InSVDF: Interface-State-Aware Virtual Device Fuzzing

Zexiang Zhang*§, Gaoning Pan†, Ruipeng Wang*§, Yiming Tao‡, Zulie Pan*§
Cheng Tu*§, Min Zhang*§✉, Yang Li*§, Yi Shen*§, Chunming Wu‡

*National University of Defense Technology, China †Hangzhou Dianzi University, China ‡Zhejiang University, China
§Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, China

***Abstract*—Hypervisor is the core technology of virtualization for emulating independent hardware resources for each virtual machine. Virtual devices serve as the main interface of the hypervisor, making the security of virtual devices crucial, as any vulnerabilities can impact the entire virtualization environment and pose a threat to the host machine's security. Direct Memory Access (DMA) is the interface of virtual devices, enabling communication with the host machine. Recently, many efforts have focused on fuzzing against DMA to discover the hypervisor's vulnerabilities. However, the lack of sensitivity to the DMA state causes these efforts to be hindered in efficiency during fuzzing. Specifically, there are two main issues: the uncertain interaction moment and the unclear interaction depth.**

In this paper, we introduce InSVDF, a DMA interface state-aware fuzzing engine. InSVDF first models the intra-interface state of the DMA interface and incorporates an asynchrony-aware state snapshot mechanism along with a depth-aware seed preservation mechanism. To validate our approach, we compare InSVDF with a state-of-the-art fuzzer. The results demonstrate that InSVDF significantly enhances vulnerability discovery speed, with improvements of up to 24.2x in the best case. Furthermore, InSVDF has identified 2 new vulnerabilities, one of which has been assigned a CVE ID.

I. Introduction

Hypervisor [1] is a system software that virtualizes the computing resources of a physical machine and simulates the necessary hardware environment required for computer systems. This allows for the creation and management of multiple virtual machines (VMs) on a single physical machine, optimizing the utilization of the host's computing resources [2], [3]. Each VM can run its own independent operating system, maintaining isolation from one another, which ensures robust security and protection for users. These capabilities make virtualization technology a fundamental component of cloud computing [4], aligning with current trends in computing power deployment.

However, the hypervisor presents significant security risks, with virtual devices representing the primary attack surface. These virtual devices [5] are used to emulate the hardware peripherals of a computer system and serve as the main interface for interaction between the guest and the host. Consequently, memory corruption vulnerabilities in virtual devices can not only lead to hypervisor crashes but also compromise isolation within virtual environments. Attackers exploiting these vulnerabilities may escape from a virtual machine, execute commands on the host machine, and threaten the security of the entire cloud computing environment [6]–[8]. Therefore, the security of virtual devices has garnered increasing attention [9]–[11].

Fuzzing [12]–[16] has become a mainstream method for security researchers due to its adaptability and ease of deployment. Current fuzzing methods for virtual devices primarily target their basic interfaces, such as MMIO/PIO and DMA. Among these interfaces, Direct Memory Access (DMA) [17] is particularly significant because it handles data transmission without requiring CPU involvement. Since the main function of a virtual device is to receive data and manage interrupt events, DMA has emerged as the interface with the highest security risk. Therefore, we focuse on the problems that arise from the interaction with the DMA interface during the fuzzing process.

The interaction features of DMA interface constrains the efficiency of fuzzing, this consists of two main aspects. **The first is the uncertain interaction moment.** The hypervisor primarily calls the DMA interface to transfer data when handling interrupt events. To efficiently schedule these events, the hypervisor employs asynchronous mechanisms. In this scenario, the interrupt handling process is initiated by the hypervisor rather than the user, occurring dynamically. This dynamic nature makes it impossible for users to determine the trigger timing through static program analysis, thereby introducing a high degree of uncertainty. **The second is the unclear interaction depth.** Virtual devices often require multiple nested interactions to process interrupt events. The presence of these nested interaction behaviors endows each DMA interaction point with a depth attribute, which changes as the interactions evolve. Multiple DMA operations may depend on each other, forming complex nested relationships. This complexity makes it challenging to analyze and understand the depth of these interactions.

Current efforts to fuzz virtual devices utilize the redirection approach proposed by V-Shuttle [18] and Morphuzz [19] to accomplish DMA interface input. Subsequent work [20]–[22] has focused on the correlations of virtual device interaction messages. For example, ViDeZZo [21] extracts intra- and

✉ Corresponding author, zhangmindy@nudt.edu.cn

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00190

inter-message dependencies to generate dependency-compliant test case sequences, while VD-Guard [22] focuses on generating input sequences that can trigger DMA interactions. However, in asynchronous scenarios, the continuity of the user's interaction with the device is disrupted, and simply generating high-quality test case sequences loses effectiveness. Additionally, current methods mainly leverage coverage to guide the fuzzing process [18], [21], [23], [24]. Nyx [23] and HyperPill [24], which are based on nested virtualization frameworks, use the PT [25] suite to obtain coverage. V-Shuttle [18] and ViDeZZo [21] use instrumentation to obtain coverage. However, calls to the same function can vary depending on the depth of the call chain. Guiding the fuzzing process solely by coverage fails to reflect differences in depth and cannot adjust the strategy accordingly.

To address these challenges, we propose InSVDF, a fuzzing framework that is attuned to the intra-interface states of virtual devices. By leveraging variables within the hypervisor runtime environment and getting dynamic feedback from the hypervisor, we model the intra-interface states to effectively guide the fuzzing process. To tackle the first challenge, we introduce an asynchrony-aware state snapshot(ASS) mechanism that enhances the fuzzer's capability to detect the trigger moment of the asynchronous interaction. For the second challenge, we implement a depth-aware seed preservation(DSP) mechanism, enabling the fuzzing process to accurately adapt to the interaction depth.

We developed InSVDF based on V-Shuttle and AFL. To verify the effectiveness, we compared InSVDF with V-Shuttle on a dataset of 9 DMA-related vulnerabilities and we outperforms V-Shuttle. Moreover, we found through ablation experiments that both mechanisms we designed had the desired effect. We discovered 2 new vulnerabilities in QEMU, one of which has been fixed by developers (has been assigned a CVE ID).

In summary, this paper makes the following contributions:

- We conduct an in-depth analysis of the virtual device's DMA interface and uncover that its internal state impedes the efficiency of fuzzing for virtual devices.
- We propose an interface state-sensitive method for discovering vulnerabilities in virtual devices and have developed a prototype called InSVDF.
- We apply InSVDF and V-Shuttle to the open-source virtualization software QEMU and VirtualBox. InSVDF outperformed V-Shuttle on a dataset of DMA-related vulnerabilities and have discovered 2 unknown vulnerabilities, one of which has been fixed and received a CVE ID.

II. BackGround

In this section, we introduce the fundamental interaction methods of virtual devices and analyze the interaction characteristics of the DMA interface. We then discuss the motivation behind our research and the challenges we anticipate. Finally, we present the core solution proposed in this paper.

A. Virtual Device Interfaces

In hypervisor implementations, users interact with virtual devices via three interfaces: MMIO, PIO, and DMA. MMIO maps a device's control registers and memory space, enabling access through standard memory instructions like `load` and `store`. PIO involves accessing a device via specific I/O ports using instructions such as `in` and `out` in x86. DMA allows peripherals to transfer data directly to memory without CPU intervention, making it ideal for high-throughput tasks like network or USB data transfers. A virtual device's core function is data processing. Since large DMA transfers heighten vulnerability risks, most issues arise from DMA input. Therefore, our work focuses on DMA interactions, analyzing their scenarios and depth in the following sections with an illustrative example.

1) DMA Interaction Scenario

Virtual device interactions are often associated with device interrupt events. DMA interactions can be categorized into synchronous and asynchronous interactions based on the invocation scenario.

