Journal on Artificial Intelligence
DOI:10.32604/jai.2021.017328
Article

Tech Science Press

PS-Fuzz: Efficient Graybox Firmware Fuzzing Based on Protocol State

Xiaoyi Li, Xiaojun Pan and Yanbin Sun*

Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou, 510000, China
*Corresponding Author: Yanbin Sun. Email: sunyanbin@gzhu.edu.cn
Received: 17 January 2021; Accepted: 17 March 2021

Abstract: The rise of the Internet of Things (IoT) exposes more and more important embedded devices to the network, which poses a serious threat to people's lives and property. Therefore, ensuring the safety of embedded devices is a very important task. Fuzzing is currently the most effective technique for discovering vulnerabilities. In this work, we proposed PS-Fuzz (Protocol State Fuzz), a gray-box fuzzing technique based on protocol state orientation. By instrumenting the program that handles protocol fields in the firmware, the problem of lack of guidance information in common protocol fuzzing is solved. By recording and comparing state transition paths, the program can be quickly booted, thereby greatly improving the efficiency of fuzzing. More importantly, the tool utilizes the synchronous execution of the firmware simulator and the firmware program, which can collect and record system information in the event of a crash from multiple dimensions, providing assistance for further research. Our evaluation results show that for the same vulnerability, the efficiency of PS-Fuzz is about 8 times that of boofuzz under ideal conditions. Even rough instrumentation efficiency can reach 2 times that of boofuzz. In addition, PS-Fuzz can provide at least 6 items more information than boofuzz under the same circumstances.

Keywords: Firmware; vulnerability mining; fuzzing

1 Introduction

With the development of emerging technologies such as 5G, the IoT has gradually become the main development trend of human life. Embedded devices, as an important part of the Internet of Things, have also become an indispensable part of our daily lives. While these embedded devices bring convenience to people's lives, they also pose unprecedented security threats. The safety of embedded devices is not only related to people's privacy and property, but may also threaten lives. Hackers often use vulnerabilities in firmware to launch attacks to control a series of IoT devices and even form a large botnet. This is because almost all the programs running in the IoT device are in the firmware. Therefore, it is very important to be able to quickly and efficiently find vulnerabilities in the firmware.

At the beginning, people focused on how to simulate the operating environment of IoT devices. Recent research has proposed a series of solutions, from pure software full-system simulation (such as Firmadyne [1]), to a combination of hardware and software simulation (such as Avatar [2] and Avatar2 [3]), to augmented process emulation (such as FirmAFL [4]). In order to further improve the efficiency of firmware vulnerability mining, firmware fuzzing has become a new research focus. However, the fuzzing of IoT firmware faces the following three challenges:

(1) The fuzzing efficiency is low. The operating environment of the firmware is relatively harsh, which is very different from ordinary software. If you are not doing research on real devices, you need to build a complex firmware simulator. And this kind of simulator itself has low efficiency. Another more important reason is that the IoT devices have strict requirements on the protocol format of the received data packet. Most of the test cases generated by some common black box fuzzing tools are likely to be invalid. Every invalid test is also executed in an inefficient firmware simulator, which seriously affects the efficiency of fuzzing.

(2) Lack of guidance information. Since firmware usually does not have source code, its fuzzing method often uses black box fuzzing technology. This kind of fuzzing will only be executed continuously according to its own mutation strategy, and it is impossible to judge the quality of each test case. The traditional branch guidance information is insufficient, and the combination ability cannot be fully utilized.

(3) Less information about crashes. Existing fuzzing tools almost only provide information about the test cases that caused the crash, and do not care about the entire system at the time of the crash. This is not conducive to further vulnerability information collation and research.

In order to solve the above challenges, we proposed PS-Fuzz–a graybox firmware fuzzing tool. We compared the time spent using PS-Fuzz and boofuzz [5] to find the same firmware vulnerability. The experimental results show that the efficiency of PS-Fuzz is about 8 times that of boofuzz under ideal conditions. Even rough instrumentation efficiency can reach 2 times that of boofuzz. At the same time, the amount of information that PS-Fuzz can provide is at least 6 more items than boofuzz under the same circumstances.

In summary, the contributions of this paper are as follows:

- We implement a protocol state-oriented fuzzing method, and based on this, create an efficient gray box firmware fuzzing tool–PS-Fuzz.
- Through a large number of tool synchronization and information integration, we can effectively monitor the execution state of the firmware program and obtain a large amount of system information when the firmware crashes.
- We use a router firmware to evaluate PS-Fuzz. Experimental results show that the efficiency of PS-Fuzz can reach 2–8 times that of boofuzz under different instrumentation conditions. At the same time, it provides much more information than boofuzz.

2 Related Work

With the development of the Internet of Things industry, more and more vulnerabilities are exposed in embedded devices. People have gradually realized the significance and value of firmware vulnerability research. These techniques can be roughly divided into static analysis and dynamic analysis.

Static analysis. Costin et al. [6] proposed a static analysis framework that can implement firmware collection, filtering, unpacking, and large-scale analysis. The framework examines the firmware information extracted from each firmware sample to determine whether it contains a private encryption key or a string of known errors.

Firmalice [7] is a symbolic analysis system used to analyze the binary code in the complex firmware of different hardware platforms and automatically identify the authentication bypass vulnerability that occurs. It is built on the Angr [8] symbolic execution engine. Firmalice tries to find the path to the privileged program point and performs authentication bypass checks on the successfully reached symbol status.

