, efficient, and dependency-aware virtual device fuzzing framework that fully explores intra- and inter-message dependencies.
- A descriptive grammar to encode intra-message dependencies and three new categories of message-aware mutators for inter-message dependencies, boosting virtual device coverage and fuzzing speed.
- Instruction of persistent fuzzing for virtual devices while handling aggregate state through reflective delta-debugging.
- Evaluation of VIDEZZO against the state-of-the-art. In addition to deep coverage, we have discovered 28 new bugs, together with 7 accepted patches.

2. Background and Motivation

Hardware interacts with software through hardware registers and interrupts. The hardware registers are mapped to PIO or/and MMIO space, allowing software to control the hardware via PIO/MMIO read/write operations, i.e., `in/out` and `load/store`. Interrupts, in reverse, are signals from the hardware that inform the software of asynchronous tasks requiring attention. A heavy asynchronous task is to transfer a large chunk of data between the hardware and the main memory through DMA channels.

Virtual devices implement I/O requests. Specifically, the hypervisor intercepts PIO and MMIO operations from the guest and forwards their requests to predefined callbacks in the virtual device. For instance, the MMIO write operation of an EHCI register is redirected into the callback `ehci_opreg_write()` in QEMU, as shown in Figure 1. Next, the control flow goes to different specialized handlers based on the value of `addr`. Importantly, the hidden implementation definitely encodes the information about how hardware works, which is helpful for virtual device fuzzing. PIO and MMIO operations follow protocols to drive virtual devices and thus can be called *virtual device messages*.

Messages	Description
PIO ops	PIO read/write operations
MMIO ops	MMIO read/write operations
Memory ops	Memory allocation, read/write, or free
Clock ops	Time adjustments

TABLE 1: Virtual device messages and their descriptions.

2.1. Virtual Device Messages

A virtual device message, e.g., an MMIO write operation, defines how a guest communicates with a virtual device. As shown in Table 1, besides PIO and MMIO messages, virtual device messages also include memory-related operations to manipulate the main memory space, and clock-related messages to adjust the current time [11].

Messages define their own *message structure* and *message fields*. Figure 2 shows the structure of an MMIO write message with four fields: a type field `E_TYPE` and three parameter fields: `ADDR`, `SIZE`, and `VALUE`. Moreover, `VALUE` contains formatted sub-fields with additional information. Multiple messages compose *message sequences* that must follow a specific order.

Leveraging virtual device messages for fuzzing brings three key benefits. First, with clear formats, a fuzzer has a precise view of inputs and thus can support fine-grained dependencies. Second, virtual device messages allow a fuzzer to build up the internal state of a virtual device by manipulating the order of message sequences. Third, a fuzzer can minimize a PoC by removing unnecessary virtual device messages to find interesting message sequences.

2.2. Motivation and Challenges

Prior work [9], [10], [11] demonstrates that coverage-based fuzzers are applicable to virtual devices. However, two challenges were overlooked from the perspective of virtual device messages: *intra-* and *inter-message dependencies*.

Intra-message Dependencies. A virtual device message usually contains multiple fields. These fields may be dependent on each other. This is usually true when virtual devices heavily interact with main memory through DMA channels [10] that can process large blocks of data. For example, Figure 3 shows a loading into `tx` at line 7. The switch at line 8 then decides the type of `array_addr` by checking the first three bits in `command`. Awareness of such dependencies reduces the search space (we only mutate the first three bits of `command`), which precludes random guessing from the fuzzer side to satisfy complicated constraints (we know `array_addr` is either a `MacAddr` or a `TxConfig`).

Inter-message Dependencies. Virtual device messages can be chained, building up complex virtual device state. Due to the narrow set of registers, interactions with devices often require multiple interactions. Figure 4 demonstrates how a specific sequence of messages triggers the code at line 10.

```
1 void ehci_opreg_write(
2     physaddr addr, uint64_t val, uint32_t size) {
3   switch (addr) {
4     case USBCMD: // do something
5     case PERIODICLISTBASE: // do something
6     case ASYNCLISTADDR: // do something
```

Figure 1: An example of I/O callback in a virtual device.

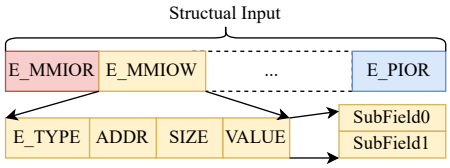


Figure 2: Virtual device messages and a structural input.

```
1  typedef struct {
2    uint32_t command; uint32_t array_addr; } tx_t;
3  void action_command(physaddr addr) {
4    tx_t tx;
5    MacAddr macaddr;
6    TxConfig config;
7    dma_read(/*addr=*/addr, /*dst=*/&tx);
8    switch (tx.command & COMMAND/*=7*/) {
9    case CmdIASetup/*=1*/:
10     dma_read(tx.array_addr, &macaddr); break;
11   case CmdConfigure/*=2*/:
12     dma_read(tx.array_addr, &config); break;
```

Figure 3: Field `array_addr` can point to different buffers when flag bits in field `command` are different.

2.3. Threat Model

We share the same threat model as previous work [9], [10], [11]. Specifically, the attacker creates a virtual machine in a cloud and controls the operating system. The attacker's goal is to compromise the hypervisor and take over other virtual machines. This setup is realistic since cloud providers grant tenants full control over the operating system of the created virtual machines.

3. Dependency-Aware Message Model

Our dependency-aware message model addresses both intra- and inter-message dependencies. Intra-message dependencies are best covered by a lightweight descriptive grammar (Section 3.1) and inter-message dependencies through a set of specialized message mutators (Section 3.2).

3.1. Intra-Message Annotation

For *intra-message dependencies*, we propose a novel lightweight descriptive grammar, which is exemplified in Figure 5. VIDEZZO uses the grammar to generate virtual device messages that satisfy intra-message dependencies. Since the annotation is device dependent, we develop an annotation inference engine that extracts specifications from the virtual device source code (Section 5.1).

Unlike the Syscall description language (Syzlang) [13], each virtual device follows ad-hoc protocols built on top of simple interfaces (e.g., only four types of messages in Table 1), which can be easily modeled by a generic grammar. To the best of our knowledge, we are the first to apply this technique to the virtual device domain. Unlike NYX-SPEC,

```
1  typedef GlobalState {
2    uint32 internal_state; } GlobalState;
3  GlobalState gs;
4  void mmio_write_dword(
5      physaddr addr, uint64_t val) {
6    switch (addr) {
7      case 0x0:
8        gs->internal_state = val; break;
9      case 0x4:
10       if (!gs->internal_state) break;
11       // do something and break
```

Figure 4: A specific ordered message sequence, i.e., {addr: 0x0, val: rand()}, {addr: 0x4, val: rand()}, triggers the code at line 10.

analysts do not require intricate knowledge of the complex hardware-level specification. Compared to specification-agnostic fuzzers, such as V-SHUTTLE or MORPHUZZ, our lightweight grammar results in higher quality seeds.

We base our grammar on the analysis of 18 devices across five categories. The grammar relies on a small type system, APIs, and statement rules. The type system and the APIs cover three requirements, i.e., field-awareness, bit-awareness, and context-awareness. The statement rules define how to develop annotations.

Type System. The grammar has a type system to reflect the field- and bit-awareness of a virtual device message, which is first introduced systematically in our study.

Field-Awareness. A virtual device message is *field-aware* if it defines boundaries and types for its fields and sub-fields. Specifically, sub-fields are sub-components of data fields, as shown in Figure 2. Sub-fields should be considered standalone and can either contain data or pointers. The value of a data field can be random or constant; while pointer fields chain nested objects [10]. For instance, in Figure 3, `command` and `array_addr` are both four bytes. The former is a data field and the latter is a pointer field. Our grammar allows us to explicitly annotate this information.

Bit-Awareness. The fields of a virtual message are *bit-aware* if they define constraints at bit granularity. For instance, some bits of data fields are used as flags, as shown on line 8 in Figure 3. Likewise, pointers often include extra information in some, e.g., lower, bits as tags.

This grammar (from line 2) first defines four basic field types with an orthogonal symbol each, i.e., `RANDOM` for a variable with a random value, `CONSTANT` for a constant, `POINTER` for a pointer, and `FLAG` for a variable with flag bits. Due to the orthogonality, a tagged pointer can be `POINTER | FLAG`. The grammar then defines a `FIELD` that is decided by its field name and field size. For example, `FIELD command` is `command#0x4`. Next, it defines how to express flag bits, i.e., `FLAG_LEN_PAIR`. For example, `command`'s first three bits are `0: 3@7`, if the initial value of these three bits is 7. In general, `0: 3` means the initial value of these bits is random.

APIs and Context-Awareness. Our grammar provides APIs to annotate intra-message dependencies. `add_struct()` defines each field in a virtual device message, `add_flag()` defines the flag bits, and `add_constant()` defines the candidate value set for a constant field. Furthermore, we can define

```
1  // type system
2  FIELD_TYPE: RANDOM | CONSTANT | POINTER | FLAG
3
4  FIELDNAME: NAME
5  FIELD : FIELDNAME '#' SIZE
6  typedef uint8_t  BEGIN
7  typedef uint8_t  LENGTH
8  typedef uint32_t INITVALUE
9  FLAG_LEN_PAIR: BEGIN ':' LENGTH [ '@' INITVALUE ]
10
11 // APIs
12 STRUCTNAME: NAME
13 STRUCT_SET: '[' STRUCTNAME+ ']'
14 FIELD_INDEX : STRUCTNAME '.' FIELDNAME
15 FIELD_TYPE_PAIR: FIELD ':' FIELD_TYPE
16 FIELD_SET: '{' FIELD_TYPE_PAIR+ '}'
17 FLAG_SET: '{' FLAG_LEN_PAIR+ '}'
18 CONDIDATES: '[' uint32_t+ ']'
19 POINT_TO_SET: '[' FIELD_INDEX+ ']'
20 CONDITION: FIELD_INDEX '.' BEGIN
21 CONDITION_SET: '[' CONDITION+ ']'
22
23 def add_struct(
24   name :-> STRUCTNAME, fields :-> FIELD_SET)
25 def add_flag(
26   field :-> FIELD_INDEX, flags :-> FLAG_SET)
27 def add_constant
28   field :-> FIELD_INDEX,
29   condidates :-> CONDIDATES)
30
31 def add_head(structs :-> STRUCT_SET)
32 def add_point_to(
33   field :-> FIELD_INDEX,
34   point_to :-> POINT_TO_SET,
35   condition :-> CONDITION_SET,
36   ALIGNMENT :-> uint8_t)
37
38 def add_point_to_linked_list(
39   head :-> FIELD_INDEX, tail :-> FIELD_INDEX,
40   point_to :-> POINT_TO_SET, links :-> FIELD_SET,
41   condition :-> CONDITION_SET,
42   ALIGNMENT :-> uint8_t)
43
44 // statements and programming model
45 MODELNAME: NAME
46 model: 'Model(' MODELNAME ',' MODELID ')'
47 api_add_field:
48   add_flag | add_constant | add_point_to
49 statement: add_struct add_field+
50 statements: model statement+ add_head
```

Figure 5: The grammar for intra-message dependencies.

tree-like objects (where each node contains a link pointer to the next node [10]) through `add_head()` to define the head (root) object and `add_point_to()` to define the descendants.

In addition to the aforementioned rules, we identify three novel types of context-aware dependencies. Specifically, a virtual device message may be *context-aware*, i.e., fields in virtual devices have specific dependencies. We identify three *context-aware* scenarios as explained in the following.

