SEDSpec: Securing Emulated Devices by Enforcing Execution Specification

Yang Chen[1,2], Shengzhi Zhang[3], Xiaoqi Jia[1,2*], Qihang Zhou[1], Heqing Huang[1], Shaowen Xu[1,2], Haochao Du[1]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3]Metropolitan College, Boston University, USA

***Abstract*—Device emulation is a vital aspect of virtualization, yet remains vulnerable to security threats. Prior research has focused on monitoring I/O data flow or identifying internal device anomalies but often falls short in precision and automation. In this paper, we propose a novel method that leverages the normal operations of an emulated device to formulate an execution specification. The specification acts as a criterion to evaluate the device's behavior and state transitions. We implement SEDSpec, a prototype system that automatically generates the execution specification for an emulated device and devises three check strategies for identifying any deviations from this specification, thereby ensuring normal operations and enhancing the security of the emulated device. We evaluate SEDSpec with five different execution specifications. The results show that SEDSpec can detect anomalies caused by vulnerability exploitation while maintaining the devices' regular functioning with minimal performance overhead.**

***Index Terms*—device emulation, anomaly detection, execution specification, program analysis**

I. Introduction

Virtualization is the cornerstone of cloud computing, facilitating resource sharing across multiple users. I/O virtualization serving as a representative virtualization technology, enables virtual machines to access various I/O devices, such as disks, video cards, and network cards. Device emulation, a typical implementation for the I/O virtualization, replicates the behavior and functions of physical devices in a software environment. This technique, however, confronts security challenges stemming from the sheer volume and diversity of emulated devices, compounded by the complexity and often unclear device specifications. Consequently, numerous vulnerabilities plague device emulation; for instance, in recently published 399 vulnerabilities in QEMU [1], 254 were attributed to device emulation [2].

These vulnerabilities can be exploited by attackers to initiate denial-of-service attacks, compromise cloud user data, or commandeer host systems, undermining the confidentiality, availability, and integrity of cloud service. Typically, these vulnerabilities are triggered by meticulously crafted I/O data streams that originate from a guest system, pass through a specific I/O interface, and reach the flawed emulated device. In pursuit of enhancing the security measures associated with I/O virtualization, efforts can be focused on two primary elements: the I/O data streams and the emulated devices themselves.

*Xiaoqi Jia is the corresponding author (email: jiaxiaoqi@iie.ac.cn).

For the I/O data streams, ensuring their authenticity can be achieved by implementing additional validation or filtering mechanisms within the I/O communication framework. For example, developing an intrusion detection system [3]–[6] serves as an impactful safeguard against attacks that exploit known, or "one-day" vulnerabilities in device emulation. By analyzing characteristics of I/O data streams from prior attacks, rules can be manually formulated to enable the intrusion detection system to intercept atypical I/O communications. However, this method requires significant manual effort and is ineffective against "zero-day" vulnerabilities. Alternatively, some researchers have endeavored to train I/O models that can detect anomalous data streams by learning from benign examples [7], [8], but the detection accuracy could be low when dealing with intricate I/O interactions.

For the virtual devices, there are two predominant approaches. The first approach involves enhancing the security of I/O virtualization by minimizing the Trusted Computing Base (TCB), which is achieved by removing certain virtual devices and narrowing the virtualization functionality scope [9], [10]. This reduction, however, may compromise the versatility of the hypervisor. The second approach involves detecting anomalous device state transitions through the construction of device state transition models, which are typically derived from the manual specifications of the devices [11]. This method, although promising, entails considerable manual effort to comprehend the device specifications and encapsulate the device state transitions.

In this paper, we propose SEDSpec, a novel approach that aims to enhance security in device emulation. In particular, SEDSpec enhances security by constructing and enforcing execution specifications for emulated devices, which serve as abstract representations of the legitimate behavior and state transitions. SEDSpec constructs the execution specification from the runtime information of the emulated device, which includes the data about the control flow and the state changes. SEDSpec then refines the execution specification by applying techniques of control flow reduction and data dependency recovery. Finally, SEDSpec integrates the execution specification into the emulated device and monitors its runtime behavior according to the execution specification and check strategies. If any anomalies are detected, SEDSpec intervenes by halting the operation or alerting a warning, depending on its working mode and the violated check strategy.

To evaluate the effectiveness and efficiency of SEDSpec, we apply it to five emulated devices in QEMU, including FDC (Floppy Disk Controller), USB EHCI (Enhanced Host Controller Interface), PCNet (PCI Network Adapter), SDHCI (Secure Digital Host Controller Interface), and SCSI (Small Computer System Interface). The experiments indicate that SEDSpec can achieve a false positive rate of no more than 0.15% in all device tests. Moreover, case studies show that SEDSpec is capable of detecting device anomalies caused by eight well-known vulnerability exploitations. These results demonstrate SEDSpec's effectiveness in ensuring the standard operation of emulated devices while identifying abnormal behaviors. Additionally, SEDSpec introduces no more than 5% performance overhead on the throughput and latency of disk devices. For network devices, the overhead is similar, with bandwidth and latency overhead of less than 8% and 10%, respectively.

Contributions. We summarize our contributions as below:

- We present a novel approach for the automatic generation of device execution specifications, leveraging the control flow and state transitions within an emulated device during legitimate I/O operations. This enables the computation of execution specifications solely by analyzing the I/O data stream, facilitating early anomaly detection prior to the activation of specific I/O operations in the emulated device.
- We design three strategies to detect violations of execution specification at runtime, i.e., a parameter check for integer overflow, an indirect jump check for control flow hijacking, and a conditional jump check for irregular device operation.
- We develop a prototype system, SEDSpec, that demonstrates efficient anomaly detection across diverse emulated devices while maintaining a low false positive rate and an acceptable level of performance overhead as shown by our experimental evaluation.

II. Background and Related works

A. Device Emulation in I/O Virtualization

Device emulation is a common method for implementing I/O virtualization, which provides various devices for virtual machines. A hypervisor must satisfy the demands of virtual machines in terms of device type and quantity, considering the limited number of physical devices in a host machine. Moreover, it should hide the real physical device details from virtual machines. To achieve these goals, device emulation replicates the behavior of relevant physical devices and provides input/output interfaces for virtual machines. As a result, a virtual machine can only access the device information constructed by device emulation. Device emulation also manages the communication between virtual machines and physical devices on the host machine., preventing direct modification of physical devices and enabling device sharing among multiple users [12].

In the architecture combining Kernel-based Virtual Machine (KVM) [13] and QEMU, the latter is responsible for emulating a range of hardware devices to service virtual machine requests. Specifically, when a guest operating system (OS) issues an I/O request via interfaces like Port-Mapped I/O (PMIO), Memory-Mapped I/O (MMIO), or Direct Memory Access (DMA), the request is first intercepted by KVM. Subsequently, this request is forwarded to QEMU, which discerns and initiates the execution of suitable device emulation routines tailored to the I/O request specifics. These routines typically encompass operations on control registers, data buffers, and DMA controllers of the emulated device, as well as the invocation of interrupts or signals directed at the guest OS. Upon completion of the emulation process, control is handed back to KVM, facilitating the continuation of the guest OS's operations. This seamless process enables the guest OS to interact with the emulated devices as if they were physical entities.