Dynamic analysis. Firmadyne is a framework dedicated to dynamically analyzing vulnerabilities in Linux-based embedded firmware. A script based on Binwalk [9] API effectively implements the extraction of the file system and optional kernel. In the simulation phase, Firmadyne uses a prepared kernel and a file system extracted from the firmware for initial simulation on the QEMU [10] simulator. In this learning process, it will continuously modify the network configuration of QEMU.

Avatar2 is a dynamic multi-target orchestration framework. Compared to Avatar using S2E [11], Avatar2 is a completely redesigned system. Each target abstracts the endpoint through the protocol and provides a high-level interface for the Avatar2 kernel. In the end, Avatar2 integrated five targets: GDB [12], OpenOCD [13], QEMU, PANDA [14], and Angr.

IoTFuzzer [15] uses the mobile App of IoT devices to design a black box fuzzing tool to avoid such problems and analyze memory error vulnerabilities on IoT devices. In order to identify those data and the content of the message to be sent to the IoT device, the data flow is tracked from selected elements to determine the content that affects certain message fields. Then, these data will be changed to the content of the field for fuzzing.

FirmFuzz [16] provides a device-independent automated simulation and dynamic analysis framework for Linux-based firmware images. It uses a graybox-based generational fuzzing method, combined with static analysis and system introspection to provide targeted and deterministic vulnerability discovery in firmware images.

FirmAFL realizes an augmented process emulation method by combining system mode simulation and user mode simulation. This new simulation method can greatly improve the efficiency of simulation execution. At the same time, FirmAFL incorporates the traditional AFL [17] into the firmware, enabling it to support augmented process emulation, and specify the input location of test cases through the virtual machine introspection system.

3 Overview

3.1 Background

Fuzzing is considered the most effective method of vulnerability analysis. However, in the current firmware vulnerability detection tools, there are fewer tools for fuzzing. There are common problems that are single in form and cannot be analyzed in depth. This is mainly limited by the particularity of the firmware itself. We believe that the main differences between firmware and ordinary binary programs are: complex format, diverse architecture, and difficulty in running [18]. These reasons all make the common fuzzing tools unable to be applied to the firmware. In addition, vulnerabilities in the firmware can usually only be exploited through various protocol ports opened by the device, which also increases the difficulty of fuzzing.

Fuzzing techniques include mutation-based fuzzing (such as AFL, LibFuzzer [19]) and generation-based fuzzing (such as Peach [20], boofuzz). The former can better generate test cases and judge program execution. However, it lacks compatibility with devices and is basically incapable of special architecture and systems. At the same time, without format constraints, test cases are difficult to accept. The latter can generate well-formed test cases. But the lack of feedback information, it belongs to the traversal of all possibilities in the black box situation. Based on the above situation, we believe that the generation-based fuzzing method is easier to combine with firmware analysis. Because it will not be affected by the architecture, nor will it be restricted by the format. Defining a feedback message for a certain type of firmware to guide the fuzzing can solve its own shortcomings.

3.2 Architecture

Based on the above problems, we design and implement an efficient firmware fuzzing tool–PS-Fuzz. Its architecture is shown in Fig. 1. PS-Fuzz mainly includes two modules: protocol state instrumentation and protocol state guided fuzzing. In order to solve the problem of the lack of guidance in common protocol fuzzing and the inability to judge whether the test cases are good or bad, PS-Fuzz uses a more direct and effective method to monitor the impact of test cases on program execution in real time, thereby simplifying the process of fuzzing fields and achieving faster fuzzing. The protocol state instrumentation module completes the preanalysis of the firmware and program instrumentation. Finally, a state detector collects the feedback information of each instrumentation point. This information will be sent to the state checker of the protocol state guided fuzzing module for state judgment. The fuzzing controller selects a new mutation strategy based on the feedback information, and further controls the mutation engine and the format checker to complete the generation of test cases. This not only makes the path of program execution deeper, but also makes the discovery of vulnerabilities faster. In addition, PS-Fuzz also includes a firmware emulator for maintaining the firmware operating environment, a monitor and a logger. More importantly, PS-Fuzz is not only aimed at discovering vulnerabilities, it will also feedback a large amount of system information to users for further research.

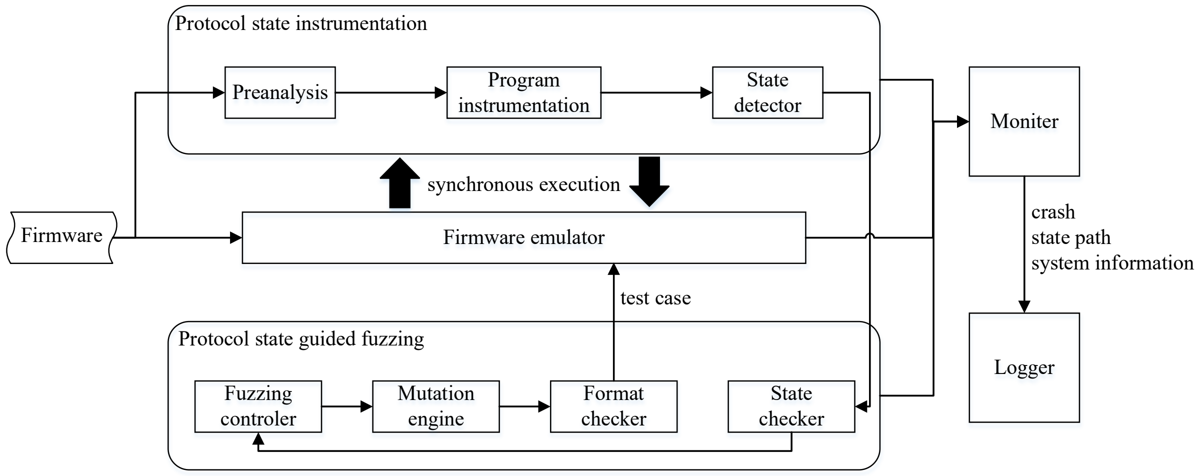


Figure 1: PS-fuzz workflow

4 Design and Implementation

4.1 Protocol State Instrumentation

The core function of the protocol state instrumentation module is to complete program execution and analysis. It is the main source of fuzzing guidance information and one of the channels for obtaining program execution information. This module mainly includes preanalysis, program instrumentation and state detector.