Head-Tail-Pointers Context. A pointer can point to a single object or a linked list of objects. Linked lists may require an additional pointer field to indicate the list's tail (Appendix Figure 14), which is reasonable since some virtual devices can handle a sequence (or ring) of commands in one message. We support this through an API

```
vd0 = Model('tx', 0)
vd0.add_head(['tx_t'])
vd0.add_struct('tx_t', {
  'command#0x4': 'FLAG',
  'array_addr#0x4': 'POINTER'})
vd0.add_flag('tx_t.command', {0: 3})
vd0.add_point_to('tx_t.array_addr',
  [None, 'macaddr', 'config', None, None, None,
//if 0       1          2       3     4     5
  None, None], condition=['tx_t.command.0'])
//  6     7             == tx_t.command.0
```

Figure 6: Intra-message annotation with our grammar. The value of `tx_t.command.0` decides `tx_t.array_addr`'s type.

named `add_point_to_linked_list()`, which is similar to `add_point_to()` but explicitly defines a linked list.

Flag/Tag-Pointer Context. Flag bits in a field may affect the fuzzing execution paths and the corresponding pointers have different types. Both data and pointer fields can have flag bits. This dependency starts from a bit-wise data field and ends with a multi-typed pointer field. Figure 3 shows an example. To support this dependency, we add a new parameter `condition :-> CONDITION_SET` to `add_point_to()` and `add_point_to_linked_list()`. This parameter shows which bits will decide the type of the corresponding pointer field.

Len-Buffer Context. A random data field may have specific semantics to indicate how long a buffer is. This dependency is required only when the virtual device checks the length of a buffer like a checksum algorithm. Appendix Figure 15 shows an example. To support this dependency, we adjust the type of the data field from `RANDOM` to `CONSTANT` and put the length of the buffer to the candidate value list.

Our grammar may be extended as long as backward compatibility is kept. Appendix A.3 lists additional intra-message dependencies.

Statements and Programming Model. Besides the type system and APIs, we introduce a programming model (from line 45) for analysts to develop the intra-message annotation. Each virtual device is encoded with a or multiple `Models` that wrap the intra-message dependencies. Analysts can extend a `Model` by including a head object (`add_head()`), structures (`add_struct()`), and fields (`add_field()`). In this way, a buffer tree is built to support field-awareness, bit-awareness, and context-awareness. Figure 6 shows the intra-message annotation for Figure 3.

Fully automating the annotation extraction is challenging. Our partial annotation inference vastly reduces human effort, which is a more scalable solution to support new virtual devices than manual inspection of the device specifications. We summarize the implementation in Section 5.1 and the uncertainty of the automation in Section 6.1.

3.2. Inter-Message Mutators

To tackle *inter-message dependencies*, we propose three new categories of message mutators that infer inter-message dependencies by composing messages and sequences of messages (Table 2). Our automatic approach overcomes tedious manual tracing and modeling [14].

Our mutators provide consistency (message-level), diversity (sequence-level), and semantics (group-level). To the best of our knowledge, we are the first to apply this technique to the virtual device domain. Our mutator design follows this intuition: once the fuzzing engine finds an interesting seed, it keeps the seed in the corpus, and consistently mutates it in the next fuzzing loop. In other words, the inter-message dependencies are not specifically defined but are gradually learned. We detail the mutators in the following.

Message-Level Mutators. We define six message-level message mutators that modify the content of a message, which ensures consistency between the original and the mutated message. For instance, the change of MMIO address can generate a sequential visit to a virtual device. Specifically, we define five primary mutators (ID 1–5) and one extensive mutator (ID 6). Mutators 1–3 modify each parameter within a message to keep the message well-balanced. Mutators 4–5 randomly delete and insert a message to adjust the length of the input. The extensive mutator (ID 6) adds several repeated messages. We observed that ID 6 would increase or decrease variables in a virtual device, allowing the fuzzer to bypass boundary checks.

Sequence-Level Mutators. We define six sequence-level message mutators that add or delete messages in a sequence. Along with local modifications via message-level mutators, this causes a higher probability of dramatic changes across message sequences to bypass hard checks [15]. Specifically, Mutators 7–9 update messages within a sequence, and mutators 10–12 handle sequences.

Group Mutator. We define the new "group mutator" category, whose purpose is to cluster dependent messages into a so-called `GROUP_MESSAGE`. Grouped messages remain intact during subsequent mutation steps. These mutators are built on top of a *trigger-action protocol*, which relies on *feedback* (trigger) from the virtual device and a *handler* (action) to decide whether to group the messages. Section 4.3 details our two types of group mutators.

4. VIDEZZO System Design

VIDEZZO (Figure 8) consists of two components: VIDEZZO-CORE, which is VMM-agnostic and provides inputs consisting of virtual device messages, and VIDEZZO-VMM, which is VMM-specific and interacts with the virtual

ID	Lvl	Mutators	Description
1	ML	ChangeValue	Mutate the value of a message
2		ChangeAddr	Mutate the address of a message
3		ChangeSize	Mutate the size if not fixed
4		EraseMessage	Randomly erase a message
5		InsertMessage	Insert one new message
6		InsertRptdMessage	Insert multple new messages
7	SL	ShuffleMessages	Shuffle a sequence of messages
8		CopyPartOfSequence	Copy messages from a sequence to another
9		CrossOverSequence	Exchange messages between two sequences
10		EraseSequence	Randomly erase part of a sequence
11		InsertSequence	Insert a sequence randomly
12		ShuffleSequence	Shuffle all messages in a sequence
13	GL	GroupMessage	Group messages for future re-use

TABLE 2: Multi-level mutators and their description. Legend: **ML**: Message-level, **SL**: Sequence-level, **GL**: Group-level.

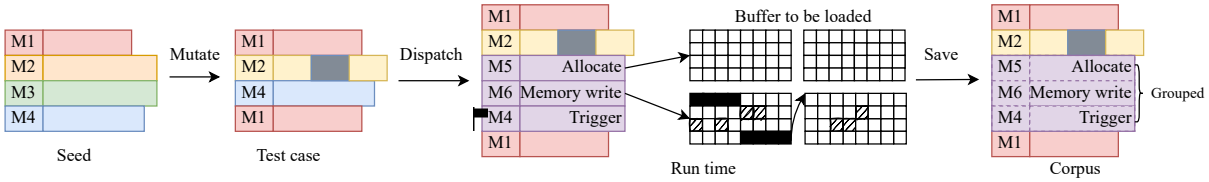


Figure 7: The workflow of ViDeZZo in a mutation-based greybox fuzzer. M: Message.

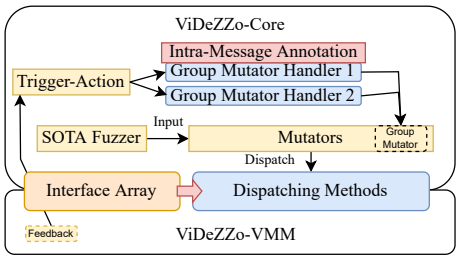


Figure 8: System design of ViDeZZo.

devices. Specifically, first, VIDEZZO picks an input, i.e., a message sequence, from the queue and passes it to message- and sequence-level mutators (Section 4.1). Next, to dispatch the messages, VIDEZZO obtains the real physical addresses from the shared testing interface array and invokes the corresponding dispatching methods (Section 4.2). Then, VIDEZZO uses feedback from the virtual device to exercise a *trigger-action protocol*. Following this protocol, the group mutators cluster messages (Section 4.3) that, e.g., are generated from the *intra-message annotation* (Section 4.4).

Figure 7 shows the fuzzing loop workflow. In the beginning, the fuzzer selects a seed from a corpus. If the corpus is empty, the first seed is empty. In the example, the seed consists of four messages, Messages 1–4. We hide their internal formats for simplicity. Then, the seed is mutated into a test case: Message 1 is kept; a parameter marked in the dark grey area of Message 2 is updated to another value (Mutator 1, 2, or 3); Message 3 is deleted (Mutator 4); Message 4 is kept; another Message 1 with different parameters is added (Mutator 5). After seed mutation, the fuzzer sequentially dispatches the messages to the target virtual device. In the example, Message 1 and Message 2 are dispatched first. Next, a *trigger* informs the fuzzer that Message 4 is about to load data from the main memory. Followed by an *action*, the fuzzer generates and injects Messages 5 and 6 to set up the main memory before Message 4 consumes them. Note that the injected message and the trigger message are locked and mutated as a group in the next fuzzing loop (Mutator 13). At this point, the fuzzer dispatches the last Message 1 and finishes exercising the test case. Finally, if the test case discovers new coverage, the fuzzer saves the mutated seed into the corpus and repeats the loop.

4.1. Input Parsing and Mutations

VIDEZZO-CORE parses a byte-array seed into message sequences through serialization and deserialization. Next, our message- and sequence-level mutators generate a new input. Section 5.2 shows the implementation.

4.2. Interface and Dispatching Methods

After the mutation (message- and sequence-level mutators), the fuzzer dispatches each virtual device message by invoking the corresponding VMM-specific dispatch methods. Specifically, VIDEZZO-CORE defines a set of high-level dispatch methods that are instantiated in VIDEZZO-VMM. This method-based abstraction allows VIDEZZO to read and write the internal state of a virtual device, facilitating VIDEZZO to scale to more virtual device categories, architectures, and hypervisors.

This mechanism uses testing interface arrays to define what messages are allowed and how they are dispatched. For instance, if an interface of a virtual device describes a four-byte aligned MMIO area from `0xe0001000` to `0xe0001200`, the guest is allowed to issue MMIO read and write messages within that range. Note that requests must be four-byte aligned. Several testing interfaces form the testing interface array that is defined in VIDEZZO-CORE and instantiated in VIDEZZO-VMM. The array includes dynamic interfaces and predefined interfaces that are described below.

Dynamic Interface. For each fuzzing instance, VIDEZZO fuzzes one virtual device, which improves the utilization of virtual device messages. For the target virtual device, VIDEZZO finds and extracts its memory regions, which define PIO or MMIO memory spaces with the information of the starting address, size, and alignment, which is feasible since each VMM sets up a well-defined struct when registering a virtual device. Obtaining this information via VMM-specific APIs needs predefined interface signatures. For example, the EHCI capability memory region is associated with the string "capabilities" that serves as a signature.

Predefined Interfaces. VIDEZZO defines several interfaces that cannot be identified automatically, such as memory allocation, read and write, and free interfaces. Specifically, except for PIO and MMIO related interfaces, we encode the interfaces for memory and clock-related messages. The number of these fixed interfaces is limited, and once defined, VIDEZZO uses them across different targets.

We associate each virtual device message with a given interface through an extra `interface_id` field. When dispatching a message, the dispatcher looks up the proper interface via `interface_id` from the testing interface array, adjusts the target address, aligns the data, and then invokes the corresponding dispatching methods.

4.3. Group Mutator

VIDEZZO proposes two group mutators: Load Miss Mutator and Record Start-End Mutator. Group mutators are instantiated over a newly proposed trigger-action protocol.

The protocol has a well-defined trigger (feedback) and an action (handler) attached to the trigger. This protocol behaves as an interrupt service routine and enables a context switch between VIDEZZO-VMM and VIDEZZO-CORE. We distinguish three phases. First, when feedback is hit, the fuzzer stops the current execution and goes to the handler. Second, the handler groups related messages into a `GROUP_MESSAGE`. Finally, the handler returns and the fuzzer resumes the driver execution.

Load Miss Mutator. Some virtual devices expect groups of messages linked to each other [10]. We introduce the *Load Miss Mutator* to handle this condition. Figure 7 shows an example: after Message 4 is issued, a `pci_dma_read()` is invoked. However, VIDEZZO must inject Messages 5 and 6 to satisfy the intra-message dependencies, otherwise, the virtual device will read meaningless information.