B. Related Work

I/O virtualization is prone to security challenges and vulnerabilities, which have been extensively studied by researchers using fuzzing techniques [14]–[19]. These investigative endeavors have uncovered numerous security deficiencies within I/O virtualization, indicating an urgent requirement for security improvement. Consequently, substantial efforts are being directed towards enhancing the security of I/O virtualization.

Enhancing security through the reduction of the TCB is a viable strategy. This approach primarily involves minimizing the hypervisor's role in facilitating virtual machine access to I/O devices. For instance, Bitvisor [9] allows the guest OS to directly access some physical devices without hypervisor intervention. Similarly, NoHype [10] and Min-V [20] selectively remove or disable some virtual devices within the hypervisor based on specific cloud environment requirements. However, these approaches modify the original hypervisor, potentially leading to compatibility issues. Furthermore, they do not address vulnerabilities inherent in the remaining virtual devices.

I/O virtualization security can be improved through the implementation of hypervisor intrusion detection systems. Systems such as ELT [3] and CloudSeer [5] exemplify this approach, monitoring the data stream in I/O communication between a guest machine and the hypervisor. These systems validate the legitimacy of I/O communication based on predefined rules. However, this methodology may not be effective against complex attacks that exhibit elusive characteristics. Furthermore, ensuring accuracy and comprehensiveness in attack identification within this framework necessitates substantial human intervention.

Model-based detection approaches upon I/O data stream [8] or virtual device states [11] serve as another alternative to defend the anomalies in I/O communication. For instance, VMDec [8] employs Markov models to represent expected I/O sequences, flagging any deviations from these models as anomalies. However, due to the lack of knowledge of the internal execution logic of I/O devices, this method may exhibit limited accuracy when detecting device anomalies triggered by complex I/O data streams. On the other hand, Nioh [11] constructs a finite state machine representing the operational state of a device based on manually written device specifications. This model delineates the possible device states and transitions between them according to the specification. Any state transition that is not accounted for in the model is deemed anomalous. Nevertheless, the scalability of this approach is limited because it relies on manual interpretation and analysis.

III. Overview

A. Threat Model

In this study, we aim to detect abnormal behaviors of emulated devices in a hypervisor caused by anomalous I/O requests that may lead to potential attacks. To achieve this goal, we define a threat model that considers various threats that could disrupt the normal operation flow and data state transitions of emulated devices. However, it should be noted that this model does not take into account attacks that arise from legitimate I/O requests or assaults stemming from external data irrelevant to the device state.

We assume that arbitrary emulated devices can be enabled and accessed by any virtual machine, without any security access policies [10], [20] that restrict the use of devices. Furthermore, we assume that an attacker has the capability to access device drivers and interact with I/O devices from a virtual machine. Such an attacker could be a root user on the guest OS or a malicious process with elevated privileges. The primary objective of the attacker is to exploit vulnerabilities in emulated devices by transmitting malicious commands or data to the device, thereby compromising the confidentiality, availability, or integrity of the hypervisor.

B. Motivation

Several factors can contribute to the introduction of vulnerabilities in device emulation implementations. Firstly, an emulated device is typically based on a device specification, which is often written in natural language and thus prone to ambiguity. This can result in bugs or inconsistencies when attempting to replicate the exact behavior of physical devices in software form. Secondly, many devices possess intricate implementation logic, and the proficiency and experience of the developers involved in the development vary widely, potentially leading to implementation bugs. Lastly, unlike hardware devices that maintain physical data separation, software devices share a common memory space. Consequently, certain corner cases that may pose minimal security risks within hardware devices could manifest as critical vulnerabilities within their emulated counterparts.

Vulnerabilities in emulated devices pose significant threats to the security and availability of multi-tenant cloud environments. A compromise in one tenant's emulated device could potentially impact the entire cloud environment, disrupting services and jeopardizing data security for all tenants. For example, certain bugs such as infinite loops can consume substantial physical resources, thereby degrading the performance of co-located virtual machines. More alarmingly, vulnerabilities that facilitate virtual machine escape provide attackers with access to sensitive data on both the host machine and other virtual machines. They may even enable the execution of high-privileged code, potentially granting control over the entire host machine.

Nioh [11] posits that exploits in emulated devices often trigger anomalous state transitions, detectable by constructing a device state transition automaton. However, their approach relies on manual interpretation and analysis of device specifications, which is labor-intensive and lacks scalability in a virtualization environment with many virtual devices. It suggests the need for a more automated and scalable solution. We observe that the anomalous state transitions identified by Nioh also manifest as execution paths not taken during normal device operations at the code level. Based on this observation, we propose a novel approach to automatically construct an execution specification of a device by analyzing its control flow and internal data. This specification can detect abnormal device behavior by deploying check strategies and can be adapted to different device commands for finer-grained anomaly detection.

Address Sanitizer (ASan) [21], Undefined Behavior Sanitizer (UBSan) [22], and Control Flow Integrity (CFI) [23] are runtime software security enhancement techniques that leverage source code instrumentation and are applicable to emulated devices. However, these techniques do not specifically target abnormal state transitions of emulated devices, thus failing to identify vulnerabilities that stem from such transitions. For example, CVE-2016-7909 [24], a vulnerability that triggers an infinite loop within PCNet due to an abnormal state transition, can be detected by Nioh but not by ASan, UBSan, or CFI. Moreover, these generic techniques monitor every address operation and variable change, resulting in substantial performance overhead. Additionally, the complex intertwining of control and data flows in these instrumentation-based techniques with the original program complicates the tracing and analysis of anomalies. In contrast, our work aims to detect abnormal state transitions in emulated devices and to focus on structures or variables within these devices that are susceptible to security issues, thereby reducing overhead. By conducting anomaly detection before the execution of emulated devices, we also aim to increase the flexibility of the detection process.

To the best of our knowledge, no existing method can automatically generate execution specifications for emulated devices without relying on manual summaries of written device specifications. Our method can automate this process for multiple emulated devices and produce a set of valid operation criteria, i.e., execution specifications. Using these specifications, we can ensure that devices remain in a legitimate operational state, thereby preventing potential attacks.

C. Challenges

We propose the construction of an execution specification that represents the legitimate operation of emulated devices.

