VIДЕZZO: 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.

VIДЕZZO, 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, VIДЕZZO 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 fuzzing framework.

In our evaluation, VIДЕZZO covers two hypervisors, four architectures, five device categories, and 28 virtual devices, and reaches competitive coverage faster. Moreover, VIДЕZZO 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 [6] 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 multidimension inputs. Fuzzing of virtual devices advanced further when starting considered 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 can be called virtual device messages.
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

<table>
<thead>
<tr>
<th>Messages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIO ops</td>
<td>PIO read/write operations</td>
</tr>
<tr>
<td>MMIO ops</td>
<td>MMIO read/write operations</td>
</tr>
<tr>
<td>Memory ops</td>
<td>Memory allocation, read/write, or free</td>
</tr>
<tr>
<td>Clock ops</td>
<td>Time adjustments</td>
</tr>
</tbody>
</table>

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

Figure 2: Virtual device messages and a structural input.

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

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,
// type system
typedef GlobalState {
  uint32 internal_state; } GlobalState;

GlobalState gs;

void mmio_write_dword(
  physaddr addr, uint64_t val) {
  switch (addr) {
    case 0x0:
      gs->internal_state = val; break;
    case 0x4:
      gs->internal_state = val; break;
    case 0x0:
      if (!gs->internal_state) break;
      // do something and break
  }
}

// APIs
typedef uint8_t BEGIN
  FIELD_TYPE: RANDOM | CONSTANT | POINTER | FLAG
  FIELDNAME: NAME
  FIELD : FIELDNAME '#' SIZE
typedef uint8_t BEGIN
  FIELD_INDEX : STRUCTNAME '.', FIELDNAME
  FIELD_TYPEPAIR: FIELD ':', FIELD_TYPE
  FIELD_SET : '{' FIELD_TYPE_PAIR+ '}
  POINT_TO_SET : '{' FIELD_INDEX+ '}
  CONDIDATES : '{' uint32_t+ '}
  CONDITION :-> CONDITION_SET,
  ALIGNMENT :-> uint8_t)

def add_struct(
  name :-> STRUCTNAME, fields :-> FIELD_SET)

def add_flag(
  field :-> FIELD_INDEX, flags :-> FLAG_SET)

def add_point_to(
  field :-> FIELD_INDEX, candidates :-> CONDIDATES)

def add_head(structs :-> STRUCT_SET)

// statements and programming model
MODELNAME: NAME
model: 'Model(' MODELNAME, ' ', MODELID ')'
api_add_field:
  add_flag | add_constant | add_point_to
statement: add_struct add_field+

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

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
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.,
FIELD_LEN_PAIR. For example, command’s first three bits are
0: 387, 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

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 com-
mands in one message. We support this through an API

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.
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 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].

At this point, the fuzzer dispatches the last Message 1 and (Mutator 5). After seed mutation, the fuzzer sequentially discards new coverage, the fuzzer saves the mutated seed as a group in the next fuzzing loop (Mutator 13). Next, the trigger dispatches the messages to the target virtual device. 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 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 fuzzier 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 fuzzier 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 malloc (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 \( \text{rand()} \&\& 0xff \) | \( (\text{rand()} \&\& 0xff) \ll 16 \). 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.

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

```
Algorithm 1: Annotation-to-Message Construction

Input: Intra-Message Annotation D
Input: Message memalloc, memread, memwrite, memfree
Result: Messages M = \{m_1, m_2, \ldots, 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(Metadata))); break;
         9. case CONSTANT do
            10. M.append(memwrite(Field.Value));
            11. case RANDOM do
                12. M.append(memwrite(rand())); break;
         13. case POINTER do
                14. M.append(memwrite(gen_pointer(Metadata))); break;

15. Function Free(Address):
   16. M.append(memfree(Address));

17. Head ← D.Head
18. HeadAddress ← Alloc(sizeof (Head));
19. Fillup(Head);
20. Free(HeadAddress);
```
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()` 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 mutants in C by hooking the LLVMFuzzerCustomMutator() 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()` in the implementation.

```c
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);
}
```

