Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 6698311, 14 pages
https://doi.org/10.1155/2021/6698311

Research Article

Framework for State-Aware Virtual Hardware Fuzzing

Hang Xu,[1] **Ganyu Qin**,[2] **Junhu Zhu**,[1] **Zimian Liu**,[1] **and Zhiqiang Liu**[2]

[1]*School of Cyber Science and Engineering, Information Engineering University, Zhengzhou 450001, China*
[2]*School of Information Engineering, Zhengzhou University, Zhengzhou 450001, China*

Correspondence should be addressed to Junhu Zhu; zhujunhu74@163.com

Received 22 December 2020; Revised 16 January 2021; Accepted 31 March 2021; Published 21 May 2021

Academic Editor: Keping Yu

Coverage-based greybox fuzzing has strong capabilities in discovering virtualization software vulnerabilities. Efficiency is one of the most important indicators while evaluating greybox fuzzing. However, the interference of virtual hardware state conditions on testcase evaluation severely impairs the efficiency of greybox fuzzing. In order to reduce the interference of virtual hardware state conditions and increase the efficiency of fuzzing, we propose a state-based virtual hardware fuzzing framework, named SAVHF (State-Aware Virtual Hardware Fuzzing). In this framework, a source-to-source instrumentation method based on the abstract syntax tree is proposed to detect the state condition of virtual hardware. Based on the source-to-source instrumentation, we afterwards propose a state-based fuzzing strategy to adapt to the state conditions of virtual hardware. We realize the prototype system of SAVHF and use it to evaluate 17 popular virtual hardware of Qemu and find 16 bugs with 1 CVE (Common Vulnerabilities and Exposures) number assigned. Evaluation results demonstrate that the proposed SAVHF framework covers an average of more than 61% of virtual hardware code branches in the 18 hours testing and can improve the average code coverage by 11.04% compared with the path-based fuzzing strategy.

1. Introduction

1.1. Background. Virtualization technology is widely used in cloud computing, software testing, daily office, and many other scenarios. Virtualization technology can provide users with with convenient service of privilege isolation protection. It is also an effective solution to reduce the cost of configuring multiple physical computing instances [1]. In order to enable the virtualization guest machine to access essential hardware (network cards, graphics cards, sound cards, etc.), the virtualization platform adopts methods of accessing the physical hardware (which needs real hardware connected to host machine) and making some full virtualized hardware [2]. The software-implemented virtual hardware runs at the same privilege level as hypervisor, which is convenient for guest user to access the virtual hardware. However, attackers can execute the exploitation program in guest machine to trigger the vulnerabilities hidden in virtual hardware and gain the same privilege as hypervisor, which may lead to the virtual machine escape [3, 4]. Attackers also send vulnerability exploit programs to victims through methods such as spam to achieve the purpose of the attack. Guo et al. use the collaborative neural network to detect robust spammer [5]. Blocking the spread of exploit programs can effectively reduce the possibility of attacks, but software vulnerabilities with independent propagation capabilities still pose a major threat to network security.

In order to remedy this, fuzzing technology has become the most popular method for software vulnerability detection, which is mainly due to its high efficiency in detecting vulnerabilities [6]. Using this technology, researchers have discovered many software vulnerabilities. The fuzzing technology was first proposed in the 1990s. The main research direction in the early stage was focused on black-box fuzzing, and the current main research directions are greybox fuzzing and white-box fuzzing. Intensive studies have been done since AFL (American Fuzzy Lop) was first proposed in 2013 [7]. At present, most endeavors on fuzzing technology refocused on efficiency performance improvement and its adaptability analysis.

In terms of performance improvement, the main research key points are testcase variation methods, testcase

set minimization methods, instrumentation methods, testcase evaluation methods, etc. The representative studies in this aspect include Driller (an augmenting fuzzing method through selective symbolic execution) [8], AFLGo (a directed greybox fuzzing method) [9], and AFLFast (a coverage-based greybox fuzzing method as Markov chain) [10]. Driller is a greybox fuzzing framework that combines symbolic execution technology. It utilizes the symbolic execution technique to deal with conditional branches that are difficult to enter for mutated testcases generated by fuzz testing. AFLGo uses LLVM (Low Level Virtual Machine) to generate the call graph and control flow graph of the test target and proposed the fuzzing strategy based on the simulated annealing algorithm. AFLGo can directionally test the code related to the specified code block. AFLFast proposes a fuzzing strategy based on the Markov chain model, which intelligently controls the number of mutation of testcases in the corpus, thereby giving more opportunities to low-frequency paths. Aschermann et al. use the processing state of the program to guide the testcase mutation of the fuzzing [11].

In terms of adaptability, the key points of the main research are kernel fuzzing, browser fuzzing, virtualization platform fuzzing, etc. Representative studies include kAFL (a fuzzing method based on hardware-assisted feedback for OS kernels) [12], syzkaller (an unsupervised coverage-guided kernel fuzzer) [13], and fuzzilli (a coverage guided fuzzer for dynamic language interpreters based on a custom intermediate language) [14]. KAFL utilizes KVM (Kernel-based Virtual Machine) and Intel-PT (Intel Processor Trace) technologies to fuzz kernel with the function of mounting image as an interaction point, and it also developes a decoder that is more effective than the official PT decoder released by Intel. Syzkaller uses a declarative language to describe the kernel system calls and uses the system calls as the interaction point to fuzz the kernel. Fuzzilli is a fuzzer for JavaScript interpreter engine of browsers. It generates testcases through grammar templates and uses intermediate representation language for mutation. It uses code branch feedback to improve the code coverage of fuzzing.

1.2. Related Works. In literature, there are some antecedent studies on fuzzing technology for virtual hardware vulnerability discovery. For instance, some researchers proposed adaptive methods for applying traditional fuzzing frameworks to virtual hardware. Tang et al. proposed a framework for adapting AFL to devices in a virtualization platform at the Blackhat Conference in 2016 [15]. This framework is an early concept of virtual hardware fuzzing and provides a rough plan for virtual hardware fuzzing. However, the state information of the virtual hardware is neglected, resulting in inaccuracy in testcase evaluation [15].

The state condition of the virtual hardware is the change condition of the processing logic determined by its control register. To this end, VDF (Virtual Device Fuzzing) [16] is proposed to use initialization testcase sequence replay to eliminate the state conditions of virtual hardware. VDF implementes a fuzzing framework for virtual devices, which uses recording and playback mechanisms to try to eliminate the state conditions of virtual hardware. Before each testcase is input, it resets the state of the virtual hardware to the initial state through the playback mechanism. VDF does not use a guest to interact with virtual hardware. It uses the memory access interfaces provided by Qemu and implements a method of input/output (I/O) with virtual hardware based on accelerator qtest. Limited by the memory access method, VDF can only fuzz the MMIO BAR (Memory Mapped I/O Base Address Registers) of the virtual hardware.

In order to solve the virtual hardware state condition interference, we propose SAVHF (Framework for State-Aware Virtual Hardware Fuzzing). In this work, we provide the instrumentation method to obtain the state information of virtual hardware, and we also use the fuzzing strategy to reduce the state condition interference and improve the code coverage of target virtual hardware. We try to answer the following key questions.

(1) How to obtain the state condition of virtual hardware?

(2) How to reduce the state condition interference of virtual hardware?

(3) How to use the state condition information to improve the adaptability of the fuzzing to the virtual hardware?