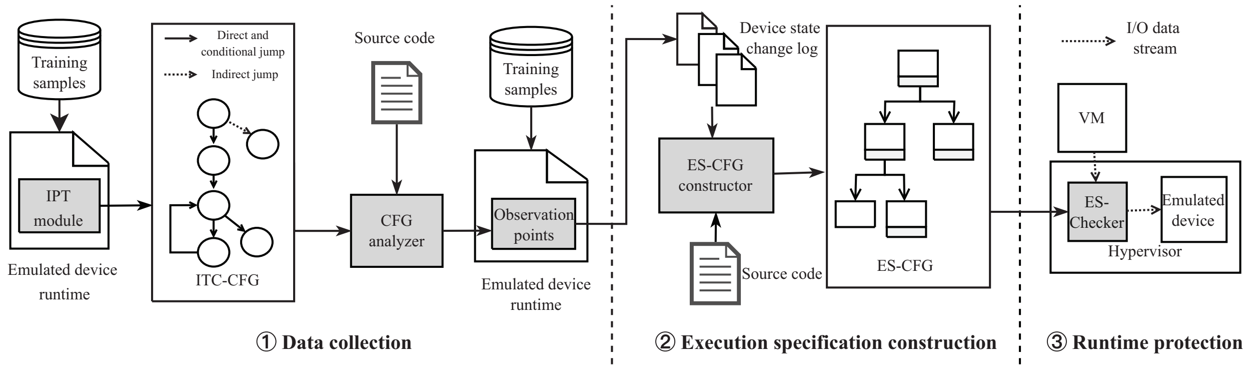


Fig. 1. The design overview of SEDSpec.

Our objective is to utilize this specification to predict and examine device execution for a given I/O interaction, thereby enabling anomaly detection. However, this approach also presents several challenges during its design and implementation.

Diversity of devices. Virtualization environments host a wide variety of emulated devices, each with distinct device specifications and implementation methods. Our approach should be flexible and applicable to commonly used emulated devices. To address it, we adopt an approach that builds execution specifications based on control flow and internal data changes, eliminating the reliance on device specifications and making our method applicable to most devices.

Complexity of control flow. The implementation of an emulated device is typically complex and code-intensive. Relying on the full control flow graph of an emulated device as its execution specification could incur significant overhead. Therefore, we require a method to trim down the control flow graph while preserving the validity of the extracted execution specification. To this end, we adopt an approach that formulates filtering rules to collect more streamlined control flow information, further simplifying the control flow during the construction of execution specifications. Meanwhile, we utilize data dependency recovery to ensure the effectiveness of execution specifications based on the reduced control flow.

Volume of device control structure data. Each emulated device is characterized by its unique control structure; for instance, the `FDCtrl` structure is associated with floppy disk controllers, and the `USBDevice` structure pertains to USB devices within the QEMU environment. Interactions through I/O operations induce modifications in the respective control structure of an emulated device. The execution specification should produce internal data changes that match the control structure data modification of the emulated device for a given I/O interaction. However, tracking and accounting for every data change is impractical due to the large amount of data in the control structure. To address this, we select variables from the device control structure that affect the control flow and filter them into execution specification internal data based on specified rules.

D. *Design Overview*

Figure 1 shows the overall workflow of SEDSpec, which integrates runtime data and source code of an emulated device to derive an execution specification for further runtime protection of the emulated device. The workflow consists of three phases.

The initial data collection phase collects the control flow and device state changes of the emulated device under benign training samples. First, the Intel Processor Trace (IPT) module [25] feeds the benign training samples to the emulated device, captures the relevant trace data, and constructs the Indirect Targets Connected Control Flow Graph (ITC-CFG) [26] from the data. Next, the CFG analyzer identifies and filters the variables in the device control structure that influence the control flow transitions, based on the source code and the ITC-CFG. These variables form the device state, which serves as the internal data structure of the execution specification, constructed in phase ②. Finally, observation points are placed inside the emulated device to record the device state changes. By inputting training samples to the emulated device again, we obtain a device state change log file that records both control flow and data state changes.

The second phase involves the construction of the execution specification construction. To achieve this, we use the Execution Specification Control Flow Graph (ES-CFG) to represent the execution specification. The ES-CFG constructor analyzes the logs in the device state change log file and locates the statements in the source code that are related to the device state parameters, which are the selected variables in the device state. These statements form the basic blocks and transition edges of the ES-CFG. The ES-CFG constructor also eliminates the redundant basic blocks and resolves the data dependencies of the device state parameters by combining data flow analysis with source code analysis.

The third phase is runtime protection. In this phase, we build a proxy system, ES-Checker, within the hypervisor to enhance the security of the emulated device with the execution specification. We design three check strategies based on the execution specification in ES-Checker. Each time an I/O interaction occurs, ES-Checker simulates the execution based on the execution specification under the I/O interaction and checks whether it violates any of the check strategies. Depending on the check strategy and the working mode (protection mode or enhancement mode), ES-Checker either halts the execution or alerts a warning if a violation is detected. Otherwise, it allows the emulated device to obtain the I/O data and operate on it.

IV. Data Collection

Data collection involves acquiring internal control flow and device state data change information from the emulated device runtime. To capture control flow information, we utilize the Intel Processor Trace (IPT) during the execution of the emulated device which enables us to generate the ITC-CFG. To further gather the change information of device state data, we select variables in the emulated device to build the device state structure and employ instrumentation to obtain the data changes on the device state parameters.

A. *ITC-CFG construction*

The control flow information is collected and processed in the IPT module of SEDSpec. The IPT is a feature of Intel CPU that traces the target executable process and generates various types of packets based on the executed jump instructions [25]. The IPT module within SEDSpec is tasked with managing and configuring IPT, parsing IPT packets, and generating the ITC-CFG by adopting the approach proposed by FlowGuard [26].

In the IPT module, we configure filtering rules for IPT such that only the control flow related to the target emulated device is tracked. Initially, the IPT module starts and stops the tracing at the locations where the I/O data stream enters and exits the target emulated device, respectively. Furthermore, to avoid contamination of the emulated device control flow by calls to shared library functions, the IPT module calculates the range of the emulated device code based on the memory layout and sets it as the range of addresses that can be collected by IPT. Lastly, to obtain a more streamlined emulated device control flow, tracing of kernel space control flow is disabled within the IPT module.

B. *Observation point setting*

The CFG analyzer within SEDSpec is tasked with examining the ITC-CFG and identifying variables that influence control flow transitions to construct the device state structure. Given that conditional and indirect jumps dictate the direction of the control flow, the CFG analyzer focuses on detecting such structures within the ITC-CFG and extracts variables influencing these structures. Subsequently, these variables are filtered based on two rules to yield the final set of variables, constituting the device state. Table I illustrates the selection of device state parameters based on the two rules.

Rule 1. Variables corresponding to the physical device registers. The code of the emulated device is derived from its physical counterpart. Given that certain device registers in the real device are crucial for controlling its behavior, we include the variables corresponding to the device registers in the device state parameters.

TABLE I
Selection of Device State Parameters

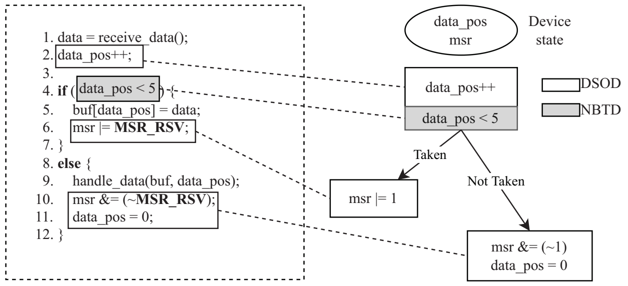
Variable types	Related Vul. or Exp.	Examples
Physical register related variables		`msr` (main status register), `dor` (digital output register), `tdr` (transfer control register).
Fixed-length buffer variables	Buffer overflow	`data_buf`, `fifo`.
Variables for counting and indexing buffer positions	Buffer overflow or integer overflow	`data_len`, `data_pos`.
Function pointer variables	Control flow hijack	`irq` (a function pointer handling device interrupt requests).

