MundoFuzz: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference

Cheolwoo Myung, Gwangmu Lee, and
Byoungyoung Lee, *Seoul National University*

https://www.usenix.org/conference/usenixsecurity22/presentation/myung

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

MUNDOFUZZ: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference

Cheolwoo Myung
Seoul National University
cwmyung@snu.ac.kr

Gwangmu Lee
Seoul National University
gwangmu@snu.ac.kr

Byoungyoung Lee*
Seoul National University
byoungyoung@snu.ac.kr

Abstract

A hypervisor is system software, managing and running virtual machines. Since the hypervisor is placed at the lowest-level in the typical systems software stack, it has critical security implications. Once compromised, the entire software components running on top of the hypervisor (including all guest virtual machines and applications running within each guest virtual machine) are compromised as well, as the hypervisor has all the privileges to access those.

This paper proposes MUNDOFUZZ, a hypervisor fuzzer to enable both coverage-guided and grammar-aware fuzzing. We find that the coverage measurement in hypervisors suffers from noises due to the hypervisor's asynchronous system event handling. In order to filter out such noises, MUNDOFUZZ develops a statistical differential coverage measurement methods, allowing MUNDOFUZZ to capture the clean coverage information for hypervisor inputs. Moreover, we observe that hypervisor inputs have complex input grammars because it supports many different devices and each device has its own input format. Thus, MUNDOFUZZ learns the input grammar through inspecting the coverage characteristics of the given hypervisor input, which is based on the idea that the hypervisor behaves in a different way if the grammatically correct (or incorrect) input is given. We evaluated MUNDOFUZZ with popular hypervisors, QEMU and Bhyve, and MUNDOFUZZ outperformed other state-of-the-art hypervisor fuzzers ranging from 4.91% to 6.60% in terms of coverage. More importantly, MUNDOFUZZ identified 40 previously unknown bugs (including 9 CVEs), demonstrating its strong practical effectiveness in finding real-world hypervisor vulnerabilities.

1 Introduction

A hypervisor is system software, managing and running virtual machines (VMs). The key technical advantage of using the hypervisor is in its resource virtualization capability, which enables a single host machine to run multiple guest VMs. This advantage becomes particularly important as the computing trends are rapidly shifting towards cloud computing. Using the hypervisor, cloud computing platforms can facilitate flexible resource managements on virtualized platforms, which can efficiently service for billions of users [18]. From the security point of view, hypervisors introduce crucial attack surface to be protected. Importantly, a hypervisor is running at the lowest layer in the typical systems software stack, so it is given the highest security privilege of the host machine. Consequently, any successful compromise exploiting hypervisor vulnerabilities would allow attackers to gain privilege of the entire host machine, including the hypervisor itself as well as all guest VMs running on top of them.

Meanwhile, fuzzing techniques have been shown its effectiveness in identifying unknown vulnerabilities in various software systems. The fuzz testing keeps running the target software with a randomly generated or mutated input. While running, it identifies the vulnerability by checking if the run leads to any unexpected behavior (e.g., a crash, a hang, or raising the assert violation). In general, the effectiveness of the fuzz testing heavily relies on the quality of its randomly generated or mutated input, as it determines the overall testing coverage with respect to the target software. In order to improve the quality of test inputs, many traditional fuzzers leverage two key features, coverage-guided fuzzing [29,39,40,52] and grammar-aware fuzzing [27,35,36,38,45]. First, the coverage-guided fuzzing improves the quality of input by leveraging code coverage measurement. Based on the coverage measurement, it keeps evaluating the quality of tested inputs. Then it only retains the good quality of inputs for the future test while discarding the bad quality of inputs, which gradually helps the fuzzer keep testing the good quality of inputs. Second, the grammar-aware fuzzing directly encodes the grammar of input semantics into the input generation and mutation logic, such that the fuzzer tests grammatically correct inputs if possible. This allows the grammar-aware fuzzing technique to explore in-depth logic of the target program, which helps to augment the testing coverage.

*Corresponding author

When it comes to hypervisor fuzzing, we find that it is challenging to employ aforementioned two key fuzzing features, coverage-guided and grammar-aware fuzzing. This is due to the following unique systems characteristics of hypervisors. First, it is challenging to employ coverage-guided fuzzing in hypervisors because the precise coverage measurement of hypervisor execution is difficult. Specifically, hypervisors are responsible of handling various events, ranging from asynchronous ones from interrupts and a timer to deterministic ones from non-target devices, which are not relevant to a specific feature targeted by the provided inputs. Since all these *noise* events will be attributed to the coverage measurement, the measured coverage for each input would be imprecise. It is worth noting that the previous work VDF [37] attempted to handle this noise issue through manual hypervisor code partitioning, but it require non-trivial manual efforts.

Second, grammar-aware fuzzing for hypervisors is challenging because of complex inputs grammars that the hypervisor accepts. Hypervisors support many different virtual devices, where each device has its own unique input semantics. Moreover, these devices accept the input through various IO interfaces (i.e., port IO, memory-mapped IO, and DMA), which further complicates the input grammar. In order to handle this issue, the previous work NYX [46] manually wrote the grammar rule based on the documentation, but this requires heavy manual efforts considering the complexity of IO interfaces as well as a large number of virtual devices.

This paper proposes MUNDOFUZZ, a hypervisor[1] fuzzer that is carefully designed to enable both coverage-guided and grammar-aware fuzzing. First, in order to enable coverage-guided fuzzing, MUNDOFUZZ proposes a new statistical differential testing approach to precisely measure the coverage. This is based on the observation that while the true coverage information would be measured in the same way, the noisy-coverage information would appear in a different way if the coverage is measured multiple times. Leveraging this observation, MUNDOFUZZ takes statistical complement and intersection from multiple coverage measurement to obtain the clean coverage information from the hypervisor input. Second, to enable the grammar-aware fuzzing, MUNDOFUZZ automatically infers grammar rules regarding the device register types (e.g., control, data and DMA address registers) and dependencies in hypervisor inputs. The key idea behind this grammar learning is that the hypervisor behaves differently if the grammatically correct (or incorrect) hypervisor input is given. Based on this idea, MUNDOFUZZ is capable of inferring the grammar through inspecting the coverage characteristics of the given hypervisor input.

We implemented the prototype of MUNDOFUZZ and evaluated it with popular hypervisors, QEMU and Bhyve. According to our evaluation, MUNDOFUZZ outperforms the state-of-the-art hypervisor fuzzer HYPER-CUBE by 4.91% and NYX by 6.60% in terms of coverage. Looking into the details, MUNDOFUZZ's precisely measured coverage significantly improved the accuracy of grammar inferences, increasing the accuracy of register types 87.2% on average. Moreover, MUNDOFUZZ's automated grammar inference showed a comparable performance to NYX, which manually specified the grammar rules. MUNDOFUZZ also identified 40 new bugs including 9 CVEs (i.e., 23 bugs in QEMU and 17 bugs in Bhyve), which demonstrate its strong practical effectiveness in finding real-world hypervisor vulnerabilities.

[1]Specifically, Type-2 hypervisor in the paper.

2 Background

2.1 Hypervisor

A hypervisor is a software application which runs multiple virtual machine (VM) instances. The major role of a hypervisor is to virtualize a set of machine resources, such as the number of processors or memory space, and to provide the functionalities of peripheral devices to VM instances.

One of the core features in hypervisors is in emulating devices, known as virtual devices. In particular, hypervisors typically use the trap mechanism to emulate privileged instructions, which delegate the access to the underlying privileged resources. For example, any operation that directly accesses a device register from a VM instance is designed to raise a trap, which hands over the control to the hypervisor. Then the hypervisor accordingly emulates the operation by forwarding it to the corresponding virtual device in the hypervisor. After the virtual device finishes the operation, the hypervisor eventually returns the results and resumes the VM.

Input Space of Hypervisors. A virtual device in a hypervisor accepts various IO operations from VM instances ranging from port IO (PIO) and memory-mapped IO (MMIO) to direct memory address (DMA). Primarily, a virtual device exposes device registers through PIO and MMIO. PIO is the most primitive input channel that presents a separate address space that is mapped to device registers, where the address space is only addressable through special instructions (`in`/`out`). On the other hand, MMIO dedicates a range of memory address to device registers, so that device registers are addressable via memory operations (e.g., `load` or `store`).

A virtual device can also accept IO operations through DMA as well. In DMA, the OS kernel (i.e., the guest kernel that resides in a VM instance) first allocates a shared memory buffer and informs the address to the virtual device. The virtual device then directly accesses the buffer to receive IO operations or place return data.

Completion Signal. When a virtual device completes a specified task by the IO operations and falls back to the idle state, it notifies the VM instance by sending a completion signal. The completion signal can be through hardware interrupt or a dedicated device register, where a certain bit signifies the

Techniques	Input Space			Coverage -guided	Noise Coverage Reduction	Semantic Input Synthesis
	PIO	MMIO	DMA			
VDF [37]	✓	✓	✗	△	△	✗
HYPER-CUBE [47]	✓	✓	✓	✗	✗	✗
NYX [48]	✓	✓	✓	✓	✗	△
MUNDOFUZZ	✓	✓	✓	✓	✓	✓

△: Manual work required.

Table 1: Comparison of major hypervisor fuzzers.

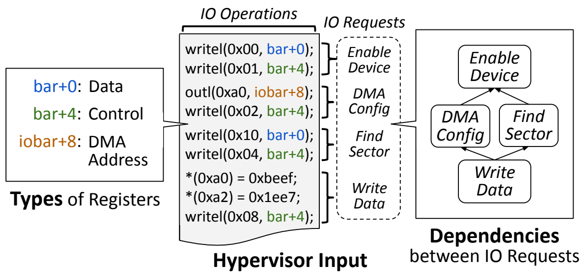

```
writel(0x00, bar+0);
writel(0x01, bar+4);
outl(0xa0, iobar+8);
writel(0x02, bar+4);
writel(0x10, bar+0);
writel(0x04, bar+4);
*(0xa0) = 0xbeef;
*(0xa2) = 0x1ee7;
writel(0x08, bar+4);
```

Figure 1: Example of a hypervisor input and its two key semantic constraints; types of registers and dependencies between IO requests.

device state. However, for most virtual devices, the method is configurable by the VM instance.

Determinism. Most virtual devices solely depend on the IO operations provided by VM instances. However, some virtual devices are asynchronous and depend on non-deterministic sources, such as physical time. For example, since timers and interrupt controllers depend on physical time, their operations are generally irrelevant to the IO operations.

2.2 Fuzzing Hypervisors

Overall, previous works on hypervisor fuzzing have adopted the general lessons from traditional fuzzing techniques, but the direction of development has been changed due to the characteristics of hypervisors.