![Figure 9: Instrumentation of Load Miss Mutator](image)

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.

```c
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](image)

Figure 10: Instrumentation of 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 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 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 `names` 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), QEMU-Fuzzer (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/union. 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 \cite{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 virtual devices. V-SHUTTLE allows adapting new virtual devices to V-SHUTTLE, and reaches competitive coverage compared with state-of-the-art virtual device fuzzers.

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.

<table>
<thead>
<tr>
<th>Device</th>
<th>VDF</th>
<th>HYPERCUBE</th>
<th>NYX-Legacy</th>
<th>V-SHUTTLE</th>
<th>QEMUFUZZER</th>
<th>ViDEZZO</th>
</tr>
</thead>
<tbody>
<tr>
<td>QEMU-x86 Audio</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AC97</td>
<td>53.0%</td>
<td>100%</td>
<td>94.06%</td>
<td>—</td>
<td>95.93%</td>
<td>95.50%</td>
</tr>
<tr>
<td>CS4233a</td>
<td>56.0%</td>
<td>74.76%</td>
<td>75.36%</td>
<td>85.89%</td>
<td>94.06%</td>
<td>92.61%</td>
</tr>
<tr>
<td>E1370</td>
<td>72.7%</td>
<td>91.34%</td>
<td>90.90%</td>
<td>91.91%</td>
<td>90.40%</td>
<td>91.36%</td>
</tr>
<tr>
<td>HS1377</td>
<td>58.8%</td>
<td>79.17%</td>
<td>84.81%</td>
<td>78.30%</td>
<td>85.87%</td>
<td>74.70%</td>
</tr>
<tr>
<td>SC16e</td>
<td>81.0%</td>
<td>83.80%</td>
<td>83.12%</td>
<td>81.52%</td>
<td>84.15%</td>
<td>87.54%</td>
</tr>
<tr>
<td>QEMU-x86 Storage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AHCI</td>
<td>—</td>
<td></td>
<td>—</td>
<td>61.60%</td>
<td>49.89%</td>
<td>62.06%</td>
</tr>
<tr>
<td>FDC</td>
<td>70.5%</td>
<td>84.51%</td>
<td>70.06%</td>
<td>—</td>
<td>69.23%</td>
<td>67.23%</td>
</tr>
<tr>
<td>Megasas</td>
<td>—</td>
<td></td>
<td>58.50%</td>
<td>76.74%</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SDHCI</td>
<td>90.5%</td>
<td>81.15%</td>
<td>73.05%</td>
<td>—</td>
<td>71.34%</td>
<td>68.52%</td>
</tr>
<tr>
<td>VioD-Blake</td>
<td>—</td>
<td></td>
<td>—</td>
<td>30.55%</td>
<td>—</td>
<td>35.39%</td>
</tr>
<tr>
<td>QEMU-x86 Network</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1000</td>
<td>61.6%</td>
<td>66.00%</td>
<td>70.30%</td>
<td>74.50%</td>
<td>38.32%</td>
<td>82.27%</td>
</tr>
<tr>
<td>E1000 (1/2)</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>63.12%</td>
<td>60.94%</td>
</tr>
<tr>
<td>E1000E (2/2)</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>35.46%</td>
<td>40.84%</td>
</tr>
<tr>
<td>EEp0.100</td>
<td>71.4%</td>
<td>83.32%</td>
<td>82.15%</td>
<td>82.13%</td>
<td>—</td>
<td>90.06%</td>
</tr>
<tr>
<td>NE2000</td>
<td>71.3%</td>
<td>71.89%</td>
<td>74.39%</td>
<td>71.90%</td>
<td>75.09%</td>
<td>98.00%</td>
</tr>
<tr>
<td>PCNet</td>
<td>36.1%</td>
<td>78.81%</td>
<td>78.87%</td>
<td>80.99%</td>
<td>93.27%</td>
<td>92.10%</td>
</tr>
<tr>
<td>RTL8139</td>
<td>63.0%</td>
<td>74.46%</td>
<td>83.35%</td>
<td>80.82%</td>
<td>83.06%</td>
<td>77.80%</td>
</tr>
<tr>
<td>QEMU-x86 Display</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATI-VGA (1/2)</td>
<td>—</td>
<td></td>
<td>—</td>
<td>79.49%</td>
<td>—</td>
<td>80.59%</td>
</tr>
<tr>
<td>ATI-VGA (2/2)</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>85.67%</td>
<td>—</td>
</tr>
<tr>
<td>CRISIS-VGA</td>
<td>—</td>
<td></td>
<td>—</td>
<td>88.55%</td>
<td>—</td>
<td>89.86%</td>
</tr>
<tr>