Synchronous DMA Interaction. In this scenario, interactions with the virtual device involve reading and writing the device's registers through MMIO or PIO. This action triggers an interrupt event. The hypervisor then processes the interrupt by invoking the DMA read/write functions to complete the data exchange. The interrupt event processing is synchronized, meaning that the event is handled sequentially, and the result is returned once the processing is complete. Thus, this type of interaction is defined as synchronous DMA interaction. In this scenario, the MMIO/PIO interface and the DMA interface operate sequentially, allowing users to control the moment of triggering DMA interactions by sending MMIO/PIO messages to trigger them.

Asynchronous DMA Interaction. To enhance event processing efficiency, the hypervisor employs timer and bottom-half (bh) mechanisms to initiate or handle interrupts [26]. As illustrated in Figure 1(a), during the initialization of the OHCI device, the hypervisor registers a timer using `timer_new_ns`. Once the timer expires, it triggers an interrupt to process the USB frame boundary. Within the `ohci_frame_boundary` function, the DMA read functions `ohci_read_ed` and `ohci_read_td` are invoked to facilitate data transfer through the DMA interface. Additionally, the hypervisor frequently uses the bh mechanism to defer the non-time-sensitive parts of interrupt events until the CPU becomes available to process them. In this context, the hypervisor controls the timing of DMA interface invocations to handle device interrupts asynchronously. This form of interaction is referred to as asynchronous DMA interaction. Here, the interactions between the MMIO/PIO interface and the DMA interface occur in a non-serial manner, and users cannot control the timing of DMA interactions.

2) DMA Interaction Depth

In typical virtual device implementations, DMA interactions often involve nested behavior. When a user interacts with the same virtual device, multiple DMA interactions may be

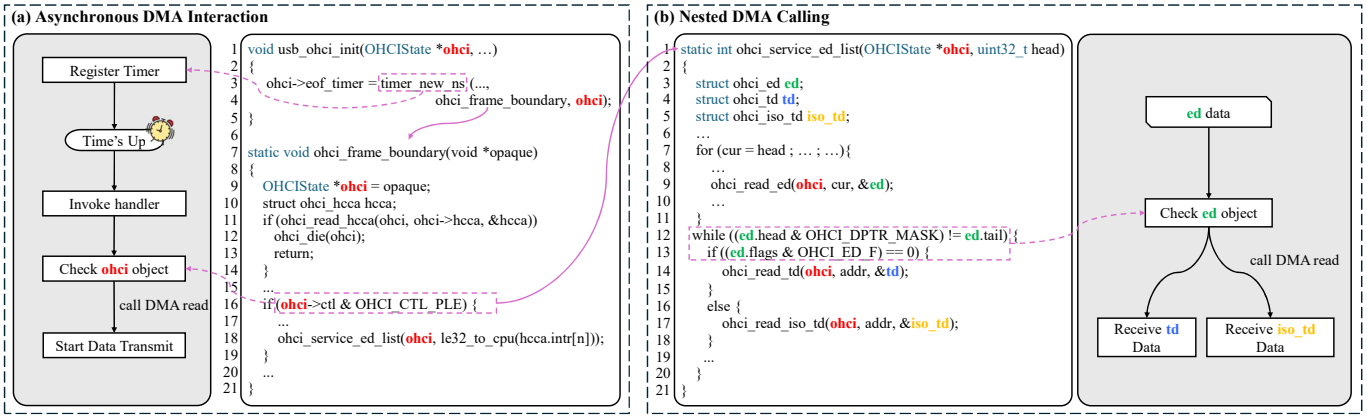


Fig. 1: An Example of OHCI device transferring data between user and virtual device. (a) demonstrates how the asynchronous interrupt events launch a DMA interaction behavior and (b) depicts the relationship among the three DMA target objects

required to transfer data into different objects within the virtual device's address space. This approach is commonly used for step-by-step processing of interrupt events. For instance, in the USB transfer protocol, it is necessary to first set up the endpoint descriptor (`ed`) before transferring data.

Figure 1(b) illustrates this process, where the hypervisor invokes DMA interactions (line 8) to input data into the `ed` object, then assesses the result (line 11 and 12) to determine whether the next DMA interaction should target the `td` or `iso_td` object. This example highlights a distinctive feature of the DMA interface: the interaction depth, which is not present in MMIO/PIO interfaces. Although the DMA interface provides data to all three target objects, the interaction depths differ, leading to varied handling of inputs. Additionally, as the interaction depth increases, the likelihood of successfully transferring data to the target object generally decreases.

B. Challenges

Current research on discovering vulnerabilities in virtual devices primarily employs fuzz testing techniques. This method involves providing random inputs to virtual devices through basic interfaces to identify potential violations or errors. However, the complexity of the interface's operational logic can affect the effectiveness of test inputs, particularly for input-intensive interfaces like DMA. We have identified two main challenges in this interface scenario, as follows:

1) ***Uncertain Invocation Moment.***

Current research works [20]–[22] trigger DMA interactions by generating a sequence of test cases that suit the interaction semantics for launching an interrupt event. However, when the DMA interaction behavior is launched by a timer or bottom half event rather than the user, they fail to input data into the DMA interface smoothly. For example, the fuzzer continuously inputs the test case sequence into multiple interfaces of the virtual device, which causes the value of `ohci->ctl` to keep changing, as shown in Figure 1(a). When the asynchronous DMA interaction is initiated, the value of `ohci->ctl` must pass the condition check at line 16 for the fuzzer to continue interacting with the virtual device. However, the fuzzer cannot proactively adjust the operational state of the target device to meet the condition when the DMA interface is invoked. Consequently, asynchronous DMA inputs can only be triggered through collision probabilities, which is extremely inefficient. *Therefore, in asynchronous scenarios, the moment of interaction initiation by the hypervisor is uncertain, which can interrupt the coherence of the test case sequence entered by the fuzzer, resulting in suboptimal interactions.*

2) ***Unclear Interaction Depth.***

Virtual devices often require multiple nested DMA interactions to process interrupt events step by step. These interactions are hierarchical, with each level potentially invoking further nested interactions. As a result, each DMA interaction point is characterized by a depth attribute that evolves dynamically as interactions progress. However, the discovery of some DMA-related vulnerabilities necessitates complex interaction processes, including nested calls to the same interaction point. In such cases, relying solely on coverage to guide the fuzzing process is insufficient for effective vulnerability discovery. *This limitation prevents the fuzzer from adjusting its strategy according to the real-time depth of interactions, thereby hindering the efficiency of vulnerability discovery.*

Both challenges stem from the interaction characteristics of DMA interfaces, which constrain fuzzing efficiency from the perspectives of smooth initiation and in-depth progression. To verify our claim, we conducted a preliminary experiment. We define the callback function of a timer or bh event as the asynchronous interaction initiation point. The first layer DMA is considered the first initiated DMA interaction of the interrupt. The second layer DMA point represents the first DMA interactions invoked by the first layer DMA, and so on, defining higher-level DMA interactions according to the call chain.

We conducted a 24-hour test on three virtual USB devices using V-Shuttle [18] and instrumented each specified point (considering only the first and second DMA layers). During the instrumentation process, we incorporated call-edge checking to ensure that the recorded number of calls at each point

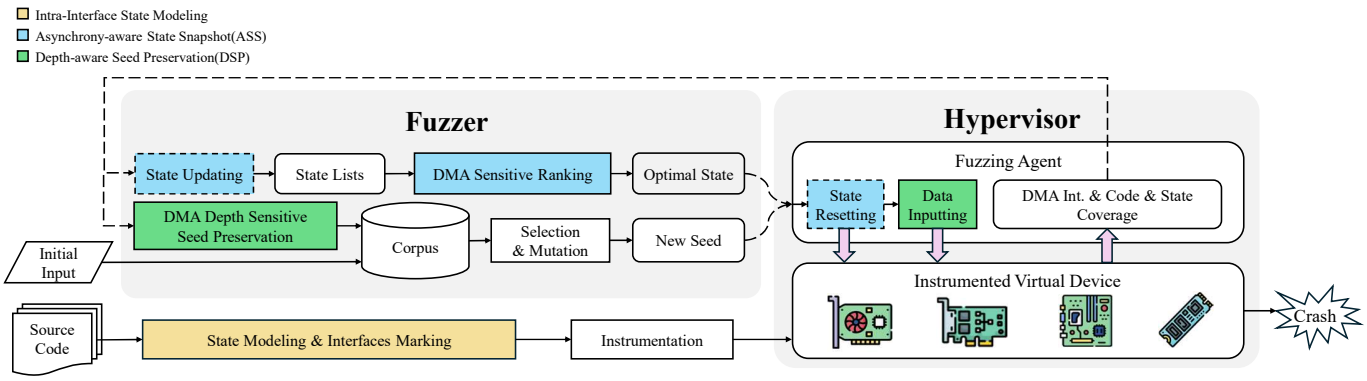


Fig. 2: Overview of InSVDF

accurately reflects asynchronous DMA interaction behavior. Additionally, we analyzed the number of calls within loops to verify that the ratio of arrivals at each recorded interaction point accurately represents the efficiency of individual seed input points.