For the *feedback*, we intercept the `load` instruction when the virtual device attempts to load data from guest memory. We call this feedback *Load Miss*. For the *action*, the mutator implements a handler, named *Load Miss Handler* that requires the destination address of `load`. In the handler, VIDEZZO generates a set of messages linked to the original one (Section 4.4).

Record Start-End Mutator. Some virtual devices require a list of messages in a given order, as shown at line 10 in Figure 4. To address this requirement, we introduce a *Record Start-End Mutator*, that records ordered messages sharing variables, e.g., `gs->internal_state`.

We define two points of *feedback*: start and end. The *start* feedback is triggered when the shared variable is written, and the *end* feedback is when the shared variable is used. On *start*, the group mutator will record all following messages until the *end* is triggered. As an *action*, VIDEZZO groups the recorded messages and keeps them intact.

Note that group mutators work together with the other two levels of mutators to learn inter-message dependencies automatically. Currently, we support two group mutators. Nevertheless, a developer can implement other feedback and handlers on top of our trigger-action protocol. For example, a group mutator could group a set of fixed virtual device messages to reset a virtual device.

4.4. Intra-Message Annotation to Messages

Algorithm 1 shows how to leverage the intra-message annotation to generate virtual device messages to support intra-message dependencies. For better understanding, we define three high-level conceptual messages: *alloc*, *fillup*, and *free*. Each of them consists of one or several basic messages defined in Table 1.

The construction starts from the head object (line 17–20), which invokes `Alloc()`, `Fillup()`, and `Free()` in order. `Alloc()` (line 1–2) appends the virtual device message `memalloc` (e.g., Message 5 in Figure 7) to the message set `M`. Moreover, `Alloc()` allocates a buffer in the guest memory. `Fillup()` (line 3–14) adds `memwrite` (e.g., Message 6 in Figure 7) to write a specific or random value to each field in the head object. If a field is a flag or a pointer, `Fillup()` sets its value according to its annotation. For example, a flag, whose flag-len-pairs are $0:16, 16:16$, has the value `(rand() && 0xff) | ((rand() && 0xff)` $\ll$ `16)`. If a field is a pointer, `Fillup()` obtains its point-to objects, checks if there exist conditional fields to be referenced, and invokes `Alloc()` and `Fillup()` recursively to generate a normal object or a linked list. After the allocation and the fillup, `Free()` (line 15–16) frees the head object. Note that, to simplify the algorithm, we assume that no-longer-used objects are freed to avoid running out of guest memory.

Algorithm 1: Annotation-to-Message Construction

```
   Input: Intra-Message Annotation D
   Input: Message memalloc, memread, memwrite, memfree
   Result: Messages M = {m_1, m_2, ..., m_n}
1  Function Alloc(Size):
2      M.append (memalloc (Size));
3  Function Fillup(Object):
4      foreach Field ← Object.Fields do
5          Metadata ← Field.Metadata
6          switch Field.Type do
7              case FLAG do
8                  M.append(memwrite(gen_flag(Matadata)));
                   break;
9              case CONSTANT do
10                 M.append(memwrite(Field.Value));
                   break;
11             case RANDOM do
12                 M.append(memwrite(rand())); break;
13             case POINTER do
14                 M.append(memwrite(gen_pointer(Matadata)));
                   break;
15 Function Free(Addres):
16     M.append(memfree(Address));
17 Head ← D.Head
18 HeadAddress ← Alloc (sizeof (Head));
19 Fillup (Head);
20 Free (HeadAddress);
```

5. Implementation

We implement the automated annotation extraction with 364 lines of CodeQL, intra-message annotation with 812 lines of Python, VIDEZZO-CORE with 1,855 lines of C and 518 lines of Python, VIDEZZO-QEMU with 1,503 lines of C, and VIDEZZO-VIRTUALBOX with 1,165 lines of C. VIDEZZO supports the latest QEMU and VirtualBox. We release our tool at https://github.com/HexHive/ViDeZZo. In the following, we discuss specific VIDEZZO modules.

5.1. Semi-Automatic Intra-message Annotation

To aid analysts, we introduce an inference engine for intra-message dependencies. Our tool generates a standalone text file with a baseline annotation for the field- and bit-awareness. An analyst can refine the baseline later. We develop the engine based on CODEQL, which integrates data-flow and taint analysis.

Field-awareness. Starting from a DMA read access, e.g., `pci_dma_read()`, our tool extracts the struct definition of

the destination buffer. For each field in the struct, the tool checks if it is a pointer by checking whether the field flows into DMA accesses code like `pci_dma_[read|write]()` via taint analysis. If the field is a pointer, the tool further obtains its type when available. In few cases, virtual devices use byte arrays and do not use any casting for the destination buffers, which forbids explicit struct definitions. This struct reconstruction needs extra models to map the real struct definition to arrays. Instead of a heavy model, we develop a Python script to split the small byte array (less than or equal to 32 bytes according to our experience) into four byte chunks and mark each chunk as data.

Bit-awareness. Starting from the struct definition obtained above, our tool traverses all binary operations with the access of a struct field as an operand. The tool parses the binary operations and identifies which bits should be considered.

5.2. VIDEZZO-CORE

VIDEZZO-CORE is VMM-agnostic and is in charge of decoding and encoding input and mutating virtual device messages. We base our VIDEZZO-CORE implementation on libFuzzer. Supporting an alternative frontend (e.g., AFL++) remains a future engineering extension.

Input Parser. In Table 3, we list the format of all the messages defined in Table 1. Each message contains two bytes that define the message type and its interface. Usually, `addr` is eight bytes and `size` is four bytes. As for `value`, most are eight bytes, but for `MEM_[READ/WRITE]`, the size of `value` depends on the dedicated `size` field. The reason for this implementation is to pre-allocate a large buffer and reduce memory allocation overheads. Finally, to decode and encode an input, we implement a serializer and a deserializer through `next_##size##_bytes()` helpers.

Message-level and Sequence-level Mutators. We implement our mutators in C by hooking the `LLVMFuzzerCustomMuatator()` in libFuzzer. First, we deserialize the binary input into messages and directly operate on them. After the mutation, we serialize the messages to a new binary input. Since we perform de/serialization only once for each input, the mutation operation is cheap.

Group Mutator. The trigger-action protocol has two parts. The first part is the trigger instrumentation in the target virtual device, and the second part is the implementation of the corresponding action handlers in VIDEZZO-CORE. For the first part, we build a Clang-based instrumentation pass that leverages the annotations of the security analyst in the target virtual device. For the second part, an analyst should develop their own handlers.

Figure 9 shows the implementation of the *Load Miss Mutator*. The *Load Miss* is detected by hooking `pci_dma_read()`

Message Type	TYPE	INTERFACE_ID	ADDR	SIZE	VALUE
[MMIO/PIO]_READ	1	1	8	4	—
[MMIO/PIO]_WRITE	1	1	8	4	8
MEM_[READ/WRITE]	1	1	8	4	SIZE
MEM_[ALLOC/FREE]	1	1	—	—	8
CLOCK_STEP	1	1	—	—	8

TABLE 3: Bytes of each field in each message.

```
static int __wrap_pci_dma_read(
    uint32_t addr, void *buf, size_t size) {
  /*handler*/LoadMissHandler(addr);
  return REAL(pci_dma_read)(addr, buf, size);
}

void ehci_state_fetchqh(EHCIState *ehci) {
  EHCIqh qh;
  /*feedback*/WARP(pci_dma_read)(
    addr, &qh, sizeof(EHCIqh));
```

Figure 9: Instrumentation of *Load Miss Mutator*. We use pci_dma_read() to be consistent with the paper jargon, while we instrument get_dword() in the implementation.

```
void mmio_write_dword(
    physaddr addr, uint64_t val) {
  switch (addr) {
    case 0x0:
      gs->internal_state = val; break;
      /*feedback*/Record(0, /*mode=*/"start");
    case 0x4:
      if (!gs->internal_state) break;
      /*feedback*/Record(0, /*mode=*/"end");
      // do something and break
```

Figure 10: Instrumentation of *Record Start-End Mutator*.

in QEMU and `PDMDevHlpPCIPhysRead()` in VirtualBox. The address of the destination buffer is passed to the *Load Miss Handler*. This handler is VMM-agnostic and implemented in VIDEZZO-CORE. Finally, we resume the control flow to the virtual device.

Figure 10 shows the implementation of the *Record Start-End Mutator*. We introduce an API named `Record(int id, char *mode)` whose first argument is a unique identifier, the second parameter indicates whether to start or end the recording. The `Record()` functions traces all the messages and groups them when the record ends.

In our prototype, all feedback is visited by automatically generated virtual device messages. An alternative strategy would require carefully manually crafted messages. However, we consider this approach infeasible because a virtual device might require a long list of messages that fulfill complex inter-message dependencies. Instead, our *trigger-action protocol* avoids this challenge and simplifies both design and implementation. Furthermore, based on our protocol, the saved crashing test case is complete and does not require regeneration [10] or message reordering [11].

Intra-Message Annotation to Messages. Following Algorithm 1, we implement a grammar interpreter in Python, which translates the encoded descriptions automatically into low-level message generators in C code. These generators are then compiled together with the VIDEZZO-CORE.

Persistent Fuzzing. Restarting virtual machines is a costly heavy-weight operation and limits the performance of a fuzzer. To overcome this limitation, we implement an in-process persistent fuzzer. However, persistent fuzzers accumulate previous states when executing test cases. The accumulation may cause unreproducible crashes as a bug may result from multiple previous test cases. Therefore, we

modify libFuzzer to optionally record *all intermediate test cases* (similar to Syzkaller). We then implement a delta-debugging approach with Picire [12] to infer the test cases responsible for the bug, and with the remaining test cases, the virtual device messages responsible for the same bug. Delta-debugging performs a binary search on sequential seeds and messages to establish the minimum crashing seed, reducing the effort to analyze messages. Based on our experience, minimized seeds are usually less than 80 messages (see last column in Appendix Table 10).

5.3. VIDEZZO-VMM

VIDEZZO-VMM is VMM-specific, however, it follows a fixed template. VIDEZZO-VMM has four steps: (1) register the targeted virtual devices, (2) initialize the VM, (3) set up the shared interface array with real physical addresses, and (4) implement VMM-specific dispatching methods. The summarized template highlights necessary knowledge to scale to a new VMM and relative virtual devices, avoiding the developer getting lost in the huge code space of hypervisors. Currently, VIDEZZO supports QEMU and VirtualBox. In the following, we detail these steps.

Targeted Virtual Device Registration. To register a targeted virtual device, VIDEZZO-VMM provides the corresponding specification: architecture, the launch command line, and the signature of fuzzing interfaces. Most importantly, the launch command lines are target-specific, and the signatures of the testing interfaces depend on the hypervisor. For the specification format, we adjust the existing implementation in QEMUFuzzer [16] by adding a new field `mrnames` as the signatures of testing interfaces. We have added virtual devices covering different categories and architectures for each hypervisor, as the evaluation shows.

VMM Initialization and Interface Identification. To initialize a VMM, we pass the launch command line in the fuzz target specification to the `main()` function of a VMM. Then, to identify the non-predefined testing interface in a virtual device, we scan all registered PIO and MMIO memory regions and choose the matched ones with the interface signatures. Finally, we extract the metadata (e.g., the physical addresses of the chosen PIO or MMIO memory region) and fill the metadata into the shared interface array. We reuse the code in QEMUFuzzer [16] for VIDEZZO-QEMU and implement similar functions for VIDEZZO-VIRTUALBOX.

VMM Specific Message Dispatching Methods. We use the QTest APIs for VIDEZZO-QEMU. QTest is an in-process testing framework based on QEMU Message Protocol (QMP). QTest can access guest memory directly and scales to multiple virtual devices and architectures. We also implement a similar functionality for VIDEZZO-VIRTUALBOX.