Rule 2. Variables associated with specific vulnerabilities. According to the indication of CVE [2], buffer overflow, out-of-bounds access, and integer overflow vulnerabilities account for over 50% of the QEMU device emulation vulnerabilities as of 2023. Consequently, we select the buffer variables, buffer length variables, and variables performing buffer indexing as part of the device state parameters for further runtime protection to detect device anomalies caused by exploiting such vulnerabilities. In addition, attackers often exploit function pointers in the device control structure to hijack the control flow in common QEMU device emulation attacks, which are also included as part of the device state parameters to detect control flow hijacking. By monitoring these parameters, SEDSpec identifies corresponding vulnerabilities or attacks.

Once the device state parameters are selected, the CFG analyzer situates observation points at locations that impact the direction of the control flows within the emulated device program to collect device state change data. These observation points are instrumented functions that record the device state parameter changes and the auxiliary information for identifying different block types during the execution specification construction phase. Thereby, the observation points are typically positioned at conditional and indirect jumps within the source code. Upon recompiling and executing the program, the device state change log can be obtained by inputting training samples.

C. *Training samples.*

Legitimate training samples are fed into the emulated device to construct the ITC-CFG and generate the device state change log. We source various samples from the web and QTest [27], and generate samples by running our test program under different settings and environments to simulate valid interactions with devices. For network card devices, we adjust parameters such as IP, MAC, gateway, interrupt mode, jumbo frame, and FlowControl. For storage devices, we configure different storage formats (e.g., FAT32, NTFS, EXT4), modes (e.g., RAID, LVM, JBOD), and parameters (e.g., partition size,



```
data = receive_data();
data_pos++;

if (data_pos < 5) {
    buf[data_pos] = data;
    msr |= MSR_RSV;
}
else {
    handle_data(buf, data_pos);
    msr &= (~MSR_RSV);
    data_pos = 0;
}
```

Fig. 2. An example for constructing basic blocks of ES-CFG.

cache size). Regarding the environment, we execute the test program on various systems with differing virtual machine configurations such as CPU cores and memory size.).

V. Execution Specification Construction

An execution specification abstracts the emulated device's internal execution information during legitimate I/O interactions, aiding in identifying potential anomalies. We define the execution specification as the control flow and device state changes that occur during legitimate operations of an emulated device. The structure of the execution specification is described using the control flow graph, termed ES-CFG.

A. ES-CFG Components

The ES-CFG is constructed based on the traditional CFG structure, and we use terms from traditional CFG to describe the components of ES-CFG. This section explains the role and functionality of each component within ES-CFG.

1) Device state: The device state is the inner data of ES-CFG. The selection process for device state parameters is detailed in Section IV-B. The device state, a separate data structure in the execution specification, does not affect the emulated device control structure. It is initialized with the values from the emulated device control structure upon booting of the emulated device. Subsequently, SEDSpec modifies the device state based solely on I/O data and ES-CFG.

2) Basic block: The basic block, a data structure encapsulating the semantics of C language code, serves as the fundamental building block of the ES-CFG. An ES-CFG basic block consists of two parts: Device State Operation Data (DSOD) and Next Block Transition Data (NBTD). DSOD comprises source code statements that manipulate the device state, while NBTD contains source code statements that facilitate transitions to subsequent basic blocks based on device state parameters. Not all basic blocks contain NBTD; those without it invariably transition to their immediate successor block. Figure 2 illustrates how DSOD and NBTD form in an ES-CFG's basic blocks derived from a device's source code when `data_pos` and `msr` are selected as device state parameters.

An ES-CFG consists of various types of basic blocks, categorized based on their position and functionality. The entry block is the first basic block that SEDSpec accesses during I/O interactions, parsing the target address/port of the I/O request as a parameter for subsequent execution. Conversely, the exit block is the last basic block of the ES-CFG, signaling the end of an I/O round. The conditional basic block is involved in conditional branching based on the device state parameters. The command decision block identifies the current device command and the accessible blocks under that command. The command end block determines whether the current command execution has concluded. These different types of blocks are constructed by parsing the block type information in the device state change log.

After an ES-CFG is deployed to an emulated device, SEDSpec initiates the traversal from the entry block of the ES-CFG for each I/O interaction round. The entry block captures and logs relevant I/O information, including the port or memory address, the data transmitted, and the I/O operation direction. Then, the process transitions to the subsequent basic blocks, which update the device state according to their DSOD and navigate to the following block using their NBTD and the updated device state parameters. Upon reaching a command decision block, SEDSpec identifies the current command type and determines accessible blocks under that command. This allows SEDSpec to derive a subgraph of the ES-CFG comprising the relevant blocks. SEDSpec then runs relevant check strategies, detailed in Section VI-A, on the subgraph for anomaly detection. If no anomaly is detected by the time SEDSpec reaches the exit block, it outputs the final device state which serves as the initial device state for the next round of I/O interactions.

B. Construction of ES-CFG

1) Construction method.: The ES-CFG constructor builds the ES-CFG utilizing both the device state change log and the source code. The device state change log encompasses control flow and device state information. Algorithm 1 presents the pseudocode for the initial construction of ES-CFG using the device state change log file and the emulated device source code.

The pseudocode illustrates how the ES-CFG constructor builds an ES-CFG. Each log in the device state change log file contains the complete control flow data, device state change data, and auxiliary information (such as device command information, and correspondence between control flow and source code) of an emulated device under a specific test case. The ES-CFG constructor, for each log, restores the control flow structure (line 2) and constructs the basic blocks and transition edges of the ES-CFG from the control flow data (line 4).

The process of constructing a basic block entails extracting statements from the source code that induce changes in the device state, leading to the generation of DSOD. To construct a transition edge associated with a specific basic block, the subsequent basic block is identified from the control flow data. Concurrently, statements from the source code that trigger this transition are identified and used to generate NBTD.