Coverage-guided Fuzzing. At the early stage, hypervisor fuzzing followed the convention of general coverage-guided fuzzing techniques [15, 52]. VDF [37] is one of the early coverage-guided hypervisor fuzzer, which fuzzes the virtual devices in QEMU using AFL [52]. To suppress coverage noises from irrelevant hypervisor parts such as non-target virtual devices and asynchronous interrupts, VDF manually partitions the hypervisor into stand-alone virtual device parts and fuzzed them separately. However, this process requires heavy manual efforts to extract virtual devices, and is also error-prone while manually partitioning the hypervisor code.

Recently, NYX [46] also employs the coverage-guided fuzzing by instrumenting the entire hypervisor. However, since NYX does not manually partition the hypervisor code, it suffers from the noises in the coverage measurement.

Throughput-oriented Fuzzing. As incorporating coverage guide is tricky, another approach preferred fuzzing performance to coverage guide. For example, HYPER-CUBE [47] drops the coverage guide support and rather randomly generates hypervisor inputs on-the-fly. However, since this approach does not recognize meaningful hypervisor inputs, it cannot develop hypervisor inputs for complex device states.

Grammar-based Fuzzing. NYX [46] employs a grammar-based fuzzing technique to explore complex device states. To enable grammar awareness, NYX utilized grammar specification in device documentations and manually wrote the grammar rules for each device. However, manually embedding grammar specification for all virtual devices requires an unacceptable manual efforts. Furthermore, the device documentation is non-trivial to interpret due to the complex details, manually organized grammars may end up with incorrect grammar rules.

3 Challenges in Hypervisor Fuzzing

In this section, we articulate two key challenges in performing hypervisor fuzzing, namely i) noises in coverage measurement, which deters the coverage-guided fuzzing (§3.1) and ii) complex input semantics, which makes difficult to enable the grammar-aware fuzzing (§3.2).

3.1 Noises in Coverage Measurement

Since hypervisors are just an application from the perspective of a host machine, measuring the coverage of hypervisors may seem straightforward, similar to measuring the coverage of typical user applications.

However, we find that coverage measurement for hypervisors is challenging largely due to its inherent system characteristics. As noted in §2, a hypervisor is designed to virtualize the entire machine resources, so it is responsible for handling various asynchronous events, notably interrupts and timer events. Because such asynchronous events are irrelevant to a certain input to the hypervisor, simply employing traditional coverage measurement techniques would suffer from severe non-deterministic noises. For instance, if any asynchronous interrupt is triggered while measuring the coverage of a certain IO operation, the resulting coverage ends up being a mixture of target device coverage and interrupt coverage. These non-deterministic noises in coverage measurements would critically harm the coverage-guided fuzzing, because the coverage-guided fuzzing assumes the deterministic coverage measurement—i.e., the same input should lead to the same coverage.

3.2 Complex Input Grammar

We find that hypervisor inputs have complex input grammar semantics, challenging the fuzzing performance for efficient

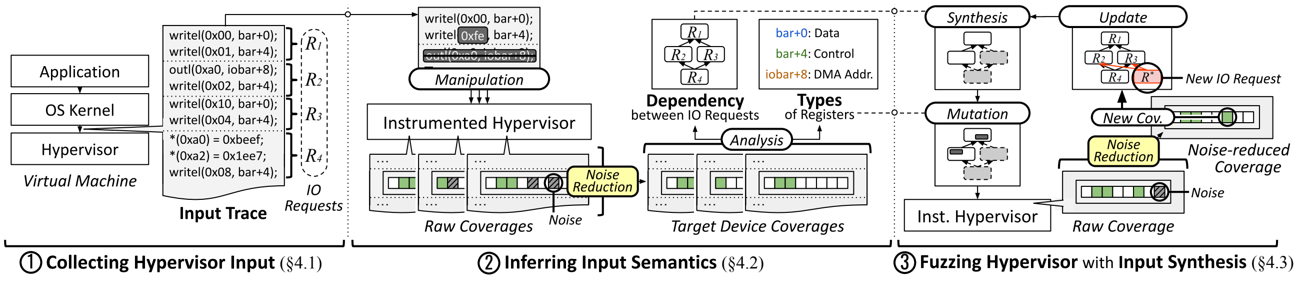


Figure 2: Overall workflow of MUNDOFUZZ.

input generation and mutation. This is largely due to the fact that most of hypervisor inputs are presented in low level memory or IO operations, and each memory operation can have its own unique semantic meanings depending on i) a memory or IO address; ii) a value it is writing; iii) a context that the memory or IO operation is performed.

In order to clearly understand this challenge, we take an example with a simplified hypervisor input for a generic disk device (illustrated in Figure 1). The input consists of a sequence of low-level IO operations that feed values through PIO/MMIO registers (i.e., using `outl()`/`writel()`) and writes data to a DMA buffer (i.e., `*(0xa0)=...`). To initialize the device, it first aligns the current sector to 0 before enabling the device ("Enable Device") and configures the DMA buffer for data transfer ("DMA Config"). After initialization, it finds the target sector `0x10` ("Find Sector") and writes data through the DMA buffer ("Write Data").

As shown in this example, hypervisor inputs are a short sequence of low-level IO operations, which is difficult to understand its semantic meaning by looking at an individual IO operation. In order to reconstruct the semantic meaning, one needs to understand the operational characteristics of hypervisor inputs: i) a collective set of IO operations forms a certain high-level task in a device; ii) a target memory address of an IO operation (i.e., device registers) has a dedicated semantic meaning; and iii) IO operations have order dependencies between them, which should be kept to correctly function. Based on these operational characteristics, we elaborate each characteristic in the followings.

IO Request: A High-level Semantic Unit. To clearly denote the high-level semantic unit, we define the term, *an IO Request*, which is a collective set of IO operations constituting a high-level semantic task. Such a high-level task is completed with a notification signal, delivered through either an interrupt or a status register. For example, IO requests in Figure 1 are "Enable Device", "DMA Config", "Find Sector" and "Write Data". We note that an IO request qualifies as a semantic unit of hypervisor inputs, because it has the property of requesting a specific functionality to its downstream (i.e., the hypervisor).

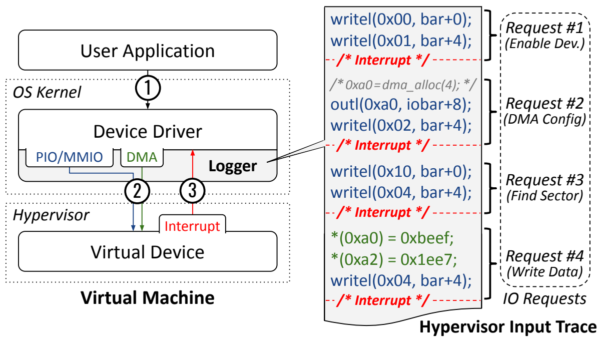
Semantics in Device Register. Device registers in hypervisor can be considered as a memory address, which is a main communication channel between a hypervisor and a device. Depending on the usage, device registers can be categorized into following three types: i) a control register, which transfers the control values that indicate the desired function or operation mode; ii) a data register, which transfers a data parameter that is necessary for device functions; and iii) a DMA address register, which transfers the base address of a DMA buffer to the device.

From the perspective of grammar-aware fuzzing, recognizing these register types is important grammar information as these critically impact the hypervisor operations. For example in Figure 1, the register `bar+4` is a control register that invokes the desired functionality inside the disk device, the register `bar+0` is a data register that specifies a desired sector number to the device, and the register `iobar+8` is a DMA register that informs the DMA buffer address. If the fuzzer is not aware of these register types when generating or mutating hypervisor inputs, its fuzzing performance would be severely limited. To be specific, while mutating control registers is likely extend the code coverage (because it would try different functionality), mutating data registers is not (because it would simply provide different value). More critically, if the fuzzer does not recognize the DMA register, it would not be able to synthesize (or mutate) any semantically correct hypervisor inputs performing DMA operations. This is because all the following DMA operations would be depending on the DMA buffer address which is delivered through the DMA register.

Dependency in IO Requests. We further observe that the IO requests also have dependencies to each other, meaning that there are necessary orders between IO requests to correctly function. For example in Figure 1, the IO request "Enable Device" needs to precede all IO requests since any IO requests are invalid before it enables the device. Similarly, "Find Sector" needs to precede "Write Data" as data can only be written properly after the disk device finds the sector. Maintaining such dependency relations is essential for synthesizing hypervisor inputs, as they are functional only when a sequence of IO requests observes the dependencies.

4 Design of MUNDOFUZZ

MUNDOFUZZ is a coverage-guided hypervisor fuzzer, which synthesizes hypervisor inputs with grammar awareness.



```
writel(0x00, bar+0);
writel(0x01, bar+4);
/* Interrupt */
/* 0xa0=dma_alloc(4); */
outl(0xa0, iobar+8);
writel(0x02, bar+4);
/* Interrupt */
writel(0x10, bar+0);
writel(0x04, bar+4);
/* Interrupt */
*(0xa0) = 0xbeef;
*(0xa2) = 0x1ee7;
writel(0x04, bar+4);
/* Interrupt */
```

Figure 3: Overview of hypervisor input collection (§4.1).

MUNDOFUZZ is designed to address the two challenges described in §3. First, to remove coverage noise, MUNDOFUZZ employs statistical and differential testing techniques to precisely measure the clean coverage information (§4.2.1). Second, to synthesize grammatically correct hypervisor inputs, MUNDOFUZZ automatically learns grammar rules regarding the device register types and dependencies between IO requests (§4.2).

The overall workflow of MUNDOFUZZ is illustrated in Figure 2. MUNDOFUZZ first collects hypervisor input traces issued by various real-world applications and partition them into *IO requests* as the units of semantics (§4.1). Next, MUNDOFUZZ infers grammar rules through leveraging the coverage characteristics of hypervisor operations, particularly focusing on two things: i) the device register types and ii) the dependencies between IO requests (§4.2). Finally, during a fuzzing run, MUNDOFUZZ synthesizes hypervisor inputs using the inferred grammar constraints (§4.3).

4.1 Collecting Hypervisor Input

The goal of hypervisor input collection is to collect a variety of inputs for virtual devices. To do this, MUNDOFUZZ collects real-world input traces by monitoring the IO interactions between the OS kernel and the device at the kernel level.

Workflow. Figure 3 shows the overview of the IO operation collection. While user applications (e.g., `dd` and `fsck`) invoke a device driver (①), the MUNDOFUZZ logger intercepts the PIO/MMIO and DMA operations issued by the kernel and records them to a hypervisor input trace (②). Upon receiving an interrupt that notifies the completion of a request, the logger slices the trace to create an IO request (③).

Recording PIO/MMIO Operations. To collect PIO and MMIO operations, MUNDOFUZZ leverages the common practice that the kernel provides PIO/MMIO APIs to communicate with peripheral devices. Notice that these primitive APIs are well-documented in most popular OSes, as these are heavily used by the third-party drivers. For example in the Linux kernel, `readl()`/`writel()` are the dedicated primitive APIs for the MMIO accesses, and `inl()`/`outl()` are the primitive APIs for the PIO accesses.