6. Evaluation

This section presents the evaluation of VIDEZZO, which we design to answer the following four research questions.

- **RQ1:** What is the overall efficiency of VIDEZZO compared to other virtual device fuzzers?
- **RQ2:** What is the difference in coverage and overhead of VIDEZZO compared to dependency-agnostic fuzzers?
- **RQ3:** What are the advantages of VIDEZZO to discover existing bugs compared to existing tools?
- **RQ4:** How does VIDEZZO behave to discover new bugs?

Fuzzer and Hypervisor Settings. For **RQ1–RQ3**, we port (i) NYX (both NYX-LEGACY and NYX-SPEC), QEMUFUZZER (the "industry" version of MORPHUZZ), and VIDEZZO to QEMU 5.1.0 and (ii) VIDEZZO to VirtualBox 6.1.14. Currently, part of NYX, MORPHUZZ, and V-SHUTTLE (also known as V-SHUTTLE-S) are open-source. NYX and MORPHUZZ support recent QEMU, but they do not support VirtualBox. V-SHUTTLE supports QEMU 5.1.0 and VirtualBox 6.1.14. For **RQ4**, we upgrade VIDEZZO to the latest QEMU and VirtualBox. We compile all hypervisors with Clang; we do not use any starting seeds for each fuzzer.

Virtual Device Fuzzing Settings. For **RQ1 and RQ2**, we select 28 virtual devices, covering five virtual device categories (USB, net, display, audio, and storage), four architectures (i386, x86_64, AArch32, AArch64), and two hypervisors (QEMU and VirtualBox).

Coverage and Bug Detector Settings: For **RQ1 and RQ2**, we rely on Clang source code coverage profiling [17]. Except for Nyx, we use a signal handler to dump live coverage. Specifically, we dump coverage every second to monitor the rapid coverage change over the first 10 minutes and then dump every 10 mins to save space. For Nyx, we re-execute all seeds to collect offline coverage. To remove the interference of early crashes, we disable all sanitizers, remove all asserts and `abort()`, and patch any bugs found in the fuzz targets. Note that we use the same patched QEMU and VirtualBox for all fuzzers. We further dump the coverage when the hypervisor crashes or times out to guarantee reliable coverage collection. For **RQ3 and RQ4**, we disable coverage profiling, enable sanitizers, i.e., ASan and UBSan, and keep all asserts and `abort()` to capture more bugs.

Server Resources: For **RQ1–4**, we conduct all experiments on six servers with hyper-threading disabled, each with 16 Intel Xeon Gold 5218 CPU (2.30GHz) cores, 64GB RAM, and Ubuntu 20.04. We fuzz each virtual device from scratch on one core for 24 hours, and we repeat each experiment ten times for statistical significance [18].

6.1. Expressive Grammar Limits Manual Effort

We execute our inference engine tool (Section 5.1) over 18 QEMU virtual devices (Appendix Table 8). The analysis automatically extracts 93% of struct definitions, determines 80% of the pointers, and finds 70% of the flags. Regarding the missing information, we identify *three* causes intrinsically related to static-analysis limitations.

Unnamed types. CODEQL 2.10.5 cannot infer the definition of unnamed structs/unions. To address this issue, the tool reports the source code location where structs/unions are defined and then relies on manual confirmation. In total, we find *four* unnamed structs and *three* unnamed unions.

State-aware executions. Two disjointed control-flow segments of a virtual device can be connected only when the

virtual device reaches specific internal states, for instance, by exercising two or more messages. Without specific knowledge, CODEQL's taint analysis shows its limitation in propagating information among disjointed control flows. In our prototype, we manually address this issue. Another viable option is to employ type-sensitive algorithms [19] that infer control-flow segments controlled by variables of the same type, e.g., global structures used in switch statements. In total, we observe *six* structs covering *four* virtual devices.

Context-aware dependencies. Static analysis struggles when extracting complex data relations such as linked lists. In these cases, we fall back to manual analysis. During our evaluation, we encounter only five virtual devices (out of 18) falling in this category.

With the support of our inference engine, we model 18 QEMU virtual devices, while NYX-SPEC models only XHCI (out of 16 QEMU virtual devices). Appendix Table 8 details the results. Moreover, to quantify the manual effort, Appendix Table 9 estimates average times for each case for the support of annotation, hypervisor and group mutator.

6.2. Efficiency

We demonstrate that VIDEZZO supports different categories of virtual devices, and reaches competitive coverage compared with state-of-the-art virtual device fuzzers.

Scalability. Table 4 shows that VIDEZZO scales to 28 virtual devices covering five device categories, four architectures, and two hypervisors. The scalability is due to three reasons. First, our flexible design abstracts the fuzzing logic (VIDEZZO-CORE) from VMM implementation (VIDEZZO-VMM), thus simplifying the porting of VIDEZZO-CORE over different virtual devices, device categories, architectures, and hypervisors. Second, our lightweight annotation allows adapting new virtual devices to VIDEZZO. Third, virtual devices already annotated can be tested over different hypervisors, thus further reducing the porting effort.

Final Coverage. For rows in Table 4 with two or more colored numbers, i.e., where we replicated related work, we observe VIDEZZO reaches competitive (8/22) or even higher (14/22) coverage.

Nyx. NYX-LEGACY works better for storage devices, while VIDEZZO works better for audio, network, and USB devices. VIDEZZO works better than NYX-SPEC in 24 hours.

V-Shuttle. VIDEZZO works better for USB devices than V-SHUTTLE. Regarding the measured results, V-SHUTTLE (without seeds) cannot reach coverage as high as other tools due to two reasons. First, the coverage is influenced by the initial corpus. The authors of V-SHUTTLE also mentioned that initial seeds can *"improve the fuzzing efficiency further"*. V-SHUTTLE uses seeds collected when the BIOS and the guest kernel initialize the virtual device. We speculate those seeds encode *intra-message dependencies* that improve the final coverage and speed. Conversely, we did not use initial seeds when reproducing V-SHUTTLE results. The comparison between VIDEZZO and V-SHUTTLE indicates that VIDEZZO produces higher quality seeds that cover more interesting test cases autonomously. Second, the authors of V-SHUTTLE confirm that they manually performed actions of storing and loading a VMM, thus adding and deleting a device to cover code that otherwise would not be reached, which implies that the final results reported in their paper are probably higher than those without manual intervention. To make the comparison fair, we did not perform these actions when reproducing V-SHUTTLE results.

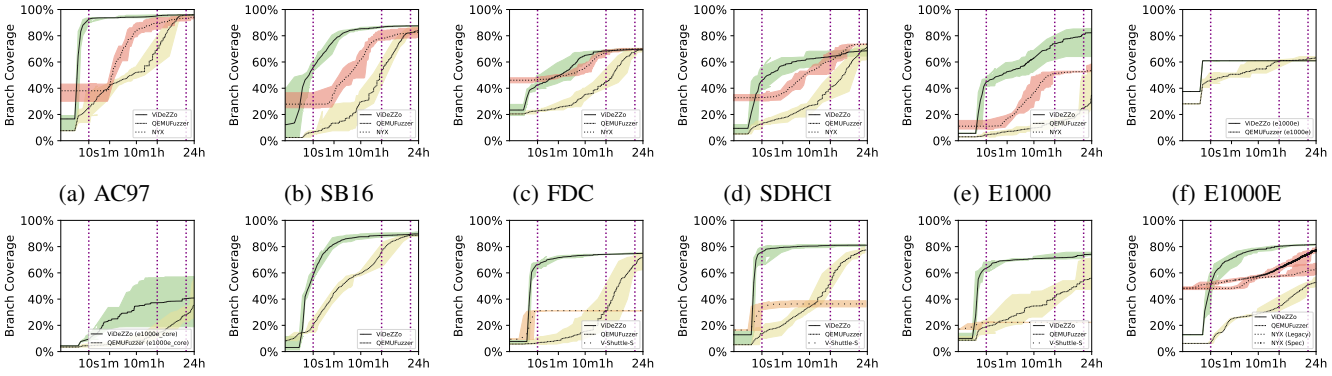
Device	VDF	HYPERCUBE	Nyx-Legacy	V-SHUTTLE	QEMUFuzzer	VIDEZZO
QEMU-x86 Audio						
AC97	53.0%	100%	94.04%	—	**95.93%**	95.90%
CS4231a	56.0%	74.76%	75.36%	85.80%	**94.06%**	92.61%
ES1370	72.7%	91.38%	89.69%	91.91%	88.40%	**91.36%**
Intel-HDA	58.6%	79.17%	62.61%	78.30%	**65.87%**	64.78%
SB16	81.0%	83.80%	83.12%	81.52%	84.15%	**87.54%**
QEMU-x86 Storage						
AHCI	—	—	—	61.60%	49.89%	**62.06%**
FDC	70.5%	84.51%	**70.06%**	—	69.23%	69.72%
Megasas	—	—	—	58.50%	58.67%	**76.74%**
SDHCI	90.5%	81.15%	**73.58%**	—	71.34%	68.52%
VirtIO-BLK	—	—	—	—	30.55%	**55.39%**
QEMU-x86 Network						
E1000	81.6%	66.08%	53.36%	74.50%	35.32%	**82.27%**
E1000E (1/2)[1]	—	—	—	—	**63.12%**	60.94%
E1000E (2/2)[1]	—	—	—	—	35.48%	**40.84%**
EEPro100	75.4%	83.32%	82.12%	—	82.13%	**90.46%**
NE2000	71.7%	71.89%	74.35%	71.90%	75.09%	**94.00%**
PCNET	36.1%	78.81%	78.87%	88.90%	**93.27%**	92.10%
RTL8139	63.0%	74.68%	**83.33%**	80.82%	83.06%	77.46%
QEMU-x86 Display						
ATI-VGA (1/2)[2]	—	—	—	79.40%	—	80.69%
ATI-VGA (2/2)[2]	—	—	—	—	—	85.67%
CIRRUS-VGA	—	—	—	—	88.65%	**89.68%**
QEMU-x86 USB						
EHCI	—	—	—	31.19%	71.84%	**71.96%**
OHCI	—	—	—	36.62%	77.33%	**83.99%**
UHCI	—	—	—	22.27%	55.90%	**72.00%**
XHCI	—	64.40%	63.24%	—	52.92%	**81.63%**
XHCI			Nyx-Spec 77.12%			
QEMU-x86_64						
VirtIO-BLK	—	—	—	—	—	55.39%
QEMU-AArch32						
PL041 (Audio)	—	—	—	—	—	83.91%
SMC91C111 (Net)	—	—	—	—	92.14%	92.98%
TC6393XB (Display)	—	—	—	—	—	76.38%
QEMU-AArch64						
XLNX-ZYNQMP-CAN	—	—	—	—	—	70.42%
XLNX-DP (Display)	—	—	—	—	—	90.42%
VirtualBox x86_64						
SB16	—	—	—	—	—	61.33%
FDC	—	—	—	—	—	39.32%
PCNET	—	—	—	—	—	48.35%
OHCI	—	—	—	—	—	36.13%

[1] We collected the coverage in `e1000e.c` and `e1000e_core.c`, respectively.
[2] We collected the coverage in `ati.c` and `ati_2d.c`, respectively. V-SHUTTLE authors confirmed that they did not consider `ati_2d.c`. Therefore, it shows — in the table.

TABLE 4: Results of final coverage over 6 fuzzers, i.e., VDF, HYPERCUBE, NYX (NYX-SPEC only supports XHCI), V-SHUTTLE, QEMUFUZZER, and VIDEZZO. "—" indicates lack of support for the virtual device. Colored numbers report average coverage across ten runs for 24 hours from our evaluation, other numbers are from corresponding papers.