Algorithm 1: Construction Algorithm of ES-CFG

```
   Input: Device state change logs: $ds\_logs$; The
          emulated device source code: $ed\_sc$.
   Output: The emulated device execution specification:
           $es\_cfg$; The emulated device command
           access control table: $cmd\_act$.
1  foreach $log$ in $ds\_logs$ do
2      $cfg \leftarrow$ RestoreRuntimeCFG(log) ;
3      $access\_vec \leftarrow$ InvalidVec();
4      foreach $basic\_block$ in $cfg$ do
5          if IsCondBlock($basic\_block$) then
6              ESCstrctBBWithTNTEdge($basic\_block$,
                $ed\_sc$);
7              UpdateAV($access\_vec$);
8          end
9          else
10             ESCstrctBB($basic\_block$, $ed\_sc$);
11             UpdateAV($access\_vec$);
12             if IsExitBlock($basic\_block$) then break;
13             ESCstrctEdge();
14             if IsCmdDecBlock($basic\_block$) then
15                 $cmd \leftarrow$ DcdCmd($basic\_block$, $log$);
16                 $access\_vec \leftarrow$ GetAV($cmd\_act$, $cmd$);
17             end
18             else if IsCmdEndBlock($basic\_block$) then
19                 UpdateCAT($cmd\_act$, $cmd$,
                    $access\_vec$);
20                 $access\_vec \leftarrow$ InvalidVec();
21             end
22         end
23     end
24 end
```

For an indirect jump in the control flow data, the ES-CFG constructor builds the basic block (line 10) and the transition edge (line 13) directly. For a conditional jump in the control flow data (line 5), the ES-CFG constructor determines whether it is taken or not to build the transition edge (line 6).

If the log carries data about the device command(line 14), the ES-CFG constructor identifies the device command type(line 15) and uses it as the key value of the command decision block's mapping table structure (line 16). In the subsequent basic block construction process, a bitmap is utilized to record the accessibility of the basic blocks under the given device command (line 7). This bitmap is ultimately stored in the mapping table entry corresponding to this command (line 19).

By following the pseudo code procedure, we can obtain a preliminary ES-CFG for an emulated device. However, this ES-CFG may contain redundant basic blocks and unresolved internal data dependency issues. Therefore, additional reduction of the control flow and recovery of the data dependencies need to be handled to build the final applicable ES-CFG.

C. Control Flow Reduction

The control flow reduction has two parts: first, excluding noisy control flow data in the data collection phase, as explained in Section IV-A; and second, deleting and merging some redundant basic blocks in the ES-CFG generation, which is presented below.

After constructing the preliminary ES-CFG, the ES-CFG constructor merges some blocks based on the internal structure of ES-CFG. The same ES-CFG basic block may be reached by both the taken and not taken branches of a conditional basic block because the construction process ignores the source code that does not affect the device state. In this case, we merge the two basic blocks and remove the NBTD of the previous ES-CFG basic block.

D. Data Dependency Recovery

The ES-CFG primarily captures alterations in device state parameters. However, control flow transitions may depend on variables other than the device state parameters. Such data dependency related variables should be properly analyzed.

The ES-CFG constructor performs a data flow analysis and selects a solution based on the analysis result. Utilizing *angr* [28], the ES-CFG constructor obtains a data flow graph according to the data dependency relations of the target variable. If the variable can be computed from the device state parameters, it is replaced by that computation in the related NBTD. Otherwise, the ES-CFG constructor inserts a `sync point` function into the code of the emulated device. At runtime, the issue is addressed by first using SEDSpec execution, then running the emulated device, and synchronizing variable values from the sync point function, before resuming SEDSpec execution.

It is noteworthy that the variables introduced by sync points may not be secure; however, this concern falls outside the scope of our security focus. These variables are unrelated to the device state and I/O data, indicating that they are not associated with the device control structure data, which is prone to security issues that we have identified. Although we cannot guarantee the trustworthiness of these variables, they are distant from the data considered hazardous within our area of concern.

VI. Runtime Protection

Runtime protection aims to monitor and enhance the security of the emulated device based on the execution specification. SEDSpec uses the dynamically collected data from the legitimate execution of the target program to construct the security constraints of the emulated device execution. This is achieved by using the ES-Checker proxy system in Figure 1 ③. ES-Checker is deployed with three check strategies. During each I/O interaction between the guest machine and the emulated device, ES-Checker performs device execution simulation in accordance with ES-CFG and employs the relevant strategies for anomaly detection. If any anomaly is detected during ES-Checker's execution, the system will either halt or alert a warning to indicate the abnormal execution, depending on the working mode and the violated strategies. If ES-Checker finds no violation of the strategies, it guarantees the real execution of the emulated device under the I/O interaction following the check strategies.

A. Check Strategies

Check strategies constitute a set of inspection rules that leverage the ES-CFG structure to perform targeted checks at specific points during the traversal of the ES-Checker. Specifically, these strategies can be employed to validate the legitimacy of the device state modifications made by the DSOD as the ES-Checker navigates into and out of each ES-CFG basic block. Additionally, at jump points governed by the NBTD, both indirect and branch jump checks can be conducted to verify the validity of the jump. In light of these considerations, we have formulated three check strategies within the ES-Checker.

Parameter check strategy is tasked with verifying the legitimacy of alterations to device state parameters, based on DSOD validation. This strategy focuses on two types of abnormal execution: integer overflow and buffer overflow.

In the case of integer overflow, if the value of an integer parameter of the device state exceeds the maximum or minimum value of its type, the ES-Checker identifies this as an integer overflow. The specific implementation method is inspired by UBSan's [29] approach, using LLVM IR metadata to denote the parameter type and employing the LLVM IR API to ascertain this type. Then, the ES-Checker determines whether an overflow has occurred based on the parameter type and changes in relevant bits in the flag register at runtime.

In terms of buffer overflow, when a buffer parameter is included in the device state, its size is recorded by ES-Checker. If a device state index parameter is used for reading or writing into this buffer according to some ES-CFG basic block's DSOD, ES-Checker verifies whether this index falls outside the buffer's range. If it does exceed this range, a buffer overflow is identified by ES-Checker.

Indirect jump check strategy examines cases where an indirect jump is utilized to go to an unauthorized address. During the ES-CFG construction process, SEDSpec retains information about indirect jump types and the mapping between the indirect jumps' target addresses and the ES-CFG's basic block. This information is implicitly utilized by the ES-Checker to implement the indirect jump check strategy. The strategy focuses on indirect jumps initiated by callbacks to a function pointer parameter within the device state, examining whether the target of the jump can correspond to a legitimate ES-CFG basic block. This strategy, however, does not consider indirect jumps propelled by 'ret' and other function pointer callbacks.

Conditional jump check strategy is performed during transitions from one ES-CFG basic block to another based on the NBTD of the preceding one. During actual executions of emulated devices, certain branches are never traversed under normal device operations. In contrast, the control flows induced by attacks on emulated devices frequently encompass numerous corner cases. A basic block associated with NBTD leads to only one subsequent basic block in the ES-CFG, indicating that either a taken or not-taken branch is absent. Upon encountering such an untraversed branch at runtime, the ES-Checker identifies this as an anomaly.

B. Working Modes.

We design two working modes for ES-Checker: protection mode and enhancement mode, each catering to different levels of false positive tolerance in various scenarios. Protection mode is for scenarios with higher security requirements and false positive tolerance. In this mode, ES-Checker halts the emulated device and virtual machine execution when it detects any anomaly. Anomalies detected by the parameter check strategy are directly related to vulnerability exploitation and do not cause false positives. However, anomalies detected by other strategies may be false positives due to corner cases that are not covered by the training samples. To address this, we provide enhancement mode for scenarios with higher availability requirements and lower false positive tolerance for emulated devices. In this mode, ES-Checker halts the execution of the device and virtual machine only upon detecting anomalies by the parameter check strategy. Concurrently, the remaining two check strategies, upon identifying anomalies, alert warnings without interrupting execution.