1.3. Contributions. SAVHF uses a syntax-based instrumentation method to monitor the state transition of virtual hardware. SAVHF also includes a state-based fuzzing strategy. It guides fuzzing to traverse the states of the virtual hardware and performs efficient testcase variation and input in various states. Meanwhile, since SAVHF does not depend on the features provided by a definite target virtualization software, SAVHF can be applied to a variety of open source virtualization softwares. The contributions of this paper can be summarized as follows:

(1) SAVHF. We propose the SAVHF, a virtual hardware fuzzing framework, which can perceive the state transition of virtual hardware and effectively detect vulnerabilities in virtual hardware

(2) Source-to-source instrumentation based on abstract syntax tree. We propose a syntax-based instrumentation method to effectively monitor the state transition of virtual hardware. By analyzing the abstract syntax tree of the target virtual hardware, we insert instrumentation at the reference code of the key structure that records the information of the virtual hardware control registers, so as to effectively detect the state transition of virtual hardware during fuzzing process

(3) State-based fuzzing strategy. We propose a strategy for fuzzing to traverse the states of the virtual hardware and to generate the testcases as the operation queue for the virtual hardware. It is a state-based fuzzing optimization strategy based on the feedback provided by instrumentation. This strategy can reduce the interference of virtual hardware state condition on the evaluation of fuzzing testcases and

improve the efficiency of fuzzing. The testcase mutation method of this strategy is guided by the feedback of both state and path. Compared with the optimization strategy of genetic algorithm solely based on path, the optimization strategy proposed in this paper can improve the performance in discovering new paths and new states in fuzzing for virtual hardware, which is demonstrated by our experiments

(4) Accurate PoC generation. By monitoring the state transition of virtual hardware, we can obtain key testcases that cause state transition of virtual hardware. On this basis, we can simplify the testcase sequence while ensuring the accuracy of the PoCs of the testcase sequences that cause the crash

1.4. Organization. The rest of this paper is organized as follows. Section 2 introduces the technical background in virtual hardware and fuzzing. Section 3 introduces SAVHF's instrumentation method and fuzzing strategy. Section 4 briefly introduces the implementation of SAVHF. In Section 5, we evaluate the performance of SAVHF from the aspects of code coverage, crash classification, PoC availability, and strategy performance. This study is finally concluded in Section 6.

2. Technical Background

2.1. Virtual Hardware

2.1.1. Access Method. MMIO (Memory Mapped I/O) and PMIO (Port Mapped I/O, also called isolated I/O) are two main methods to access the peripheral hardware in normal computers. Dedicated I/O processors are also used to handle I/O requests, but they still use the MMIO and PMIO interfaces. In a virtualization platform, most virtual hardware uses MMIO or PMIO (one or both) to provide access for the CPU, which are the methods discussed in this paper. For hardware mounted on the PCI bus, the BIOS scans them when the computer is launched and reads the BAR (Base Address Registers) of them and performs address mapping according to its defined mapping type (MMIO or PMIO) [17].

The method to access MMIO address space is the same as accessing normal memory address space, which makes it convenient for guest system to access the IO address space of the hardware using the MMIO mapping method. In contrast, to access the PMIO address space, guest system must use the specific instructions provided by CPU instruction set. For systems running on x86/amd64, OUT and IN can be used to access PMIO address space.

The Linux operating system designs a pseudofilesystem called sysfs to export kernel objects, which provides an interface to access kernel data structures. The directory */sys/devices* provides the hierarchy of Linux kernel device tree, so that users can obtain the device model information and device address space by accessing the filesystem node in this directory [18]. For example, after users map the file */sys/devices/pci0000 : 00/0000 : 00:0* f.0/*resource1* to the virtual address space of the process by using MMAP system call, the read and write operation on the virtual address will be transferred by the Linux kernel to the I/O operation on the BAR 0 of the graphic card device mounted on PCI bus 0, which is a memory mapped device address space. Thus, with the help of Linux sysfs, users can access the device address space using unified I/O representation, which is a tuple consisting of address and size for read operation or a triple consisting of address, size, and value for write operation.

2.1.2. State Transition. The implementation of virtual hardware needs to follow the specifications of the simulated hardware to provide the same functions as real hardware. It has state information at runtime, and when the virtual hardware is in a different state, the logic for handling commands or external interference may be different [19]. Generally, the operation logic of the virtual hardware is controlled by the configuration registers of the virtual hardware, which can be customized by users. Configuration registers and other registers are accessed by the way of address space mapping, which can be described as a unified representation.

For example, users can disable the CRC checksum appended after each package by setting the 16th bit (CRC disable bit) of the rtl8139 transmit configuration register to 1. With the help of Linux pseudofilesystem sysfs, users only need to map the file resource 0 in the file system directory corresponding to the rtl8139 to the process virtual memory and set the 16th bit of the 2-byte word at the 64th offset of the mapped memory to 1 [20].

As the configuration registers changes, the operation logic of the hardware changes, which means the hardware has transformed from the prior state to a new one. The state of the hardware will also be changed due to external interference, which may not be user-controllable. Normally, the hardware will return to the waiting state after completing the current task to handle the next user's instruction or external interference.

2.1.3. Vulnerability. Virtual hardware is one of the relatively independent and direct ways for users to interact with the software implementation of certain specific modules in the virtualization platform. Coupled with the wide variety of virtual hardware, there are errors in their implementation, which makes virtual hardware a vulnerable target for attackers. Hardwares with high usage rates, such as display adapters, network adapters, disk drives are the main targets for attackers. As shown in Table 1, statistics on high-risk Qemu virtual hardware vulnerabilities exposed by the CVE vulnerability database before 2019 [21] show that display adapters, network adapters, disk drives, and Virtio devices account for more than half of virtual hardware vulnerabilities. Virtual hardware vulnerabilities generally have the characteristics of low exploration complexity and great danger, and most of the vulnerabilities can lead to the impact of code execution attacks.

2.2. Greybox Fuzzing Framework

2.2.1. Instrumentation and Feedback. The instrumentation and feedback are necessary conditions for the greybox fuzzing and are also a specific measure that different from the black-box fuzzing. Instrumentation is used to obtain

Table 1: Qemu virtual hardware high-risk vulnerabilities statistics.

Hardware	Vulnerablity	Complexity		Effect	
		Low	Medium	Code Exucution	Other
Display adapters	5	5	0	4	1
Network adapters	5	4	1	5	0
Disk drives	5	4	1	4	1
Virtio	4	3	1	3	1
Other	16	15	1	11	5

feedback information of the target during the testing process. For targets with and without source code, compile-time instrumentation and runtime instrumentation are used to gain feedback information. For the test target with the source code, the instrumentation technology can be more adaptable to the test target through the custom instrumentation strategy, which includes the instrumentation strategy based on the basic blocks [7], the instrumentation strategy based on the function call operations, and the instrumentation strategy based on the system call operations [22]. For test targets without source code, binary translation and just-in-time compilation are typically used to place instrumentation code between the basic blocks of the test target (Qemu, DynamoRIO, Intel Pin, etc.). With the promotion of Intel PT technology [23], the feedback acquisition technology based on Intel PT is also applied to greybox fuzzing.

Based on different instrumentation strategies, the obtained feedback is also different, but the feedback must provide a matching basis for the optimization function of the greybox fuzzing.

2.2.2. Testcase Input and Target Reset. Testcases of fuzzing are usually applied to the target software in the form of standard I/O or files, which is also the way most software processes input. In order to accurately obtain the effect of each testcase on the target software, it is necessary to ensure that each testcase has an independent effect on the target software, which is achieved by resetting the target software each time a testcase is entered. If we reset the target software, it will cost more computing and time resources. In order to remedy this, AFL proposed a fork server mechanism to reduce the resource consumption before the process is loaded into memory.