4.1.1 Preanalysis

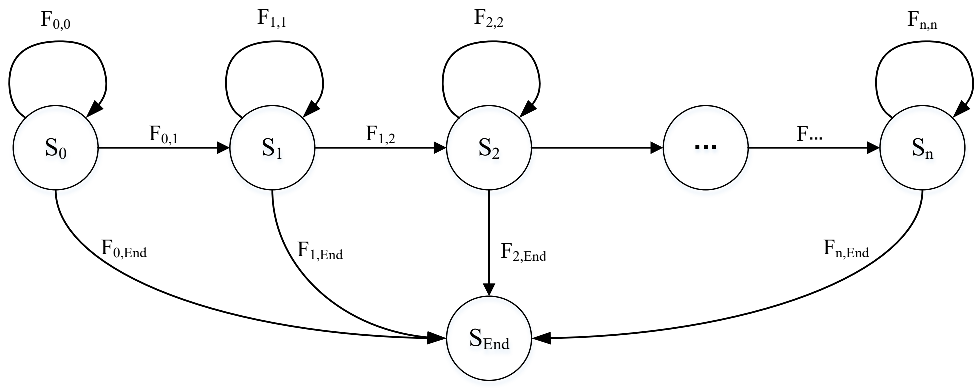
The preanalysis has two main steps: protocol format acquisition and firmware program analysis. In the protocol format acquisition step, we need to get as many data packets as possible to the emulation device, so as to obtain the format of the data packets that the firmware can accept. The correct protocol format is used for subsequent format checking of the generated test cases to ensure that each test is valid. After obtaining the protocol format, we have to enter the firmware program analysis step. In this step, we need to find a program used to process data packets in the firmware file system, and analyze the program in the next process. This is because vulnerabilities that cause firmware security threats are usually triggered by data packets. Therefore, the program that processes the data packet is the most likely to have vulnerabilities and the most likely to be exploited in actual situations. The main function of preanalysis is to provide PS-Fuzz with the correct protocol format and firmware programs that need to be fuzzed. It determines the general fuzzing direction for PS-Fuzz.

4.1.2 Program Instrumentation

Program instrumentation needs to find the most suitable point in the program for instrumentation according to the protocol format provided by the preanalysis and firmware program. Our idea is to segment the correct protocol format by field. Then find the position of the function that processes each field in the program. These functions will produce obvious branching phenomenon after processing fields. When a function finds that the data in the field it is processing is abnormal, it will immediately jump to the abnormal state and end the processing of the data packet. In addition, the order of these processing functions is obvious. This means that the subsequent field processing is skipped, that is to say, the current test case does not reach deeper in the program. This is why it can be the main criterion for judging guidance. It is simple to use IDA to find these field-processing functions in the program, because these functions must store the field name in the register before processing the field. We can use this to quickly find the addresses of these functions in the program. It should be noted that whether some fields exist or whether there is content in the fields does not affect the processing of the data packet. We need to exclude these processing functions to avoid unnecessary instrumentation. Finally, we instrumented each valid field processing function, and tracked which branch it went to after the end of the run, and collected the execution information of the program.

4.1.3 State Detector

PS-Fuzz defines each instrumentation point as a state. And from one instrumentation point to another, we call it state transition. Fig. 2 illustrates the basic process of state transition in PS-Fuzz. The state set $S = \langle S_0, S_1, S_2, \dots, S_n, S_{End}\rangle$ is the state set of a certain firmware program. The set of protocol field information required from one state to another state is $F = \{F_{i,j}, 0 \le i, j \le n\}$. For a state S_i, if the program wants to further reach the state S_j, it must add or satisfy the corresponding field information $F_{i,j}$ in the new test case. If it is not satisfied, the program will jump to the state S_{End}. Obviously, each test case will generate a corresponding state transition path. This is the complete execution flow of a test case in the program. According to the state transition path, we can clearly understand the depth of program execution, thereby judging the quality of test cases. It should be noted that, in order to further improve the efficiency, the state transition of PS-Fuzz is designed to only transition in a deeper direction, but not in the opposite direction. We will explain the content of this part later.



$$\begin{tikzcd}
S_0 \arrow[loop above, "F_{0,0}"] \arrow[r, "F_{0,1}"] \arrow[drr, bend right, "F_{0,End}"] & S_1 \arrow[loop above, "F_{1,1}"] \arrow[r, "F_{1,2}"] \arrow[dr, bend right, "F_{1,End}"] & S_2 \arrow[loop above, "F_{2,2}"] \arrow[r] \arrow[d, "F_{2,End}"] & \cdots \arrow[r, "F\dots"] & S_n \arrow[loop above, "F_{n,n}"] \arrow[dll, bend left, "F_{n,End}"'] \\
 & & S_{End} & &
\end{tikzcd}$$

Figure 2: Protocol state transition driven by field information

4.2 Protocol State Guided Fuzzing

The protocol state guided fuzzing module is constructed by adding guidance information to the common protocol fuzzing architecture. It includes all the common functions of fuzzing. This module mainly includes fuzzing controller, mutation engine, format checker and state checker.