<td>QEMU-x86 USB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HiCZ</td>
<td>—</td>
<td></td>
<td>—</td>
<td>31.19%</td>
<td>73.84%</td>
<td>71.96%</td>
</tr>
<tr>
<td>GTZC</td>
<td>—</td>
<td></td>
<td>—</td>
<td>77.33%</td>
<td>83.99%</td>
<td>—</td>
</tr>
<tr>
<td>UHCI</td>
<td>—</td>
<td></td>
<td>—</td>
<td>55.90%</td>
<td>72.00%</td>
<td>—</td>
</tr>
<tr>
<td>XHCI</td>
<td>—</td>
<td>64.40%</td>
<td>63.24%</td>
<td>—</td>
<td>52.92%</td>
<td>81.63%</td>
</tr>
<tr>
<td>XHZC</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>77.12%</td>
</tr>
<tr>
<td>QEMU-x86-44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VioD-Blake</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>85.39%</td>
<td>—</td>
</tr>
<tr>
<td>QEMU-Aarch32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PL041 (Audio)</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>83.91%</td>
<td>—</td>
</tr>
<tr>
<td>HDMI11a (Neo)</td>
<td>—</td>
<td></td>
<td>—</td>
<td>92.14%</td>
<td>92.08%</td>
<td>—</td>
</tr>
<tr>
<td>TCS3363X (Display)</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>76.38%</td>
<td>—</td>
</tr>
<tr>
<td>QEMU-Aarch64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XNLX-ZYNS6P-CAN</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>70.62%</td>
<td>—</td>
</tr>
<tr>
<td>XNLX-DP (Display)</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>80.82%</td>
<td>—</td>
</tr>
<tr>
<td>Virtuozel 8b-44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B16</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>84.53%</td>
<td>—</td>
</tr>
<tr>
<td>FDC</td>
<td>—</td>
<td></td>
<td>—</td>
<td>48.32%</td>
<td>—</td>
<td>48.32%</td>
</tr>
<tr>
<td>PCNet</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>48.35%</td>
<td>—</td>
</tr>
<tr>
<td>GTZC</td>
<td>—</td>
<td></td>
<td>—</td>
<td>—</td>
<td>86.15%</td>
<td>—</td>
</tr>
</tbody>
</table>

We collected the coverage in at1000e.c and at1d2e.c, c, respectively. 

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.

QEMUFUZZER. ViDEZZO works better than QEMUFUZZER 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
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.

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 support 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 fuzzier 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 distinct 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 QEMUfuzzer. Second, the performance of intra-message annotations appears to be target dependent. Third,

<table>
<thead>
<tr>
<th>Variants</th>
<th>IntrA-Message Dependency</th>
<th>InteR-Message Dependency</th>
<th>Persistent Fuzzing</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViDeZZo-ARP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ViDeZZo-AP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ViDeZZo-RP</td>
<td>N/A</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ViDeZZo-P</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
</tr>
<tr>
<td>QEMUfuzzer</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
</tr>
<tr>
<td>V-Shuttle</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
</tr>
<tr>
<td>NYX-Spec</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
</tr>
<tr>
<td>ViDeZZO++-ARP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**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 QEMU-Fuzzer, V-Shuttle, NYX-Spec, and ViDeZZO++.