In greybox fuzzing with virtual hardware as the target, the state of the virtual hardware is changed with the input of testcases. If the state information of the virtual hardware is not monitored, it is difficult to evaluate the effectiveness of testcases and to effectively guide the optimization of fuzzing. Without resetting the target virtual hardware module, if the testcases that cause the state changes to the virtual hardware are ignored, all the testcase sequences must be recorded to ensure the validity of the final testcase in the event of a crash, resulting in extreme waste of storage resource and the inaccurate assessment of the role of testcases, which has a lot of unknowable influence on the fuzzing strategy. However, the fork server mechanism cannot adapt to multiprocess or multithreaded test targets. The most feasible way to reset virtual hardware is to restart the virtualization platform, which will consume a lot of time.

3. State-Aware Fuzzing Framework

The key problem of virtual hardware fuzzing is the interference of state transitions of the evaluation of fuzzing testcase. The state transition of the virtual hardware is complicated. The behavior of the virtual hardware under the current testcase is affected by the previous state of the virtual hardware and the current testcase itself. Moreover, the current testcase may also cause the state transition of the virtual hardware, thereby generating a new state.

To briefly explain the state condition of fuzzing virtual hardware, we hereby give an example as shown in Figure 1. When $\text{testcase}_{[x]}$ is input to virtual hardware, its initial state is $\text{state}_{[n]}$. The virtual hardware processes $\text{testcase}_{[x]}$ and generates corresponding feedback. However, the testcase fails to trigger the state transition of the virtual hardware, so when $\text{testcase}_{[x+1]}$ is input, the state of the virtual hardware inherits from $\text{state}_{[n]}$. $\text{Testcase}_{[x+1]}$ triggers the state transition of the virtual hardware, so when $\text{testcase}_{[x+2]}$ is input, its initial state is $\text{state}_{[n+1]}$. The impact of testcases on the virtual hardware is fed back to fuzzer. The feedback of a testcase in virtual hardware fuzzing is based on its initial state, which interferes with testcase evaluation and PoC generation in fuzzing. At the same time, utilizing the state transition information of virtual hardware can improve the efficiency of virtual hardware fuzzing and further explore the code branches of virtual hardware.

The state of the virtual hardware is determined by the virtual hardware control registers, so we can effectively detect the state transition of the virtual hardware by monitoring the modification of these registers. To find the state transition of virtual hardware accurately by detecting the modification of the control registers, we proposed syntax-based source-to-source instrumentation in Subsection 3.1.

In order to utilize the state transition information of virtual hardware to discover more code branches, we propose a state-based fuzzing strategy in Subsection 3.2. It is the basic optimization strategy for our fuzzing framework and guides fuzzing process. In this case, we can explore more states and more code branches.

3.1. Syntax-Based Instrumentation. In this module, we used a compiler to generate the abstract syntax tree of the source code of the target virtual hardware and insert

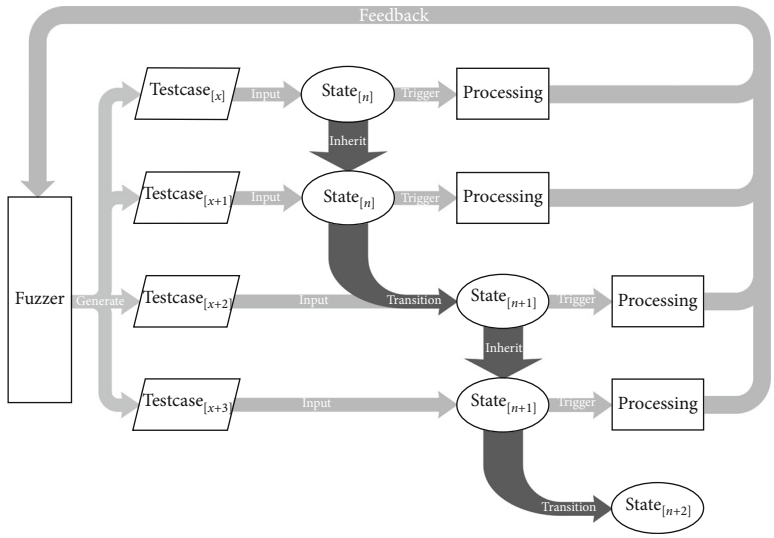

Figure 1: Testcases affect state transition of virtual hardware and generate feedback.

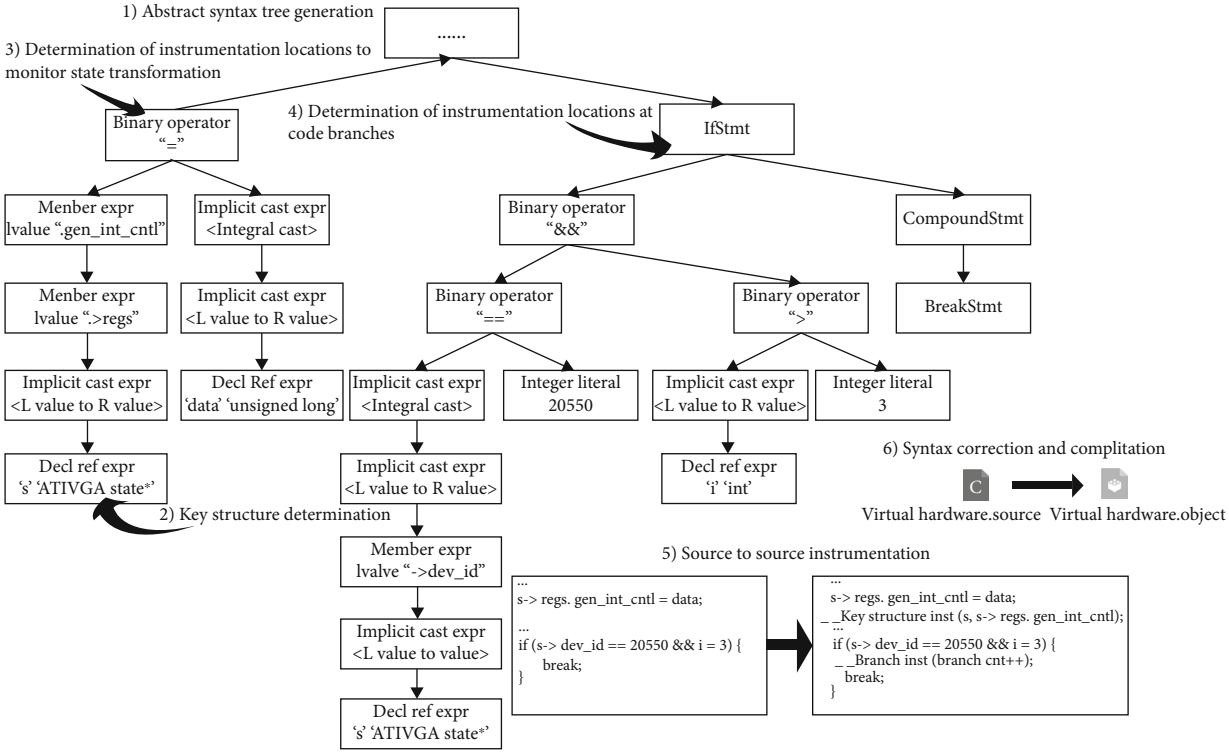

Figure 2: Syntax-based instrumentation steps of our model.

instrumentation at the root node of the abstract syntax subtree that caused the change of the key structure of target virtual hardware. As shown in Figure 2, the routines of this module include the following steps.