The results shown in Table I indicate that the efficiency of the data input DMA interface is significantly reduced during fuzzing. This inefficiency arises because the fuzzer lacks control over the timing of asynchronous calls and is unclear about the depth of interaction at runtime, which aligns with our previous claim. Addressing these challenges can significantly enhance the fuzzer's capability to uncover unique vulnerabilities.

TABLE I: The number of times that V-Shuttle triggers DMA-related interaction points in a 24h test

Device	Access Point		
	Initiation Point	1st Layer DMA	2nd Layer DMA
ohci	692,399 (100%)	234,031 (33.8%)	84,472 (12.2%)
ehci	849,898 (100%)	133,434 (15.7%)	50,705 (6.0%)
uhci	2,368,435 (100%)	234,031 (24.2%)	51,487 (2.2%)

C. Intuition: Implementing Interface State-Aware Fuzzing

The challenges mentioned above arise from the fuzzer's insensitivity to the state of virtual device interfaces. To address these challenges, we propose monitoring the interaction state of the DMA interface in real time during testing and providing feedback to guide the fuzzing process. However, directly obtaining the state of the virtual device interface is not feasible. The hypervisor's runtime environment hosts numerous virtual devices, each with a diverse set of interfaces, making it difficult and resource-intensive to monitor the hypervisor and extract the interface status of the target device.

We observe that the hypervisor maintains the state of each virtual device using variables, such as `ohci` in Figure 1, which influence the interaction behavior of the virtual device. The state of the interface is closely tied to the interaction behavior, so we can indirectly monitor the interaction state of the interface by tracking these variables.

Based on this observation, we propose the concept of the virtual device **intra-interface state**. This approach allows us to monitor the state of the virtual device interface and guide the fuzzing process by adjusting and controlling the interface's state, thereby improving the efficiency of fuzzing.

III. Design

In this section, we introduce InSVDF, a virtual device fuzzing framework that is sensitive to the interface state of the target virtual device. First, InSVDF models the intra-interface state of a virtual device based on its state variable and the running status of the DMA interface. Then, InSVDF constructs an asynchrony-aware state snapshot(ASS) mechanism and a depth-aware seed preservation(DSP) mechanism to address the above two challenges in II-B.

A. Threat Model

We assume that the attacker is a privileged guest user with full memory access inside the virtual machine, capable of sending arbitrary data to the virtual device through any interface. This assumption is reasonable in a cloud computing scenario because the user has system privileges within their virtual machine. The attacker constructs malicious data packets and sends them to the virtual device with the intent to cause a virtual machine crash or to escape from the virtual machine's isolation environment onto the host machine.

B. System Overview

The framework of InSVDF is shown in Figure 2. InSVDF is composed of a fuzzing agent and a fuzzer. The following list summarizes the high-level functionalities of the two main components.

Fuzzing Agent. The fuzzing agent is running inside the hypervisor, which (1) sets the intra-interface state of the tested device as specified by the fuzzer before transmitting input, (2) helps the fuzzer to input data into the tested virtual device, and (3) captures the tested device's intra-interface state after the device processing the data and passes this information to the fuzzer.

Fuzzer. The fuzzer is placed outside the hypervisor, which (1) ranks the captured intra-interface state and passes the

optimal state to the fuzzing agent, (2) preserves seeds that can trigger the deeper DMA interaction behavior. The fuzzer is the core component of InSVDF, responsible for implementing the primary algorithm of our method.

Through the coordination of these two components, InSVDF establishes an asynchrony-aware atate anapshot mechanism to ensure that fuzzing is sensitive to the invocation moment of asynchronous DMA interactions. Additionally, a depth-aware seed preservation mechanism has been developed to enhance the fuzzer's sensitivity to the depth of DMA interactions.

C. Intra-Interface State Monitoring

1) Constitution of the Intra-Interface State

To effectively monitor the DMA interface state of the target virtual device during the fuzzing process, we construct the intra-interface state, which primarily encompasses two aspects: the interaction behavior of the virtual device and the interaction features of the DMA interface.

Interaction Behavior. Since the primary function of the virtual device is to process input data, variations in interaction behavior will affect the accessibility of its interfaces. To effectively understand the behavior of the virtual device under test, we monitor the state variables involved in branch conditions.

Interaction Features. In this paper, we monitor both the invocation scenarios and the interaction depth characteristics of the interface. For invocation scenarios, we focus on asynchronous DMA interaction behavior, tracking its trigger status to determine which intra-interface states can complete asynchronous DMA interactions. Regarding interaction depth, differential monitoring at various interaction depth points provides insights into the depth of interactions. InSVDF adjusts its energy allocation strategy based on this depth.

In summary, InSVDF constructs a model of the virtual device's intra-interface state, grounded in the interaction behavior of the virtual device and the interaction features of the DMA interface.

2) Capturing the Intra-Interface State

Figure 3 illustrates the method for capturing a virtual device's intra-interface state. We perform static analysis on the hypervisor source code and apply instrumentation during compilation, enabling real-time state retrieval during fuzzing.

We identify and analyze interaction behavior based on the device's state variables. Initially, we intercept its registration function to extract these variables, which are crucial for registration, operation, and destruction. To enhance relevance, we apply live variable analysis to filter out irrelevant elements, consolidating the remaining variables into a set for effective monitoring of virtual device behaviors.

To avoid significant overhead from distinguishing behavior changes with low thresholds, we utilize static analysis to investigate the relationship between value intervals of state variables and corresponding behavioral changes. These intervals are recorded and encoded for differentiation. We perform real-time hashing on the intervals to which each element belongs during the fuzzing procedure. The resulting hash values are then used to characterize the current interaction behavior.

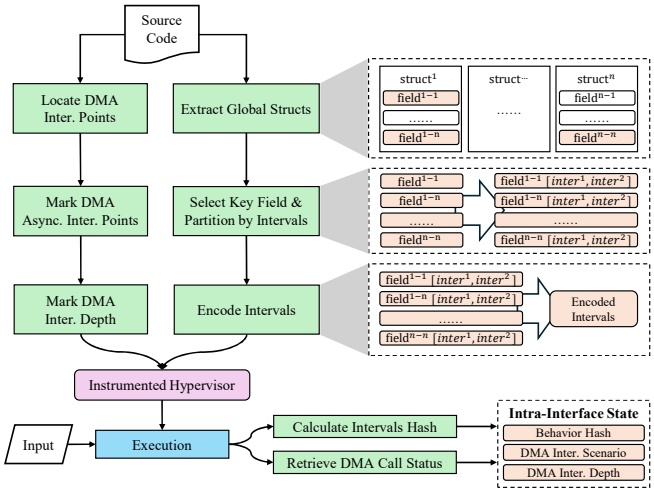


Fig. 3: Illustration of the Intra-Interface State Model

We determine the target device's interface characteristics by automatically labeling its DMA interaction points. First, we identify all DMA interaction points within the virtual device. Second, we classify them based on their invocation scenarios following Li et al. [27], we label those triggered by timers or bh events as `Asy`, and others as `Syn`. Third, we analyze the call chain to assign interaction depths, marking points from direct to nested triggering. For those appearing in multiple call chains, we create and label distinct clones to accurately represent different depths.