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 $33 \times 33$) 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, QEMU FUZZER, 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 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 EHCl 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 mutable fields [33]. ViDEZZO takes virtual messages as input and mutates the whole input sequence, which is different from the above approaches. Like Syzcaler [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 HR00119S0089-AMPFP-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


Appendix A.

A.1. Head-Tail-Pointers Context

Figure 14 shows how pointer fields collaborate. Pointer field head and ta11 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 ta11 are unlikely equal.
typedef struct {
  uint32_t head; uint32_t tail;
} ed_t;

typedef struct {
  uint32_t addr1;
} td_t;

void *ptr, hwaddr reg,
void handle_end_descriptor(physaddr head) {
  td_t td;
  dma_read(physaddr addr0, /dst=*/&td);
  int n_copied = custom_memcpy(/*src=*/bpl.addr1, /*dst=*/buf);
  if (bpl.len == n_copied) {
    xhci_process_commands(xhci);
  } else {
    epid = val & 0xff;
    streamid = (val >> 16) & 0xffff;
    xhci_kick_ep(xhci, reg, epid, streamid);
  } //----------
}

void handle_hda(physaddr addr0)
while (1) {
  reg >>= 2;
  if (reg == 0) {
    xhci_process_commands(xhci);
  } else {
    trb->addr = ring->dequeue;
    pci_dma_read(pci_dev, ring->dequeue, trb, TRB_SIZE);
    //----------
    vdi = Model('xhci_doorbell_write_0', 3)
    vdi.add_head('mmio_write')
    vdi.add_struct('mmio_write', {
      'addr#0x8': 'POINTER', 'len#0x4': 'CONSTANT',
    })
    vdi.add_flag('mmio_write.valu', {
      'valu#0x8': 'CONSTANT',
    })
    vdi.add_constant('mmio_write.len', 0x4)
    vdi.add_constant('mmio_write.addr', 0x0)
    vdi.add_constant('mmio_write.len', 0x4)
    vdi.add_constant('mmio_write.valu', 0x0)
    vdi = Model('xhci_doorbell_write_0', 4)
    vdi.add_head('mmio_write')
    vdi.add_struct('mmio_write', {
      'addr#0x8': 'POINTER', 'len#0x4': 'CONSTANT',
      'valu#0x8': 'FLAG',
    })
    vdi.add_flag('mmio_write.valu', {
      'valu#0x8': 'CONSTANT',
    })
    vdi.add_constant('mmio_write.addr', [0 for i in range(4, 0x20)])
    vdi.add_constant('mmio_write.len', 0x4)
    vdi.add_flag('mmio_write.valu', {0: 8, 8: 16, 16: 32, 32: 6400})
  }
  //----------
  vdi2 = Model('ed_t', 1)
  vdi2.add_head('ed_t')
  vdi2.add_struct('ed_t', [
    'head#0x4': 'POINTER|FLAG',
    'tail#0x4': 'POINTER',
  ])  
  vdi2.add_flag('ed_t.head', {0: 1})
  vdi2.add_struct('ed_t', [
    'td_t', ['next#0x4': 'POINTER'],
  ])  
  vdi2.add_linked_list('ed_t', ['td_t', ['next#0x4': 'POINTER'])
  vdi2.add_flag('ed_t.head', {0: 1})
  vdi2 = Model('ed', 1)
  //---------
  ed->head |= td.next & 0xffffff00;
  if (ed.head & 0x1) /* be invalid and return */
  physaddr addr = ed->head & 0xffffff00;
  td_t td;
  while ((ed.head & 0xffffff00) != ed.tail) {
    //----------
    vdi3 = Model('xhci_doorbell_write_0', 3)
    vdi3.add_head('mmio_write')
    vdi3.add_struct('mmio_write', {
      'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT',
      'valu#0x8': 'CONSTANT',
    })
    vdi3.add_constant('mmio_write.addr', 0x0)
    vdi3.add_constant('mmio_write.len', 0x4)
    vdi3.add_constant('mmio_write.valu', 0x0)
    vdi4 = Model('xhci_doorbell_write_0', 4)
    vdi4.add_head('mmio_write')
    vdi4.add_struct('mmio_write', {
      'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT',
      'valu#0x8': 'FLAG',
    })
    vdi4.add_flag('mmio_write.valu', {
      'valu#0x8': 'CONSTANT',
    })
    vdi4.add_constant('mmio_write.addr', [0 for i in range(4, 0x20)])
    vdi4.add_constant('mmio_write.len', 0x4)
    vdi4.add_flag('mmio_write.valu', {0: 8, 8: 16, 16: 32, 32: 6400})
  }
  //----------
  vdi4 = Model('xhci_doorbell_write_0', 4)