4.2.1 State Checker

The state checker corresponds to the state detector in the protocol state instrumentation module. It is used to receive the state transition path information sent by the state detector. And compare with the state transition path generated by the previous test case. In order to determine whether the current test case triggers a new state, whether the mutation strategy can be updated according to the new state. When the state checker thinks that the answers to the above questions are all yes, it will send the new state information to the fuzzing controller for processing.

4.2.2 Fuzzing Controller

The fuzzing controller is the brain of the protocol state guided fuzzing. It stores a table, which we call the state strategy table. In the state strategy table, each state has a corresponding mutation strategy. When the fuzzing controller receives the state information from the state checker, it will look for the corresponding mutation strategy in the state strategy table according to the latest triggered state. It will follow this new mutation strategy in the next fuzzing process. In terms of defining mutation strategy, we think that when the program reaches a new state S_i, it means all the fields in front of the corresponding field information of S_i in protocol format $(F_{1,2}, F_{2,3}, \dots, F_{i-1,i})$ have met the conditions of program execution. These fields have already completed the mutation process or do not need to be mutated at all. In order to run the program more in-depth, it should mutate the following fields $(F_{i,i+1}, F_{i+1,i+2}, \dots, F_{n-1,n})$ as much as possible. Therefore, our mutation strategy for the state Si is to keep the $F_{1,2}, F_{2,3}, \dots, F_{i-1,i}$ field information unchanged, and use $F_{i,i+1}, F_{i+1,i+2}, \dots, F_{n-1,n}$ as the main mutation objects. At the same time, we also need to define the field types of $F_{i,i+1}, F_{i+1,i+2}, \dots, F_{n-1,n}$, which is to make the mutation engine work better. At this time, if a reverse jump is performed, the previously fixed field has to be mutated again, which is obviously redundant. This is also the main reason why PS-Fuzz is designed to be unable to perform reverse state transitions. In this way, fields that have been mutated and do not need to be mutated at all in $F_{1,2}, F_{2,3}, \dots, F_{i-1,i}$ are skipped, thereby greatly improving efficiency.

4.2.3 Mutation Engine

The mutation engine will mutate according to the mutation strategy currently in use. It usually maintains a mutation database. According to the different definitions of each field type in our mutation strategy, the mutation engine will traverse all the malformed data under this type until the fuzzing process of the field is completed. This mutation database comes with boofuzz, which contains a lot of malformed data that can trigger common protocol vulnerabilities. But for PS-Fuzz, if a new state cannot be triggered, using traversal fuzzing for invalid fields is equivalent to a waste of time. And we cannot guarantee that the new state will be reached after the traversal is over. And boofuzz's mutation database is obviously not so easy to trigger new states. So we added some necessary fields to boofuzz's mutation database to ensure that it will enter a new state after a period of mutation. The mutation engine will complete the generation of test cases and pass the test cases to the format checker.

4.2.4 Format Checker

The job of the format checker is very simple. It will get the correct protocol format extracted in the preanalysis stage in the protocol state instrumentation module. The correct protocol format will be accurately divided into fields here. The test cases generated by the mutation engine will also be divided into fields. The format checker will compare the two data packets field by field to determine whether the test case can be accepted by the emulation device. Finally, the format checker will send the test cases that successfully pass the check to the emulation device through the socket to complete a test.

4.3 Monitor

The monitor needs to monitor three parts: emulation device, protocol state instrumentation module and protocol state guided fuzzing module. PS-Fuzz calls all information related to the crash that can be obtained as system information I_S. After boofuzz can get the crash feedback, it will record all the test information in a fuzzing database. It includes the initial interaction information between each test case and the emulation device, the content of the data packet sent, and the feedback information after sending. We define this information as boofuzz information I_B. In addition, the GDB used by the protocol state instrumentation module can also provide us with rich information. This is because there is a gdbserver in the firmware emulator. Gdbserver can be connected to the GDB in the protocol status instrumentation module to realize the synchronous execution of the firmware program. In other words, in the protocol state instrumentation module, we can obtain exactly the same information as in the emulation device. We define this information as GDB information I_G. I_G mainly contains information from common GDB analysis, including memory information, stack information, register information, program execution, and so on. This information is usually obtained by manually entering multiple commands. PS-Fuzz uses the pwndbg tool to integrate all crash information and provide intuitive feedback. The entire information acquisition process does not require user involvement. The last is the crash information that the QEMU system simulation mode itself can provide I_Q, so we can get Eq. (1). PS-Fuzz provides more system information than most protocol fuzzing tools, and it can provide help for firmware vulnerability mining and further firmware research from different angles.