Since the intra-interface state of the virtual device is primarily influenced by data input, during the fuzzing process, we obtain the real-time intra-interface state of the tested device each time one test case is executed. This state includes the interaction behavior hash value, the invocation scenario label, and the interaction depth.

D. Asynchrony-Aware State Snapshot

The invocation scenario of asynchronous DMA interactions is governed by the hypervisor and remains outside the user's control. As a result, the fuzzer cannot actively and reliably trigger asynchronous DMA interactions by constructing specific sequences of test cases. To address this issue, we develop an **asynchrony-aware state snapshot mechanism (ASS)**. As demonstrated in Algorithm 1, we enhance the success rate of completing asynchronous interactions by saving, sorting, and resetting the intra-interface state of devices.

State Saving. InSVDF maintains a library of intra-interface states during fuzzing. Each time one test case is executed, InSVDF captures the intra-interface state of the tested virtual device and updates the state library. This involves creating a snapshot of the new state and updating the counters for existing states. Creating a snapshot of the new state involves recording the real-time values of relevant elements in the virtual device's state variables. These values are then saved in a writable segment of the hypervisor's address space.

Algorithm 1 Asynchrony-aware State Snapshot

Require: Seed_Pool S, Target_Device T
initialize Lib_{state};
set $state_{opt} = 0$, $state_{cur} = 0$;
for each test case c in S **do**
  if $Lib_{state} \neq \varnothing$ **then**
    $state_{opt} \leftarrow$ `Sort_State`(Lib_{state});
    $state_{opt}.selected\_times$++;
    `Reset_State`($state_{opt}$, T);
  end if
  `Execute_Input`(c);
  $state_{cur} \leftarrow$ `Capture_State`(T);
  if triggerAsy($state_{cur}$) **then**
    if isNewState($state_{cur}$) **then**
      `Save_State`($state_{cur}$, Lib_{state});
    else if accessDeeperDep($state_{cur}$) **then**
      `Update_State`($state_{cur}$, Lib_{state});
    end if
  end if
  $state_{cur}.reached\_times$++;
end for

State Sorting. Each time the state library is updated, InSVDF scores each intra-interface state based on its invocation scenario label, interaction depth, arrival and selection count. The states are then ranked, and the optimal state is chosen for the current test process. In scoring the intra-interface states, those capable of triggering asynchronous DMA interaction behaviors receive higher scores. The score of a state decreases with increasing arrival and selection count, which helps prevent InSVDF from becoming trapped in a local optimum.

State Resetting. Before processing each test case, InSVDF checks the running status of the hypervisor and resets the state of the tested device to the selected optimal state accordingly, meaning it restores the relevant elements of the state variables to their recorded values. At the same time, InSVDF updates the selection count of the optimal state. This process completes the resetting of the virtual device's intra-interface state, after which InSVDF resumes the fuzzing test. Notably, all intra-interface states are collected during the fuzzing process, ensuring accessibility and preventing false positives in state resetting. To fully capture the intra-interface states of the tested virtual device, InSVDF only saves and sorts states during the first round of fuzzing. State resetting is carried out only after the second round begins. These steps constitute the asynchrony-aware state snapshot mechanism, enhancing InSVDF's stability in conducting asynchronous DMA interactions.

E. Depth-Aware Seed Preservation

In nested interaction behavior, the likelihood of accessing deeper interaction points diminishes as the interaction depth increases. To address this, we have designed the **depth-aware seed preservation mechanism(DSP)**. This mechanism focuses on allocating more resources to deeper DMA interactions and accelerating vulnerability discovery in such behaviors. By preserving seeds that trigger maximum interaction depths and maintaining seed pools of varying sizes according to different interaction depths, we enhance the probability of targeting deeper DMA interactions effectively. Based on this idea, we further design the following two strategies:

Max-Interaction Depth Record. During the fuzzing process, InSVDF tracks the maximum DMA interface interaction depth. Whenever a test case is executed, InSVDF extracts the interaction depth value from the intra-interface state and updates the global maximum DMA interface interaction depth if the current interaction depth exceeds the recorded maximum.

Depth-Based Seed Preservation. InSVDF focuses more energy on deeper interactions by increasing the number of seeds that can trigger such interactions. During fuzzing, after processing each test case, InSVDF evaluates its interaction depth and compares it with the current global maximum DMA interaction depth. If the test case's interaction depth is greater than or equal to the global maximum, it is preserved in the seed pool due to its potential for exploring deeper interaction behaviors. Additionally, InSVDF sets thresholds for the number of seeds at each level of interaction depth, with the maximum number of seeds increasing as the interaction depth grows. This strategy helps counteract the low probability of triggering deeper interaction behaviors.

With these two features, we enhance the fuzzing process's sensitivity to the depth of virtual device interactions, allowing us to focus more on testing deeper interaction behaviors.

IV. Implementation

We implemented InSVDF[1] with 3.5k LoC and 380 lines of CodeQL [28] code. The static analysis is conducted using the CodeQL suite, and the fuzzing framework is based on V-Shuttle and AFL [12]. In the following sections, we will discuss the details of InSVDF.

A. Intra-Interface State: Access and Adjustment

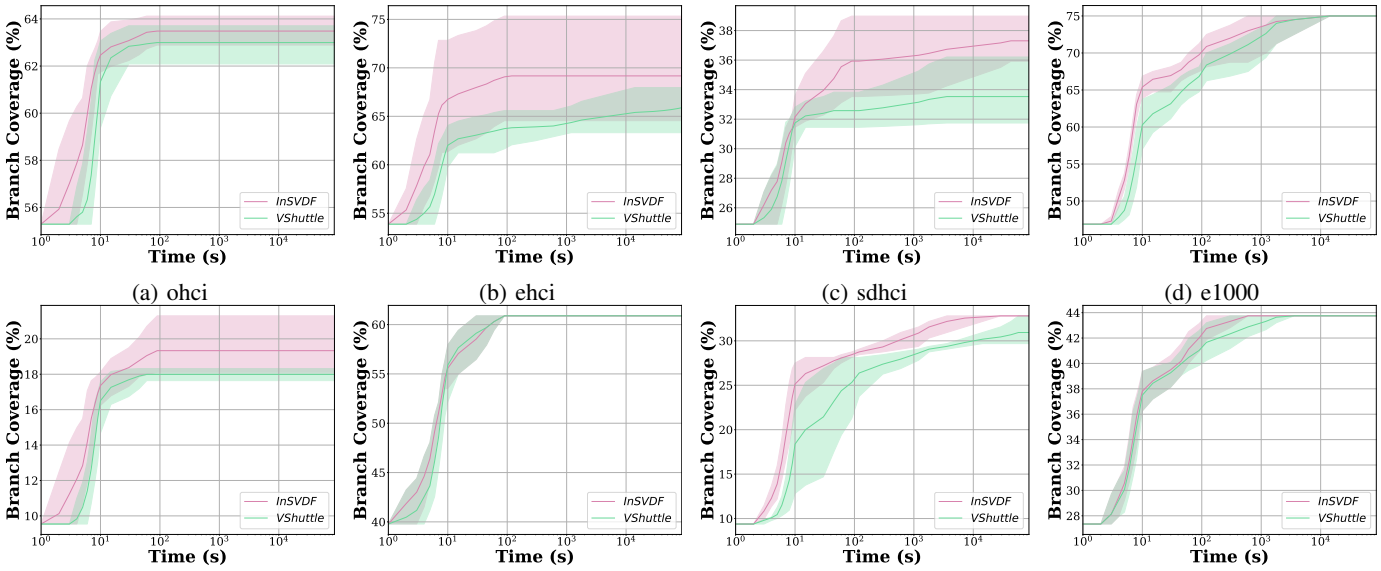
To accurately obtain the real-time values of useful elements in state variables that affect interaction behavior within the hypervisor, We instrument the target device's initialization function to identify the base address of state variables affecting interaction behavior within the hypervisor. Through static analysis, we determine the type and offset of each element. A harness function is then constructed to retrieve these values in real time during fuzzing, assessing their intervals for accurate coding, and generating a hash key that effectively describes the interaction behavior.