  vdi4.add_head('mmio_write')
  vdi4.add_struct('mmio_write', {
    'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT',
    'valu#0x8': 'FLAG',
  })
  vdi4.add_flag('mmio_write.valu', {
    'valu#0x8': 'CONSTANT',
  })
  vdi4.add_constant('mmio_write.addr', [0 for i in range(4, 0x20)])
  vdi4.add_constant('mmio_write.len', 0x4)
  vdi4.add_flag('mmio_write.valu', {0: 8, 8: 16, 16: 32, 32: 6400})
  //----------
  vdi5 = Model('xhci_doorbell_write_0', 5)
  vdi5.add_head('xhci_doorbell_write_0')
  vdi5.add_struct('xhci_doorbell_write_0', {
    'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT',
    'valu#0x8': 'CONSTANT',
  })
  vdi5.add_constant('xhci_doorbell_write_0.addr', 0x0)
  vdi5.add_constant('xhci_doorbell_write_0.len', 0x4)
  vdi5.add_constant('xhci_doorbell_write_0.valu', 0x0)
  //----------
  vdi6 = Model('xhci_doorbell_write_0', 6)
  vdi6.add_head('xhci_doorbell_write_0')
  vdi6.add_struct('xhci_doorbell_write_0', {
    'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT',
    'valu#0x8': 'FLAG',
  })
  vdi6.add_flag('xhci_doorbell_write_0.valu', {
    'valu#0x8': 'CONSTANT',
  })
  vdi6.add_constant('xhci_doorbell_write_0.addr', [0 for i in range(4, 0x20)])
  vdi6.add_constant('xhci_doorbell_write_0.len', 0x4)
  vdi6.add_flag('xhci_doorbell_write_0.valu', {0: 8, 8: 16, 16: 32, 32: 6400})
  //----------
  vdi7 = Model('xhci_doorbell_write_0', 7)
  vdi7.add_head('xhci_doorbell_write_0')
  vdi7.add_struct('xhci_doorbell_write_0', {
    'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT',
    'valu#0x8': 'FLAG',
  })
  vdi7.add_flag('xhci_doorbell_write_0.valu', {
    'valu#0x8': 'CONSTANT',
  })
  vdi7.add_constant('xhci_doorbell_write_0.addr', [0 for i in range(4, 0x20)])
  vdi7.add_constant('xhci_doorbell_write_0.len', 0x4)
  vdi7.add_flag('xhci_doorbell_write_0.valu', {0: 8, 8: 16, 16: 32, 32: 6400})

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

A.2. Len-Buffer Context

Figure 15 shows how a data field collaborates with a pointer field. If a fuzzier 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++

Figure 16: Intra-message Dependency in MMIO Accesses.

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 byetmaps (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.

Figure 17: State Instrumentation in QEMU XHCI.
<table>
<thead>
<tr>
<th>Device</th>
<th>Generic Suffix</th>
<th># of</th>
<th># of</th>
<th># of Context-Awareness</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACF7</td>
<td>ACF7_2D3.A.</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>ACF7</td>
<td>ACF7_3D3.A.</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>CS4231a</td>
<td>CS4231A_BUP.A</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>ES1150</td>
<td>ES1150_BUP.A</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>Intel-HDA</td>
<td>INTEL-HDA_BUP.A</td>
<td>1/4</td>
<td>1/3</td>
<td>1/4</td>
</tr>
<tr>
<td>Ind-386</td>
<td>INTEL-386_BUP.A</td>
<td>1/4</td>
<td>1/3</td>
<td>1/4</td>
</tr>
<tr>
<td>SB16</td>
<td>SB16_BUP.A</td>
<td>1/4</td>
<td>1/3</td>
<td>1/4</td>
</tr>
<tr>
<td>ARCTe</td>
<td>ARCTe_CMPS.A.A</td>
<td>2/0</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>ARCTe</td>
<td>ARCTe_CMPS.A</td>
<td>2/0</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>ARCTe</td>
<td>ARCTe_CMPS.A.A</td>
<td>2/0</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>ARCTe</td>
<td>ARCTe_CMPS.A</td>
<td>2/0</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>FCC</td>
<td>FCC_CMPS.BUP.A</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>FCC2</td>
<td>FCC2_CMPS.BUP.A</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>MEGASAS</td>