As such, MUNDOFUZZ attaches a logger to these kernel primitive APIs for PIO/MMIO. Logging these APIs, MUNDOFUZZ is able to record all the PIO/MMIO operations with their access type (e.g., read or write), access address, and the write value if applicable. It is worth noting that device drivers may directly utilize host instructions to communicate with PIO/MMIO. In this case, MUNDOFUZZ may utilize page fault traps as employed for DMA below, but we have not encountered such cases during our evaluation with Linux.

Recording DMA Operations. Unlike PIO/MMIO, DMA operations are not performed through dedicated kernel APIs, because its operations can be seen as ordinary memory read-/write instructions. Thus, MUNDOFUZZ leverages the fact that most OSes provide a primitive API to allocate a DMA memory buffer (e.g., `dma_map_single()` in the Linux kernel). More specifically, the MUNDOFUZZ logger monitors all DMA buffer allocation events, then it installs a page fault trap for all the memory pages allocated for new DMA buffers. Later when a DMA operation attempts to access the DMA buffer, the MUNDOFUZZ logger intercepts the page fault handler and records the DMA operation with its access address and the write value. To transparently resume the execution, MUNDOFUZZ temporarily allows the DMA operation to access the buffer again and reinstalls a page fault trap right after the access using the single-step debugging feature [3].

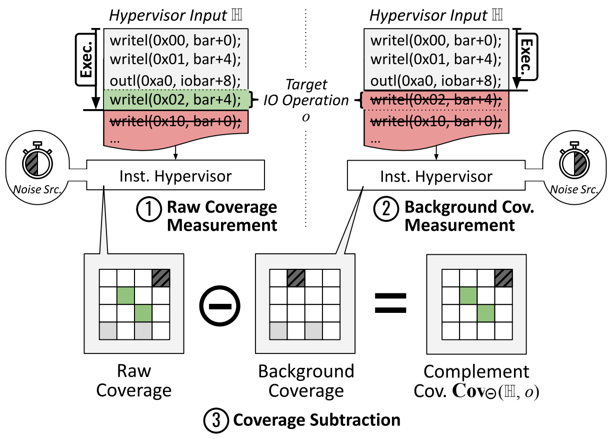
Recording DMA Buffer Allocations. Along with recording DMA operations, MUNDOFUZZ additionally records the DMA buffer allocation events for the future inference task that we describe later in §4.2.2 and §4.2.3. Specifically, MUNDOFUZZ intercepts the DMA buffer allocation API and records the allocation events with their addresses and sizes.

4.2 Inferring Input Semantics

Using the collected input trace, MUNDOFUZZ infers the semantic constraints (i.e., the types of registers and the dependencies between IO requests) by observing the coverage characteristics of the target device while algorithmically manipulating the trace. To this end, MUNDOFUZZ first constructs a primitive algorithm to remove coverage noise from an instrumented hypervisor (§4.2.1). Using this, MUNDOFUZZ then infers the register types and IO request dependencies by inspecting coverage characteristics (§4.2.2 and §4.2.3).

4.2.1 Removing Noises in Coverage

To remove coverage noise, MUNDOFUZZ takes a statistical differential testing approach as follows. First, MUNDOFUZZ first obtains the coverage of an individual IO operation, which still contains non-deterministic noises. Second, MUNDOFUZZ removes the non-deterministic noise by iteratively intersecting the coverage. Finally, MUNDOFUZZ reconstructs the target device coverage (i.e., an IO request coverage) by merging the coverages from individual IO operations.



```
writel(0x00, bar+0);
writel(0x01, bar+4);
outl(0xa0, iobar+8);
writel(0x02, bar+4);
~~writel(0x10, bar+0);~~
...
```

```
writel(0x00, bar+0);
writel(0x01, bar+4);
outl(0xa0, iobar+8);
~~writel(0x02, bar+4);~~
~~writel(0x10, bar+0);~~
...
```

Figure 4: Workflow of IO operation coverage capture (§4.2.1). The green box (🟩) and the lighter grey box (⬜) represent the target and non-target IO operation coverage, respectively. The dark gray box (⬛) with hatch patterns represents non-deterministic noise.

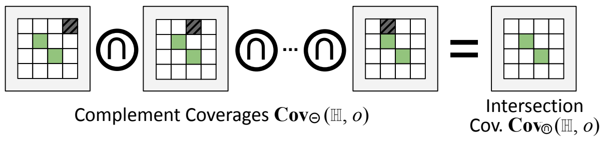
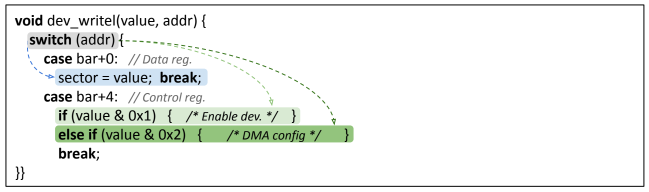

Complement Coverages $\mathbf{Cov}_{\ominus}(\mathbb{H}, o)$
Intersection Cov. $\mathbf{Cov}_{\circledcirc}(\mathbb{H}, o)$

Figure 5: Overview of noise coverage removal (§4.2.1). The green box (🟩) represents the target IO operation coverage, and the grey box (⬛) with hatch patterns represents non-deterministic noise.

Capturing Noisy IO Operation Coverage. Notice that as the hypervisor executes more IO operations, it is more likely that the coverage feedback suffers from more noises. To minimize this side-effect as much as possible, this stage first narrows down the scope of coverage measurement to the smallest level, namely a single target IO operation.

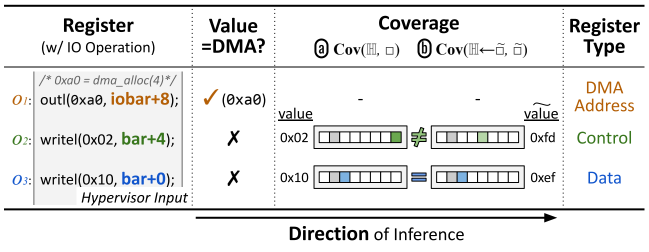
It is worth noting that simply executing a single IO operation alone would not show the original coverage, because it depends on the device state established by the preceding IO operations. Thus, we take the differential testing approach: we measure the coverage of two hypervisor inputs, one input includes the target IO operation and the other does not. Then we take the complement of these so as to obtain the coverage of the target IO operation.

For instance, Figure 4 describes the procedure step by step. First, MUNDOFUZZ measures the raw coverage by executing the hypervisor input up to the target IO operation, which captures non-deterministic noises (⬛) as well as the true target IO operation coverage (🟩) (①). Next, MUNDOFUZZ measures the background coverage by re-executing the same hypervisor input but excluding the target IO operation, which does not contain the true target IO operation coverage (②). Finally, MUNDOFUZZ takes the complement coverage $\mathbf{Cov}_{\ominus}$ by subtracting two coverages (③). This results in the coverage of the target IO operation as well as non-deterministic noises. As a next step, we describe how to filter out non-deterministic noises to obtain the clean target IO operation.



```
void dev_writel(value, addr) {
  switch (addr) {
    case bar+0:  // Data reg.
      sector = value;  break;
    case bar+4:  // Control reg.
      if (value & 0x1)  {  /* Enable dev. */  }
      else if (value & 0x2)  {    /* DMA config */    }
      break;
}}
```

(a) Example MMIO handler in the virtual device. The shade colors (⬛, 🟦, 🟩 and 🟩) represent the coverage activated when the corresponding code is touched.



	Register (w/ IO Operation)	Value =DMA?	Coverage ⓐ $\mathbf{Cov}(\mathbb{H}, \square)$				Coverage ⓑ $\mathbf{Cov}(\mathbb{H} \leftarrow \tilde{\square}, \tilde{\square})$	Register Type
	/* 0xa0 = dma_alloc(4)*/		value				$\widetilde{\text{value}}$	
o_1:	outl(0xa0, iobar+8);	✓ (0xa0)		-		-		DMA Address
o_2:	writel(0x02, bar+4);	✗	0x02	▭	$\neq$	▭	0xfd	Control
o_3:	writel(0x10, bar+0);	✗	0x10	▭	=	▭	0xef	Data
	Hypervisor Input							

(b) Inferring the type of registers in Figure 3. The colored boxes (⬛, 🟦, 🟩 and 🟩) represent the coverage activated by touching the corresponding code in Figure 6a. □ represents the placeholder for the IO operations (o_1, o_2 and o_3) on the corresponding row. $\tilde{\square}$ denotes the IO operation with the bitwise inverted value.

Figure 6: Workflow of device register type inference (§4.2.2).

Removing Noises from IO Operation Coverage. Since non-deterministic noises result in different coverages in every execution, simply taking the complement (as we have done before) does not remove non-deterministic noises. To handle this issue, we take the statistical approach based on the following idea: even if non-deterministic noises keep changing, the true target IO operation coverage would always be measured. In order to leverage this idea, we take a sufficiently large number of complement coverages and effectively remove any changing coverage by intersecting them all.

More formally, let $\mathbf{Cov}_{\ominus}(\mathbb{H}, o)$ be the complement coverage of the target IO operation o in the hypervisor input $\mathbb{H}$. If we have n-many complement coverages, we can define the intersection coverage $\mathbf{Cov}_{\circledcirc}(\mathbb{H}, o)$ as $\mathbf{Cov}_{\circledcirc}(\mathbb{H}, o) = \cap_{i \in [1,\dots,n]} \mathbf{Cov}^{i}_{\ominus}(\mathbb{H}, o)$.

For example, Figure 5 illustrates this procedure. While each complement coverage includes its own non-deterministic noises (⬛), taking the intersection of those cancels out those noises. As a result, the intersection coverage only includes the target device coverage (🟩).

Reconstructing IO Request Coverage. Based on the clean coverage of each IO operation, we can now reconstruct the coverage of an IO request by summing up the coverage of IO operations. Formally, we denote $\mathbf{Cov}(\mathbb{H}, \vec{R})$ as the coverage of $\vec{R}$ (i.e., an IO request) in $\mathbb{H}$ (i.e., a hypervisor input), which can be computed as $\mathbf{Cov}(\mathbb{H}, \vec{R}) = \cup_{o \in \vec{R}} \mathbf{Cov}_{\circledcirc}(\mathbb{H}, o)$.

4.2.2 Device Register Type Inference

We identify three types of registers through which a hypervisor accepts as input. As noted in §2.1, these include data, control, and DMA address registers. In order to identify three register types, MUNDOFUZZ goes through the procedure as follows. First, MUNDOFUZZ identifies DMA address registers by checking whether their values are rooted from the DMA buffer allocation API. Next, MUNDOFUZZ identifies control registers by inspecting whether the coverage exhibits differently when they are provided with alternative values. Finally, MUNDOFUZZ classifies the rest as data registers.