(1) Abstract syntax tree generation. We use a compiler to generate the abstract syntax tree of the source code of the target virtual hardware module. The abstract syntax tree provides convenient interfaces for key structure determination, reference code determination, and instrumentation code insertion. The nodes of the abstract syntax tree include declaration, statement, expression, and other types. Declaration is the root node of the variable (or function) declaration, statement is the root node of the basic statement of the program, and expression is the root node of the basic expression

(2) Key structure determination. The state of the virtual hardware implemented by software is determined

by its key structure. So, we monitor the state of the virtual hardware by monitoring the key structure changes. According to the specific situation of the virtualization platform where the target virtual hardware is implemented, we determine the key structure to be monitored manually. Take vga-pci as an example, its key structure is *PCIVGAState*. The registers related to the state such as *mmio region* and *mrs* of the vga-pci are stored in this structure

(3) Determination of instrumentation locations to monitor state transition. Search the abstract syntax tree to find the abstract syntax subtrees that may modify the key structure and determine the root nodes of the abstract syntax subtrees as the instrumentation locations. We search for codes that modify the key structure through matching rules. This rule matches the reference node of the key structure (or its member variables) as the value code of the assignment operation to find the code that causes the key structure to be modified. After locking the node that causes the modification of the key structure, we use the nearest statement type node among its ancestor nodes as the instrumentation root node

(4) Determination of instrumentation locations at code branches. Determine the root nodes of the logical branches in the abstract syntax tree as the instrumentation locations. The code branch jump is composed of statement nodes such as conditional judgments or loops. For the statement nodes of the conditional judgment type, we use the statement root node of each conditional branch as the instrumentation root node. For the statement node of the loop type, we use the first statement node of the loop body as the instrumentation root node

(5) Source-to-source instrumentation. According to the type of instrumentation locations, the instrumentation code is inserted into the source code of the target virtual hardware module. For code branch type instrumentation locations, insert code branch jump monitoring instrumentation code. For key structure monitoring type instrumentation locations, insert their corresponding structure monitoring instrumentation code. The code branch jump monitoring instrumentation code will feed back branch jump information to the fuzzer when the target virtual hardware is running, and the structure monitoring instrumentation code will give state change information to the fuzzer when the key structure is modified, which means the state of virtual hardware is changed

(6) Syntax correction and compilation. Modify the newly generated code to ensure the syntax is correct and compile it to generate virtual hardware module with instrumentation inserted. This module will be inserted into the corresponding executable of the virtualization platform during the linking operation of the compilation process, so that it will be used when the virtualization platform is running

Remark 1. Through the above instrumentation method, the modification of control registers of the virtual hardware will be detected during the fuzzing process, and the code branches covered during the processing of testcases can also be accurately recorded. Code branch feedback and state transition feedback provide the basis for customizing a fuzzing strategy that meets the characteristics of virtual hardware.

3.2. State-Based Fuzzing Strategy. The change of the key structure does not completely mean that the state of the virtual hardware changes, but the change of the state of the virtual hardware basically depends on the key structure of the virtual hardware. By monitoring the modification of the key structure of the virtual hardware, we obtained the effect of testcases on the state of the virtual hardware. We define the testcases that cause the change of the state of target virtual hardware as high-value testcases. The current state of the virtual hardware is the state affected by the high-value testcase sequence after the virtual hardware is reset. We replay a certain segment of the sequence from the first testcase to roll back the state of the virtual hardware. Based on a certain state of the virtual hardware, we can fully explore the processing logic that the virtual hardware completes in this state by entering mutated testcases. As is shown in Algorithm 1, our fuzzing strategy includes the following steps.

(1) Reset the virtual hardware by restarting the guest instance and then use randomly mutated testcases to find testcases that may independently cause changes in the key structure of the virtual hardware. If a new high-value testcase is found, the virtual hardware is reset. On the premise of ensuring that this testcase does not overlap with the testcases in the current testcase corpus, we add the testcase to the high-value testcase corpus. If the number of testcases in the high-value testcase corpus is greater than the threshold T_{high}, then go to step 2

(2) Conduct fuzzing of one round with R_{init} as the length of the initialization testcase sequence of rollbacks of this round and traverse the permutation of R_{init} testcases in the current high-value testcase corpus to generate possible testcase sequences of rollback. After the rollback with the testcase sequence Q_{now} as the current initialization testcases, we try to enter mutated testcases to fully explore the processing logic of the target virtual hardware based on this state. If the number of input mutated testcases is greater than the threshold $N_{explore}$, we discard this rollback testcase sequence and try the next rollback testcase sequence

(3) During the exploration process, if a testcase triggers the modification of a key structure, we insert it at the end of $S_{current}$, and on the premise of ensuring that it is not duplicated with the testcases in the high-value testcase corpus, we add it to the high-value testcase corpus. If a testcase can find a new code branch or can find some code branches with less

```
1: S_high = ∅;
2: S_normal = ∅;
3: R_init = 0;
4: while Count(S_high) < T_high do
5:   t_now = TestcaseRandomGen();
6:   feedback = FuzzOne(t_now);
7:   if AffectKeyStruct(feedback) then
8:     ResetHardware();
9:     if NotDuplicated(S_high, t_now) then
10:      AddTo(S_high, t_now);
11:    end if
12:  end if
13: end while
14: while True do
15:  R_init = (R_init + 1) mod |S_high|;
16:  for Q_now ∈ A^{R_init}_{|S_high|} do
17:    RollBack(Q_now);
18:    for n ⟵ 1 to N_explore do
19:      tnow = TestcaseMutate(Shigh,Snormal);
20:      feedback = FuzzOne(t_now);
21:      if AffectKeyStruct(feedback) then
22:        if NotDuplicated(S_high, t_now) then
23:          AddTo(Shigh,tnow);
24:        end if
25:      end if
26:      if DetectNewBranch(t_now) then
27:        AddTo(S_normal);
28:      end if
29:      if CauseCrash(feedback) then
30:        LogCrash(Q_now, t_now);
31:      end if
32:    end for
33:  end for
34: end while
```

Algorithm 1: Fuzzing algorithm based on state-based fuzzing strategy.

consumption, we add it to the normal testcase corpus. If a testcase causes the target virtualization platform to crash, insert it at the end of the current testcase sequence $S_{current}$ and generate the PoC based on $S_{current}$

(4) After one round fuzzing, the results are counted, and the high-value testcase corpus is optimized. Increase the length of the initialization testcase sequence by 1, jump to step 2, and start fuzzing of next round

The testcases of virtual hardware can be described by triples (address, size, value) for write operations and tuples (address, size) for read operations. The main method of mutation is to randomize address, size, and value. For the mutation based on a high-value testcase, we perform random value addition and subtraction operations on the address, size, or value of the basis testcase. For randomization mutation, we use the input address range to randomize the address, use 1, 2, 4, and 8 to randomize the size, and use the range corresponding to the size to randomize the value.

Figure 3 shows the fuzzing strategy when three state transitions are found, and each state transition corresponds to a high-value testcase. When performing rollback operations, we try to replay these testcases to achieve random reorganization of the virtual hardware state. This strategy can traverse the state of the virtual hardware as much as possible and discover the behavior of the virtual hardware in various states, thereby improving the code coverage of fuzzing.