<td>MEGASAS_CMPS.BUP.A</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>MEGASAS</td>
<td>MEGASAS_CMPS.BUP.A</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>MEGASAS</td>
<td>MEGASAS_CMPS.BUP.A</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
<tr>
<td>MEGASAS</td>
<td>MEGASAS_CMPS.BUP.A</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
</tbody>
</table>

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

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.

![Figure 19: State and state transition coverage.](image-url)
<table>
<thead>
<tr>
<th>Id</th>
<th>Target Category</th>
<th>VMM</th>
<th>Version</th>
<th>Arch</th>
<th>Short Description</th>
<th># of Messages</th>
<th>Reported By</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ac97 audio</td>
<td>qemu</td>
<td>7.0.94</td>
<td>i386</td>
<td>Aborted in audio_calloc()</td>
<td>1</td>
<td>An, Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>2</td>
<td>am53c974 storage</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Null pointer access in do_busid_cmdl()</td>
<td>N/A</td>
<td>An</td>
<td>Fixed</td>
</tr>
<tr>
<td>3</td>
<td>atu display</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Out of bounds write in atu_md8_bit()</td>
<td>N/A</td>
<td>Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>4</td>
<td>ati display</td>
<td>qemu</td>
<td>7.0.94</td>
<td>i386</td>
<td>Out of bounds write in ati_md8_bit()</td>
<td>4</td>
<td>Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>5</td>
<td>cadence_uart serial</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Deivision by zero in uart_parameters_setup()</td>
<td>2</td>
<td>Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>6</td>
<td>dwc2 usb</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Asserted assertion in hwusb/ocore from dwc2</td>
<td>N/A</td>
<td>Vi</td>
<td>Patch submitted</td>
</tr>
<tr>
<td>7</td>
<td>e1008 net</td>
<td>qemu</td>
<td>7.2.50</td>
<td>i386</td>
<td>Infinite loop in process_rx_desc()</td>
<td>N/A</td>
<td>Vi, QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>8</td>
<td>ehci usb</td>
<td>qemu</td>
<td>7.1.50</td>
<td>i386</td>
<td>Aborted in usb_ep_get()</td>
<td>N/A</td>
<td>Vi, QF</td>
<td>Patch submitted</td>
</tr>
<tr>
<td>9</td>
<td>ehci usb</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Assertion failure in address_space_unmap()</td>
<td>N/A</td>
<td>Vi, QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>10</td>
<td>exynos421_fimd display</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Asserted assertion in fimd_update_memory_allocation()</td>
<td>2</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>11</td>
<td>fgmae100 net</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Heap buffer overflow in aosped_smk flashes()</td>
<td>2</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>12</td>
<td>huck_usb_phy</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Out of bounds in inux_usbphy_read()</td>
<td>1</td>
<td>Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>13</td>
<td>intel-hda audio</td>
<td>vbox</td>
<td>7.0.7</td>
<td>i386</td>
<td>Global buffer overlow in hluMmioWrite()</td>
<td>1</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>14</td>
<td>lan9118 net</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Out of bounds read in lan9118</td>
<td>535</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>15</td>
<td>lan9118 net</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Aborted in lan9118_tbit_mode_read()</td>
<td>1</td>
<td>Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>16</td>