VII. Experiments and Evaluations

A. Experiment Setup

We construct execution specifications and deploy SEDSpec on an x86 architecture server with Intel PT support. We set up a full-fledged virtualization environment using KVM and QEMU. The experimental environment consists of a host machine and a guest virtual machine. The host machine runs Ubuntu 18.04 LTS on an Intel® Core™ i9-10900X CPU (10 cores, 3.70GHz, and 20MB cache) and 64GB RAM. The guest virtual machine runs Ubuntu 18.04 LTS with 4 vCPUs and 8GB RAM. During experiments, ES-Checker is the only SEDSpec component that affects the security and performance of emulated devices.

B. Security Evaluation

We evaluate the effectiveness of SEDSpec by generating and deploying execution specifications for five different devices: FDC, USB EHCI, PCNet, SDHCI, and SCSI. These devices are emulated by QEMU and represent various types of device controllers for managing floppy disks, USB devices, network adapters, SD cards, and storage devices, respectively. Our selection of these five devices is motivated by three major considerations. First, they are common and representative storage devices and network devices that can demonstrate the generality of SEDSpec. Second, they have a large number and variety of vulnerabilities, which enable us to select diverse vulnerabilities for case studies to validate the effectiveness of SEDSpec. Last, these devices include all the devices used in Nioh's experiments, allowing for comparisons with Nioh.

TABLE II
False Positives Over Time

Device	10 hours	20 hours	30 hours
FDC	1	2	5
USB EHCI	3	3	3
PCNet	1	5	6
SDHCI	4	7	7
SCSI	1	3	4

To evaluate the effectiveness of SEDSpec, we carry out two distinct types of experiments. First, we test whether the execution specifications generated by SEDSpec can maintain the normal operation of devices under both long-term and multi-dimensional device interactions. Second, we perform case studies where the vulnerability exploitations are simulated to breach the execution specifications. This allows us to demonstrate SEDSpec's effectiveness in defending against such exploitations. Additionally, we analyze and explain the underlying mechanism of how the relevant check strategy can identify vulnerability exploitation.

1) Ensuring Normal Operation: We classify the devices with execution specifications into two categories: storage devices and network devices. The former includes FDC, USB EHCI, SDHCI, and SCSI, which perform read and write operations on files. The latter includes PCNet, which performs read and write operations on network packets.

To evaluate the functionality of these devices, we devise a test program in the guest OS that interacts with the emulated device using device drivers. This test program is utilized to conduct a long-term and multi-dimensional interaction with the device to ascertain whether the execution specification generated by SEDSpec can ensure the normal operation of the device.

We set three different interaction modes: sequential, random, and random with delay. In sequential mode, the test program follows a predetermined order of read and write operations to interact with the device. In the random mode, the test program randomly chooses read and write operations to interact with the device. In the random with delay mode, the test program randomly chooses read and write operations to interact with the device and introduces a random delay between each operation. The volume of data in each test case varies randomly, ranging from thousands to tens of thousands of I/O sequences. Utilizing test cases that encompass a substantial volume of data enables a more nuanced identification of the false positive rate. Conversely, if the test samples were limited to only a single or a few I/O sequences, albeit with a large number of such samples, the resulting false positives, while potentially numerous, would seem inconsequential when compared against an extensive baseline. This scenario would lead to an apparent false positive rate for all devices nearing zero. We apply each interaction mode to each device for 10 hours, 20 hours, and 30 hours.

We manually analyze the samples that SEDSpec identifies as abnormal and determine whether they are false positives generated by SEDSpec. Furthermore, we quantified the incidence of these false positives across various time intervals, as detailed in Table II. The calculation of the false positive rate is shown in the following formula.

$$FPR = \frac{N_L}{N_T}$$

In this formula, FPR is the false positive rate, N_L is the number of legal test cases reported as abnormal by SEDSpec, and N_T is the total number of test cases for the device. We test and calculate the false positive rates for each emulated device. Table III shows the false positive rate of SEDSpec. The false positive rates of SEDSpec for FDC, USB EHCI, PCNET, SDHCI, and SCSI are 0.14%, 0.10%, 0.11%, 0.09% and 0.17%, respectively.

In our assessment of SEDSpec, we focused on the coverage metric pertaining to legitimate behaviors across various devices. SEDSpec aims to encompass a comprehensive range of legitimate device behaviors, with effective coverage being determined by the ratio of code paths covered in a given device's codebase relative to the totality of paths representing all legitimate behaviors. Empirical evidence suggests that while it is time-consuming for fuzzing techniques to cover and reach exceptional control flows, they are notably efficient at accessing the most common control flows within a brief period. Our experiments revealed that the coverage rates for different devices began to converge approximately after one hour of testing. Consequently, we employ fuzzing to approximate the coverage path of legitimate behavior by running it on a device for one hour, thereby calculating the effective coverage rate of each device. The results from this process are summarized as shown in Table III: which presents the effective coverage rates for the devices evaluated as follows: 95.9%, 97.3%, 96.2%, 93.5%, and 93.8% respectively.

2) Preventing Execution Specification Violations: To evaluate the security enhancement of emulated devices by SEDSpec, we use one-day vulnerabilities from the CVE [2] list to construct the I/O data stream and conduct case studies. We use different versions of QEMU in the experiment, depending on the CVE vulnerabilities that affect the emulated device. The I/O data stream contains proof-of-concepts for at least one vulnerability targeting the emulated device. We measure the accuracy of SEDSpec in detecting anomalies by comparing its execution outcome with the ground truth. We configure SEDSpec to operate in protection mode, which halts QEMU execution upon detecting anomalies in emulated devices. To show the effectiveness of different check strategies, we activate only one check strategy for each experiment. It's important to note that a single vulnerability could induce multiple forms of anomalies, which may in turn trigger different check strategies.

We conducted case studies on 8 vulnerabilities and successfully prevented their exploitation in 5 devices, as shown in Table III. We selected these vulnerabilities for our case studies for two primary reasons. Firstly, it enables a comparison with typical works such as Nioh, which tests the same cases. Secondly, these cases exhibit diverse causes and exploitation

TABLE III
THE MAIN RESULT OF SEDSPEC EXPERIMENTS.

Device	CVE ID	QEMU Version	Check Strategies			False Positive Rate	Effective Coverage
			Parameter Check	Indirect Jump Check	Conditional Jump Check		
FDC	CVE-2015-3456	v2.3.0	√		√	0.14%	95.9%
USB EHCI	CVE-2020-14364	v5.1.0	√	√		0.10%	97.3%
PCNet	CVE-2015-7504	v2.4.0		√		0.11%	96.2%
	CVE-2015-7512	v2.4.0	√	√			
	CVE-2016-7909	v2.6.0			√		
SDHCI	CVE-2021-3409	v5.2.0	√			0.09%	93.5%
SCSI	CVE-2015-5158	v2.4.0			√	0.17%	93.8%
	CVE-2016-4439	v2.6.0			√		

processes, thereby testing the effectiveness of SEDSpec's check strategies.