QEMUFuzzer. VIDEZZO works better than QEMU-FUZZER for storage, network, and USB virtual devices that have more annotations. We attribute QEMUFUZZER's high final coverage to its DMA access pattern support during fuzzing as these simple patterns can slowly guess partial intra-message dependencies.

Importantly, no fuzzer can reach 100 % coverage as some



(a) AC97 (b) SB16 (c) FDC (d) SDHCI (e) E1000 (f) E1000E

(g) E1000E Core (h) CIRRUS-VGA (i) EHCI (j) OHCI (k) UHCI (l) XHCI

Figure 11: Branch coverage over 24 hours of virtual devices fuzzed by NYX, V-SHUTTLE, QEMUFUZZER, and VIDEZZO. The shadows show the minimum/maximum coverage, and the black lines show the average coverage.

code is never reachable in the evaluated environment due to: (1) conditional compilation flags; (2) configuration error handlers; (3) migration callbacks; (4) cleaning up functions. For instance, compared to QEMU, the basic block coverage of VirtualBox virtual devices is lower and one of the reasons is more unreachable functions.

Branch Coverage over Time. Figure 11 shows that, during the first ten seconds, the coverage reached by VIDEZZO sharply increases, then reaches a plateau within one hour. Even though NYX-LEGACY has higher coverage at the beginning, VIDEZZO catches up in seconds. In general, VIDEZZO achieves competitive final coverage results faster than NYX-LEGACY, QEMUFUZZER, and V-SHUTTLE. Interestingly, code with deeper callstacks (Figure 11g) is more slowly explored compared to shallower code (Figure 11f).

6.3. Sensitivity Analysis of Design Choices

We conduct a sensitivity analysis of our design choices in terms of coverage and overhead. Specifically, we design four VIDEZZO variants with a suffix indicating the enabled features, i.e., *intrA-message annotation* (A), *inteR-message mutators* (R), and *Persistent fuzzing* (P). For example, VIDEZZO-ARP shows results with all three features enabled. VIDEZZO-AP does not have inter-message mutators but enables byte mutations in libFuzzer. In general, these variants show the impact of each feature independently and combined.

Variants	IntrA-Message Dependency	InteR-Message Dependency	Persistent Fuzzing
VIDEZZO-ARP	✓	✓	✓
VIDEZZO-AP	✓	N/A	✓
VIDEZZO-RP	N/A	✓	✓
VIDEZZO-P	N/A	N/A	✓
QEMUFUZZER	N/A	N/A	N/A
V-SHUTTLE	N/A	N/A	N/A
NYX-SPEC	✓	✓	N/A
VIDEZZO++-ARP	✓	✓	✓

TABLE 5: VIDEZZO variants and baselines. Specifically, VIDEZZO-ARP keeps all our design options and techniques, e.g., *intrA-message annotation* (A), *inteR-message mutators* (R) and *Persistent fuzzing* (P). Furthermore, we test QEMUFUZZER, V-SHUTTLE, NYX-SPEC, and VIDEZZO++.

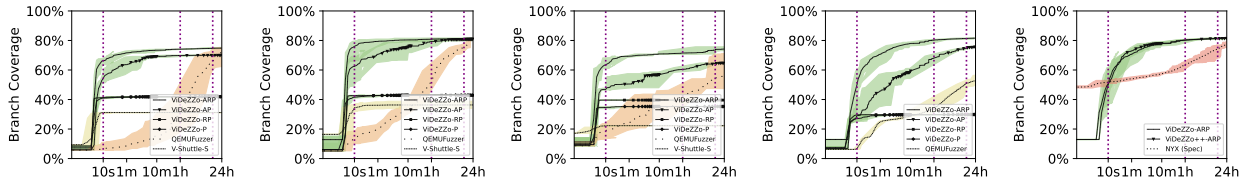
Table 5 shows the setting. We select QEMUFUZZER and V-SHUTTLE as baselines, and include NYX-SPEC and VIDEZZO++ to evaluate statefulness. We implement VIDEZZO++ based on ViDeZZo-ARP, which contains a simple state-aware mechanism similar to FUZZUSB (see Appendix A.4). VIDEZZO and QEMUFUZZER supports all four controllers, V-SHUTTLE does not support XHCI, while NYX-SPEC and VIDEZZO++ only support XHCI.

Efficient Dependency-aware Fuzzing. Figure 12 presents four insights. First, the difference between VIDEZZO-ARP and VIDEZZO-AP indicates our inter-message mutators contribute to both new coverage and coverage speed over time. Second, the difference between VIDEZZO-ARP and VIDEZZO-RP indicates our intra-message annotations have more impact on the new coverage. Third, the differences among VIDEZZO-ARP, VIDEZZO-RP, and VIDEZZO-P indicate our inter-message mutators are more effective when intra-message annotations are enabled. The third observation is expected since, without intra-message dependencies, the fuzzer is spinning in the shallow code space. Fourth, the slight difference between VIDEZZO and VIDEZZO++ shows that state awareness does not significantly increase code coverage. Later shown in Figure 13b, however, VIDEZZO++ finds different state transitions (paths), i.e., different coverage, presenting a distince exploration angle to find new bugs.

Overhead. Table 6 shows the executions per second. Here, we observe three insights. First, VIDEZZO is generally faster than QEMFUZZER. Second, the performance of intra-message annotations appears to be target dependent. Third,

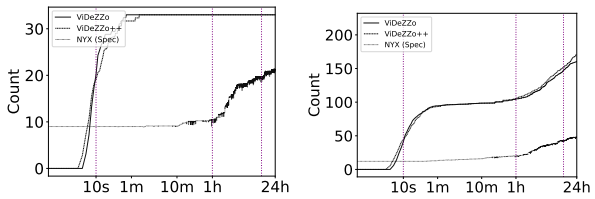
Fuzzer	EHCI	OHCI	UHCI	XHCI
	Average Speed (exec/s)			
VIDEZZO-ARP	3,679.18	763.77	4,263.97	4,350.51
VIDEZZO-AP	1,135.82	198.69	850.31	1,111.21
VIDEZZO-RP	3,221.41	6,735.11	3,006.41	5,393.13
VIDEZZO-P	1,135.20	1,358.38	820.81	1,136.90
QEMUFuzzer	120.38	141.70	93.27	169.54

TABLE 6: Overhead of VIDEZZO variants over 24 hours.



(a) EHCI (b) OHCI (c) UHCI (d) XHCI (e) XHCI++

Figure 12: Branch coverage of VIDEZZO variants and baselines.



(a) State coverage (b) Transition coverage

Figure 13: State and transition coverage over 24 hours.

inter-message mutators increase the speed by four to five times due to a reduced number of mutations per iteration.

State Coverage. In the following, we show the number of states (e.g., QEMU XHCI has 33 distinct states) and of state transitions (the state transition space is 33×33) over time of NYX-SPEC, VIDEZZO, and VIDEZZO++. As shown in Figure 13a, VIDEZZO and VIDEZZO++ cover states faster than NYX-SPEC. Conversely, VIDEZZO++ performs slightly better than VIDEZZO in Figure 13b, which is expected since the former is state-aware. We further include the final state machines in Appendix Figure 19a, Figure 19b and Figure 19c.

6.4. Quick Discovery of Existing Bugs

Table 7 shows the comparison how well V-SHUTTLE, QEMUFUZZER, and VIDEZZO discover the five bugs listed in the V-SHUTTLE paper. We execute each trial 10 times for at most 24 hours and then average the number of executions to discover the bugs. Initially, we fail to reproduce CVE-2020-25085 in 24 hours because due to a not annotated a 12-bit sub-field. We address this issue by manually annotating the sub-field, thus allowing VIDEZZO to bypass the block.

6.5. Discovery of Bugs in Long-time Running

We run VIDEZZO for 24 hours and successfully reproduce 24 existing bugs and found 28 new bugs on and above

Bug	Description	V-SHUTTLE	QEMFUZZER	VIDEZZO
CVE-2020-11869	ATI-VGA IO	35.6M	—	782K (98.0K–2.85M)
CVE-2020-25084	EHCI UAF	79.4M	1.80M (1.36–2.23M)	44.0M (11.7M–88.8M)
CVE-2020-25085	SDHCI HBO	8.88M	1.58M (1.28M–1.85M)	32.3M (1.74M–114M)
CVE-2020-25625	OHCI IL	40.5M	TIMEOUT	2.22K (1.02K–6.22K)
CVE-2021-20257	E1000 IL	235K	TIMEOUT	283K (101K–618K)

TABLE 7: Average number of, along with minimum and maximum, executions that a virtual device fuzzer requires to trigger a vulnerability. "–" indicate a lack of support. Colored numbers are from our evaluation and uncolored ones are from corresponding papers. IO: integer overflow, UAF: use after free, HBO: heap buffer overflow, IL: infinite loop.

QEMU 6.1.50 and on VirtualBox 7.0.0 (Appendix Table 10). Specifically, we fuzz every virtual device for 24 hours, with one core each. Whenever a crash appears, we consider it as a bug, triage it, and report it. Moreover, we are actively engaging the QEMU and VirtualBox communities to patch existing and newly found bugs.

Impact Analysis. Appendix Table 10 shows that virtual devices have different types of bugs, including not only missing or faulty checks, but also spatial and temporal memory corruption. Assertion failures and aborts represent 59.61% (31/52), causing a denial of service on Ubuntu which enables assertions in its QEMU binary. In the discovered bugs, we obtain 1 CVE recognized and provide 7 accepted patches. Bugs in virtual devices are common. Bugs diversity (virtio/non-virtio) in virtual devices enriches the attack primitives and reduces the attack difficulty.

The following two case studies show why the intra-message annotation and inter-message mutator matter.

Case Study 1. VIDEZZO triggers a previously known user-after-free in the QEMU EHCI controller (CVE-2020-25084). Having a minimized PoC, we reached the following two conclusions. The inter-message mutators automatically picked up a "write and wait" message sequence pattern. The first stage of the PoC sets up the internal state of EHCI with multiple write messages. The sequence ends with a `clock_step` message, that advances the system time, forces the expiration of all timers, and activates the asynchronous `ehci_work_bh` to process USB packets. This "write and wait" pattern is common in driver development. The intra-message annotation affects the control flow resulting in a UAF bug. The second stage of the PoC loads a buffer from the physical memory. EHCI reaches different states according to the previous internal state and the values in the buffer. Then, EHCI tries to map a USB packet via `usb_packet_map()` to the host address space but fails. The error handler, i.e., `usb_packet_unmap()`, rolls back the previous mappings. However, EHCI does not synchronize the mapping failure. Therefore, `usb_packet_unmap()` is invoked again, thus resulting in a use-after-free bug.

Case Study 2. VIDEZZO triggers a previously unknown inconsistency bug inside a QEMU USB storage device by fuzzing the QEMU OHCI controller. Specifically, in the storage device, USB_MSDM_DATATOUT conflicts with SCSI_XFER_FROM_DEV. This inconsistency triggers an assertion failure in the storage device.

We constructed a minimized PoC benefiting from intra-message annotation and inter-message mutators: (1) The intra-message annotation helps pass critical constraint checks.

Specifically, VIDEZZO models endpoint descriptors chains and transfers descriptors chains of OHCI [20] through flag and pointer fields, guaranteeing that the OHCI traverses the descriptors and allowing the storage device to access the data inside the descriptors themselves. Using the constraints enables VIDEZZO to produce a correct SCSI command with a state SCSI_XFER_FROM_DEV. (2) The inter-message mutators select a specific message sequence allowing the storage device to reach the buggy state. In particular, the storage device must traverse multiple stages from USB_MSDM_CBW to USB_MSDM_DATAOUT, which is achieved via the inter-message mutators that gradually learn the message order.