$$I_S = I_B + I_G + I_Q \quad (1)$$

5 Experimental Evaluation

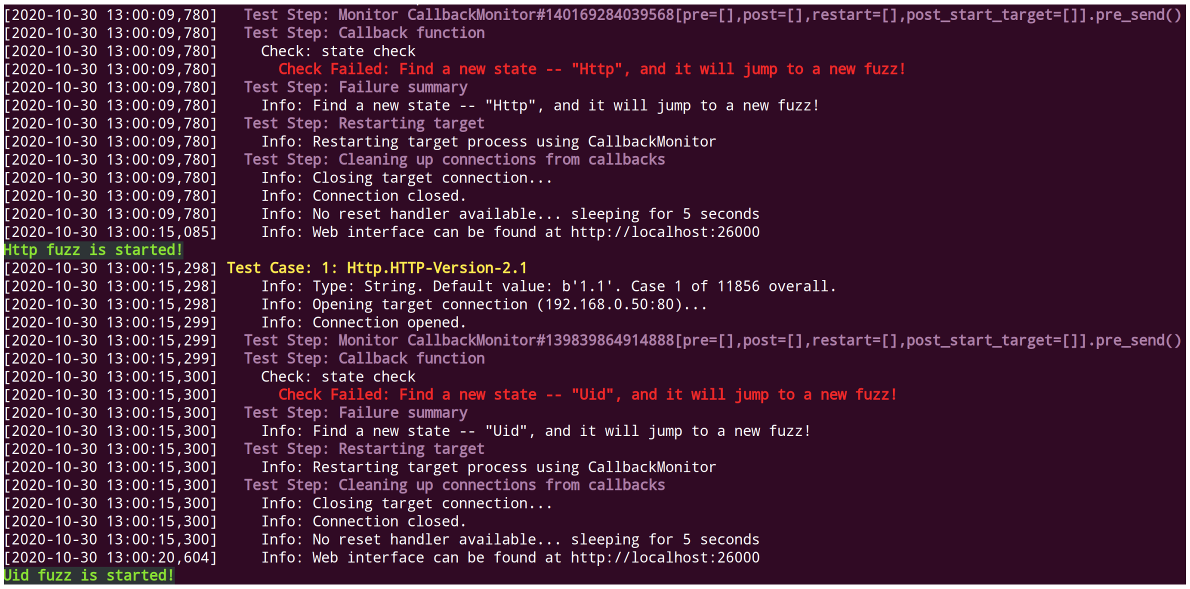
We evaluate PS-Fuzz through a typical firmware vulnerability. D-Link's DAP-2695 router has a buffer overflow vulnerability CVE-2016-1558 in its firmware. The vulnerability is due to the firmware not checking the length of the dlink_uid parameter when processing HTTP packets. As a case study, we will discuss the vulnerability and how it was detected. We will also evaluate the efficiency of vulnerability detection and discuss the system information that can be obtained when the vulnerability is triggered. We evaluated PS-Fuzz on a virtual machine running Ubuntu 16.04 with two processors and 4 GB of RAM. In the firmware emulator, PS-Fuzz uses QEMU version 2.5.0 as the backend of fuzzing. In addition, PS-Fuzz uses GDB version 9.2 for instrumentation. And make changes on boofuzz version 0.2.0 to fuzzing the emulation device.

5.1 Experimental Steps

First use Firmadyne to simulate the firmware. After the emulation device is started, we can use some packet capture tools to capture the data packets in the process of interacting with its Web page to get the correct protocol format and necessary fields. Then we can use Binwalk to extract the complete file system of the firmware and find the program for processing HTTP packets for preanalysis. After getting the instrumentation points and mutation strategy, PS-Fuzz can automatically fuzz the emulation device. We mainly evaluate the efficiency and amount of information of PS-Fuzz. In order to achieve evaluation, we use boofuzz for comparison.

5.2 Efficiency

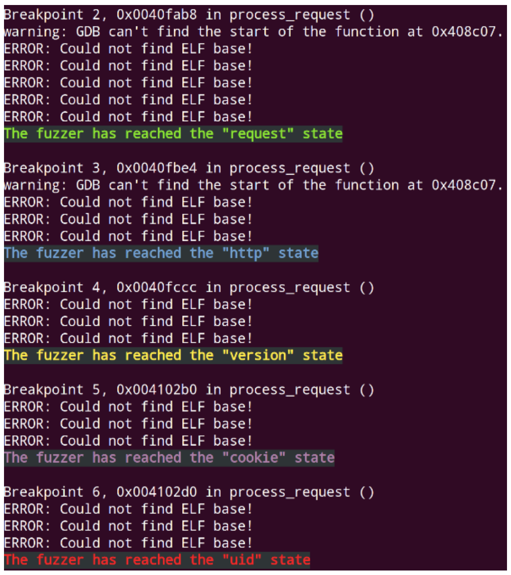
As shown in the Fig. 3, when PS-Fuzz starts to run, it will quickly transfer its state. The state triggered by each test case will be displayed on the terminal in real time as shown in the Fig. 4. In the end, PS-Fuzz can quickly find vulnerabilities and give prompts on both the emulator side and the fuzzing side, as shown in the Fig. 5 and Fig. 6. However, boofuzz cannot be found under normal circumstances even if the vulnerability is triggered. After adding a monitor to it, we compared the efficiency of the two.



```
[2020-10-30 13:00:09,780]   Test Step: Monitor CallbackMonitor#140169284039568[pre=[],post=[],restart=[],post_start_target=[]].pre_send()
[2020-10-30 13:00:09,780]   Test Step: Callback function
[2020-10-30 13:00:09,780]     Check: state check
[2020-10-30 13:00:09,780]       Check Failed: Find a new state -- "Http", and it will jump to a new fuzz!
[2020-10-30 13:00:09,780]   Test Step: Failure summary
[2020-10-30 13:00:09,780]     Info: Find a new state -- "Http", and it will jump to a new fuzz!
[2020-10-30 13:00:09,780]   Test Step: Restarting target
[2020-10-30 13:00:09,780]     Info: Restarting target process using CallbackMonitor
[2020-10-30 13:00:09,780]   Test Step: Cleaning up connections from callbacks
[2020-10-30 13:00:09,780]     Info: Closing target connection...
[2020-10-30 13:00:09,780]     Info: Connection closed.
[2020-10-30 13:00:09,780]     Info: No reset handler available... sleeping for 5 seconds
[2020-10-30 13:00:15,085]     Info: Web interface can be found at http://localhost:26000
Http fuzz is started!
[2020-10-30 13:00:15,298] Test Case: 1: Http.HTTP-Version-2.1
[2020-10-30 13:00:15,298]     Info: Type: String. Default value: b'1.1'. Case 1 of 11856 overall.
[2020-10-30 13:00:15,298]     Info: Opening target connection (192.168.0.50:80)...
[2020-10-30 13:00:15,299]     Info: Connection opened.
[2020-10-30 13:00:15,299]   Test Step: Monitor CallbackMonitor#139839864914888[pre=[],post=[],restart=[],post_start_target=[]].pre_send()
[2020-10-30 13:00:15,299]   Test Step: Callback function
[2020-10-30 13:00:15,300]     Check: state check
[2020-10-30 13:00:15,300]       Check Failed: Find a new state -- "Uid", and it will jump to a new fuzz!
[2020-10-30 13:00:15,300]   Test Step: Failure summary
[2020-10-30 13:00:15,300]     Info: Find a new state -- "Uid", and it will jump to a new fuzz!
[2020-10-30 13:00:15,300]   Test Step: Restarting target
[2020-10-30 13:00:15,300]     Info: Restarting target process using CallbackMonitor
[2020-10-30 13:00:15,300]   Test Step: Cleaning up connections from callbacks
[2020-10-30 13:00:15,300]     Info: Closing target connection...
[2020-10-30 13:00:15,300]     Info: Connection closed.
[2020-10-30 13:00:15,300]     Info: No reset handler available... sleeping for 5 seconds
[2020-10-30 13:00:20,604]     Info: Web interface can be found at http://localhost:26000
Uid fuzz is started!
```