Identifying DMA Address Registers. To identify DMA address registers, MUNDOFUZZ leverages the fact that their role is providing valid DMA buffer addresses to the device, This means that they always accept the address of valid DMA buffers. Since we record the DMA allocation events in §4.1, we can recognize DMA address registers by checking whether they are provided with legitimate DMA buffer addresses.

For example, Figure 6b illustrates how MUNDOFUZZ identifies the DMA address register. When the DMA buffer allocation API is invoked `dma_alloc(0xa0, 4)`, MUNDOFUZZ marks [`0xa0`, `0xa4`) as an address range of the DMA buffer. When the first IO operation o_1 writes the value `0xa0` to the register `iobar+8`, MUNDOFUZZ recognizes that the value is within the valid DMA buffer address range, and accordingly marks the register `iobar+8` as a DMA address register.

In addition to registers, we observe that some bytes in DMA buffers also occationally accept the address of another DMA buffer, mainly to chain them together and make a linked list. In this case, we also mark the offset of such bytes in the DMA buffer as a DMA address type and treat them as such.

Identifying Control Registers. The key idea behind identifying control registers is that a control register exhibits a different control flow depending on its value. Figure 6a illustrates an example when MMIO handler receives the kernel-side MMIO register writes (`writel`). Unlike the data register `bar+0` that simply updates the local variable to the given value, the control register `bar+4` uses the value to decide the control logic (i.e., `if` and `else if`).

As such, MUNDOFUZZ infers control registers by inspecting if they exhibit different coverages when it is provided with alternative values. Currently, MUNDOFUZZ uses a bit-inverted value as an alternative value based on the observation that a control register often contains a set of control flags. As this observation may not always be true, MUNDOFUZZ also assigns a random value with a low probability (i.e., 25% in the current configuration) to handle the case that the control register takes unique constant values.

More formally, let $\tilde{o}$ be the IO operation o with a modified value, and $\mathbb{H} \leftarrow \tilde{o}$ be the hypervisor input $\mathbb{H}$ where the IO operation o is replaced to $\tilde{o}$. MUNDOFUZZ considers the type of the destination register as control if $\mathbf{Cov}(\mathbb{H}, o) \neq \mathbf{Cov}(\mathbb{H} \leftarrow \tilde{o}, \tilde{o})$.

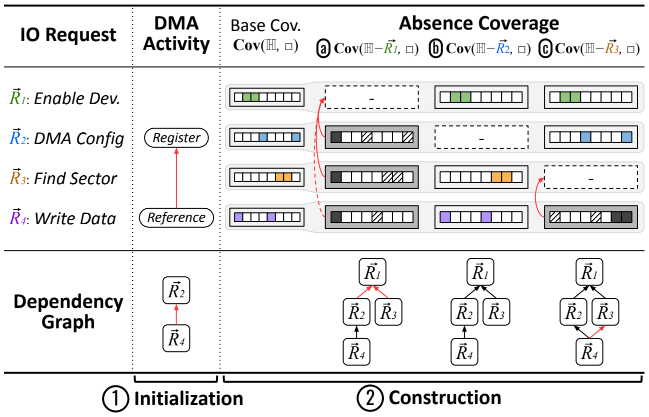


Figure 7: Workflow of IO request dependency inference (§4.2.3). IO requests are from Figure 1. $\mathbb{H}$ is the hypervisor input that concatenates the IO requests $\vec{R}_1$, $\vec{R}_2$, $\vec{R}_3$, and $\vec{R}_4$ in order. □ represents the placeholder for the IO requests ($\vec{R}_1$, $\vec{R}_2$, $\vec{R}_3$ and $\vec{R}_4$) on the corresponding row. The hatched boxes represent the different coverage from the base coverage. The red lines represent the new dependency edges created by the DMA activity or the absence coverages above.

Figure 6 demonstrates how MUNDOFUZZ infers control and data registers with an example IO sequence in Figure 3. MUNDOFUZZ first changes the value of the second IO operation o_2 to the bit-inverted value `0xfd` and inspects if it changes the coverage of o_2. Notice that while the original value `0x02` leads to the second condition (🟩) in the device-side MMIO handler in Figure 6a, the bit-inverted value `0xfd` leads to the first condition (🟩) as `0xfd&0x01` $= 1$. So MUNDOFUZZ detects the resulting coverage difference and marks the register `bar+4` as a control register. Meanwhile, the coverage of the third IO operation o_3 does not change regardless of the value, since the register `bar+0` always executes the same code (🟦). Accordingly, MUNDOFUZZ marks `bar+0` as a data register.

4.2.3 IO Request Dependency Inference

To facilitate inferring the dependencies between IO requests, we first identify two kinds of dependencies between them, namely operational and logical. The operational dependencies are the dependencies established by the IO operations that constitute the IO requests. For example, the IO request "Write Data" in Figure 1 is operationally dependent to the IO request "DMA Config", because the DMA operations in "Write Data" are only legitimate after the buffer is registered by "DMA Config". On the other hand, the logical dependencies are the dependencies established by the internal device logic. For example, the IO request "Write Data" is logically dependent to the IO request "Find Sector", because the internal device logic requires "Find Sector" to find the target sectors before "Write Data" writes.

Workflow. To address each dependency kind, MUNDOFUZZ takes separate approaches as follows. While collecting IO requests, MUNDOFUZZ first initializes the dependency graph

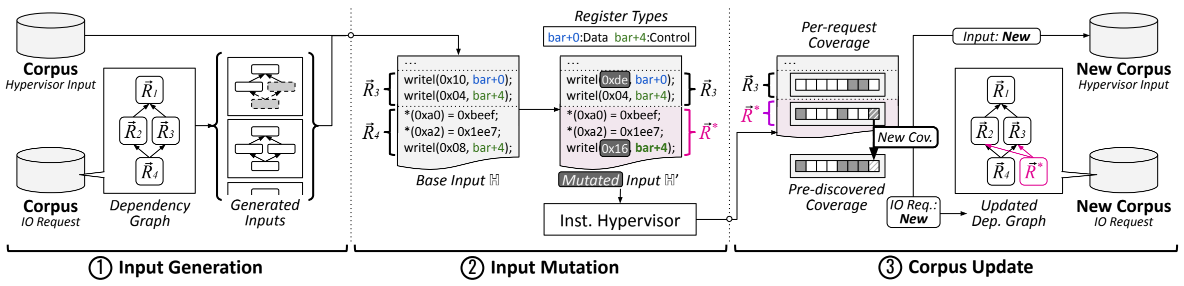


Figure 8: Workflow of fuzzing and input synthesis (§4.3). The magenta-shaded shapes represent the new request $\vec{R}^*$ created by mutating the control register `bar+4` (`0x08` $\rightarrow$ `0x16`) of the request $\vec{R}_4$. The hatched box represents new coverage discovered by $\vec{R}^*$.

using the operational dependencies revealed by the DMA operations. After initialization, MUNDOFUZZ infers the logical dependencies by inspecting the coverage variation of each IO request when other IO requests are alternately absent. If the absence of a certain IO request severely distorts the coverage of others, MUNDOFUZZ creates the dependency edges between the absent IO request and the influenced ones.

Initialization with Operational Dependencies. As noted, the operational dependencies are established by the constituting IO operations, whose dependency relations are dictated by the life cycle of DMA buffers. To review the life cycle, a kernel first allocates a DMA buffer and *registers* its address through a DMA address register. After the device recognizes the address of the DMA buffer through the DMA address register, the kernel *references* the DMA buffer with DMA operations to transfer data to the device. This life cycle suggests that the IO request registering the address of the DMA buffer must precede any IO requests referencing the buffer.

Figure 7 illustrates an example that leverages this principle to initialize the dependency graph between the IO requests in Figure 3. Since the IO request "DMA Config" registers the DMA buffer that is referenced by the IO request "Write Data", "DMA Config" must be located before "Write Data". MUNDOFUZZ accordingly creates the dependency edge between them to provision the minimum dependency graph, which must be maintained in the final result.

Construction with Inferred Logical Dependencies. Unlike the operational dependencies that can be reconstructed from the collected IO operations, the logical dependencies arise from the internal device logic, which makes the dependencies hard to figure out. In order to infer the logical dependency, we observe that an IO request works differently if the logically-dependent IO request is absent, resulting in significant coverage differences.

Thus, MUNDOFUZZ detects the logical dependencies by inspecting whether the absence of an IO request changes the coverage of another IO request. Formally, let $\mathbf{Cov}(\mathbb{H}, \vec{R})$ be the base coverage of the IO request $\vec{R}$, and $\mathbf{Cov}(\mathbb{H} - \vec{R}', \vec{R})$ be the absence coverage of $\vec{R}$ when the IO request $\vec{R}'$ is skipped. The IO request $\vec{R}$ is then logically dependent to $\vec{R}'$ if $\mathbf{Cov}(\mathbb{H}, \vec{R}) \neq \mathbf{Cov}(\mathbb{H} - \vec{R}', \vec{R})$.

Figure 7 illustrates how MUNDOFUZZ constructs the dependency graph using the hypervisor input in Figure 1 as an example. First, MUNDOFUZZ detects the dependencies between $\vec{R}_1$ and all other IO requests, as the absence of $\vec{R}_1$ changes all the following coverages (ⓐ). MUNDOFUZZ does not create the dependency edge between $\vec{R}_1$ and $\vec{R}_4$, as it is already dependent to $\vec{R}_1$ by the transitive nature of the dependency graph. Next, MUNDOFUZZ retains the dependency graph the same, as the absence of $\vec{R}_2$ does not change any following coverages (ⓑ). It is worth noting that $\vec{R}_2$ and $\vec{R}_4$ may not be logically dependent even if they are operationally dependent. Finally, MUNDOFUZZ detects the dependency edge between $\vec{R}_3$ and $\vec{R}_4$, since the absence of $\vec{R}_3$ changes the coverage of $\vec{R}_4$ (ⓒ). Notice that the final dependency graph coincides with the complete dependency graph illustrated in Figure 1.

4.3 Fuzzing Hypervisor with Input Synthesis

Using the inferred semantic constraints, MUNDOFUZZ synthesizes hypervisor inputs at fuzzing time. Since the synthesis process also requires the clear coverage from the target device, MUNDOFUZZ first provisions a mechanism to reduce the noise coverage on-the-fly (§4.3.1). Using the noise-removed coverage, MUNDOFUZZ further synthesizes hypervisor inputs and updates the dependency graph between IO requests with newly identified IO requests (§4.3.2).

4.3.1 Noise Coverage Reduction

The overall concept of noise coverage reduction is similar to §4.2.1 (i.e., removing the deterministic and non-deterministic noises separately). However, it needs to be computationally light because input synthesis is performed at fuzzing time.