Nioh experiment tested 5 vulnerabilities (CVE-2015-3456, CVE-2015-5158, CVE-2016-4439, CVE-2016-7909, CVE-2016-1568), and SEDSpec detected all except CVE-2016-1568, which results from the Use-After-Free (UAF) causing by missing data initialization in certain cases. The unpatched code lacks the initialization code, so the execution specification does not include the relevant state transition, raising a mis-detection by SEDSpec. The other four vulnerabilities stem from input data that disrupts the device operation, causing an internal anomaly that violates the execution specification involved in the conditional jump check strategy.

CVE-2015-3456, aka Venom, is a vulnerability collected in the FDC, which can be detected by both conditional jump check and parameter check. The vulnerability stems from the fact that the `data_pos` variable in data structure `FDCtrl` is incremented indefinitely without being reset, resulting in an out-of-bounds access to the buffer pointed by the `fifo` variable. The parameter check strategy prevents this issue by checking whether the `data_pos` variable exceeds the size of the `fifo` buffer.

SEDSpec successfully detected the exploitation of CVE-2020-14364, a vulnerability that affects the emulated USB EHCI. The exploitation for CVE-2020-14364 creates a `USBDevice` structure with a value for `setup_len` that exceeds the size of `data_buf`. This overwrites the variables behind `data_buf` and ultimately alters the control flow. It is worth noting that the exploitation involves two instances of out-of-bounds access. The first instance occurs when the `setup_len` is larger than the `setup_buf`. The second instance occurs when the overwritten variable `setup_index` is set to a negative integer. The variables above the `data_buf` can be overwritten because the `data_buf` is indexed by the `setup_index`. The parameter check strategy detected both instances in the experiment. Moreover, the indirect jump check strategy revealed the vulnerability when the handler pointer in the fake `irq` was invoked as a function.

We tested SEDSpec on the PCNet using the PoCs of CVE-2015-7504 and CVE-2015-7512, both of which resulted from out-of-bounds buffer access in the `PCNetState` structure. In CVE-2015-7504, the out-of-bounds access to the buffer relies on a temporary pointer variable unrelated to device state parameters. Therefore, the parameter check approach fails to detect the anomaly caused by the vulnerability. In the exploitation for CVE-2015-7504, the `irq` variable adjacent to the buffer is overwritten and tampered with by writing 4 bytes beyond the allowed range. The indirect jump check strategy discovered the vulnerability before the handler pointer variable is invoked by another function. In CVE-2015-7512, when the variable `xmit_pos` in data structure `PCNetState` is larger than 4092 in the `pcnet_receive_function`, the buffer would be written as out-of-bounds. The parameter check revealed the vulnerability when an out-of-bounds writing occurred since `xmit_pos` is linked to the buffer index variable. The attack for CVE-2015-7512 is identical to that of CVE-2015-7504, which was also discovered by the indirect check strategy.

CVE-2021-3409 is a vulnerability of the SDHCI that occurs when the `blksize` is changed during an ongoing data transformation, which also causes out-of-bounds access issues. The exploitation changes the value of the `blksize` to be less than the `data_count`, triggering the vulnerability. The value of the expression (`blksize - data_count`) raises an unsigned integer overflow, which is successfully detected by the parameter check strategy.

C. Performance Evaluations

To evaluate the performance of SEDSpec, we conduct the comparison experiment for devices of both storage and network. For the storage devices, USB EHCI, SDHCI, and SCSI are designed as the interfaces for USB storage, SD card, and disk respectively. In practice, the performance is evaluated by measuring the additional overhead of modifying the interface. We use *iozone* [30], a file system benchmarking tool, to measure the read and write throughput and latency of storage devices. For the network devices, PCNet serving as a typical device is treated as the objective of measurement, which can be measured by the bandwidth (obtained by *iperf* [31]) and latency (obtained by *ping*).

In the storage device benchmark, we normalize the results according to the ranges of throughput and latency. The *iozone* tool offers different sizes of blocks for measuring the read and write throughput and latency of storage devices. We normalize the throughput and latency of the original emulated device to 1 and then compute the normalized value of the updated

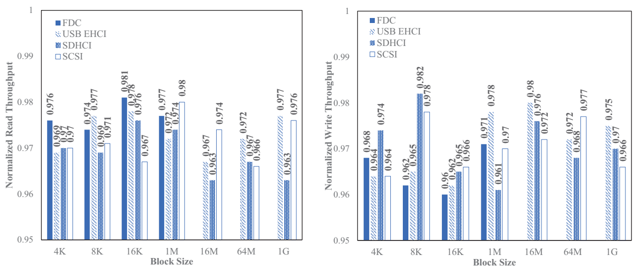


Fig. 3. Normalized throughput results of storage devices.

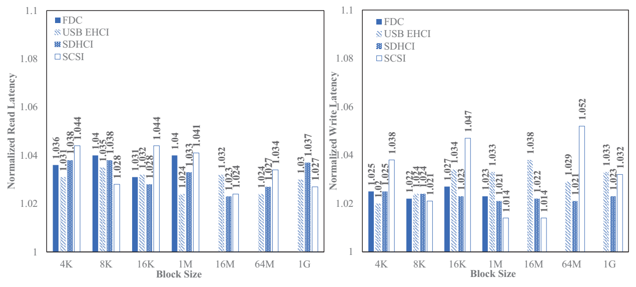


Fig. 4. Normalized latency results of storage devices.

emulated device based on the *iozone* results. Generally, due to the extra overhead brought by SEDSpec, the throughput decreases, and the latency increases.

The normalized throughput and latency are illustrated in Figures 3 and 4, respectively. It is crucial to note that the FDC's capacity is only 2.88 MB, so we can only evaluate its performance using blocks smaller than its limitation. Based on the experimental results, we can conclude that SEDSpec incurs a performance loss of both throughput and latency with less than 5 percent.

For the PCNet benchmark, we set up the user-mode network environment in QEMU and configured the host forwarding port for *iperf* communication before booting. After that, we executed *iperf* in either server mode or client mode to communicate with the *iperf* running on the server, where the bandwidth for both upstream and downstream is obtained. Such bandwidth is also measured in the TCP and UDP communication. In addition, we assessed the network latency by pinging the user-mode IP address, which is transformed from the host IP address using Network Address Translation (NAT).

The results of the PCNet bandwidth benchmark are shown in Figure 5. SEDSpec reduces bandwidth by 6.9%, 7.3%, 5.7%, and 6.6% for TCP upstream, TCP downstream, UDP upstream, and UDP downstream, respectively. During 100 *ping* times of tests, the average PCNet latency is 0.65 milliseconds, while that of PCNet with SEDSpec is 0.71 milliseconds, resulting in only a 9.2% increase in overhead.