7. Discussion

Testing Instruction Emulation. VIDEZZO targets virtual devices; we consider testing instruction emulation orthogonal work. Recent achievements on this topic [21], [22], [23], [24] also show that it needs specific techniques to overcome the high overhead and the difficulty of constructing more semantic-aware instructions.

Testing Closed-source VMMs. VIDEZZO relies on source code to annotate virtual device messages and, at first approximation, does not support closed-source virtual devices like EHCI in VirtualBox and devices in VMWare. Nevertheless, VIDEZZO shows that reusing existing annotations for the same virtual device for different hypervisors is feasible. The remaining future work is to identify testing interfaces in the closed-source virtual device via a reverse engineering.

Probing Hypervisor Internal State. In our implementation, we removed the heavy fork server and let the fuzzed virtual device accumulate its internal state in the fashion of persistent fuzzing. V-SHUTTLE [10] claims that this approach limits the effectiveness of fuzzing. However, as demonstrated in our paper, combining dedicated dependency-aware mutators and delta-debugging mitigates unwanted side effects.

8. Related Work

Virtual Device Security Analysis. To our best knowledge, Ormandy et al. first discussed the problems of hypervisor security in 2007 [3]. Ormandy et al. developed a tool generating random I/O port activities to test virtual devices. Later, Yu et al. used static analysis and differential testing to build a virtual device testing framework named VDTEST [5] and showed that with more meaningful analysis, the testing results outperform random testing [3]. Tang et al. tried to fuzz virtual devices [6] and then Deng et al. published a tool named VDF on fuzzing virtual devices [7]. VDF records initialization MMIO messages as seed inputs, mutates them, and then dispatches them via "record and replay". Increasing the testing interfaces of VDF, later, Schumilo et al. proposed HYPERCUBE which shows the improvement via multiple testing interfaces even without coverage feedback [8]. Schumilo et al. extended their work through NYX by introducing coverage feedback and a new mutation engine to generate more grammar-aware inputs [9]. Conversely, our approach is more scalable than NYX-SPEC due to our lightweight intra-message annotation. Pan et al. propose V-SHUTTLE to handle DMA accesses and design a seed pool to feed the virtual devices [10]. At the same time, MORPHUZZ also observes handling DMA accesses is essential. Even though VIDEZZO is concurrently proposed, we go further than V-SHUTTLE and MORPHUZZ by supporting intra-message dependencies (by our grammar) and inter-message dependencies (by our mutators).

Grammar and Format-based Mutation. Mutation-based fuzzing is one of the most effective techniques to discover bugs in complex software. These fuzzers leverage different mutators to generate inputs and increase code coverage. For non-structural inputs, the mutators operate random bits or bytes [25], [26], however, these techniques are ineffective with structured inputs. Grammar-aware mutators operate on nodes or subtrees in the AST [27], [28], [29], [30], [31], [32], [15] to generate valid inputs. To improve the effectiveness of current grammar-aware fuzzers, Prashast et al. leverage grammar automatons and redesign the grammar-aware mutators to change inputs at a large-scale [15]. Format-aware mutators usually operate on each field of the test case [33], [34], [35], [36]. Van-Thuan et al. design several high-level mutators to add, delete, and splice mutatable fields [33]. VIDEZZO takes virtual messages as input and mutates the whole input sequence, which is different from the above approaches. Like Syzcaller [37], our mutators handle inter-message dependencies. We are the first to apply such mutators to fuzz virtual devices.

9. Conclusion

Virtual device fuzzing remains challenging as fuzzers must keep track of intra-message and inter-message dependencies. We propose a dependency-aware fuzzing framework VIDEZZO for virtual devices combining both of the dependencies. In this framework, a lightweight grammar, and three categories of new message mutation rules are presented, working together boosting to the fuzzing results. Compared to previous work, VIDEZZO is both scalable (covering two hypervisors, four architectures, five device categories, and 28 virtual devices) and efficient (hitting competitive code coverage faster). We successfully reproduced 24 existing bugs and found 28 new bugs with 1 CVE assigned covering diverse bug types. We have provided 7 accepted patches. VIDEZZO is available at https://github.com/HexHive/ViDeZZo.

Acknowledgement

We thank the anonymous reviewers for their insightful feedback and appreciate the shepherd for additional experiments. This project was supported by the National Key R&D Program of China (No. 2022YFE0113200), the National Natural Science Foundation of China (NSFC) under Grant U21A20464, DARPA under grant HR001119S0089-AMP-FP-034, SNSF PCEGP2_186974, and ERC StG 850868. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] W. Contributors, "Virtual machine escape," 2021, https://en.wikipedia.org/wiki/Virtual_machine_escape. Accessed: 2021/10/06.

[2] T. M. Corporation, "CVE search results matched "qemu"," 2021, https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=qemu. Accessed: 2021/10/06.

[3] T. Ormandy, "An empirical study into the security exposure to hosts of hostile virtualized environments," in *CanSecWest*, 2007.

[4] K. Cong, F. Xie, and L. Lei, "Symbolic execution of virtual devices," in *International Conference on Quality Software (ICQS)*, 2013.

[5] T. Yu, X. Qu, and M. B. Cohen, "VDTEST: An automated framework to support testing for virtual devices," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2016.

[6] J. Tang and M. Li, "When virtualization encounter AFL," in *Black Hat Europe*, 2016.

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

[8] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "HYPER-CUBE: High-dimensional hypervisor fuzzing," in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[9] 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.

[10] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. We, "V-SHUTTLE: Scalable and semantics-aware hypervisor virtual device fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

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

[12] renatahodovan, "renatahodovan/picire: Parallel delta debugging framework," 2022, https://github.com/renatahodovan/picire. Accessed: 2022/07/01.

[13] S. Contributors, "syzkaller/syscall_descriptions_syntax.md at master," 2022, https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md. Accessed: 2022/07/03.

[14] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing os fuzzer seed selection with trace distillation," in *USENIX Security Symposium (USENIX Security)*, 2018, pp. 729–743.

[15] P. Srivastava and M. Payer, "GRAMATRON: Effective grammar-aware fuzzing," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[16] Q. Contributors, "Fuzzing," 2021, https://qemu.readthedocs.io/en/latest/devel/fuzzing.html.

[17] T. C. Team, "Source-based code coverage," 2021, https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.

[18] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[19] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 278–289, 2007.

[20] Microsoft, "Openhci: Open host controller interface specification for usb," 2022, =https://composter.com.ua/documents/OHCI_Specification_Rev.1.0a.pdf. Accessed: 2022/11/29.

[21] P. Fonseca, X. Wang, and A. Krishnamurthy, "MULTINYX: A multi-level abstraction framework for systematic analysis of hypervisors," in *European Conference on Computer Systems (EuroSys)*, 2018.

[22] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "Formally verified memory protection for a commodity multiprocessor hypervisor," in *USENIX Security Symposium (USENIX Security)*, 2021.

[23] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A secure and formally verified linux kvm hypervisor," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[24] 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.

[25] A. Contributors, "google/afl: american fuzzy lop - a security-oriented fuzzer," 2021, https://github.com/google/AFL. Accessed: 2021/10/06.

[26] L. Contributors, "libfuzzer – a library for coverage-guided fuzz testing," 2017, https://llvm.org/docs/LibFuzzer.html. Accessed: 2021/10/06.

[27] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFUZZER: An evolutionary interpreter fuzzer using genetic programming," in *European Symposium on Research in Computer Security (ESORICS)*, 2016.

[28] J. Wang, B. Chen, L. Wei, and Y. Liu, "SKYFIRE: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (S&P)*, 2017.

[29] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for deep bugs with grammars," in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[30] H. Han, D. Oh, and S. K. Cha, "CODEALCHEMIST: Semantics-aware code generation to find vulnerabilities in javascript engines," in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[31] J. Wang, B. Chen, L. Wei, and Y. Liu, "SUPERION: Grammar-aware greybox fuzzing," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.

[32] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[33] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," in *IEEE Transactions on Software Engineering (TSE)*, 2019.

[34] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "PROFUZZER: On-the-fly input type probing for better zero-day vulnerability discovery," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

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

[36] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic grey-box fuzzing for structured binary formats," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[37] S. Contributors, "google/syzkaller: Syzkaller is an unsupervised coverage-guided kernel fuzzer," 2021, https://github.com/google/syzkaller. Accessed: 2021/10/06.

Appendix A.

A.1. Head-Tail-Pointers Context

Figure 14 shows how pointer fields collaborate. Pointer field `head` and `tail` point to the head and the tail of a singly linked list of `td_t`. Fuzzers hooking each dma_read API [10], [11] cannot handle this because they do not know dependencies across dma_read APIs. If the context is not handled, the while loop in line 10 cannot terminate because random `head` and `tail` are unlikely equal.

```
typedef struct {
  uint32_t head; uint32_t tail;
} ed_t;
typedef struct {
  uint32_t next;
} td_t;
void handle_end_descriptor(physaddr head)
  ed_t ed;
  dma_read(/*addr=*/head, /*dst=*/&ed)
  while ((ed.head & 0xffffff00) != ed.tail) {
    td_t td;
    phsyaddr addr = ed->head & 0xffffff00;
    if (ed.head & 0x1) /* be invalid and return */
    dma_read(/*addr=*/addr, /*dst=*/&td);
    ed->head |= td.next & 0xfffffff00;
//----------
vd1 = Model('ed', 1)
vd1.add_head(['ed_t'])
vd1.add_struct('ed_t', {
  'head#0x4': POINTER|FLAG,
  'tail#0x4': 'POINTER'})
vd1.add_flag('ed_t.head', {0: 1@0})
vd1.add_struct('td_t', {'next#0x4': 'POINTER'})
vd1.add_linked_list(
  'ed_t.head', 'ed_t.tail',
   ['td_t'], ['next'], alignment=8)
vd1.add_head(['ed_t'])
```

Figure 14: Example of Head-Tail-Pointers Context.

```
typedef {
  uint64_t addr1; uint32_t len;
} bpl_t;
void handle_hda(physaddr addr0)
  bpl_t bpl;
  dma_read(/*addr=*/addr0, /*dst=*/&bpl);
  int n_copied = custom_memcpy(
    /*src=*/bpl.addr1, /*dst=*/buf);
  if (bpl.len == n_copied) {
    // do something
//----------
vd2 = Model('bpl', 2)
vd2.add_head('bpl_t')
vd2.add_struct('bpl_t', {
  'addr1#0x8': 'POINTER', 'len#0x4': 'CONSTANT'})
vd2.add_strcut('bpl_buf', {
  'buf#0x1000': 'RANDOM'})
vd2.add_point_to('bpl_t.addr1', ['bpl_buf'])
vd2.add_constant('bpl_buf.len', [0x1000])
```

Figure 15: Example of Len-Buffer Context.

A.2. Len-Buffer Context

Figure 15 shows how a data field collaborates with a pointer field. If a fuzzer does not control the value of `len`, the branch at line 9 is unlikely to be token.

A.3. Intra-message Dependency in MMIO Accesses

Figure 16 shows other intra-message dependencies when handling MMIO accesses. First, line 7 requires `reg` and `val` that are fields of a MMIO write message to be both equal to zero. If not, `xhci_process_commands()` is unlikely to be taken. Second, line 11 highlights the sub-fields in `val`.