To address this, MUNDOFUZZ optimizes the algorithm in §4.2.1 with two ways. First, rather than recording the coverage of each individual IO operation, MUNDOFUZZ records the coverage of an IO request (i.e., a set of IO operations) as a whole to reduce the number of coverage measurements. This indeed widens the window of deterministic noise, but a typical IO request is still much smaller than an entire hypervisor input (63 vs. 68,574 IO operations on average).

Second, rather than intersecting coverages multiple times, MUNDOFUZZ pre-records all candidate non-deterministic noises and subtracts them from the raw coverage at once. Currently, MUNDOFUZZ finds candidate non-deterministic noises using a dummy hypervisor input that does not reference any device registers. This may miss some non-deterministic noises, but is able to reveal the coverage from some asynchronous hypervisor components (e.g., a timer).

4.3.2 Fuzzing and Input Synthesis

While maintaining input semantics is important, supplying registers with correct values also has crucial importance. To address this, we adopt two input corpora that are specialized for capturing general dependency relations and specific register values, respectively. In particular, the *hypervisor input corpus* reserves entire hypervisor inputs with specific register values as a whole, and the *IO request corpus* reserves individual IO requests with the dependency graph of them.

Workflow. Figure 8 shows a workflow of hypervisor input synthesis that consists of three stages; generation, mutation and update. First, the generation stage provisions a base hypervisor input by either i) creating a new hypervisor input from the IO request corpus or ii) choosing a hypervisor input in the hypervisor input corpus (①). Next, the mutation stage mutates the base hypervisor input by modifying the register values with reference to their inferred types (②). Finally, the update stage analyzes the coverage feedback from the mutated input and, if it discovers new coverage, updates the dependency graph in the IO request corpus accordingly and reserves the mutated input in the hypervisor input corpus (③).

Input Generation. MUNDOFUZZ first generates a base hypervisor input by either randomly selecting a hypervisor input in the hypervisor input corpus, or creating a new hypervisor input using the dependency graph and the reserved IO requests in the IO request corpus. In the current configuration, we set the equal chances for both selecting a pre-reserved input and creating a new one from the scratch. When creating a new one, MUNDOFUZZ starts from the root IO request and picks a random child IO request while going downward in the dependency graph. The detailed algorithm is described in Appendix A.

Input Mutation. Given a base hypervisor input from the generation stage, MUNDOFUZZ mutates the value of registers according to their types. The mutation method is universal for all register types (i.e., AFL-style mutation including bit-flips and random value assignments), but it depends on the register types about how frequently it is applied or how the mutated IO requests are regarded.

For control registers, we note that the value of control registers largely determines the control flow inside the device. This suggests that mutating their values is likely to alter the functionality of the original IO request. Noticing this, MUNDOFUZZ first regulates the mutation probability

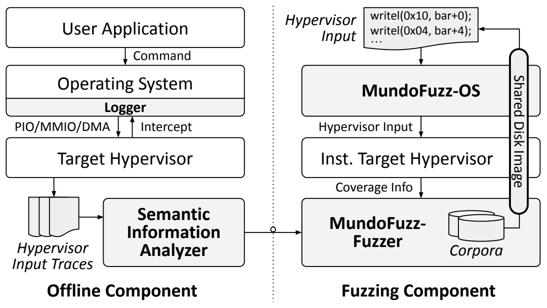


Figure 9: Overview of the MUNDOFUZZ system.

to a small value (18% in the current configuration) to maintain the original functionality, and if the value is mutated, MUNDOFUZZ regards the containing IO request as a new IO request. For example in Figure 8, MUNDOFUZZ regards the magenta-shaded IO request $\vec{R}^*$ as a new IO request, since it mutates the control register `bar+4` in $\vec{R}_4$ from `0x08` to `0x16`.

For data registers, we note that they generally accept a wide range of value while they still do not commonly affect the control flow. Therefore, MUNDOFUZZ mutates their value with a higher probability than control registers (30% in the current configuration). For DMA address registers, MUNDOFUZZ does not mutate the value but rather supplies a valid DMA buffer address. Specifically, MUNDOFUZZ replays the buffer allocation events at the beginning of the corresponding IO requests and feeds the allocated addresses to such registers.

In addition, MUNDOFUZZ also mutates the value of DMA buffers in a similar way to register values. In particular, MUNDOFUZZ handles DMA-address-typed buffer bytes (marked in §4.2.2) in the same way as DMA address registers. For the rest of DMA buffer bytes, MUNDOFUZZ currently mutates them similar to data registers.

Corpus Update. After executing the mutated hypervisor input, MUNDOFUZZ updates two corpora according to the coverage feedback. Specifically, if the mutated input discovered new coverage, MUNDOFUZZ adds the input to the hypervisor input corpus for future fuzzing runs. Furthermore, if the new coverage belongs to the new IO request, MUNDOFUZZ adds the new IO request to the IO request corpus and update the dependency graph accordingly. MUNDOFUZZ currently constructs the new dependencies by copying the parent dependency edges of the base IO request to the new IO request, as the new IO request has replaced the base IO request in the mutated input. For example in Figure 8, the new IO request $\vec{R}^*$ copies the parent dependencies of the base IO request $\vec{R}_4$.

5 Implementation

Figure 9 shows the overview of the MUNDOFUZZ system. The system is largely divided into two components, namely offline and fuzzing components. The offline component collects hypervisor input traces and analyzes semantic information (i.e., register types and IO request dependencies) be-

Devices		# of Regs.	Accuracy	
			Raw	Noise-reduced
QEMU	AC97	31	16.1% (-)	**87.9% (+71.8%)**
	ES1370	13	38.5% (-)	**92.3% (+53.8%)**
	Intel-HDA	33	15.2% (-)	**87.9% (+72.7%)**
	Floppy	4	75.0% (-)	50.0% (-25.0%)
	NVMe	9	44.4% (-)	**100.0% (+55.6%)**
	E1000	299	2.3% (-)	**98.0% (+95.7%)**
	E1000E	380	5.0% (-)	**96.1% (+91.1%)**
	RTL8139	24	37.5% (-)	75.0% (+37.5%)
	PCNET	3	0.0% (-)	**100.0% (+100%)**
	AM53C974	16	23.1% (-)	**84.6% (+61.5%)**
	MEGASAS	3	100.0% (-)	100.0% (0.0%)
	OHCI	18	38.9% (-)	55.6% (+16.7%)
	EHCI	7	57.1% (-)	85.7% (+28.6%)
	XHCI	24	50.0% (-)	91.7% (+41.7%)
Bhyve	Intel-HDA	34	14.7% (-)	**91.2% (+76.5%)**
	E1000	305	3.0% (-)	**98.7% (+95.7%)**
Average			32.6% (-)	**87.2% (+54.6%)**

Table 2: Accuracy of inferred device register types using raw and noise-reduced coverage. Highlighted numbers are +50% greater than the result using raw coverage. All devices are from QEMU and Bhyve.

fore fuzzing. The fuzzing component fuzzes the coverage-instrumented hypervisor using the semantic information.

Logging Operating System. To collect hypervisor input traces, we modified the Linux kernel 5.8.0 [16] to intercept PIO (`in/out{b,w,l}`) and MMIO (`write/read{b,w,l}`) API functions. For DMA operations, we modified the DMA buffer allocation function (`dma_map_page_attrs()`) to install page fault traps on DMA buffers at runtime, and modified the kernel-level page fault handler (`handle_page_fault()`) to record trapped DMA operations. Moreover, we implemented x86-based instruction decoder based on [42] to interpret DMA operations. To subdivide a hypervisor input trace into IO requests, we modified the interrupt handler (`handle_irq_event()`) to monitor completion signals.

Semantic Information Analyzer. We developed a semantic information analyzer that implements the analysis algorithm shown in §4.2.2 and §4.2.3. To obtain the coverage feedback from the target hypervisor, we incorporated the instrumented version of the target hypervisor with AFL's instrumentation compiler (`afl-clang-fast`).

MUNDOFUZZ-Fuzzer. We implemented MUNDOFUZZ-Fuzzer based on AFL 2.57b [52] by making it compatible with hypervisors. In particular, we enabled AFL to feed multiple inputs to the target hypervisor without shutting down the running hypervisor. Again, we instrumented the target hypervisor with AFL's instrumentation compiler.

MUNDOFUZZ-OS. We implemented an agent OS called MUNDOFUZZ-OS based on xv6 [32] to relay hypervisor inputs from MUNDOFUZZ-Fuzzer to the target hypervisor. To supply MUNDOFUZZ-OS with hypervisor inputs, we bridged MUNDOFUZZ-OS to MUNDOFUZZ-Fuzzer with a shared disk image, where MUNDOFUZZ-OS polls new hypervisor inputs that MUNDOFUZZ-Fuzzer presents. Since xv6 does not provide a basic PCI device enumeration function, we added a GRUB 2.0 support [10] with the Multiboot2 specification [33]. To support Bhyve, we added a basic UEFI support [23], which can only boot UEFI-compatible OSes.

6 Evaluation

In this section, we compare MUNDOFUZZ to the state-of-the-art hypervisor fuzzers, HYPER-CUBE and NYX, and demonstrate the effectiveness of the key techniques in MUNDOFUZZ. Specifically, we present the answers to the following research questions through evaluation.

- **RQ1.** Does noise-reduced coverage help inferring semantic constraints accurately? (§6.2)
- **RQ2.** How much does MUNDOFUZZ outperform state-of-the-art hypervisor fuzzers in coverage wise? (§6.3)
- **RQ3.** Can MUNDOFUZZ discover unknown vulnerabilities in hypervisors? (§6.4)

6.1 Evaluation Setup

We evaluated hypervisor fuzzers on a server-class machine with Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz and 512GB RAM running Ubuntu 18.04 LTS. We also used the latest version of each hypervisor, i.e., QEMU 5.2.0 and Bhyve 13.0-release. An input trace obtained from individual user applications (i.e., `dd` and `fsck`) was 68,574 IO operations on average, 104,386 at maximum for USB-EHCI.

6.2 Register Type Inference Accuracy

To verify how much noise-reduced coverage improves the accuracy of inferred semantic constraints, we compared the accuracy of inferred register types when using raw coverage feedback and noise-reduced coverage (**RQ1**). We referred to the device specifications [1, 2, 4–7, 11, 17, 19–22, 24] for ground-truth register types and only measured the accuracy of control and data types. We only included the registers that the corresponding virtual device implements, excluding the registers that only exist in the documentation but are never implemented in virtual devices.

Table 2 shows the accuracy of register types when they are inferred with raw and noise-reduced coverage. Overall, noise-reduced coverage significantly improves inference accuracy, more than 50% in 10 of 16 virtual devices. On average, noise-reduced coverage achieved 87.2% of register type inference accuracy, while raw coverage stands at 32.6%. In particular, noise-reduced coverage enables almost perfect inference in E1000 and E1000E, while it merely works with raw coverage.