Remark 2. The above fuzzing strategy explores the various states of virtual hardware by traversing the transition sequence of state variables and using it as the operation to initialize the test target. This strategy reduces the interference of the state information to the evaluation of testcases by periodically rolling back the fuzzing target. Meanwhile, the strategy defines the testcases for discovering new branches, and the testcases for discovering new states as the input of the testcase mutation function to make the state information and branch detection information together play a guiding role in fuzzing.

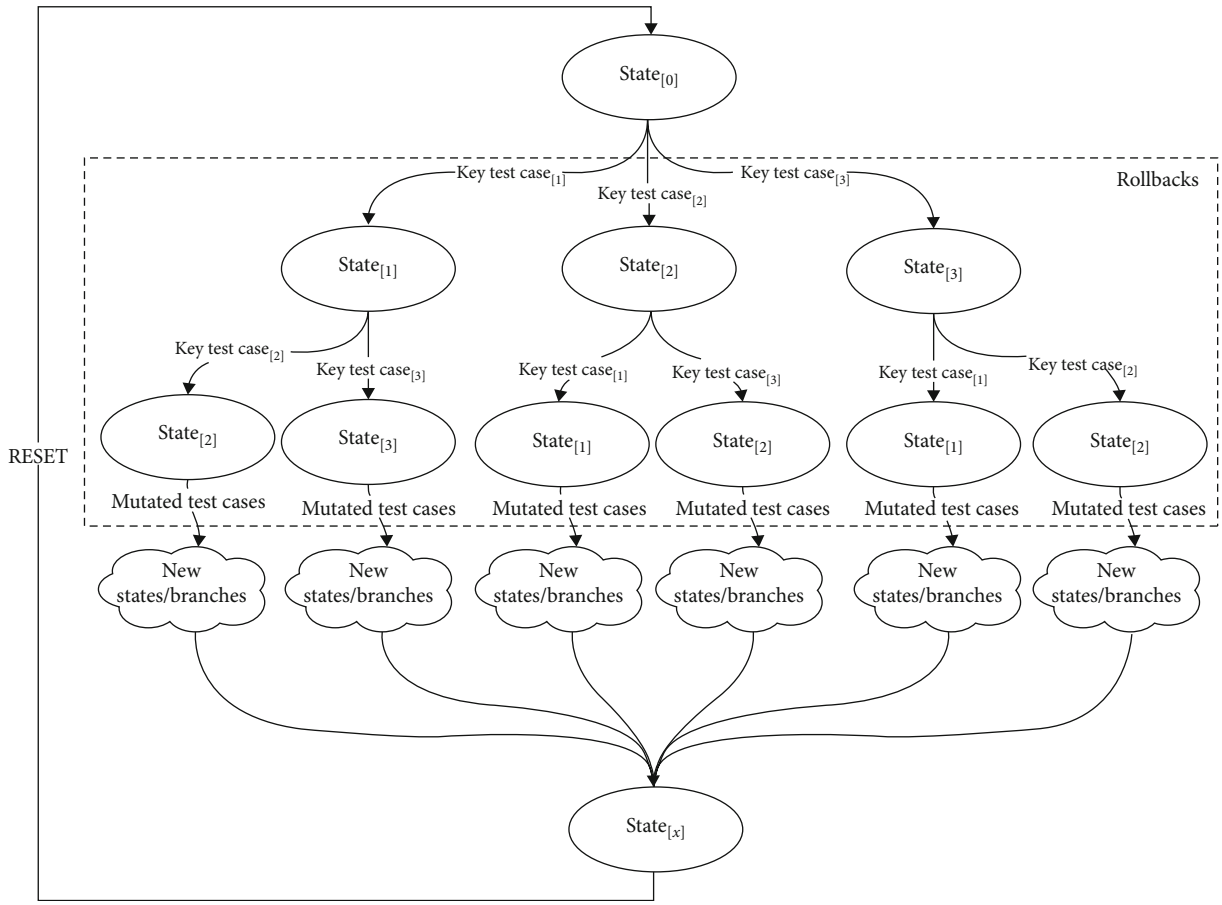

Figure 3: State-based fuzzing strategy.

4. Implementation

Based on the model we introduced in Section 3, we implement SAVHF. The key components are elaborated as follows.

(1) Fuzzing framework. As shown in Figure 4, our fuzzing framework is mainly composed of three modules: the instrumentation inserter, the fuzzing back-end, and the fuzzing front-end. Among them, the instrumentation inserter implements the instrumentation at key structure modification parts and branch jump parts of the target virtual hardware source code based on the abstract syntax tree analysis; the fuzzing back-end is responsible for obtaining and analyzing the feedback information of fuzzing, generating the mutated testcases and guiding the fuzzing process according to the optimization strategy; the fuzzing front-end is responsible for applying the testcases transmitted by the fuzzing back-end to the target virtual hardware

(2) Instrumentation-inserter. Based on the abstract syntax tree analysis interfaces provided by the clang compiler, we implemented the instrumentation-inserter. The key structure is determined by a simple analysis of the source code of the target virtual hardware. We found that the virtual hardware in the same virtualization platform follows the same implementation rules. So, it is relatively easy to find the key structure. We then input the key structure information into the source-to-source instrumentation component, which is based on the AST matcher interfaces provided by clang and uses defined rules to find the root node of the abstract syntax subtree that may cause the key structure to change. At the same time, by traversing the abstract syntax tree of the target virtual hardware source code, we can also find the code branches. After analyzing the types of the abstract syntax tree nodes that need to be inserted with instrumentations, the instrumentation code is inserted into the source code. Finally, after correcting possible syntax errors in the instrumentation code, the target virtual hardware is compiled

(3) Fuzzing back-end. Fuzzing back-end interacts with the instrumentation code in the target virtual hardware through shared memory and pipes. Shared memory is used to transfer the description information of key structure and code branch jump information, and pipes are used to send and receive commands between fuzzing back-end and instrumentation code. The testcases are sent to the fuzzing front-end in the form of uniform I/O tuples, and mutation methods include randomization, splicing,

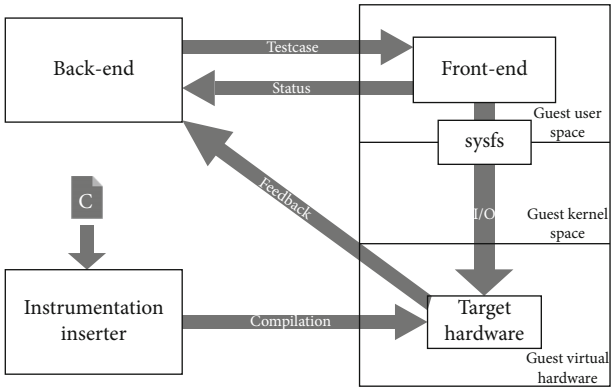

Figure 4: Our fuzzing framework.

insertion, and segmentation. In order to reduce the memory resource consumption of fuzzing, we only save the changed parts of key structure corresponding to the high-value testcases, not the entire structure

(4) Fuzzing front-end. In order to ensure that the fuzzing environment and the real environment have the same condition to interact between guest and virtual hardware, the fuzzing front-end is set in the guest instance of the virtualization platform where the target virtual hardware is located. According to the device manufacturer number and device number specified by the fuzzing back-end, the front-end finds the corresponding file node in *sysfs* to interact with the virtual hardware through this file node. The front-end uses a network adapter in the virtualization platform that has nothing to do with the target virtual hardware to interact with the back-end to eliminate the uncertainties caused by the front-end and back-end interactions for fuzzing. Due to the use of the instrumentation method based on abstract syntax tree, we only insert instrumentation into the code closely related to the target virtual hardware. Therefore, the operation of the network adapter during the interaction between the back-end and front-end will not trigger the inserted instrumentation

5. Evaluation

We selected a set of representative virtual hardware of the open source virtualization software Qemu under different versions as the dataset of the experiment, which contained network devices, display adapters, sound cards, bus controllers, and a virtual disk drive under version 4.0 of Qemu.