VIII. Discussion

Anomaly Defence. SEDSpec halts the virtual machine or issues warnings when detecting any anomalies in the emulated devices. The main objective of SEDSpec is to detect the anomalies rather than to defend them, so it uses a straightforward processing method in the current stage. In future work, several avenues can be employed to handle the anomalies. For example, using rollback to restore the virtual machine state to a previous point before the exploitation, directly terminate the anomalous process in the virtual machine, and classify the alert levels based on different check strategies.

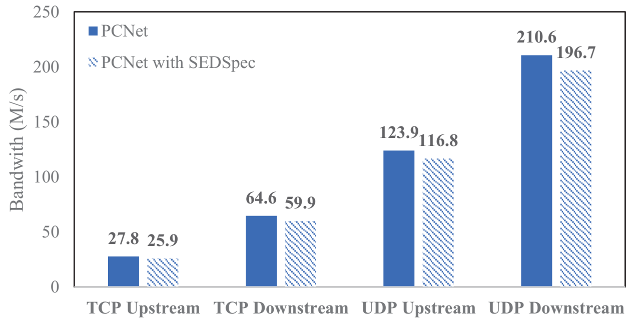


Fig. 5. PCNet bandwidth benchmark

False Positive and Remedy. Due to limitations in acquiring an exhaustive collection of test samples, the generation of false positives by SEDSpec is an inevitable outcome. Our analysis of these instances indicates that they are exclusively linked to exceedingly rare device commands. Such commands fall outside the realm of standard operational procedures and necessitate the manual input of specific commands alongside particular I/O sequences for activation. As a result, the occurrence of false positives has a negligible impact on SEDSpec's practical functionality and its operational efficacy. To further mitigate the incidence of false positives, an approach involves distributing SEDSpec among device developers and testers. This strategy enables the utilization of extensive test cases to formulate precise execution specifications, thereby enhancing SEDSpec's accuracy and reliability.

Limitation. SEDSpec's capability is confined to detecting anomalies associated with the device state during runtime protection. Since the device state is generated upon the device control structure, the variables out of the control structure, e.g., temporary variables and global variables beyond SEDSpec's scope for checking and protection. Despite this limitation, SEDSpec's external inspection mechanism for emulated devices ensures its high compatibility with a range of security mechanisms, such as canary, ASLR, CFI, AppArmor, and seccomp. This compatibility facilitates SEDSpec's seamless integration with these security mechanisms, thereby mitigating its inherent limitations and enhancing overall security.

IX. Conclusion

This paper introduces a novel approach to identifying anomalies caused by vulnerabilities in emulated devices along with a newly developed prototype system named SEDSpec. Unlike existing methods that rely on predefined device specifications, our approach is devised upon execution information of emulated devices under benign I/O interactions through process tracing and program instrumentation. Correspondingly, specific execution specification is built by capturing the device state and control flow data under different I/O interactions. The execution specification supports us in predicting the emulated device behavior and state changes for a given I/O data stream. We also devise three strategies to detect violations induced by vulnerability exploitations and misuse based on the execution specification. Furthermore, we implement the ES-Checker module to monitor the emulated device execution. We evaluate our approach and system on various versions of QEMU with five constructed execution specifications. The experimental results show that SEDSpec can effectively identify anomalies caused by certain emulated device vulnerabilities with a low runtime overhead.

Acknowledgment

This work was partially supported by the Program of Key Laboratory of Network Assessment Technology, the Chinese Academy of Sciences, the Program of Beijing Key Laboratory of Network Security and Protection Technology, the National Key Research and Development Program of China (No.2021YFB2910109), and the National Natural Science Foundation of China (NO. 62202465). The content of this material, including any opinions, findings, conclusions, or recommendations, solely reflects the views of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of any funding agencies.

References

[1] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*. USENIX, 2005, pp. 41–46.

[2] A. Kumar and I. Sharma, "Common vulnerabilities and exposures." https://cve.mitre.org/, 2022.

[3] K. Kc and X. Gu, "ELT: efficient log-based troubleshooting system for cloud computing infrastructures," in *30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011), Madrid, Spain, October 4-7, 2011*. IEEE Computer Society, 2011, pp. 11–20.

[4] J. Nikolai and Y. Wang, "Hypervisor-based cloud intrusion detection system," in *2014 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2014, pp. 989–993.

[5] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, T. Conte and Y. Zhou, Eds. ACM, 2016, pp. 489–502.

[6] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li, "Deconstructing xen," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[7] D. Cotroneo, L. D. Simone, F. Fucci, and R. Natella, "Moio: Run-time monitoring for I/O protocol violations in storage device drivers," in *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. IEEE Computer Society, 2015, pp. 472–483.

[8] X. Chen, J. Chen, D. Zhao, and X. Jin, "Anomaly detection based on IO sequences in a virtual machine with the Markov mode," *Journal of Tsinghua University (Science and Technology)*, vol. 58, no. 4, pp. 395–401, 2018.

[9] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai *et al.*, "Bitvisor: a thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2009, pp. 121–130.

[10] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 401–412.

[11] J. Ogasawara and K. Kono, "Nioh: Hardening the hypervisor by filtering illegal I/O requests to virtual devices," in *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*. ACM, 2017, pp. 542–552.

[12] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," in *Proceedings of the Fourth Symposium on Operating System Principles, SOSP 1973, Thomas J. Watson, Research Center, Yorktown Heights, New York, USA, October 15-17, 1973*, H. Schorr, A. J. Perlis, P. Weiner, and W. D. Frazer, Eds. ACM, 1973, p. 121.

[13] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, no. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[14] C. Myung, G. Lee, and B. Lee, "Mundofuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 1257–1274.

[15] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "HYPER-CUBE: high-dimensional hypervisor fuzzing," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[16] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "Morphuzz: Bending (input) space to fuzz virtual devices," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 1221–1238.

[17] 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 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 2597–2614.

[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 *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 2197–2213.

[19] 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*, ser. Lecture Notes in Computer Science, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds., vol. 10453. Springer, 2017, pp. 3–25.

[20] A. Nguyen, H. Raj, S. K. Rayanchu, S. Saroiu, and A. Wolman, "Delusional boot: securing hypervisors without massive re-engineering," in *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, P. Felber, F. Bellosa, and H. Bos, Eds. ACM, 2012, pp. 141–154.

[21] AddressSanitizer — Clang 19.0.0git documentation. [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html

[22] UndefinedBehaviorSanitizer — Clang 19.0.0git documentation. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[23] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[24] NVD - CVE-2016-7909. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016-7909

[25] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3C: System programming Guide, Part*, vol. 3, no. 36, 2016.

[26] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, “Transparent and efficient CFI enforcement with intel processor trace,” in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. IEEE Computer Society, 2017, pp. 529–540.
[27] “Qtest device emulation testing framework,” https://qemu.readthedocs.io/en/latest/devel/qtest.html, 2022.
[28] Angr. [Online]. Available: https://angr.io/
[29] T. C. Team, “UndefinedBehaviorSanitizer,” https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2022.
[30] W. D. Norcott, “Iozone filesystem benchmark,” https://www.iozone.org/, Jan. 2016.
[31] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu, “iperf - the ultimate speed test tool for tcp, udp and sctp,” https://iperf.fr/, 2023.