Exceptionally for Floppy, noise-reduced coverage adversely affects the inference accuracy. Our investigation found that Floppy uses some device registers for both data transfer and command specification, which is incompatible to the current inference mechanism that assumes a single type for each

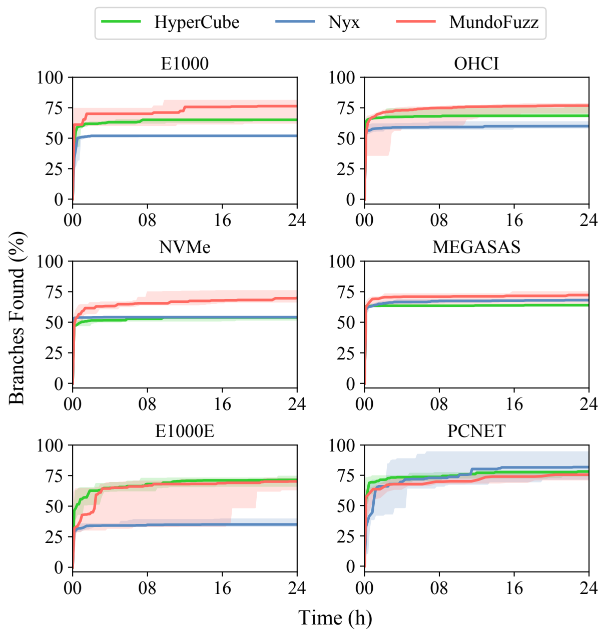


Figure 10: Branch coverage plot over the 24-hour fuzzing time. The colored lines show the median of 8 trials on each hypervisor. The shaded areas confine their best and worst coverage.

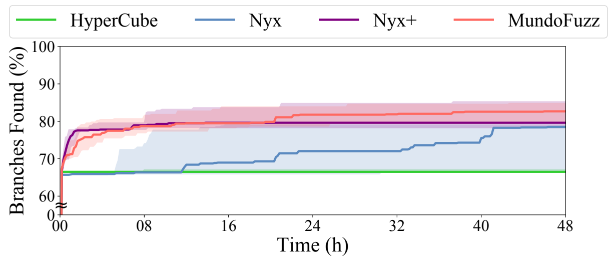


Figure 11: Coverage change over fuzzing time on XHCI. NYX+ represents NYX with grammar specification (NYX-SPEC in [48]).

register. This case can be easily handled by incorporating a data/control mixed type to inference, but we did not observe such a mixed-typed register in other devices.

For OHCI, noise-reduced coverage does not improve much accuracy. This is because some data registers in OHCI slightly alter the control flow as a side-effect. In particular, they affects the control flow of the internal queue update logic, which is triggered every time data registers are updated. To address this, the inference mechanism can adopt a threshold level to ignore minor coverage disturbance.

6.3 Coverage Comparison

To demonstrate the performance benefit of MUNDOFUZZ, we compared the code coverage of MUNDOFUZZ to the state-of-the-art hypervisor fuzzers, HYPER-CUBE and NYX (**RQ2**). We used the open-source version of NYX and reconstructed HYPER-CUBE by referring to the base implementation of NYX. In the following, we call NYX to indicate the version without a grammar specification (i.e., NYX-Legacy in [48]) on which most of the evaluation were performed in the original NYX paper. In addition, since NYX provides one manual grammar specification only for XHCI (NYX-Spec in [48]), we also compared MUNDOFUZZ to their grammar-specified version on XHCI.

Devices		Branch Coverage			Improvement	
		HYPER-CUBE	NYX	MUNDOFUZZ	vs.HC	vs.NYX
QEMU	AC97	**100.00%**	99.00%	**100.00%**	0.00%	+1.00%
	ES1370	87.50%	**91.07%**	**91.07%**	+3.57%	0.00%
	Intel-HDA	**79.83%**	79.00%	**79.83%**	0.00%	+0.83%
	Floppy	**85.15%**	**85.15%**	82.29%	-2.24%	-2.24%
	RTL8139	75.62%	**76.99%**	76.92%	+1.63%	-0.07%
	PCNET	78.13%	**81.78%**	75.52%	-2.61%	-6.26%
	E1000E	**71.51%**	35.04%	70.23%	-1.28%	+35.19%
	E1000	65.13%	51.97%	**76.32%**	+11.19%	+24.35%
	NVMe	53.89%	54.15%	**69.98%**	+16.09%	+15.83%
	AM53C974	63.45%	64.29%	**65.13%**	+1.68%	+0.84%
	MEGASAS	63.97%	68.12%	**72.27%**	+8.30%	+4.15%
	OHCI	68.49%	59.97%	**76.79%**	+8.51%	+16.82%
	EHCI	67.06%	66.02%	**69.26%**	+2.20%	+9.29%
	XHCI	66.48%	72.03%	**81.77%**	+15.29%	+9.74%
Bhyve	Intel-HDA	50.00%	55.17%	**56.89%**	+6.89%	+1.72%
	E1000	38.13%	47.93%	**50.18%**	+12.05%	+2.25%
Geomean					+4.91%	+6.60%

Table 3: Branch coverage on various virtual devices after 24 hours of fuzzing. Each presented number is the median of 8 trials. Bold percentages are the largest coverage on each device. vs.**HC** and vs.**NYX** denote the relative improvement of MUNDOFUZZ over HYPER-CUBE and NYX, respectively.

To compare coverage, we ran each hypervisor for 24 hours and repeated 8 times, following the evaluation guideline suggested by [41]. We measured coverage with Gcov [9] by executing the entire corpus from each hypervisor with Gcov-instrumented hypervisors every 10 minutes. We also restarted hypervisors with the same interval to minimize the influence of previous fuzzing campaign affecting the latter one.

Table 3 shows the median of final branch coverage and the relative improvement of MUNDOFUZZ over HYPER-CUBE and NYX. Overall, MUNDOFUZZ shows better performance compared to HYPER-CUBE and NYX, outperforming on 12 out of all 16 virtual devices by an average margin of 4.91% and 6.60%, respectively. Figure 10 shows the coverage plots of the selected virtual devices that have exhibited significant differences compared to HYPER-CUBE and NYX. The full set of coverage plots can be found in Appendix D. We found most of the under-performing virtual devices, such as E1000E and PCNET, were related to networking. On such devices, MUNDOFUZZ could not easily reach the branches inside packet receive functions. This was mainly because MundoFuzz was not able to collect hypervisor inputs for packet receiving, possibly due to the fact that the collecting hypervisor setup did not receive network packets destined to those devices.

NYX with Manual Grammar Specification. Since [48] incorporated manual grammar specification for XHCI to NYX, we also compared NYX with the XHCI grammar specifica-

Hypervisor	Bug Types	Numbers
QEMU	Use-after-free	3
	Heap Overflow	2
	Segmentation Fault	3
	Infinite Loop	3
	Stack Overflow	1
	Assertion	11
Bhyve	Segmentation Fault	4
	Floating Point Exception	1
	Assertion	12

Table 4: New bugs found by MUNDOFUZZ.

tion [26] to MUNDOFUZZ. Figure 11 shows the time-course change of XHCI coverage, where MUNDOFUZZ approaches NYX with grammar specification (NYX+) in a slower pace. We note that this is due to the incomplete dependency graph from the inference stage, which MUNDOFUZZ requires time to tune at fuzzing time. Nevertheless, MUNDOFUZZ eventually outpaces NYX+ in 20 hours and discovers more coverage in the end at 48 hours, even without any manual information.

6.4 New Vulnerabilities

To verify if MUNDOFUZZ is able to discover unknown vulnerabilities, we fuzzed the latest version of hypervisors with ASan [49] and MSan [51] (**RQ3**). Table 4 summarizes all 40 bugs that we newly discovered with MUNDOFUZZ, including 9 CVEs from QEMU. For Bhyve, we have requested CVEs and are waiting for response. The details of the bugs are shown in Appendix C. In the following, we examine a couple of new bugs and explain how MUNDOFUZZ was able to discover them.

Case Study: Heap buffer overflow in QEMU AM53C974. MUNDOFUZZ found a heap buffer overflow in the AM53C974 virtual SCSI controller. In Figure 12, AM53C974 first allocates `cache` with a large buffer `cache->buf` (Line 6) and initializes the limiting capacity `cache->cap` with a user-provided value (Line 8). Then, it performs PIO writes to copy data from `PIO_buf` to `cache->buf` (Line 13) and decrements the copied amount from `cache->cap` (Line 14). This decrement, however, can actually go over the initial capacity and eventually makes `cache->cap` underflow. While this underflow does not lead to a buffer overflow immediately, it later allows a DMA write with a huge write length (`DMA_len`) to pass the safety check (Line 20) and trigger the out-of-bound write (Line 21).

Reproducing this bug essentially requires two steps. First, it has to keep underflowing `cache->cap` by performing PIO writes enough times. Second, it must trigger the out-of-bound access to `cache->buf` by performing a DMA write with a huge write length (`DMA_len`). Thus, reproducing imposes multiple PIO requests followed by a specific DMA write operation. In this case, MUNDOFUZZ's design significantly helps to generate such an input—(i) MUNDOFUZZ is aware of the individual request format, the PIO request and DMA write operation (as presented in MUNDOFUZZ's input collection §4.1); (ii) MUNDOFUZZ knows how the individual request can be connected—i.e., a PIO request can be followed by either a PIO request or a DMA write (as presented in MUNDOFUZZ's dependency graph §4.2.3).

```
1  struct C {char buf[MAX]; char* top; uint cap;};
2  struct C *cache; char PIO_buf[16];
3
4  // Step 0: init cache capacity.
5  void init_cache(uint capacity) {
6    cache = malloc(sizeof(struct C));
7    cache->top = &cache->buf[0];
8    cache->cap = capacity;
9  }
10
11 // Step 1: make cache capacity underflow.
12 void write_from_pio(uint PIO_len) {
13   memcpy(cache->top, PIO_buf, PIO_len);
14   cache->cap -= PIO_len;  // underflow
15   cache->top += PIO_len;
16 }
17
18 // Step 2: access cache buffer out of bound.
19 void write_from_dma(char *DMA_addr, uint DMA_len) {
20   if (DMA_len <= cache->cap)  // huge DMA_len - OK
21     memcpy(cache->top, DMA_addr, DMA_len);  // crash
22 }
```

Figure 12: The heap buffer overflow crash in AM53C974.

However, it would be challenging for grammar-unaware fuzzers to generate such inputs, as it is aware of neither the individual request format nor how the request can be connected.

Case Study: Infinite Loop in QEMU E1000E. MUNDOFUZZ discovered an infinite loop issue in the E1000E virtual network device. When dequeueing a packet from the receive ring buffer, E1000E checks whether it is a null packet whose fields are all 0's and accordingly discards the packet to dequeue again. This causes E1000E to dequeue an infinite number of null packets if the receive ring buffer is filled with null packets, since a ring buffer never depletes the element.