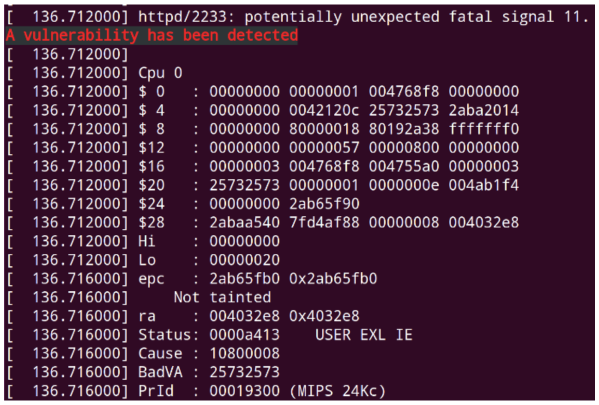
Figure 3: Fast state transition in PS-Fuzz

As shown in Tab. 1, since we have obtained the correct protocol format and know the parameters that trigger the vulnerability, a series of optimizations can be carried out when formulating the mutation strategy. The experimental results show that the time required for PS-Fuzz is much less than the time spent by boofuzz when the same vulnerability is found. In addition, we find that the efficiency of PS-Fuzz depends on the method of instrumenting. This requires a full understanding of the program execution process in the preanalysis stage. We did a lot of work in the preanalysis stage during the experiment. Under ideal circumstances, the efficiency of PS-Fuzz is about 8 times that of boofuzz. Even rough instrumentation efficiency can reach 2 times that of boofuzz.



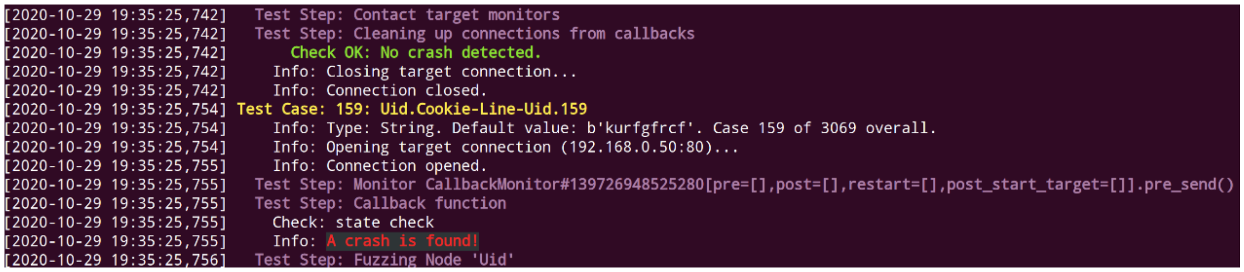
```
Breakpoint 2, 0x0040fab8 in process_request ()
warning: GDB can't find the start of the function at 0x408c07.
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
The fuzzer has reached the "request" state

Breakpoint 3, 0x0040fbe4 in process_request ()
warning: GDB can't find the start of the function at 0x408c07.
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
The fuzzer has reached the "http" state

Breakpoint 4, 0x0040fccc in process_request ()
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
The fuzzer has reached the "version" state

Breakpoint 5, 0x004102b0 in process_request ()
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
The fuzzer has reached the "cookie" state

Breakpoint 6, 0x004102d0 in process_request ()
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
ERROR: Could not find ELF base!
The fuzzer has reached the "uid" state
```