<td>ls13cx895a storage</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Assertion failure in ls13cx895a_emulator</td>
<td>N/A</td>
<td>Ny, Vi, QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>17</td>
<td>megasas storage</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Assertion failure in scsi_dma_complete()</td>
<td>N/A</td>
<td>QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>18</td>
<td>megasas storage</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Assertion failure in hlvio_co_write_seq_prepare()</td>
<td>N/A</td>
<td>QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>19</td>
<td>mve storage</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Null pointer access in memory_region_set_enabled()</td>
<td>1</td>
<td>Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>20</td>
<td>ohci usb</td>
<td>qemu</td>
<td>7.0.50</td>
<td>i386</td>
<td>Assertion failure in usb_msd_transfer_data()</td>
<td>29</td>
<td>Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>21</td>
<td>ohci usb</td>
<td>qemu</td>
<td>7.0.50</td>
<td>i386</td>
<td>Aborted in ohci_frame_boundary()</td>
<td>8</td>
<td>VS, QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>22</td>
<td>ohci usb</td>
<td>qemu</td>
<td>7.0.50</td>
<td>i386</td>
<td>Heap use after free in usb_cancel_packet()</td>
<td>67</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>23</td>
<td>ohci usb</td>
<td>qemu</td>
<td>7.0.91</td>
<td>i386</td>
<td>Assertion failure in usb_cancel_packet()</td>
<td>79</td>
<td>An</td>
<td>Open</td>
</tr>
<tr>
<td>24</td>
<td>omap_dss display</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch32</td>
<td>Out of memory in hw omap-dss for aarch32</td>
<td>3</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>25</td>
<td>pl041 audio</td>
<td>qemu</td>
<td>7.0.94</td>
<td>aarch32</td>
<td>Aborted in audio_budg() triggered by pl041</td>
<td>1</td>
<td>An</td>
<td>Fixed</td>
</tr>
<tr>
<td>26</td>
<td>pl041 audio</td>
<td>qemu</td>
<td>7.0.94</td>
<td>aarch32</td>
<td>Aborted in audio_budg() triggered by pl041</td>
<td>2</td>
<td>An</td>
<td>Open</td>
</tr>
<tr>
<td>27</td>
<td>sb16 audio</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Aborted in audio_calloc() caused by sb16</td>
<td>N/A</td>
<td>An</td>
<td>Fixed</td>
</tr>
<tr>
<td>28</td>
<td>sb16 audio</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Aborted in audio_calloc()</td>
<td>4</td>
<td>Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>29</td>
<td>sdhci storage</td>
<td>qemu</td>
<td>7.1.50</td>
<td>i386</td>
<td>Heap buffer overflow in sdi_channel_dataport()</td>
<td>9</td>
<td>QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>30</td>
<td>smc91c111 net</td>
<td>qemu</td>
<td>7.1.93</td>
<td>arc32</td>
<td>Out of bounds read/write in smc91c111</td>
<td>5</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>31</td>
<td>tc6393xb device</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch32</td>
<td>Negative size param in nds blk load 512()</td>
<td>23</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>32</td>
<td>tc6393xb device</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch32</td>
<td>Heap buffer overflow in nds blk write 512()</td>
<td>7</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>33</td>
<td>virtioblk storage</td>
<td>qemu</td>
<td>7.0.94</td>
<td>i386</td>
<td>Assertion failure in address_space_write_cw_cached()</td>
<td>5</td>
<td>An</td>
<td>Fixed</td>
</tr>
<tr>
<td>34</td>
<td>virtioblk storage</td>
<td>qemu</td>
<td>7.0.94</td>
<td>i386</td>
<td>Infinite loop in virtio blk handle vq()</td>
<td>16</td>
<td>An</td>
<td>Fixed</td>