B. Intra-Interface State Library: Design and Workflow

In InSVDF, the intra-interface state is managed by the fuzzing agent and the fuzzer. The fuzzing agent handles the acquisition and setting of the intra-interface state, while the fuzzer is responsible for scoring and sorting the state. Accordingly, the state library is divided into two parts, as illustrated in Figure 5.

State List: Located within the fuzzer's address space, this component stores data necessary for state sorting, including the hash key, invocation scenario label, interaction depth value, arrival count, selected count, and the state score.

[1]https://github.com/Chan9Yan9/InSVDF



(a) ohci (b) ehci (c) sdhci (d) e1000

(e) virtio-gpu (f) mptsas1068 (g) ohci(VirtualBox) (h) pcnet(VirtualBox)

Fig. 4: Branch coverages over 24 hours. The shaded regions represent the maximum and minimum coverage achieved by fuzzers across ten runs.

State Pool: Maintained within the hypervisor, this component stores the hash key and state buffer. The state buffer is a memory area allocated by the fuzzing agent within the hypervisor to hold the real-time values of useful elements in the state variables whenever a new state is discovered.

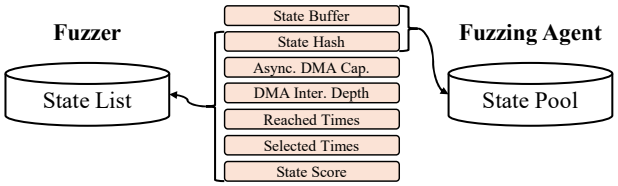


Fig. 5: The Composition of the State Library

The hash key uniquely corresponds to the interaction behavior, serving as a synchronization signal between the fuzzing agent and the fuzzer in InSVDF.

V. Evaluation

We compared InSVDF with the state-of-the-art virtual device fuzzer, V-Shuttle [18], to answer the following research questions:

- **RQ1.** What is the performance of InSVDF on virtual device vulnerability discovery?
- **RQ2.** Are the asynchrony-aware state snapshot mechanism and the depth-aware seed preservation mechanism effective?
- **RQ3.** Can InSVDF effectively help to find new virtual device vulnerabilities?

A. Evaluation Setup

We perform our experiments on a server with an Intel(R) Xeon(R) Gold 6330 CPU and 256GB RAM. We conducted fuzzing on each virtual device with one CPU core for 24 hours and repeated each test 10 times.

B. Overall Effectiveness

To answer **RQ1**, we will analyze both the coverage and the speed of discovery of known vulnerabilities.

1) Coverage Comparison

To evaluate the impact of InSVDF on code coverage, we selected six devices from various subsystems in QEMU for testing. These devices were chosen for two key reasons. First, they span USB, storage, network and graphics categories, all of which involve intensive DMA transfers. Second, these devices have recently been found to have DMA-related vulnerabilities, making their security evaluation particularly important. Additionally, to demonstrate the scalability of our method, we randomly selected two devices from VirtualBox for testing. We used the state-of-the-art fuzzer V-Shuttle for comparison and configured our system with the same settings as V-Shuttle to collect initial seeds under standard full-system emulation. Each device was tested ten times, with the maximum, minimum, and average coverage recorded at each time point. The branch coverage results are displayed in Figure 4.

From the figure, we can observe that our system exhibits a more significant coverage growth rate compared to V-Shuttle across all eight devices. We attribute this improvement primarily to the asynchrony-aware state snapshot (ASS) mechanism. By resetting the intra-interface state to one that can trigger asynchronous DMA interactions, the ASS mechanism helps the fuzzer to explore branching structures associated with asynchronous interactions more effectively, thereby boosting the coverage growth rate.

Regarding total coverage, our system outperforms V-Shuttle in 5 out of 8 devices. This gain is mainly attributed to

the depth-aware seed preservation (DSP) mechanism, which preserves more seeds capable of triggering deep interaction behaviors. This allows InSVDF to achieve higher final coverage compared to V-Shuttle. However, these enhancements are modest, with a growth range of 1-4%. This limited improvement is due to our approach focusing primarily on the sensitivity to the interface's inner state, which mainly enhances the validity of our inputs.

2) Bug Discovery Comparison

To compare the actual performance of InSVDF and V-Shuttle in detecting known vulnerabilities. We collected nine DMA-related bugs in QEMU and ran InSVDF and V-Shuttle on the devices associated with these vulnerabilities to evaluate their effectiveness in discovering these issues. We measured the time taken by each system to uncover the vulnerabilities, setting the maximum testing time to 24 hours. If a system failed to discover a vulnerability within this period, the discovery time was recorded as 24 hours. We conducted ten tests per target device to calculate the average discovery time. The results are presented in Table II.

TABLE II: Time consumption of discovering previous known vulnerabilities by InSVDF and V-Shuttle

Category	Virtual Device	Vulnerability	InSVDF	V-Shuttle	Gain
USB	ohci	CVE-2020-25624	**24mins**	132mins	5.5x
	ohci	CVE-2024-8354	**13mins**	315mins	24.2x
	ehci	CVE-2021-3750	**72mins**	871mins	12.1x
Storage	sdhci	CVE-2022-3872	**58mins**	890mins	15.3x
	sdhci	Issue #451	**87mins**	963mins	11.1x
	mptsas1068	CVE-2021-3392	**173mins**	215mins	1.2x
	lsi53c895a	CVE-2023-0330	**81mins**	470mins	5.8x
	nvme	CVE-2021-3929	**57mins**	314mins	5.5x
Network	e1000e	Issue #1543	**67mins**	144mins	2.1x

From the table, it is evident that InSVDF significantly enhances the efficiency of detecting DMA-related vulnerabilities compared to V-Shuttle. The improvement for CVE-2021-3392 is the smallest, at just 1.2x, which aligns with the coverage results. This vulnerability is due to an error in initializing and adding a new object into the queue within the `mptsas_process_scsi_io_request()` function. The root cause of this vulnerability is minimally connected to the interface state addressed in this paper, resulting in a less pronounced gain.

Our system demonstrated its highest performance with CVE-2024-8354, achieving up to a 24x improvement in elapsed time. We attribute this significant gain to factors beyond merely increased coverage. To highlight the effectiveness of our approach, we will conduct a detailed analysis of this vulnerability, showcasing how our method accelerates the identification and resolution of this particular bug.

Case Study: CVE-2024-8354 involves an assertion failure in QEMU's USB module. Figure 6 shows the call chain leading to this bug. Triggered by a timer, the event exemplifies asynchronous DMA behavior, completing when `ohci` meets the required conditions. As discussed in Section II, existing fuzzers struggle to precisely control `ohci`, making it unlikely

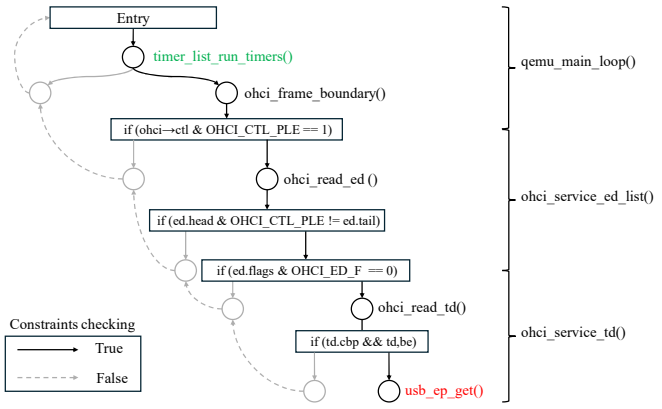


Fig. 6: The call chain of triggering CVE-2024-8354

to trigger this asynchronous DMA event. Our asynchrony-aware state snapshot (ASS) mechanism increases this likelihood, accelerating vulnerability discovery. Additionally, the vulnerable function `usb_ep_get` resides at the third level of nested DMA calls. By employing depth-aware seed preservation (DSP), we retain test cases at various depths, improving efficiency in reaching the vulnerable function.