A.4. Implementation of VIDEZZO++

```
static void xhci_doorbell_write(
    void *ptr, hwaddr reg,
    uint64_t val, unsigned size) {
  reg >>= 2;
  if (reg == 0) {
    if (val == 0) {
      xhci_process_commands(xhci);
  } else {
    epid = val & 0xff;
    streamid = (val >> 16) & 0xffff;
    xhci_kick_ep(xhci, reg, epid, streamid);
}
//----------
vd3 = Model('xhci_doorbell_write_0', 3)
vd3.add_head('mmio_write')
vd3.add_struct('mmio_write', {
  'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT',
  'valu#0x8': 'CONSTNAT'})
vd3.add_constant('mmio_write.addr', 0x0)
vd3.add_constant('mmio_write.len', 0x4)
vd3.add_constant('mmio_write.valu', 0x0)

vd4 = Model('xhci_doorbell_write_!0', 4)
vd4.add_head('mmio_write')
vd4.add_struct('mmio_write', {
  'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT',
  'valu#0x8': 'FLAG'})
vd4.add_constant('mmio_write.addr', [
  i for i in range(4, 0x20)])
vd3.add_constant('mmio_write.len', 0x4)
vd4.add_flag('mmio_write.valu', {
  0: 8, 8: 16, 16: 32, 32: 64@0})
```

Figure 16: Intra-message Dependency in MMIO Accesses.

```
while (1) {
  TRBType type;
  pci_dma_read(
    pci_dev, ring->dequeue, trb, TRB_SIZE);
+ __sanitizer_cov_trace_state(0, 1);
  trb->addr = ring->dequeue;
  trb->ccs = ring->ccs;
```

Figure 17: State Instrumentation in QEMU XHCI.

VIDEZZO++ mimics FUZZUSB to support state and state transition feedback in QEMU XHCI (currently, VIDEZZO++ only supports QEMU XHCI). First, like FUZZUSB, a state transition point is defined as a DMA access, e.g, `pci_dma_read()`. Then, whenever a state transition point is visited, the virtual device enter into a new state. Next, we manually instrument QEMU XHCI, with `__sanitizer_cov_trace_state()` whose first argument is always 0 for QEMU XHCI and second arugment is the identifier of the new state, to explicitly pass the new state after the transition point to libFuzzer (Figure 17). We implemented two bytemaps (the first is of one dimension for states and the other is of two dimensions for state transitions) to count how many times (at most 255) a state or a state transition is encountered. Besides, the two byte maps are involved into the libFuzzer's feature collection. If covering any new state or state transition, this seed will be added to the corpus.

Device	Generic Struct	# of Flags	# of Pointers	# of Fields	# of Context -Awareness
AC97	AC97_BD.8.A	2/2	1/1	2/2	—
AC97	AC97_TMPBUF.8.A	—	—	1/1	—
CS4231a	CS4231A_BUF0.8.A	—	—	1/0	—
ES1370	ES1370_TMPBUF.8.A	—	—	1/1	—
Intel-HDA	INTEL_HDA_BUF0.8.A	1/0	1/0	3/4	III
Intel-HDA	INTEL_HDA_VERB.32	1/1	—	1/1	—
SB16	SB16_BUF0.8.A	—	—	1/0	—
AHCI	AHCI_CMFIS.8.A	2/0	—	37/32	—
AHCI	AHCI_SG.S	—	1/1	3/3	—
AHCI	AHCI_RESFIS.8.A	—	—	1/1	—
AHCI	AHCI_LST.8.A	—	—	1/1	—
FDC	FLOPPY_BUF.8.A	—	—	1/0	—
MEGASAS	MEGASAS_REPLY_QUEUE_TAIL.32	—	—	1/1	—
MEGASAS	MEGASAS_REPLY_QUEUE_HEAD.32	—	—	1/1	—
MEGASAS	MEGASAS_MFI_FRAME_HEADER_SENSE.64	—	—	1/2	—
MEGASAS	MEGASAS_MFI_INIT_QINFO.S	1/1	—	5/5	—
MEGASAS	MEGASAS_MFI_FRAME_INIT.S	1/1	2/1	15/15	II
MEGASAS	MEGASAS_MFI_FRAME_DCMD.S	1/1	—	17/17	II
MEGASAS	MEGASAS_MFI_FRAME_ABORT.S	1/1	—	15/17	II
MEGASAS	MEGASAS_MFI_FRAME_SCSI.S	1/1	—	12/12	II
MEGASAS	MEGASAS_MFI_FRAME_IO.S	1/1	1/1	17/17	II
SDHCI	SDHCI_FIFO_BUFFER0.8.A	—	—	1/1	—
SDHCI	SDHCI_FIFO_BUFFER1.8.A	—	—	1/1	—
SDHCI	SDHCI_ADMA2.64	0/1	1/1	4/2	—
SDHCI	SDHCI_ADMA1.32	1/1	1/1	1/1	—
SDHCI	SDHCI_ADMA2_64.64	—	1/1	4/2	—
E1000	E1000_RX_DESC.S	2/1	1/1	6/6	—
E1000	E1000_TX_DESC0.S	2/2	1/1	3/3	—
E1000E	E1000_TX_DESC0.S	2/2	1/1	3/3	II
E1000E	E1000_CONTEXT_DESC.S	4/4	—	4/0	II
E1000E	E1000E_READ_RX_DESC.8.A	—	1/0	4/4	—
EEPRO100	MAC_ADDR0.8.A	0/1	—	6/6	II
EEPRO100	CONFIGURATION.8.A	22/2	—	22/22	II
EEPRO100	TX_BUFFER_ADDRESS.32	—	1/1	1/1	II
EEPRO100	TX_BUFFER_SIZE.16	—	—	1/1	II
EEPRO100	TX_BUFFER_EL.16	0/1	—	1/1	II
EEPRO100	EEPRO100_TX.S	4/0	3/0	11/4	—
EEPRO100	MAC_ADDR1.8.A	—	—	6/6	—
EEPRO100	EEPRO100_RX.S	2/0	1/0	6/4	—
PCNET	PCNET_XDA.S	2/0	—	3/2	—
PCNET	PCNET_TMD.S	2/3	1/1	5/5	—
PCNET	PCNET_RDA.S	2/0	—	3/2	—
PCNET	PCNET_RMD.S	3/4	1/0	5/5	—
PCNET	PCNET_INITBLK32.S	10/12	—	13/13	—
PCNET	PCNET_INITBLK16.S	3/10	—	10/10	—
RTL8139	RTL8139_RX_RING_DESC_RXDW0.32	1/1	—	1/1	—
RTL8139	RTL8139_RX_RING_DESC_RXDW1.32	1/0	—	1/1	—
RTL8139	RTL8139_RX_RING_DESC_RXBUFLO.32	—	1/1	1/1	—
RTL8139	RTL8139_RX_RING_DESC_RXBUFHI.32	—	—	1/1	—
RTL8139	RTL8139_TXBUFFER.8.A	—	—	1/1	—
RTL8139	RTL8139_TX_RING_DESC_TXDW0.32	1/1	—	1/1	—
RTL8139	RTL8139_TX_RING_DESC_TXDW1.32	1/1	—	1/1	—
RTL8139	RTL8139_TX_RING_DESC_TXBUFLO.32	—	1/1	1/1	—
RTL8139	RTL8139_TX_RING_DESC_TXBUFHI.32	—	—	1/1	—
EHCI	entry.32	1/1	1/0	1/1	—
EHCI	EHCIqtd.S	3/4	7/7	8/8	—
EHCI	EHCIqh.S	6/7	9/4	12/12	—
EHCI	EHCItd.S	11/11	8/15	16/16	—
EHCI	EHCIsitd.S	3/1	1/0	7/7	—
OHCI	OHCI_HCCA.S	—	32/32	35/35	—
OHCI	OHCI_ED.S	3/3	3/2	4/4	I
OHCI	OHCI_TD.S	4/4	3/1	4/4	I
OHCI	OHCI_ISO_TD.S	3/3	1/1	12/12	I
UHCI	link.32	1/1	1/1	1/1	—
UHCI	UHCI_QH.S	2/2	2/2	2/2	—
UHCI	UHCI_TD.S	3/3	2/2	4/4	—
XHCI	XHCITRB0.S	2/2	1/3	5/5	—
XHCI	XHCIEvRingSeg.S	1/0	1/0	3/3	—
XHCI	XHCI_POCTX.64	—	1/1	1/1	—
XHCI	XHCI_CTX.32.A	1/1	—	2/2	—
XHCI	XHCI_SLOT_CTX.32.A	4/4	—	4/4	—
XHCI	XHCI_EP0_CTX.32.A	3/3	—	5/5	—
	Missing (False Negative)	38/128 29.69%	19/95 20.00%	27/396 6.82%	
	Wrong (False Positive)	16/128 12.50%	9/95 9.47%	4/396 1.01%	

TABLE 8: Statistics of semi-automatic intra-message annotation. We list the supported generic structs (we use different suffixes to tell the difference, e.g., S for struct, A for array, 32 for uint32_t), the number of flag fields, pointer fields, all fields of the structs (verified results v.s. automatic results), also the context-awareness for a virtual device (I is for Head-Tail-Pointer Context, II is for Flag/Tag-Pointer Context, and III is for Len-Buffer Context).

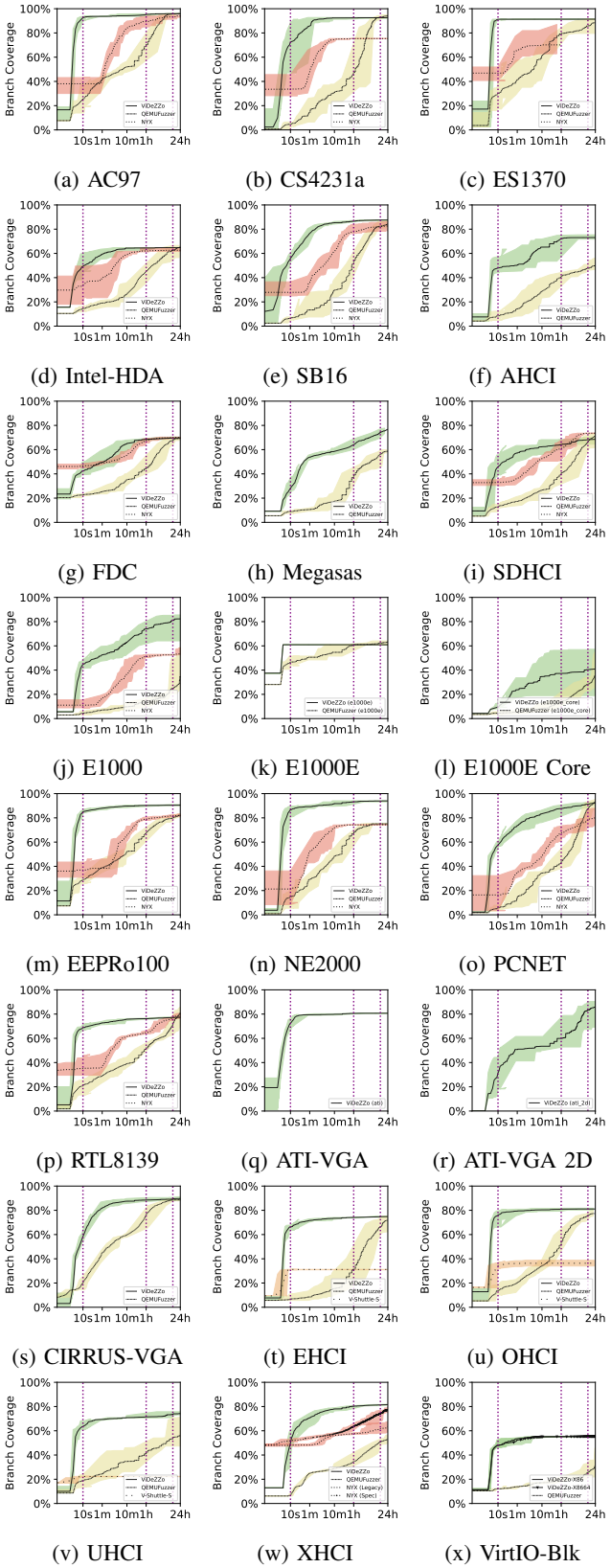
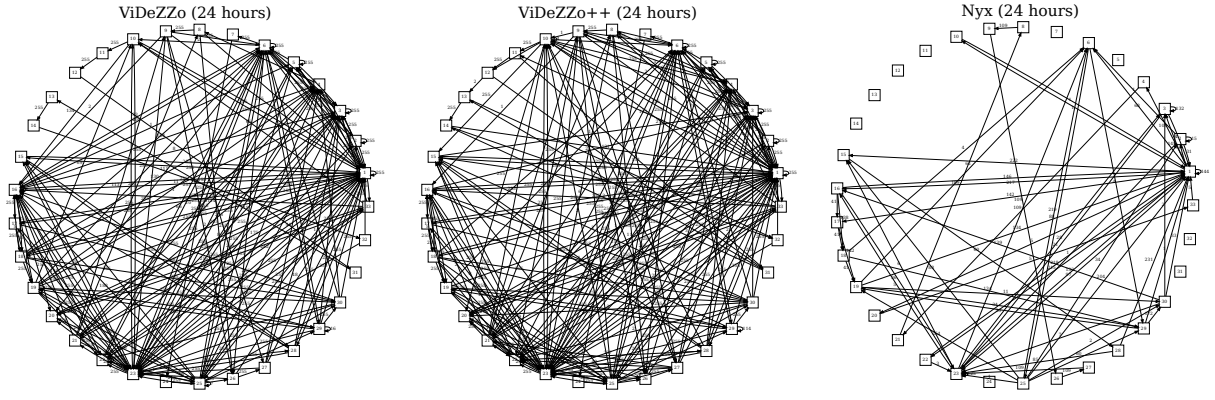