</tr>
<tr>
<td>35</td>
<td>vmxnet3 net</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Code should not be reached vmxnet3_co_write_seq()</td>
<td>N/A</td>
<td>Vi, QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>36</td>
<td>vmxnet3 net</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>These loop errors in vmxnet3 validate genes()</td>
<td>N/A</td>
<td>QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>37</td>
<td>vmxnet3 net</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Assertion failure in vmxnet3_co_write_seq()</td>
<td>N/A</td>
<td>QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>38</td>
<td>vmxnet3 net</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Out of memory net iv xplan unit()</td>
<td>N/A</td>
<td>QF, VS</td>
<td>Fixed</td>
</tr>
<tr>
<td>39</td>
<td>vmxnet3 net</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>Assertion failure in net iv xplan reset()</td>
<td>N/A</td>
<td>QF</td>
<td>Fixed</td>
</tr>
<tr>
<td>40</td>
<td>vmxnet3 net</td>
<td>qemu</td>
<td>6.1.50</td>
<td>i386</td>
<td>eh_get_gw_type: code should not be reached</td>
<td>N/A</td>
<td>QF, VS</td>
<td>Fixed</td>
</tr>
<tr>
<td>41</td>
<td>xhci usb</td>
<td>qemu</td>
<td>7.0.94</td>
<td>i386</td>
<td>Aborted in xhci_find_stream()</td>
<td>56</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>42</td>
<td>xilinx_dp display</td>
<td>qemu</td>
<td>7.0.91</td>
<td>aarch64</td>
<td>Aborted in xilinx_dp_aux_set_command()</td>
<td>1</td>
<td>Vi</td>
<td>Fixed</td>
</tr>
<tr>
<td>43</td>
<td>xilinx_dp display</td>
<td>qemu</td>
<td>7.0.50</td>
<td>aarch64</td>
<td>Out of bounds read in xilinx_dp_read()</td>
<td>1</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>44</td>
<td>xilinx_dp display</td>
<td>qemu</td>
<td>7.1.50</td>
<td>aarch64</td>
<td>Out of bounds in xilinx_dp_ribbon_read()</td>
<td>N/A</td>
<td>An</td>
<td>Fixed</td>
</tr>
<tr>
<td>45</td>
<td>xilinx_dp display</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Overflow in xilinx_dp_aux flush rx_fifo()</td>
<td>3</td>
<td>Vi</td>
<td>Patch submitted</td>
</tr>
<tr>
<td>46</td>
<td>xilinx_dp display</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Aborted in xilinx_dp_change_graphic_fmt()</td>
<td>1</td>
<td>Vi</td>
<td>Patch submitted</td>
</tr>
<tr>
<td>47</td>
<td>xilinx_dp display</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Underflow in xilinx_dp_aux_pop rx_fifo()</td>
<td>1</td>
<td>Vi</td>
<td>Patch submitted</td>
</tr>
<tr>
<td>48</td>
<td>xilinx_dp display</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Overflow in xilinx_dp_aux_flush rx_fifo()</td>
<td>17</td>
<td>Vi</td>
<td>Patch submitted</td>
</tr>
<tr>
<td>49</td>
<td>xilinx_zynqmp_can net</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Fifo underflow in transfer fifo()</td>
<td>2</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>50</td>
<td>xilinx_zynqmp_can net</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Fifo overflow in transfer fifo()</td>
<td>291</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>51</td>
<td>xilinx_zynqmp_uart0 net</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Out of bound in xilinx_uart0_write()</td>
<td>1</td>
<td>Vi</td>
<td>Open</td>
</tr>
<tr>
<td>52</td>
<td>xilinx_zynqmp_uart0 net</td>
<td>qemu</td>
<td>7.2.50</td>
<td>aarch64</td>
<td>Underflow in xilinx_dp_aux_pop rx_fifo()</td>
<td>2</td>
<td>Vi</td>
<td>Open</td>
</tr>
</tbody>
</table>

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 ViDiZZO (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 “Fixed(us)”; particularly, if our patch is accepted, we mark it Fixed(us).