C. Analysis of Method Validity

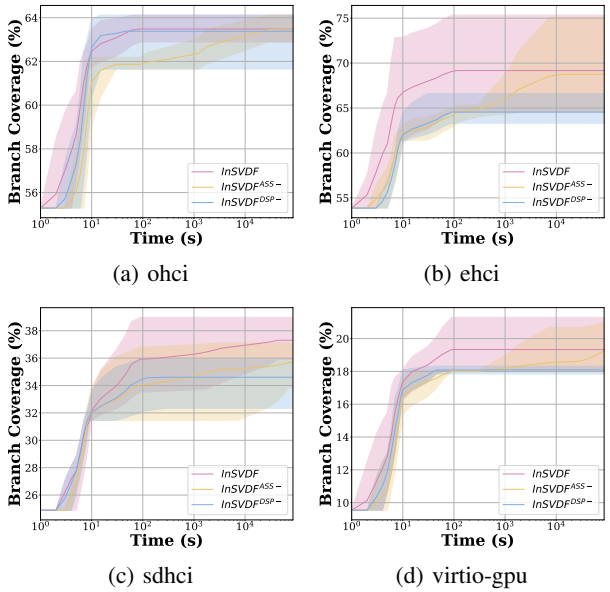
To address **RQ2**, we conducted ablation experiments on ASS and DSP to assess their contributions to vulnerability detection and analyze the overhead introduced by each component. We also combined our method with ViDeZZo's inter- and intra-message sensitivity techniques to evaluate whether our approach and semantics-aware fuzzing can effectively complement each other.

1) Per-Component Effectiveness

We chose four devices and performed InSVDF, InSVDF without ASS(InSVDF^{ASS-}), InSVDF without DSP(InSVDF^{DSP-}) on them, We collected branch coverage data for each configuration and the results are in Figure 7.

From the figure, it is evident that InSVDF demonstrates the best overall performance among the three configurations. Specifically, InSVDF^{ASS-} achieved superior final coverage compared to InSVDF^{DSP-} across all four devices, though its initial coverage growth was slower than that of both InSVDF and InSVDF^{DSP-}. Conversely, InSVDF^{DSP-} exhibited a faster rate of coverage growth early in the test but ultimately achieved weaker final coverage compared to InSVDF and InSVDF^{ASS-}, which aligns with our deduction in Section V-B1.

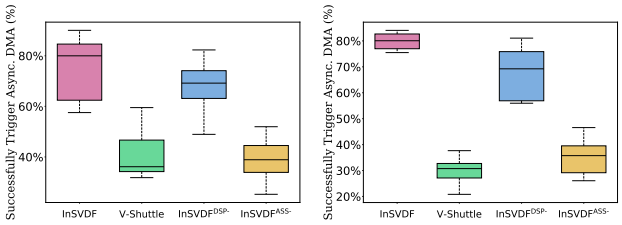
InSVDF^{ASS-} retains only DSP. This configuration facilitates a gradual increase in seeds that can explore deep interaction behaviors. As the experiment progresses, these seeds accumulate in the seed pool, enhancing the ability to discover previously uncovered deep interaction branches. Conversely, InSVDF^{DSP-} incorporates only the ASS, This approach allows for frequent resetting of the intra-interface state, which helps the fuzzer effectively trigger asynchronous DMA interactions



(a) ohci (b) ehci

(c) sdhci (d) virtio-gpu

Fig. 7: The ablation study of branch coverages



(a) ohci (b) virtio-gpu

Fig. 8: The boxplot of successfully triggering asynchronous DMA interaction portion for InSVDF, V-Shuttle, InSVDF^{DSP-}, InSVDF^{ASS-}

early in the testing phase. This results in quicker exploration of branches related to asynchronous behaviors. However, the frequent state resets hinder the execution of branches requiring deep and repeated interactions, resulting in weaker final coverage compared to InSVDF^{ASS-} and the full InSVDF configuration. Incorporating both ASS and DSP, InSVDF achieves the optimal balance, providing the best overall performance.

To validate our conclusions, we designed two auxiliary experiments for ASS and DSP usage scenarios, respectively, to better illustrate their roles.

Effectiveness of Asynchrony-Aware State Snapshot. In this experiment, to demonstrate the effectiveness of ASS, we evaluated InSVDF, InSVDF without ASS (InSVDF^{ASS-}), InSVDF without DSP (InSVDF^{DSP-}), and V-Shuttle on `ohci` and `virtio-gpu`, both of which involve asynchronous DMA interactions in their interrupt handling functions. We focused on monitoring the initiation points of asynchronous interrupts, such as timers or bh events, as well as the asynchronous DMA interaction points. By analyzing the ratio of occurrences where the fuzzer successfully triggers these asynchronous events, we assessed the system's effectiveness in initiating asynchronous DMA interaction behaviors.

Figure 8 illustrates the ratio of successful DMA data transmissions to the total number of asynchronous interrupts initiated over a 24-hour period. The results indicate that both InSVDF and InSVDF^{DSP-} significantly improve the percentage of successfully triggered asynchronous DMA interactions compared to V-Shuttle. In contrast, InSVDF^{ASS-} does not exhibit a similar improvement. This demonstrates that the asynchrony-aware state snapshot (ASS) design effectively enhances the fuzzer's sensitivity to asynchronous interaction behaviors at the virtual device interface, leading to a higher success rate in triggering asynchronous DMA interactions.

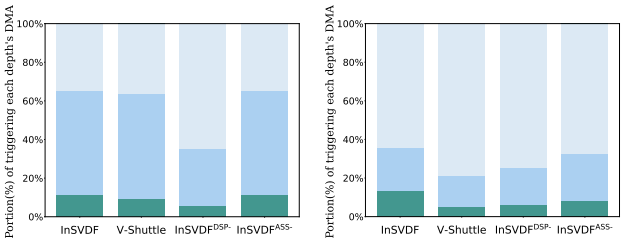
As discussed in Section V-B1, we identify ASS as key to accelerating vulnerability detection by ensuring stable input and overcoming the asynchronous interaction challenge. As shown in Table II, the detection speed improvement is most notable for CVE-2024-8354, CVE-2021-3750 and CVE-2022-3872, where vulnerability chains are triggered by timers or bh events, providing strong evidence for our hypothesis.

Effectiveness of Depth-Aware Seed Preservation. In this experiment, we aimed to assess the effectiveness of the depth-aware seed preservation (DSP) mechanism. We chose `ohci` and `virtio-gpu` for this evaluation due to their complex DMA interaction depths of at least three layers. Tests were conducted using InSVDF, InSVDF^{ASS-}, InSVDF^{DSP-}, and V-Shuttle on these devices.

We tracked the frequency of DMA interactions at each depth level, using the number of triggered first-layer DMA interactions as the baseline. We then calculated the percentage of occurrences for second-layer DMA interactions and interactions at depths greater than or equal to three layers. The results are illustrated in Figure 9.

The figure reveals that both InSVDF and InSVDF^{ASS-} outperforms V-Shuttle, indicating that the depth-aware seed preservation (DSP) mechanism significantly improves the proportion of deep DMA interactions within the overall DMA interaction behavior. In contrast, InSVDF^{DSP-} shows only a slight advantage over V-Shuttle on `virtio-gpu` and is even less effective than V-Shuttle on `ohci`.