The virtual network devices we selected included ne2000, rtl8139, eepro100 (i82550), vmxnet3, and rocker; the selected virtual display adapters included cirrus-vga, vmware-svga, vga-pci, ati-vga, and bochs-display; the selected virtual sound cards included ac97, es1370, and intel-hda; the selected bus controllers included piix4-usb-uhci, pvscsi, and sdhci-pci; the selected virtual disk drive was nvme. The virtual hardware in the data set contained virtual hardware with high usage rates, and the total number of the virtual hardware in our dataset was 17.

We ran our experiments on a workstation equipped with a modern Intel processor and 64 gigabytes RAM. For each virtual hardware which is the target of fuzzing, we gave it 8 gigabytes of RAM and 2 processor cores and continue fuzzing for more than 18 hours. During the experiment, we recorded the code branch coverage of the target virtual hardware, the number of triggered state transition, and the number of crashes found during fuzzing.

5.1. Code Coverage and Crash Discovery. In our test, we collected four metrics to objectively reflect the performance of SAVHF, and these metrics include the number of target virtual hardware code branches currently discovered, code coverage (the percentage of basic code blocks found), the number of found crashes, and the number of found unique crashes.

We counted and averaged the code coverage in all tests, shown in Figure 5. The first hour of fuzzing is very efficient, and the efficiency gradually decreases with time. The code coverage eventually stabilizes around a value and grows slowly, and this value is about 61% after we averaged the coverage of all tests.

Figure 6 shows the trend of detecting unique crashes overtime during fuzzing. The total number of unique crashes was gradually increasing, but the step-like growth occurred after discovering some new crashes. This aspect indicates that the defective code may be triggered in many ways; on the other hand, it indicates that the discovery of unique crashes is based on improving coverage.

After the fuzzing test for each virtual hardware, we record the final data of these metrics, which are shown in Table 2. Although our fuzzing experiment generally only took about 18 hours, the average code coverage reached more than 61%, of which the highest coverage reached 89.41% (es1370). Of the 17 virtual hardwares as test datasets, we found unique crashes (or hangs) in 4 of them. Considering that the tested Qemu virtualization platform was a relatively new stable version, this data shows that SAVHF has good performance in detecting virtual hardware bugs.

As is presented in Table 3, all the unique crashes we obtained were manually analyzed one by one and were divided into the following 3 categories according to the effect of crashes.

(1) Memory corruption. Crashes of the category of memory corruption are generally illegal memory accesses caused by programs failing to control the use of variables such as arrays and pointers. The test with ati-vga as the target detected 11 unique crashes, 2 of which were memory-destruction crashes, which crashed in the function *ati_cursor_define* of ati-vga

(2) Assertion. Crashes of the category of assertion refer to failed restrictions on unintended behavior or unimplemented functions in the release version, which usually result in unexpected termination of the program. The only unique crash of vmxnet3

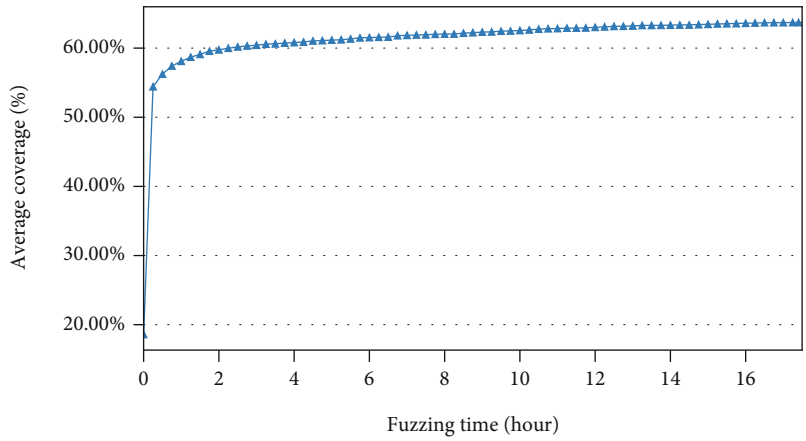

Figure 5: Average percentage of coverage over time during fuzzing.

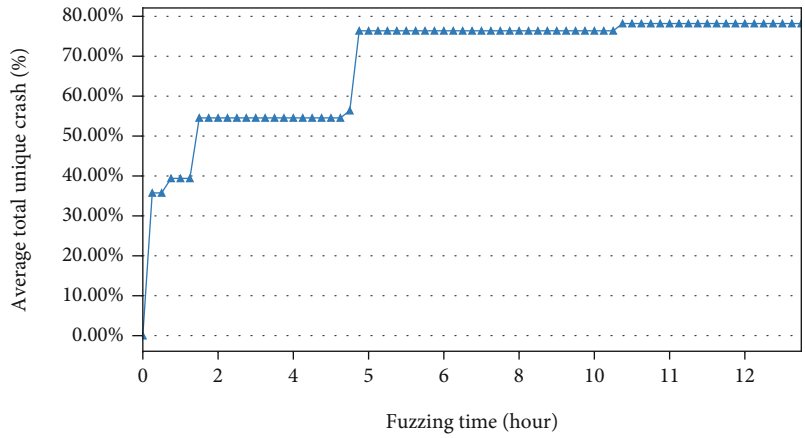

Figure 6: Average percentage of total unique crashes over time during fuzzing.

was caused by the failed assertion *g_assert_not_reached* and triggered a SIGABRT error. 7 unique crashes of ati-vga are assertion errors and finally crashed at the assertion *assert* (*bpp!*=0). The only unique crash produced by the fuzzing with pvscsi as the target was an assertion error, which was caused by a failed assertion *assert* (*s- > rings_info_valid*).

(3) Hang. Crashes of the category of hang generally refer to crashes that cause the program to fall into long-term processing (or endless loop) due to logic errors and prevent users from interacting with it. 2 unique crashes of ati-vga were of this category and were causes by unchecked recursive call in the *ati_mm_read function*, which ultimately caused the virtual platform to die. 3 unique crashes of cirrus-vga were caused by unchecked loop limitation

5.2. PoC Availability. For each unique crash, SAVHF generated a PoC program that may trigger the crash and the basic block passing flow of the target virtual hardware. SAVHF records the testcase sequence that affected the virtual hardware state, and the PoC program was generated after processing the pretestcase sequence and the testcase that triggered the crash. In order to analyze the effectiveness of PoCs generated in tests, we recompiled the corresponding version of the original Qemu virtualization platform and launched it with the same configuration to ran each PoC program in the guest environment. We divided PoCs according to their corresponding virtual hardware and crash types. The results are given in Table 4.

The availability of PoCs of category assertion and memory corruption reached 100%, but of the hang category PoC, their availability is lower, with an average of 41.67%. This result indicated that the hang category crashes of the virtual hardware were relatively complicated, which may be affected by other conditions besides the virtual hardware state.