Figure 4: Discover the state triggered by each test case



```
[  136.712000] httpd/2233: potentially unexpected fatal signal 11.
A vulnerability has been detected
[  136.712000]
[  136.712000] Cpu 0
[  136.712000] $ 0   : 00000000 00000001 004768f8 00000000
[  136.712000] $ 4   : 00000000 0042120c 25732573 2aba2014
[  136.712000] $ 8   : 00000000 80000018 80192a38 fffffff0
[  136.712000] $12   : 00000000 00000057 00000800 00000000
[  136.712000] $16   : 00000003 004768f8 004755a0 00000003
[  136.712000] $20   : 25732573 00000001 0000000e 004ab1f4
[  136.712000] $24   : 00000000 2ab65f90
[  136.712000] $28   : 2abaa540 7fd4af88 00000008 004032e8
[  136.712000] Hi    : 00000000
[  136.712000] Lo    : 00000020
[  136.716000] epc   : 2ab65fb0 0x2ab65fb0
[  136.716000]     Not tainted
[  136.716000] ra    : 004032e8 0x4032e8
[  136.716000] Status: 0000a413    USER EXL IE
[  136.716000] Cause : 10800008
[  136.716000] BadVA : 25732573
[  136.716000] PrId  : 00019300 (MIPS 24Kc)
```

Figure 5: Display on the emulation device end when the crash occurs



```
[2020-10-29 19:35:25,742]   Test Step: Contact target monitors
[2020-10-29 19:35:25,742]   Test Step: Cleaning up connections from callbacks
[2020-10-29 19:35:25,742]       Check OK: No crash detected.
[2020-10-29 19:35:25,742]     Info: Closing target connection...
[2020-10-29 19:35:25,742]     Info: Connection closed.
[2020-10-29 19:35:25,754] Test Case: 159: Uid.Cookie-Line-Uid.159
[2020-10-29 19:35:25,754]     Info: Type: String. Default value: b'kurfgfrcf'. Case 159 of 3069 overall.
[2020-10-29 19:35:25,754]     Info: Opening target connection (192.168.0.50:80)...
[2020-10-29 19:35:25,755]     Info: Connection opened.
[2020-10-29 19:35:25,755]   Test Step: Monitor CallbackMonitor#139726948525280[pre=[],post=[],restart=[],post_start_target=[]].pre_send()
[2020-10-29 19:35:25,755]   Test Step: Callback function
[2020-10-29 19:35:25,755]     Check: state check
[2020-10-29 19:35:25,755]     Info: A crash is found!
[2020-10-29 19:35:25,756]   Test Step: Fuzzing Node 'Uid'
```

Figure 6: Display on the fuzzing end when the crash occurs

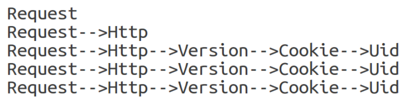
Table 1: Comparison of the time of finding the same vulnerability with two different instrumentation methods of PS-Fuzz and boofuzz

Tool	Boofuzz	PS-Fuzz (Rough instrument)	PS-Fuzz (Accurate instrument)
Time	8 min 34 s	3min 47s	58s

5.3 Amount of Information

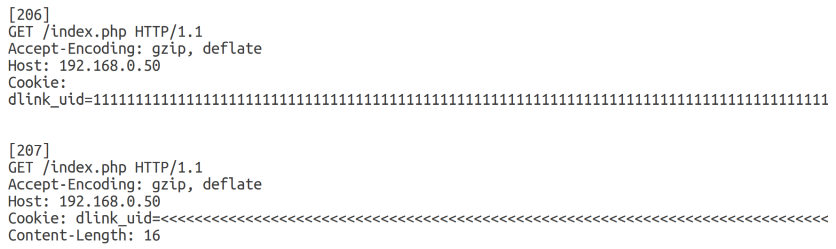
After the fuzzing is over, boofuzz will record the information of each test in a fuzzing database. It includes the initial interaction information between each test case and the emulation device, the content of the data packet sent, and the feedback information after sending.

In addition to the information that boofuzz itself can provide, PS-Fuzz can also provide a large amount of system information. As shown in the Fig. 7, PS-Fuzz will record the state transition path of each test case. At the same time, PS-Fuzz will immediately record the information provided by the firmware simulator and the protocol state instrumentation module after the vulnerability is triggered. This mainly includes simulation system information, register information, stack information, and command execution information. In addition, since boofuzz cannot identify the crash of the simulated device, it cannot record the data packet that caused the crash. PS-Fuzz can accomplish this job well, as shown in the Fig. 8.



```
Request
Request-->Http
Request-->Http-->Version-->Cookie-->Uid
Request-->Http-->Version-->Cookie-->Uid
Request-->Http-->Version-->Cookie-->Uid
```

Figure 7: Log state transition information



```
[206]
GET /index.php HTTP/1.1
Accept-Encoding: gzip, deflate
Host: 192.168.0.50
Cookie:
dlink_uid=111111111111111111111111111111111111111111111111111111111111111111111111111111111111111


[207]
GET /index.php HTTP/1.1
Accept-Encoding: gzip, deflate
Host: 192.168.0.50
Cookie: dlink_uid=<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Content-Length: 16
```

Figure 8: Log the packets that caused the crash

Finally, we compared the information provided by PS-Fuzz and boofuzz to get Tab. 2. From the table, we can clearly see that PS-Fuzz far exceeds boofuzz in its ability to collect and provide information.