This performance discrepancy can be attributed to two main factors. First, in the case of `virtio-gpu`, V-Shuttle has a lower baseline for deep DMA interactions. Therefore, the ASS mechanism retains states capable of triggering second or third-layer DMA interactions, which provided a notable enhancement. Second, for `ohci`, V-Shuttle already achieves a high rate of triggering second-layer DMA interactions (around 63%). In this device, the primary challenge obstacle the fuzzing is the uncertain moment of initiating asynchronous interactions. The frequent state resets by ASS consume fuzzing resources that could otherwise be used to explore deeper interactions. This results in a lower percentage of configurations that exclusively retain ASS triggering deeper interaction behaviors. Overall, InSVDF effectively combines the strengths of both DSP and ASS, delivering the optimal performance in discovering DMA-



(a) ohci (b) virtio-gpu

Fig. 9: Proportion(%) of triggering the first-layer, the second-layer and the the higher-layer DMA interactions for InSVDF, V-Shuttle, InSVDF^{DSP-}, InSVDF^{ASS-}

related vulnerabilities.

Case Study: CVE-2021-3929 is a use-after-free vulnerability. Due to the lack of isolation between DMA and MMIO addresses in the hypervisor implementation, an unintended operation during the NVMe device's DMA write to MMIO control registers invokes the `nvme_write_bar()` function, triggering a memory release. Normally, this function does not cause such issues, and typical fuzzers cannot distinguish this scenario. However, with DSP's ability to detect interaction depth, the fuzzer identifies the interaction depth of 1 instead of 0, facilitating faster vulnerability detection.

Based on the results from the two auxiliary experiments, we can confirm that both asynchrony-aware state snapshot (ASS) and depth-aware seed preservation (DSP) effectively achieve their design objectives. These two designs are not entirely independent in their benefits, within certain operational ranges, they interact and complement each other. Their combined effect enhances the fuzzer's sensitivity to the interface's operational state, thereby significantly improving the efficiency of vulnerability discovery.

2) Overhead Analysis

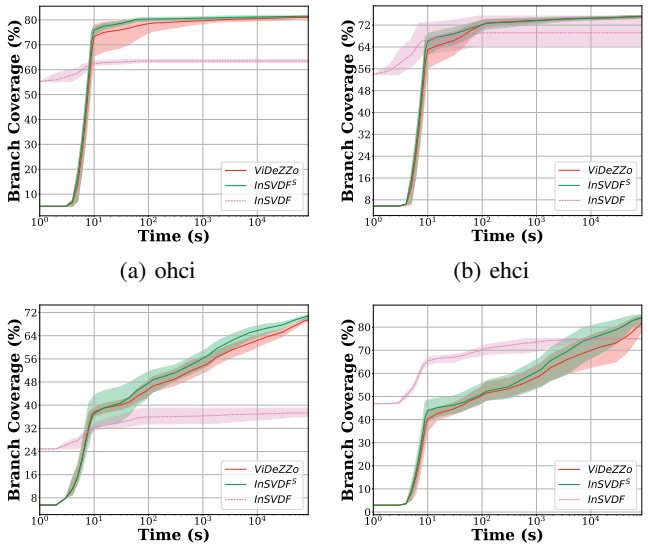
In addition to verifying the effectiveness of each component, we also analyzed the overhead introduced by each individually, where we measured the throughput of each configuration during the fuzzing process, and the results are shown in Table III. The table shows that InSVDF has a lower testing speed than V-Shuttle, primarily due to the ASS component, which resets the target device to its optimal intra-interface state before input and captures the state post-execution. These operations inevitably reduce throughput, but the impact is limited to around 15% due to our lightweight design. Moreover, as seen in Figure 4, InSVDF's fuzzing efficiency is comparable to V-Shuttle, suggesting that the overhead does not significantly affect performance.

3) Relationship with semantic analysis

Our work focuses primarily on enhancing the stability of input, in contrast to mainstream semantics-aware approaches [20], [21], which prioritize input effectiveness. In this section, we integrated our method with ViDeZZo's intra- and inter-message dependency analysis to explore whether these approaches can complement each other.

TABLE III: Average speed(*exec/s*) of V-Shuttle and InSVDF

Device	V-Shuttle	InSVDF	InSVDF^{DSP-}	InSVDF^{ASS-}
ohci	7125.9	6183.7	6317.4	6989.7
ehci	2452.3	2067.2	2310.2	2439.8
sdhci	326.7	281.5	290.1	320.7
virtio-gpu	3125.4	2767.5	2805.7	3104.2



(a) ohci (b) ehci

(c) sdhci (d) e1000

Fig. 10: Branch coverages over 24 hours. The shaded regions represent the maximum and minimum coverage achieved by fuzzers across ten runs.

We tested four devices using InSVDF, ViDeZZo, and InSVDF with semantic analysis (InSVDF^{S}). To highlight the impact of semantic analysis, both InSVDF^{S} and ViDeZZo were configured with no initial seeds. The coverage statistics are presented in Figure 10. We can see that semantic analysis significantly improves coverage. Meanwhile, we observe that our method still maintains an advantage in terms of coverage speed. We attribute this gain to the combined effects of ASS and DSP, where ASS enhances the stability of fuzzing inputs, and the integration of semantic-awareness techniques with DSP improves the effectiveness of inputs at varying interaction depths. Therefore, we consider that our approach and semantic analysis can mutually reinforce each other.

D. Discovery New Vulnerabilities

During our experiments, InSVDF discovered a total of 2 previously unknown QEMU virtual device vulnerabilities. The details of these vulnerabilities are shown in the Table IV. We have reported both vulnerabilities to the manufacturer, one has been fixed and assigned a CVE ID. To answer **RQ3**, we will analyze the CVE to demonstrate how our methodology contributed to uncovering this vulnerability.

TABLE IV: New bugs discovered by InSVDF

Virtual Device	Bug ID	Fix Time	Bug Type	CVSS v3.0
virtio-gpu	CVE-2024-3446	2024.04	use-after-free	8.2
ohci	Issue #2435	-	assertion failure	-

Case Study: CVE-2024-3446 is a vulnerability we identified in the virtio-gpu device, which has an 8.2 CVSS score according to CVE Detail. A malicious privileged guest could exploit this flaw to crash the QEMU process on the host, leading to a denial of service, or potentially execute arbitrary code within the context of the QEMU process on the host. This vulnerability is also caused by unintended DMA write operations on the MMIO address region, requiring multiple interactions with the same DMA interface to trigger. InSVDF's enhanced capability to detect vulnerabilities in deep DMA interactions facilitated the discovery of this issue.

VI. Threat Validity

More Complete State Monitoring. InSVDF focuses on monitoring the virtual device's DMA interface to enhance interaction efficiency and accelerate vulnerability discovery. However, it does not address other aspects of the virtual device's operation, such as memory usage and event scheduling. While our method improves vulnerability detection efficiency, it does not fundamentally enhance the overall detection capability. We recognize this limitation and plan to explore broader solutions in future research.

State Variation-Based Vulnerability Discovery. In our system design, we establish an interface state model to enhance vulnerability mining efficiency by monitoring and recovering interface states. However, we recognize that our approach does not fully leverage the potential of this state model. By integrating data flow analysis, we can uncover the relationships between state variables and interaction behaviors. Additionally, through the mutation of existing states, we could explore unique states that may trigger previously unknown vulnerabilities. We believe that there is substantial potential for further exploration in this area.

Specific-Type Vulnerability Triggering Model. In addition to developing a virtual device interface model, it is also valuable to construct a virtual device vulnerability model. While most current research methods focus on enhancing the efficiency of vulnerability discovery and improving coverage, creating specific-type vulnerability-triggering models is essential for discovering vulnerabilities that align with expected interaction characteristics and processes. This approach can help identify higher-risk vulnerabilities within virtual devices.