5.3. Strategy Performance. To evaluate the performance of our fuzzing strategy, we conducted comparative experiments. We use the path-based fuzzing strategy to experiment on the same virtual hardware. This strategy is also a basic optimization strategy for greybox fuzzing. In addition, we use the same code branch instrumentation method to ensure the same granularity of instrumentation. Under the same

Table 2: Virtual hardware tested with SAVHF.

Hardware type	Module	I/O regions		Branches detected	Coverage	Crashes found	Unique crashes	Duration
		Mapping type	Mapping size					
Network devices	ne2000	PMIO	0 × 100	169	78.76%	0	0	18 h
	rtl8139	PMIO	0 × 100	421	67.74%	0	0	18 h
		MMIO	0 × 100					
	eepro100 (i82550)	MMIO	0 × 4000	270	81.39%	0	0	18 h
		PMIO	0 × 40					
		MMIO	0 × 20000					
	vmxnet3	MMIO	0 × 1000	56	13.30%	6630	1	18 h
		MMIO	0 × 2000					
		MMIO	0 × 1000					
	Rocker	MMIO	0 × 2000	206	47.26%	0	0	18 h
		MMIO	0 × 2000					
Display adapter	cirrus-vga	MMIO	0 × 2000000	738	81.60%	4	3	18 h
		MMIO	0 × 1000					
	vmware-svga	MMIO	0 × 1000	106	44.62%	0	0	18 h
		MMIO	0 × 1000000					
		MMIO	0 × 10000					
	vga-pci	MMIO	0 × 1000000	32	73.68%	0	0	18 h
		MMIO	0 × 4000					
	ati-vga	MMIO	0 × 1000000	356	83.88%	3251	11	18 h
		PMIO	0 × 100					
	bochs-display	MMIO	0 × 4000	22	42.11%	0	0	18 h
		MMIO	0 × 1000000					
		MMIO	0 × 1000					
Sound card	ac97	PMIO	0 × 400	467	81.29%	0	0	18 h
		PMIO	0 × 100					
	es1370	PMIO	0 × 100	265	89.41%	0	0	18 h
	Intel-hda	MMIO	0 × 4000	104	56.12%	0	0	18 h
Bus controller	piix4-usb-uhci	PMIO	0 × 20	219	77.39%	0	0	18 h
	pvscsi	MMIO	0 × 8000	98	52.59%	582	1	18 h
	sdhci-pci	MMIO	0 × 100	306	62.40%	0	0	18 h
Block	nvme	MMIO	0 × 2000	169	49.06%	0	0	18 h
		MMIO	0 × 1000					

Table 3: Classification of unique crashes.

Virtual hardware	Crash category	Unique crashes
vmxnet3	Assertion	1
cirrus-vga	Hang	3
ati-vga	Memory corruption	2
	Assertion	7
	Hang	2
pvscsi	Assertion	1

Table 4: Availability of PoC.

Virtual hardware	Crash category	PoCs	Available PoCs
vmxnet3	Assertion	1	1
cirrus-vga	Hang	3	1
ati-vga	Memory corruption	2	2
	Assertion	7	7
	Hang	2	1
pvscsi	Assertion	1	1

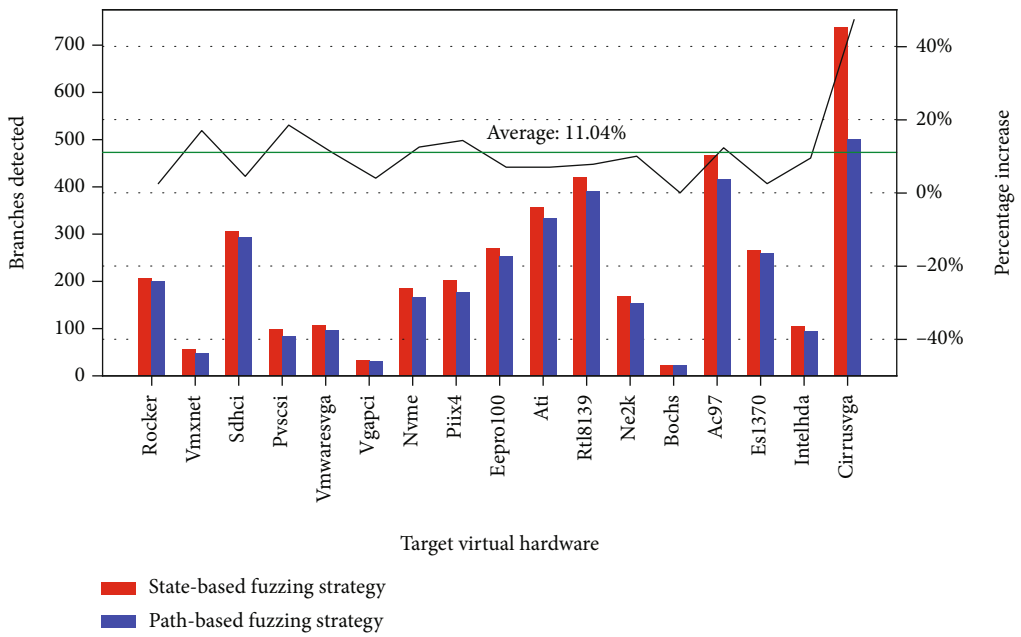

Figure 7: The difference in the number of code branches found by fuzzing under the guidance of two strategies.

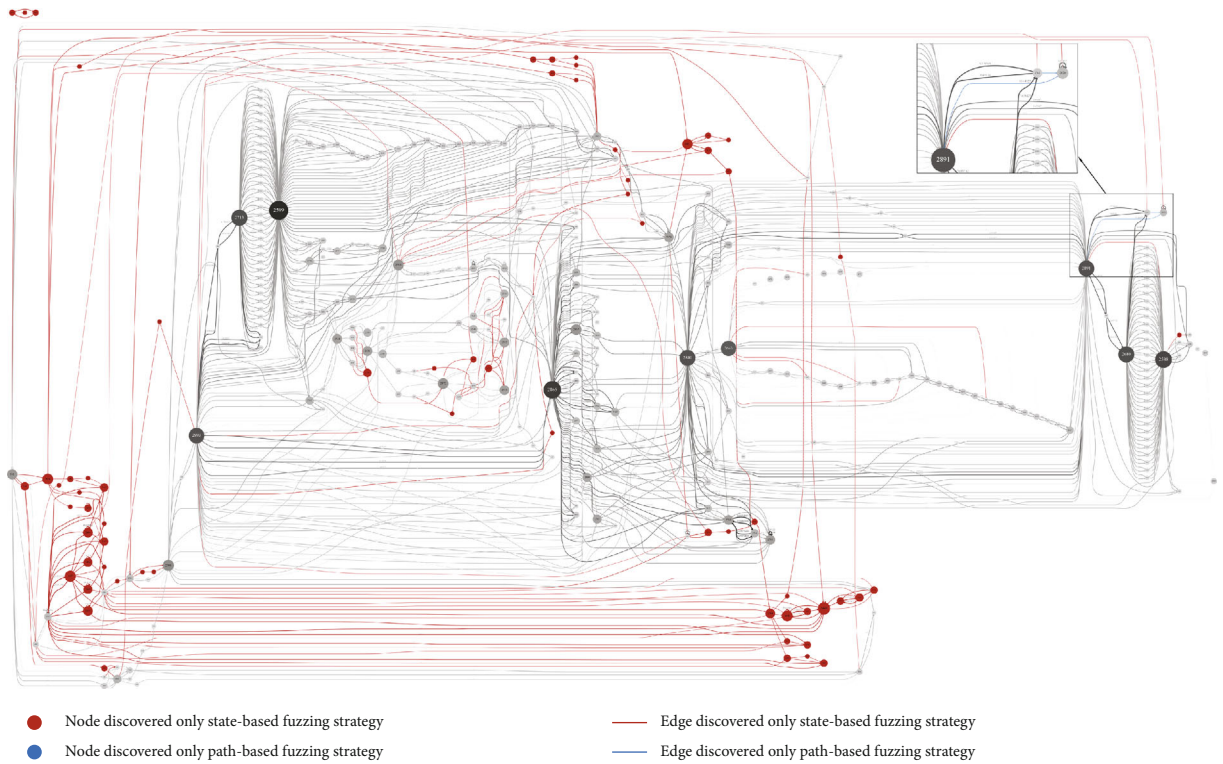

Figure 8: Branches found by fuzzing under the guidance of two strategies with cirrus-vga as the target.