To reproduce the bug, E1000E first needs to be in the loop-back mode, so that it can receive anything by transmitting to itself. However, exploring the loop-back mode is difficult as it is controlled by a single bit in a control register, making it easily overlooked while fuzzing. MUNDOFUZZ, on the other hand, recognizes the IO request that enables the loop-back mode as it modifies a control register. Furthermore, MUNDOFUZZ incorporates it into the dependency graph to synthesize more hypervisor inputs that run in the loop-back mode. As a result, MUNDOFUZZ succeeds to transmit a packet to itself no matter how difficult it is.

Moreover, an input needs to transmit a packet as soon as it configures the receive buffer, so that any other operations do not clutter the zero-initialized queue with any non-zero value. In this regard, the dependency graph readily guides MUNDOFUZZ the way how to generate a minimum hypervisor input that configures the receive buffer, sets the loop-back mode, and transmits a packet.

7 Discussion

Control Bytes in DMA Buffers. While MUNDOFUZZ handles major characteristics of DMA, including IO request dependencies established by DMA operations and DMA address bytes inside DMA buffers, MUNDOFUZZ currently cannot distinguish the control bytes in DMA buffers from plain data bytes. In particular, the semantic roles of control bytes are similar to the control registers. However, mutating control bytes values do not necessarily change the following coverage immediately, unlike the case of mutating control register values which immediately change the coverage. This makes making the current inferring algorithm of MUNDOFUZZ inaccurate for the control bytes in DMA buffers. Correctly identifying such control bytes is our future task.

Benefits of Statistical Coverage Measurement. When it comes to the coverage measurement, MundoFuzz takes a generic approach so that it does not need manual code modification. Furthermore, MundoFuzz leverages a statistical method to remove coverage noise from such a generic approach. Manual approaches such as Kcov [13] may not suffer from the coverage noise by manually skipping the coverage measurement from interrupt handler threads, but this manual modification should be individually done for each hypervisor.

Fuzzing Type-1 Hypervisors. The current implementation and evaluation of MUNDOFUZZ mainly focus on Type-2 hypervisors. In the future, MUNDOFUZZ can be extended to support Type-1 hypervisors by nesting a Type-1 hypervisor in an Intel PT-instrumented [12] Type-2 hypervisor, similar to [48]. We note that this does not invalidate any design points of MUNDOFUZZ, as all challenges remain the same in such an environment. In particular, the coverage information from an Intel PT-instrumented hypervisor is also tampered with an asynchronous noise, and the virtual devices of Type-1 hypervisors still have complex input grammars.

8 Related work

Hypervisor Fuzzing. Early hypervisor fuzzing works focused on relatively simple IO interfaces, such as PIO, MMIO, or hypercall interface [28, 30, 31, 37, 40, 43, 44]. For instance, IOFuzz [44] creates random sequences of PIO operations. Viridian [30] and Xen Test Framework [28, 31] invoke malicious hypercalls within Hyper-V and Xen. To fuzz PIO/MMIO interfaces more efficiently, VDF [37] and Tang et al. [40] utilize the collected IO traces to generate hypervisor inputs. However, they do not aim to keep the input semantics correct. Moreover, they modify the hypervisor itself to collect IO traces and suppress coverage noise. On the other hand, MUNDOFUZZ does not modify hypervisors at all.

The recent state-of-the-art hypervisor fuzzers are designed to handle a complete set of IO interfaces, including DMA. HYPER-CUBE [47] handles the DMA interface by writing data to a scratch buffer. However, it does not generate semantically correct inputs to explore complex device states.

NYX [48] addresses this limitation of HYPER-CUBE by utilizing the grammar specification of each virtual device. However, it requires a huge amount of manual efforts to embed grammar specification for a number of devices. Moreover, as mentioned in §6.3, MUNDOFUZZ demonstrates a higher coverage only after 20 hours, while MUNDOFUZZ does not require any kind of manual efforts.

Grammar Inference. In order to generate semantically valid inputs, recent fuzzing techniques have been developed by inferring grammars with given inputs [27,35,36,38,45]. In the user level, various fuzzers [27, 35, 38] generated syntactically valid inputs to fuzz programs accepting highly structured input languages such as interpreters, compilers. Moreover, kernel fuzzers [36, 45] also generate a sequence of correct system calls by referring to system call dependencies.

IO Trace Monitor. From a utility software to kernel module, the IO trace monitor is designed to debug or analyze the IO event in the kernel. For the utility, iostat in sysstat [34] is user application that monitors and report how many IO operations happen. However, since the utility software have no privilege to monitor IO trace directly, it simply provides statistic of IO operation by reading a status file in `sysfs`.

For the kernel module, kprobe [14] enable the user to break into any kernel routine and monitor IO events dynamically. And, ftrace [8] provides several options to monitor dynamic IO events. For example, ftrace gives by mmiotrace [25] options to trace MMIO operation utilizing the page fault mechanism. These modules have capabilities to monitor IO trace, however, it cannot fully utilized to monitor the complete interfaces because kprobe is inadequate to monitor the IO trace, and ftrace is only possible to collect for MMIO interface.

To monitor extended IO trace in DMA interface, Periscope [50] introduced IO trace monitor based on mmiotrace and enable the monitor to observe DMA operations. However, it is designed for Android kernel (AArch64), we implemented to monitor x86-based DMA operation.

9 Conclusion

Despite the advancement of hypervisor fuzzing techniques, exploring complex device states still remains challenging due to the coverage noise and the low-level inputs that complicate input generation. In this paper, we present MUNDOFUZZ, the hypervisor fuzzer that enables complex device states exploration, which removes coverage noise in raw coverage feedback and synthesizes semantically correct inputs using inferred semantic constraints. The evaluation shows that MUNDOFUZZ outperforms the state-of-the-art hypervisor fuzzer HYPER-CUBE and NYX by 4.91% and 6.60%, respectively. MUNDOFUZZ also found 40 new bugs in QEMU and Bhyve, 9 of which are acknowledged as CVEs.

10 Acknowledgment

We would like to thank anonymous reviewers for their insightful comments, which significantly improved the final version of this paper. We also thank QEMU security team for their helpful feedback and responses. This work is the result of commissioned research project supported by the affiliated institute of ETRI (2020-026). This work was partially supported by Institute for Information and communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). This work was partially supported by National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

References

[1] AC97 data sheet. https://www.ti.com/lit/ds/snas276g/snas276g.pdf.

[2] AM53C974 data sheet. https://www.digchip.com/datasheets/parts/datasheet/014/AM53C974-pdf.php.

[3] *Debugging in AMD64 64-bit Mode in Theory*. [Online]. Available: http://fdbg.x86asm.net/debugging-in-long-mode-in-theory.html.

[4] E1000/E1000E data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf.

[5] EHCI data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/ehci-specification-for-usb.pdf.

[6] ES1370 data sheet. http://ftp.kolibrios.org/users/Asper/sound/Ensoniq/ES1370.PDF.

[7] Floppy data sheet. http://www.osdever.net/documents/82077AA_FloppyControllerDatasheet.pdf.

[8] *Function Tracer (ftrace)*. [Online]. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[9] *gcov: a Test Coverage Program*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html.

[10] *GNU GRUB Manual 2.04*. [Online]. Availalbe: https://www.gnu.org/software/grub/manual/grub/grub.html.

[11] Intel-HDA data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf.

[12] *Intel Processor Trace*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html.

[13] *kcov: code coverage for fuzzing — The Linux Kernel documentation*. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kcov.html.

[14] *Kernel probes (kprobes)*. [Online]. https://www.kernel.org/doc/Documentation/kprobes.txt.

[15] libFuzzer. https://llvm.org/docs/LibFuzzer.html.

[16] The linux kernel. https://github.com/torvalds/linux.

[17] MEGASAS data sheet. https://docs.broadcom.com/doc/pub-005183.

[18] Number of consumer cloud-based service users worldwide in 2013 and 2018. https://www.statista.com/statistics/321215/global-consumer-cloud-computing-users/.

[19] NVMe data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf.

[20] OHCI data sheet. https://composter.com.ua/documents/OHCI_Specification_Rev.1.0a.pdf.

[21] PCNET data sheet. https://pdf1.alldatasheet.com/datasheet-pdf/view/89876/AMD/AM79C970.html.

[22] RTL8139 data sheet. https://www.cs.usfca.edu/~cruse/cs326f04/RTL8139D_DataSheet.pdf.

[23] *Unified Extensible Firmware Interface (UEFI) Specification 2.8*. [Online]. Availalbe: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf.

[24] XHCI data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/extensible-host-controler-interface-usb-xhci.pdf.

[25] *Memory Mapped I/O Trace*, 2014. [Online]. Available: https://nouveau.freedesktop.org/MmioTrace.html.

[26] *eXtensible Host Controller Interface for Universal Serial Bus (xHCI)*, 2019. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/extensible-host-controler-interface-usb-xhci.pdf.

[27] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.

[28] Sören Bleikertz. *Fuzzing the Xen Hypervisor*, 2010. https://www.openfoo.org/blog/xen-fuzz.html.

[29] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[30] Amardeep Chana. *MWR-Labs: Ventures into Hyper-V - Fuzzing hypercalls*, 2019. https://labs.f-secure.com/blog/ventures-into-hyper-v-part-1-fuzzing-hypercalls.

[31] Citrix. *Xen Test Framework*, 2014. https://xenbits.xenproject.org/docs/xtf/index.html.

[32] Russ Cox, M Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2011.

[33] Bryan Ford and Erich Stefan Boleyn. *Multiboot specification version 2.0.*, 2016. [Online]. https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html.

[34] Sebastien Godard. *sysstat: System performance tools for the Linux operating system*. [Online]. http://sebastien.godard.pagesperso-orange.fr/.

[35] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289*, 2018.

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

[37] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted evolutionary fuzz testing of virtual devices. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Atlanta, GA, September 2017. https://www.cs.ucr.edu/~heng/pubs/VDF_raid17.pdf.

[38] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725. IEEE, 2016.

[39] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[40] Moony Li Jack Tang. When virtualization encounter AFL. https://www.blackhat.com/docs/eu-16/materials/eu-16-Li-When-Virtualization-Encounters-AFL-A-Portable-Virtual-Device-Fuzzing-Framework-With-AFL-wp.pdf, 2016.

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

[42] Nikita Komarov. *RaceHound*. [Online]. https://github.com/kmrov/racehound.

[43] Airbus Security Lab. *crahsOS*, 2017. https://github.com/airbus-seclab/crashos.

[44] Tavis Ormandy. *An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments*, 2007. http://taviso.decsystem.org/virtsec.pdf.

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

[46] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types.

[47] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Hyper-Cube: High-dimensional hypervisor fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS*, pages 23–26, 2020.

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

[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[50] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019. https://www.cspensky.info/pdfs/ndss2019_04A-1_Song_paper.pdf.