VII. Related Works

Virtual Device Fuzzing. The existing fuzzing of virtual devices mainly focuses on their basic interfaces, such as MMIO/PIO and DMA. Current research mainly focuses on the data input methods and the dependencies between messages. Significant efforts have been made to address data entry issues [18], [19], [23], [29]–[31]. The interface redirection methods proposed by V-Shuttle [18] and Morphuzz [19] provide excellent solutions to these challenges. Subsequent work [20]–[22], [24] has concentrated on the correlations between virtual device interaction messages. For example, MundoFuzz [20] employs differential testing to infer inter-message dependencies, while ViDeZZo [21] utilizes static analysis to extract both intra- and inter-message dependencies. These approaches leverage these correlations to guide the generation of high-quality test case sequences, effectively enhancing code coverage. These approaches fail to consider the implications of the interface state. Therefore, our work addresses the interface states.

State-Aware Fuzzing. State-guided fuzzing can supplement the deficiency of coverage-guided fuzzing. This approach first appeared in the network protocol fuzzing [32]–[34]by using protocol status code [35], [36] as feedback. In recent years, security researchers have begun modeling the state of software to guide the fuzzing process [37]–[42]. Two mainstream methods are viewing runtime memory as the state [38] and considering environment variables as the state [42]. InSVDF models the intra-interface state using variables, which shares similarities with StateFuzz [42]. This similarity arises from the inherent characteristic of both the kernel and hypervisor, where the behaviors of their components are maintained through numerous variables. However, relying solely on feedback to guide fuzzing is not sufficient to address the challenges posed by asynchronous interactions. Overcoming this challenge requires external control over the state. InSVDF goes a step further by actively controlling the state during fuzzing. We achieve this by saving state snapshots and resetting them, which helps to enhance fuzzing efficiency.

VIII. Conclusion

In this paper, we propose an interface state-aware fuzzing method for hypervisor virtual devices to effectively uncover DMA-related vulnerabilities. We developed a prototype system, InSVDF, which monitors the operational state of DMA interfaces. This system first models the intra-interface state of virtual devices and then employs an asynchrony-aware state snapshot mechanism alongside a depth-aware seed preservation mechanism. These techniques enhance the fuzzer's sensitivity to both the invocation moment and the interaction depth of the DMA interactions. We compared our system with the state-of-the-art fuzzer V-Shuttle on QEMU and VirtualBox. InSVDF significantly accelerates vulnerability discovery. During the experiments, InSVDF also identified two new vulnerabilities in QEMU, one of which has been fixed and assigned a CVE ID.

Acknowledgment

We sincerely thank all the anonymous reviewers for their valuable comments and suggestions to help us improve this work. This work is partially supported by the National Key Research and Development Program of China (No.2022YFB3102904) and the National Natural Science Foundation of China (Grants No.62402147).

REFERENCES

[1] A. Desai, R. Oza, P. Sharma, and B. Patel, "Hypervisor: A survey on concepts and taxonomy," International Journal of Innovative Technology and Exploring Engineering, vol. 2, no. 3, pp. 222–225, 2013.

[2] Q. Developers, "Qemu: Open source machine emulator and virtualizer," https://www.qemu.org/, 2024.

[3] Oracle, "Virtualbox: Powerful x86 and amd64/intel64 virtualization," https://www.virtualbox.org/, 2024.

[4] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong, "The characteristics of cloud computing," in 2010 39th International Conference on Parallel Processing Workshops. IEEE, 2010, pp. 275–279.

[5] R. Shacham, S. Schulzrinne, S. Thakolsri, and W. Kellerer, "The virtual device: Expanding wireless communication services through service discovery and session mobility," in WiMob'2005), IEEE International Conference on Wireless And Mobile Computing, Networking And Communications, 2005., vol. 4. IEEE, 2005, pp. 73–81.

[6] Red Hat, "Venom: Qemu vulnerability (cve-2015-3456)," 2015. [Online]. Available: https://access.redhat.com/articles/1444903

[7] Wikipedia, "Virtual machine escape," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Virtual_machine_escape

[8] National Vulnerability Database, "Cve-2009-1244 detail," 2009. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2009-1244

[9] Google Cloud, "Esxi hypervisors: New malware persistence methods," 2023. [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/esxi-hypervisors-malware-persistence/

[10] Microsoft, "Virtualization-based security (vbs) and secure boot," 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs

[11] VMware, "Virtualized security," 2023. [Online]. Available: https://www.vmware.com/topics/virtualized-security

[12] M. Zalewski, "American fuzzy lop (afl)," https://lcamtuf.coredump.cx/afl/, 2024.

[13] Google, "syzkaller: Linux kernel fuzzer," https://github.com/google/syzkaller, 2024.

[14] L. Developers, "Libfuzzer: A library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2024.

[15] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," IEEE Transactions on Software Engineering, vol. 47, no. 11, pp. 2312–2331, 2019.

[16] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," ACM Computing Surveys (CSUR), vol. 54, no. 11s, pp. 1–36, 2022.

[17] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled direct memory access: Isolating cpu and io traffic by leveraging a dual-data-port dram," in 2015 International Conference on Parallel Architecture and Compilation (PACT). IEEE, 2015, pp. 174–187.

[18] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 2197–2213.

[19] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "Morphuzz: Bending (input) space to fuzz virtual devices," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 1221–1238.

[20] C. Myung, G. Lee, and B. Lee, "{MundoFuzz}: Hypervisor fuzzing with statistical coverage testing and grammar inference," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 1257–1274.

[21] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer, "Videzzo: Dependency-aware virtual device fuzzing," in 2023 IEEE Symposium on Security and Privacy (SP). IEEE, 2023, pp. 3228–3245.

[22] Y. Liu, S. Chen, Y. Xie, Y. Wang, L. Chen, B. Wang, Y. Zeng, Z. Xue, and P. Su, "Vd-guard: Dma guided fuzzing for hypervisor virtual device," in 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023, pp. 1676–1687.

[23] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 2597–2614.

[24] A. Bulekov, Q. Liu, M. Egele, and M. Payer, "Hyperpill: Fuzzing for hypervisor-bugs by leveraging the hardware virtualization interface," in 33rd USENIX Security Symposium (USENIX Security 24), 2024.

[25] Intel, "Collecting processor trace in intel system debugger," https://www.intel.com/content/www/us/en/developer/videos/collecting-processor-trace-in-intel-system-debugger.html, 2024.

[26] QEMU, "Timers and replay," 2023. [Online]. Available: https://www.qemu.org/docs/master/devel/replay.html#timers

[27] Q. Li, G. Pan, H. He, and C. Wu, "Matryoshka trap: Recursive mmio flaws lead to vm escape," in CanSecWest, 2022.

[28] GitHub, "Codeql: Semantic code analysis engine," https://codeql.github.com/, 2024.

[29] T. Yu, X. Qu, and M. B. Cohen, "Vdtest: An automated framework to support testing for virtual devices," in Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 583–594.

[30] A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng, "Vdf: Targeted evolutionary fuzz testing of virtual devices," in Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings. Springer, 2017, pp. 3–25.

[31] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Hyper-cube: High-dimensional hypervisor fuzzing." in NDSS, 2020.

[32] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, 2020, pp. 460–465.

[33] Y. Yu, Z. Chen, S. Gan, and X. Wang, "Sgpfuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations," IEEE Access, vol. 8, pp. 198 668–198 678, 2020.

[34] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019, pp. 748–758.

[35] M. D. Network, "Http status codes," https://developer.mozilla.org/en-US/docs/Web/HTTP/Status, 2024.

[36] "Restful api design," https://restfulapi.net, 2024.

[37] S. Zhou, Z. Yang, D. Qiao, P. Liu, M. Yang, Z. Wang, and C. Wu, "Ferry:{State-Aware} symbolic execution for exploring {State-Dependent} program paths," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 4365–4382.

[38] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," Empirical Software Engineering, vol. 27, no. 7, p. 191, 2022.

[39] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 765–777.

[40] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in 2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020, pp. 1597–1612.

[41] A. Fioraldi, "Program state abstraction for feedback-driven fuzz testing using likely invariants," arXiv preprint arXiv:2012.11182, 2020.

[42] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "{StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 3273–3289.