Table 2: Comparison of the information obtained by boofuzz and PS-Fuzz after the crash is found

Information	**Boofuzz**	**PS-Fuzz**
Interactive information	√	√
Data packets	√	√
Feedback information	√	√
State transition paths	×	√
Simulation system	×	√
Registers	×	√
Stack	×	√
Instructions	×	√
Crash packets	×	√

6 Conclusion

In this paper, we propose PS-Fuzz, a gray-box fuzzing technology that detects embedded firmware vulnerabilities by comparing protocol state transition paths. On the basis of the traditional protocol fuzzing, we can perform instrumentation in the firmware program according to the protocol field state. And by collecting synchronization information from different modules to guide PS-Fuzz for fuzzing. So as to improve the efficiency of fuzzing test, and provide help for further firmware research. The experimental results on the actual embedded firmware show that for the same firmware vulnerability, PS-Fuzz has a significant efficiency improvement under different instrumentation situations, and the efficiency is 2-8 times that of boofuzz. PS-Fuzz not only has a significant increase in the speed of finding vulnerabilities, it can also provide a large amount of crash system information. Compared with traditional fuzzing tools, PS-Fuzz can provide at least 6 additional system internal information. Our current method is limited by the inability to fully automate the preanalysis stage. And We plan to further expand the versatility of PS-Fuzz by optimizing the instrumentation method to help the security protection of embedded devices.

Funding Statement: This work is funded by the National Key Research and Development Plan (Grant No. 2018YFB0803504), the National Natural Science Foundation of China (Nos. 62072130, 61702223, 61702220, 61871140, 61872420, U1636215), the Guangdong Province Key Area R&D Program of China (No. 2019B010137004), the Guangdong Basic and Applied Basic Research Foundation (No. 2020A1515010450), Guangdong Province Universities and Colleges Pearl River Scholar Funded Scheme (2019), and the Opening Project of Shanghai Trusted Industrial Control Platform (TICPSH202003014-ZC).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] D. D. Chen, M. Egele, M. Woo and D. Brumley, “Towards automated dynamic analysis for linux-based embedded firmware,” in *Proc. of 23rd Annual Network and Distributed System Security Sym.*, San Diego, CA, USA, vol. 1, pp. 1–16, 2016.

[2] J. Zaddach, L. Bruno, D. Balzarotti and A. Francillon, “Avatar: a framework to support dynamic security analysis of embedded systems’ firmware,” in *Proc. of 21st Annual Network and Distributed System Security Sym.*, San Diego, CA, USA, vol. 23, pp. 1–16, 2014.

[3] M. Muench, D. Nisi, A. Francillon and D. Balzarotti, “Avatar2: a multi-target orchestration platform,” in *Proc. of the Workshop Binary Analysis Research*, San Diego, CA, USA, vol. 18, pp. 1–11, 2018.

[4] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu *et al.,* “Firm-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation,” in *Proc. of the 28th USENIX Conf. on Security Sym.*, Berkeley, CA, USA, pp. 1099–1114, 2019.

[5] J. Pereyda, “Boofuzz: network protocol fuzzing for humans,” 2017. [Online]. Available: https://github.com/jtpereyda/boofuzz.

[6] A. Costin, J. Zaddach, A. Francillon and D. Balzarotti, “A large-scale analysis of the security of embedded firmwares,” in *Proc. of the 23rd USENIX Conf. on Security Sym.*, San Diego, CA, USA, pp. 95–110, 2014.

[7] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel and G. Vinga, “Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware,” in *Proc. of 22nd Annual Network and Distributed System Security Sym.*, San Diego, CA, USA, vol. 1, 2015.

[8] F. Wang and Y. Shoshitaishvili, “Angr-the next generation of binary analysis,” in *Proc. of 2017 IEEE Cybersecurity Development,* Cambridge, MA, USA, pp. 8–9, 2017.

[9] C. Heffner, “Binwalk: firmware analysis tool,” 2010. [Online]. Available: https://code. google. com/p/binwalk/.

[10] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *Proc. of the 14th USENIX Annual Technical Conf.*, Anaheim, CA, USA, vol. 41, pp. 46, 2005.

[11] V. Chipounov, V. Kuznetsov and Candea G, "S2e: a platform for in-vivo multi-path analysis of software systems," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[12] G. D. B. Developers, "GDB: the GNU project debugger," 2017. [Online]. Available: https://www. gnu. org/software/gdb.

[13] H. Högl, D. Rath and D. De, "OpenOCD-Open on-chip debugger" 2006.

[14] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek and R. Whelan, "Repeatable reverse engineering with PANDA," in *Proc. of the 5th Program Protection and Reverse Engineering Workshop*, New York, NY, USA, pp. 1–11, 2015.

[15] J. Chen, W. Diao, Q. Zhao, C. Zuo and K. Zhang, "Iotfuzzer: discovering memory corruptions in IoT through app-based fuzzing," in *Proc. of the Network and Distributed System Security Sym.*, San Diego, CA, USA, 2018.

[16] P. Srivastava, H. Peng, J. Li, H. Okhravi and M. Payer, "Firmfuzz: automated IoT firmware introspection and analysis," in *Proc. of the 2nd Int. ACM Workshop on Security and Privacy for the Internet-of-Things*, New York, NY, USA, pp. 15–21, 2019.

[17] M. Zalewski, "American fuzzy lop," 2015. [Online]. Available: http://lcamtuf. coredump. cx/afl.

[18] X. Li, L. Qiao, Y. Sun and Q. Guan, "Research on automated vulnerability mining of embedded system firmware," in *Proc. of Int. Conf. on Artificial Intelligence and Security*, Hohhot, China, vol. 1254, pp. 105–117, 2020.

[19] K. Serebryany, "Libfuzzer: a library for coverage-guided fuzz testing (within llvm)." [Online]. Available: https://github.com/Dor1s/libfuzzer-workshop.

[20] M. Eddington, "Peach fuzzing platform," 2011. [Online]. Available: https://community.peachfuzzer.com/.