Figure 18: A full version of the branch coverage over 24 hours of the virtual devices fuzzed by NYX, V-SHUTTLE, QEMUFUZZER, and VIDEZZO. The shadow shows the minimum and the maximum coverage, and the black line shows the average coverage.

Step (where in the text)	Manual effort	Estimated average time
Add a new VMM (Section 5.3)	Register a virtual device by searching its architecture, the launch command line, and the signature of PIO/MMIO regions.	10 minutes per virtual device
	Initialize a VMM and identify the testing interfaces by following the main() in an existing VMM frontend.	A week per VMM (up to two weeks for debugging)
	Decide and implement the dispatching methods by looking for guest memory access functions.	An hour per VMM
Finish the rest of the annotation extraction after scanning the source code of a virtual device with our static analysis engine (Section 6.1)	Extract the definition of unnamed types by looking at the source code.	Two minutes per case
	Match two taint analysis results touching the same variable due to disjointed control flow by reading the source code.	15 minutes per case
	Extract the head-tail pointer context by reading the source code.	10 minutes per case
	Extract the flag/tag pointer context by reading the source code.	20 minutes per case
	Extract the length and buffer context by reading the source code.	Five minutes per case
Add a new group mutator (end of Section 4.3)	Obtain the insight about what group mutator is necessary by fuzzing virtual devices.	N/A
	Decide the feedback and develop the handler with the help of our action-trigger protocol.	Hours per case (up to two days for debugging)

TABLE 9: During the whole process of the virtual device fuzzing, three steps are manual, i.e., adding a new VMM, addressing the uncertainty of the static analysis, and adding a new group mutator. For each step, this table details the necessary manual effort and the estimated average time. Note that our week has 40 working hours.



(a) ViDeZZo (b) ViDeZZo++ (c) Nyx-Spec

Figure 19: State and state transition coverage.

Id	Target	Category	VMM	Version	Arch	Short Description	# of Messages	Reported By	Status
1	ac97	audio	qemu	7.0.94	i386	Abort in audio_calloc()	1	An, Vi	Fixed
2	am53c974	storage	qemu	6.1.50	i386	Null pointer access in do_busid_cmd()	N/A	An	Fixed
3	ati	display	qemu	6.1.50	i386	Out of bounds write in ati_2d_blt()	N/A	VS	Fixed
4	ati	display	qemu	7.0.94	i386	Out of bounds write in ati_2d_blt()	4	Vi	$\widetilde{Fixed}$
5	cadence_uart	serial	qemu	7.2.50	aarch64	Devision by zero in uart_parameters_setup()	2	Vi	$\widetilde{Fixed}$
6	dwc2	usb	qemu	6.1.50	aarch32	Assertion failed in hw/usb/core.c from dwc2	N/A	Vi	Patch submitted
7	e1000	net	qemu	6.1.50	i386	Infinite loop in process_tx_desc()	N/A	Ny, VS, QF	Fixed
8	ehci	usb	qemu	6.1.50	i386	Abort in usb_ep_get()	N/A	Ny, VS, QF	Patch submitted
9	ehci	usb	qemu	6.1.50	i386	Assertion failure in address_space_unmap()	N/A	Ny, VS, QF	Fixed
10	exynos421_fimd	display	qemu	7.2.50	aarch32	Assertion failure in fimd_update_memory_section()	2	Vi	Open
11	ftgmac100	net	qemu	7.2.50	aarch32	Heap buffer overflow in aspeed_smc_flash_do_select()	2	Vi	Open
12	imx_usb_phy	usb	qemu	7.2.50	aarch32	Out of bounds in imx_usbphy_read()	1	Vi	$\widetilde{Fixed}$
13	intel-hda	audio	vbox	7.0.7	i386	Global buffer overflow in hdaMmioWrite()	1	Vi	Open
14	lan9118	net	qemu	7.2.50	aarch32	Out of bounds read in lan9118	535	Vi	Open
15	lan9118	net	qemu	7.2.50	aarch32	Abort in lan9118_16bit_mode_read()	1	Vi	$\widetilde{Fixed(us)}$
16	lsi53c895a	storage	qemu	6.1.50	i386	Assertion failure in lsi53c810 emulator	N/A	Ny, VS, QF, An, Vi	Fixed
17	megasas	storage	qemu	6.1.50	i386	Assertion failure in scsi_dma_complete()	N/A	QF	Fixed
18	megasas	storage	qemu	6.1.50	i386	Assertion failure in bdrv_co_write_req_prepare()	N/A	QF	Fixed
19	nvme	storage	qemu	6.1.50	i386	Null pointer access in memory_region_set_enabled()	1	Vi	$\widetilde{Fixed(us)}$
20	ohci	usb	qemu	7.0.50	i386	Assertion failure in usb_msd_transfer_data()	29	Vi	$\widetilde{Fixed}$
21	ohci	usb	qemu	7.0.50	i386	Abort in ohci_frame_boundary()	8	VS, QF, Vi	$\widetilde{Fixed(us)}$
22	ohci	usb	qemu	7.0.50	i386	Heap use after free in usb_cancel_packet()	67	Vi	Open
23	ohci	usb	qemu	7.0.91	i386	Assertion failure in usb_cancel_packet()	79	An	Open
24	omap_dss	display	qemu	7.2.50	aarch32	Out of memory in hw/omap-dss for aarch32	3	Vi	Open
25	pl041	audio	qemu	7.0.94	aarch32	Abort in audio_bug() triggered by pl041	1	An	Fixed
26	pl041	audio	qemu	7.0.94	aarch32	Abort in audio_bug() triggered by pl041	2	An	Fixed
27	sb16	audio	qemu	6.1.50	i386	Assertion failure in audio_calloc() caused by sb16	N/A	An	Fixed
28	sb16	audio	qemu	6.1.50	i386	Abort in audio_calloc()	4	Vi	$\widetilde{Fixed(us)}$
29	sdhci	storage	qemu	7.1.50	i386	Heap buffer overflow in sdhci_read_dataport()	9	QF	Fixed
30	smc91c111	net	qemu	7.1.93	aarch32	Out of bounds read/write in smc91c111	5	Vi	Open
31	tc6393xb	display	qemu	7.2.50	aarch32	negative-size-param in nand_blk_load_512()	23	Vi	Open
32	tc6393xb	display	qemu	7.2.50	aarch32	Heap buffer overflow in nand_blk_write_512()	7	Vi	Open
33	virtio-blk	storage	qemu	7.0.94	i386	Assertion failure in address_space_stw_le_cached()	5	An	Fixed
34	virtio-blk	storage	qemu	7.0.94	i386	Infinite loop in virtio_blk_handle_vq()	16	An	Fixed
35	vmxnet3	net	qemu	6.1.50	i386	Code should not be reached vmxnet3_io_bar1_write()	N/A	VS, Vi	Fixed
36	vmxnet3	net	qemu	6.1.50	i386	Three hw_error() in vmxnet3_validate_queues()	N/A	QF	Fixed
37	vmxnet3	net	qemu	6.1.50	i386	Assertion failed in vmxnet3_io_bar0_write()	N/A	QF	Fixed
38	vmxnet3	net	qemu	6.1.50	i386	Out of memory net_tx_pkt_init()	N/A	QF, VS	Fixed
39	vmxnet3	net	qemu	6.1.50	i386	Assertion failure in net_tx_pkt_reset()	N/A	QF	Fixed
40	vmxnet3	net	qemu	6.1.50	i386	eth_get_gso_type: code should not be reached	N/A	QF, VS	Fixed
41	xhci	usb	qemu	7.0.94	i386	Abort in xhci_find_stream()	56	Vi	$\widetilde{Fixed(us)}$
42	xlnx_dp	display	qemu	7.0.91	aarch64	Abort in xlnx_dp_aux_set_command()	1	Vi	$\widetilde{Fixed(us)}$
43	xlnx_dp	display	qemu	6.1.50	aarch64	Out of bounds read in xlnx_dp_read()	1	Vi	$\widetilde{Fixed(us)}$
44	xlnx_dp	display	qemu	6.1.50	aarch64	Out of bounds in xlnx_dp_vblend_read()	N/A	An	Fixed
45	xlnx_dp	display	qemu	7.2.50	aarch64	Overflow in xlnx_dp_aux_push_rx_fifo()	3	Vi	Patch submitted
46	xlnx_dp	display	qemu	7.2.50	aarch64	Abort in xlnx_dp_change_graphic_fmt()	1	Vi	Patch submitted
47	xlnx_dp	display	qemu	7.2.50	aarch64	Underflow in xlnx_dp_aux_pop_tx_fifo()	1	Vi	Patch submitted
48	xlnx_dp	display	qemu	7.2.50	aarch64	Overflow in xlnx_dp_aux_push_tx_fifo()	17	Vi	Patch submitted
49	xlnx_zynqmp_can	net	qemu	7.2.50	aarch64	Fifo underflow in transfer_fifo()	2	Vi	Open
50	xlnx_zynqmp_can	net	qemu	7.2.50	aarch64	Fifo overflow in transfer_fifo()	291	Vi	Open
51	xlnx_zynqmp_qspips	spi	qemu	7.2.50	aarch64	Out of bound in xilinx_spips_write()	1	Vi	Open
52	xlnx_zynqmp_qspips	spi	qemu	7.2.50	aarch64	Underflow in xlnx_dp_aux_push_rx_fifo()	2	Vi	Open

TABLE 10: List of QEMU and VirtualBox bugs. For column "# of Messages", we present the least number of messages to trigger this bug with the help of the delta debugging. If the bug has been fixed, we do not do the delta debugging and mark the cell "N/A". For column "Reported-By", if a previously known bug, we list the tool names that triggered the bug, i.e., NYX (Ny), VSHUTTLE (VS), QEMUFUZZER (QF), and VIDEZZO (Vi), and we use ANONYMOUS (An) for these tools we do not know. For column "Status", if a bug does not have patch, we mark it Open; if we submit a patch, we mark it "Patch submitted"; if a bug has been fixed and we are not involved, we mark it "Fixed"; if we are involved, we mark it "$\widetilde{Fixed}$"; particularly, if our patch is accepted, we mark it $\widetilde{Fixed(us)}$.