condition, the difference in the performance of fuzzing is mainly caused by the fuzzing strategy. We mainly evaluate the performance of the fuzzing strategy based on the number of virtual hardware code branches found at the same time.

For each virtual hardware, the difference in the number of code branches found by fuzzing under the guidance of these two strategies is shown in Figure 7. Compared with the path-based fuzzing strategy, the state-based strategy proposed in this paper can increase the number of discovered virtual hardware branches by 11.04%. Particularly, with the virtual display adapter cirrus-vga as the test target, the increase reached more than 40%.

We compared the code branches found during fuzzing using these two strategies. The results are given in Figure 8.

The parts marked in red in Figure 8 are the code branches, and basic code blocks discovered using the state-based fuzzing strategy; the parts marked in blue are the code branches, and basic code blocks discovered using the path-based strategy; the other parts were discovered by both two strategies.

The branch and basic blocks found in Figure 8 using the state-based fuzzing strategy have clustering characteristics. We checked these code branches and found that most of the code branches need to meet the specified value of the virtual hardware control register. The control registers used by these checks include *cirrus_blt_srcaddr* and *cirrus_srccounter*, which control whether cirrus-vga can perform certain logic processing activities.

The behavior of the virtual hardware is determined by both the state and input of the virtual hardware. Traversing more states helps to discover the behavior of virtual hardware in various states, thereby improving the code coverage of fuzzing.

6. Conclusion and Discussion

In this paper, we propose SAVHF, a state-based virtual hardware fuzzing framework. To effectively detect the state transition of the virtual hardware, we proposed a source-to-source instrumentation method based on the abstract syntax tree. We also introduced a state-based fuzzing strategy, which can effectively reduce the interference of virtual hardware state conditions on fuzzing and increase the efficiency of fuzzing. We used SAVHF to fuzz 17 representative virtual hardware of Qemu and found 16 unique crashes. For the ones that have not been patched, we actively contacted the vendor and obtained the CVE vulnerability number or positive response. SAVHF covered an average of more than 61% of virtual hardware code branches within 18 hours during testing and can improve the average code coverage by 11.04% compared with the path-based fuzzing strategy.

The instrumentation method based on abstract syntax tree proposed in this paper mainly detects the state transition of virtual hardware by detecting the modification of the key structure of virtual hardware. However, the effect of each member variable in the key structure on the processing logic is complicated. The framework proposed in this paper does not further analyze the specific effect of each control register on the processing logic, but only monitors the modification of control registers. The main purpose of this method is to add state transition information to the optimization strategy of fuzzing, so as to further improve the efficiency of fuzzing. If methods such as symbolic execution can be used to analyze the state conditions, it may be able to further improve the efficiency of fuzzing.

Data Availability

(1) The [evaluation] data used to support the findings of this study have been deposited in the https://github.com/F1r/SAVHF_data.git repository. (2) The [evaluation] data used to support the findings of this study are included within the article. (3) The [evaluation] data used to support the findings of this study were supplied by [Hang Xu] under license and so cannot be made freely available. Requests for access to these data should be made to [Hang Xu, Email: flier_m@outlook.com].

Conflicts of Interest

The author(s) declare(s) that they have no conflicts of interest.

Acknowledgments

This work is supported by the SAST Industry-University-Research Cooperation Foundation, the Henan Provincial Key Research, National Key Research, and Development Project under grant: 2017YFB0802902, Development and Promotion Project under grant: 551, and the Henan Provincial Key Scientific Research Project for College and University under grant: Â 21A510011.

References

[1] N. Goel, A. Gupta, and S. N. Singh, "A study report on virtualization technique," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 1250–1255, Greater Noida, India, 2016.

[2] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: a survey on concepts, taxonomy and associated security issues," in *2010 Second International Conference on Computer and Network Technology*, pp. 222–226, Bangkok, Thailand, 2010.

[3] P. You, Y. Peng, W. Liu, and S. Xue, "Security issues and solutions in cloud computing," in *2012 32nd International Conference on Distributed Computing Systems Workshops*, pp. 573–577, Macau, China, 2012.

[4] A. Gkortzis, S. Rizou, and D. Spinellis, "An empirical analysis of vulnerabilities in virtualization technologies," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 533–538, Luxembourg, Luxembourg, 2016.

[5] Z. Guo, Y. Shen, A. K. Bashir et al., "Robust spammer detection using collaborative neural network in internet of thing applications," *IEEE Internet of Things Journal*, p. 1, 2020.

[6] V. J. Manès, H. Han, C. Han et al., "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, p. 1, 2019.

[7] M. Zalewski, *American fuzzy lop*, 2017, https://lcamtuf.coredump.cx/afl/.

[8] N. Stephens, J. Grosen, C. Salls et al., "Driller: Augmenting fuzzing through selective symbolic execution," *NDSS*, vol. 16, no. 2016, pp. 1–16, 2016.

[9] M. Bohme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, Dallas, TX, USA, 2017.

[10] M. Bohme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox¨ fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

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

[12] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for {OS} kernels," *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 167–182, 2017.

[13] Google, *Syzkaller: an unsupervised, coverage-guided linux system call fuzzer*, 2020, https://opensource.google.com/projects/syzkaller.

[14] S. Groß, *Fuzzil: Coverage guided fuzzing for javascript engines*, Ph.D. dissertation, Master's thesis, Karlsruhe Institute of Technology, 2018, https://saelo.github.io/papers/thesis.pdf.

[15] J. Tang and M. Li, "When virtualization encounter afl," *Blackhat*, 2016.

[16] A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng, "Vdf: Targeted evolutionary fuzz testing of virtual devices," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 3–25, Springer, 2017.

[17] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers: Where the Kernel Meets the Hardware*, O'Reilly Media, Inc., 2005.

[18] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*, O'Reilly Media, Inc., 2005.

[19] K. Cong, F. Xie, and L. Lei, "Symbolic execution of virtual devices," in *2013 13th International Conference on Quality Software*, pp. 1–10, Najing, China, 2013.

[20] Realtek, *Single chip multifunction 10/100mbps ethernet controller w/power management rtl8139d(l) datasheet*, 2020, http://realtek.info/pdf/rtl8139d.pdf.

[21] CVE Details, *Qemu: Security vulnerabilities*, 2020, https://www.cvedetails.com/vulnerability-list/vendorid-7506/productid12657/Qemu-Qemu.html.

[22] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: analyzing impact of coverage metrics in greybox fuzzing," *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pp. 1–15, 2019.

[23] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37302–37313, 2018.