[51] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[52] Michael Zalewski. *American Fuzzy Lop*. [Online]. Available: https://lcamtuf.coredump.cx/afl/.

Algorithm 1: CreateNewInput

```
Input G: Dependency graph between the IO requests in the corpus.
Output List of IO requests.
R = RootOf(G)      // R: IO request
S = {R}            // S: set of chosen IO requests
while HasChild(R) do
    R = PickOneChildRandomly(R)
    S ← R
    // Backtracking to resolve missing dependencies
    Q = [R]        // Q: IO request queue
    while not Empty(Q) do
        R' = Dequeue(Q)
        if not ParentsOf(R') in S then
            if BreakDependencyRandomly() then break
            S ← ParentsOf(R')
            Enqueue(Q, ParentsOf(R'))
return ToList(S)
```

A Input Generation Algorithm

Algorithm 1 describes how MUNDOFUZZ creates a new hypervisor input given a dependency graph between IO requests. Starting from the root IO request, the algorithm picks a random child IO request and recursively repeats the process until it reaches the leaf IO request. If there are multiple root IO requests that are mutually independent to each other, MUNDOFUZZ creates an empty IO request that joins all such IO requests to make a unified root. If it encounters any unresolved dependency in this course, the algorithm probabilistically resolves the dependency by backtracking the dependency graph, or deliberately leaves it unresolved to break the dependency and examine the corresponding behavior.

B Case Study: Null Dereference in QEMU MEGASAS

MUNDOFUZZ found a null dereference issue in the MEGASAS virtual RAID disk device. MEGASAS uses a DMA buffer to perform several commands. The DMA buffer contains parameters, such as control ID that determines which command should be performed. When receiving a disk write command, MEGASAS copies the DMA buffer to its internal queue. However, this copied buffer is set to 0 when MEGASAS receives a device initialization command afterwards, while leaving the disk write command still in the queue. This causes MEGASAS to dereference a null pointer when it is triggered to flush all pending commands.

In order to find this bug, following two challenges should be addressed. First, it requires the DMA buffer should include a specific control ID to perform MEGASAS commands. Second, the bug requires following three commands in a right order; i) write data to the disk, ii) initialize MEGASAS, iii) flush all pending commands.

In this case, MUNDOFUZZ can easily generate the needed DMA buffer from hypervisor inputs. Moreover, since MUNDOFUZZ maintains the dependencies between IO requests, it can try all three commands in order and quickly discover the bug. However, it would be difficult to grammar-unaware fuzzers, as they do not recognize the DMA buffer and are hard to keep the orders.

C List of Discovered Bugs

Hypervisor	Description	Device	Status	ID
QEMU	Infinite loop issue in `e1000e_write_packet_to_guest`	E1000E	Assigned	CVE-2020-28916
	Infinite loop issue in `e1000e_ring_advance`	E1000E	Assigned	CVE-2020-25707
	Infinite loop issue in `process_tx_desc`	E1000	Assigned	CVE-2021-20257
	NULL pointer dereference issue in `megasas_command_cancelled`	MEGASAS	Assigned	CVE-2020-35503
	NULL pointer dereference issue in `scsi_req_continue`	AM53C974	Assigned	CVE-2020-35504
	NULL pointer dereference issue in `do_busid_cmd`	AM53C974	Assigned	CVE-2020-35505
	Use-after-free issue in `flatview_write_continue`	AM53C974	Assigned	CVE-2020-35506
	Use-after-free issue in `mptsas_process_scsi_io_Request`	MPTSAS1068	Assigned	CVE-2021-3392
	Use-after-free issue in `esp_do_nodma`	AM53C974	Confirmed	-
	Stack overflow issue in `flatview_read_continue`	AM53C974	Confirmed	-
	Heap buffer overflow issue in `scsi_req_parse_cdb`	AM53C974	Confirmed	-
	Heap buffer overflow issue in `esp_fifo_pop_buf()`	AM53C974	Requested	-
	Assertion in `usb_packet_unmap` at `hcd-ehci.c`	EHCI	Assigned	CVE-2020-25723
	Assertion in `usb_msd_handle_data` at `dev-storage.c`	XHCI	Fixed	-
	Assertion in `fifo8_pop_buf` at `fifo8.c`	AM53C974	Fixed	-
	Assertion in `fifo8_push_all` at `fifo8.c`	AM53C974	Fixed	-
	Assertion in `mptsas_interrupt_status_write`	MPTSAS1068	Confirmed	-
	Assertion in `address_space_stw_le_cached`	VirtIO-blk	Requested	-
	Assertion in `lsi_do_dma` at `lsi53c895a.c`	LSI53C810	Requested	-
	Assertion in `esp_transfer_data()`	AM53C974	Requested	-
	Assertion in `scsi_read_data()`	AM53C974	Requested	-
	Assertion in `esp_do_dma` at `esp.c`	AM53C974	Fixed	-
	Assertion in `esp_do_dma` at `esp.c`	AM53C974	Fixed	-
Bhyve	Floating point exception issue in `pci_nvme_handle_io_cmd` at `pci_nvme.c`	NVMe	Requested	-
	Segfault issue in `pci_nvme_pci_nvme_handle_admin_cmd` at `pci_nvme.c`	NVMe	Requested	-
	Segfault issue in `pci_nvme_append_iov_req` at `pci_nvme.c`	NVMe	Requested	-
	Segfault issue in `pci_nvme_handle_io_cmd` at `pci_nvme.c`	NVMe	Requested	-
	Assertion in `pci_nvme_cq_update` at `pci_nvme.c`	NVMe	Requested	-
	Assertion in `e82545_write_ra` at `pci_e82545.c`	E1000	Requested	-
	Assertion in `e82545_transmit` at `pci_e82545.c`	E1000	Requested	-
	Assertion in `e82545_write_register` at `pci_e82545.c`	E1000	Requested	-
	Segfault issue in `e82545_transmit` at `pci_e82545.c`	E1000	Requested	-
	Assertion in `hda_stream_start` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_corb_run` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_set_rirbctl` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_set_corbctl` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_stream_start` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_set_sdctl` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_set_dpiblbase` at `pci_hda.c`	Intel-HDA	Requested	-
	Assertion in `hda_stream_start` at `pci_hda.c`	Intel-HDA	Requested	-

Table 5: List of new bugs.

D Branch Coverage Plot

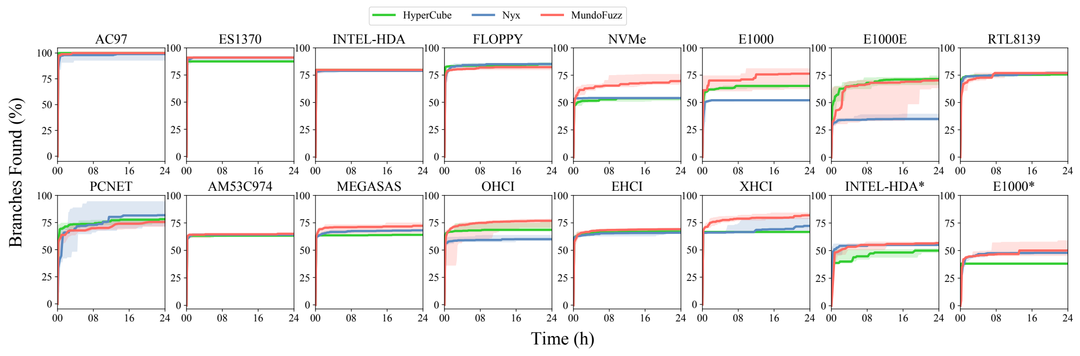


Figure 13: The branch coverage over fuzzing time for 24 hours in QEMU and Bhyve (each 8 runs). Each plot shows maximum, minimum and median result. * represents the virtual devices of Bhyve.

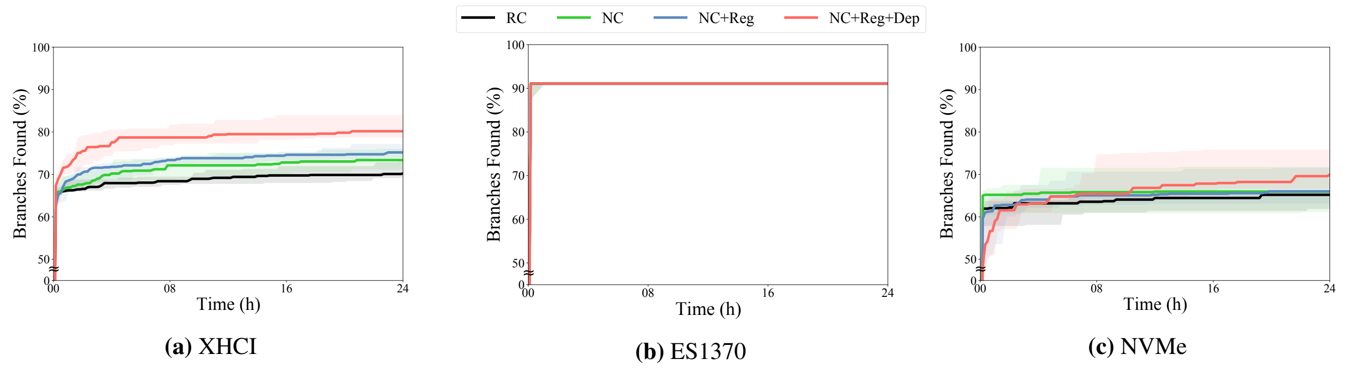


Figure 14: Coverage change over time with various combinations of key techniques. RC and NC indicate pure coverage-guided fuzzing with raw and noise-reduced coverage, respectively. NC+Reg and NC+Reg+Dep use inferred register types and dependency graphs in addition.

E Effectiveness Analysis

To elicit the performance benefit of each technique, we prepared four combinations that cumulatively incorporate techniques as follows; RC performs coverage-guided fuzzing with raw coverage. NC reduces coverage noise with noise coverage reduction. NC+Reg enables semantic input synthesis with inferred register types. NC+Reg+Dep additionally utilizes inferred dependency graphs for input synthesis.

Figure 14 shows the average coverage over fuzzing time with the four combinations. For XHCI in Figure 14a, each technique evenly contributes to the coverage expansion. In particular, NC improves the coverage as the noise reduction prevents MUNDOFUZZ being misled by coverage noise and falsely reserving redundant inputs. Furthermore, NC+Reg mutates register values in reference to their types, resulting in less destructive mutation to input semantics. Finally, NC+Reg+Dep proactively synthesizes semantically correct inputs and explores more complex control flows.

For ES1370 in Figure 14b, all coverages reach the saturation point in a few minutes. This is because such simple devices do not have complex control flow. For NVMe in Figure 14c, although NC+Reg+Dep did not show better performances to others, it eventually showed better performances after 10 hours of run. Another thing to note is that NC+Reg converges to NC in the end. This is because NVMe only has 9 registers, which is small enough to explore most value combinations in 24 hours through random mutation.