Received: 19 November 2019 | Revised: 16 January 2020 | Accepted: 22 February 2020

DOI: 10.1002/cpe.5756

WILEY

SPECIAL ISSUE PAPER

Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware

Zicong Gao[1] | Weiyu Dong[1] | Rui Chang[2,3] | Yisen Wang[1]

[1]Cyberspace Security Department, State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China

[2]College of Computer Science and Technology, Zhejiang University, Hangzhou, China

[3]62 Science AvenueZhengzhou City, Henan Province,China

Correspondence
Rui Chang, College of Computer Science and Technology, Zhejiang University, Hangzhou, China.
Email: crix1021@zju.edu.cn

Summary

Fuzzing is an effective approach to detect software vulnerabilities utilizing changeable generated inputs. However, fuzzing the network protocol on the firmware of IoT devices is limited by inefficiency of test case generation, cross-architecture instrumentation, and fault detection. In this article, we propose the Fw-fuzz, a coverage-guided and crossplatform framework for fuzzing network services running in the context of firmware on embedded architectures, which can generate more valuable test cases by introspecting program runtime information and using a genetic algorithm model. Specifically, we propose novel dynamic instrumentation in Fw-fuzz to collect the running state of the firmware program. Then Fw-fuzz adopts a genetic algorithm model to guide the generation of inputs with high code coverage. We fully implement the prototype system of Fw-fuzz and conduct evaluations on network service programs of various architectures in MIPS, ARM, and PPC. By comparing with the protocol fuzzers Boofuzz and Peach in metrics of edge coverage, our prototype system achieves an average growth of 33.7% and 38.4%, respectively. We further verify six known vulnerabilities and discover 5 0-day vulnerabilities with the Fw-fuzz, which prove the validity and utility of our framework. The overhead of our system expressed as an additional 5% of memory growth.

KEYWORDS

code coverage, firmware, fuzzing, instrumentation, security

1 | INTRODUCTION

With the rapid growth of the IoT, the IoT environments suffer significant security threat from three layers: perception layer, network layer, application layer.[1] Many researchers have made great efforts to develop more secure and trusted IoT environments. In the application layer, malicious application detection and analysis on Android[2,3] and IOS[4,5] is a hot spot for researchers. As for the network layer, preventing attacks from inside[6] and outside[7,8] of the network is the focus of research. As the low-level layer of IoT, the perception layer is composed of a large number of devices, in which various methods for preventing attacks on the devices are proposed[9-11] in recent years. However, the connection part between the application layer and the perception layer lacks concerns.

In the context of the IoT scenario, the firmware is a class of software existed in devices and provides necessary instructions to connect the low-level hardware with high-level applications. The vulnerabilities in firmware program for network protocol processing are a serious attack surface.[12-14] These programs are usually implemented in languages such as C and C++. And their process of handling protocols does not follow the protocol specification, which results in lots of vulnerabilities.

Fuzzing[15] is an effective software testing technique for finding vulnerabilities by providing complicated inputs to a program. Fuzzing can be divided into three categories: blackbox fuzzing, whitebox fuzzing, and greybox fuzzing.[16] Blackbox fuzzers randomly generate inputs to target program with no internal inspection inside the program, while the whitebox fuzzers generate inputs based on knowledge of the structure of the program (usually needs source code). Greybox fuzzers, which are in middle, observe limited information from the program by utilizing lightweight code instrumentation. Recent years, a greybox fuzzer based on code coverage and heuristic rules named AFL[17] makes great success in practice and in the research community. Many advanced fuzzing are further developed on the basis of AFL.[18-22]

However, a few advanced fuzzers are applied for protocol fuzzing on firmware of embedded system. On the one hand, as Marius Muench[23] concluded, there are challenges of fuzzing firmware: fault detection, performance axsnd scalability, and instrumentation. These challenges limit the state-of-art greybox fuzzing techniques to gain the insight information of the firmware program to guide the generation of input cases. Some researchers attempt to fuzz the firmware in full emulation[24,25] or partial emulation environment.[26] However, the data flow of the simulated execution is frequently different from the actual execution, which may fail to capture errors in the firmware program. And the type of architecture supported by the simulation is limited by Qemu.[27]

On the other hand, current protocol fuzzing approaches such as Peach[28] and Boofuzz[29] are blackbox fuzzing, using rules defined by the users to generate inputs. DELTA,[30] SecFuzz,[31] and Ruiiter[32] pay more attention to generating test cases for encrypted protocols by different learning methods rather than improving the test cases of fuzzing based on the feedback information of the test program.

This article introduces the idea of code coverage-guided to the protocol fuzzing on firmware. We propose a universal firmware fuzzing framework Fw-fuzz, which is suitable for network service programs on different architecture firmware. By using a novel dynamic instrumentation on the physical device, the Fw-fuzz obtains the cross-architecture capability and can easily observe the running status of the firmware program. Once having the ability to get real-time information during firmware program running, the inefficient generation of fuzzing inputs can be improved.[33] We further build a genetic algorithm model based on code coverage to guide the generation of test cases.

The contributions of this article are as follows.

- We propose Fw-fuzz, a code coverage-guided fuzzing framework that can fuzz the network program on firmware in IoT device efficiently.
- We propose a dynamic firmware instrumentation approach for successfully obtaining the code coverage of the MIPS, ARM, and PPC architecture firmware programs.
- We establish a genetic algorithm model to guide the generation of inputs with high code coverage in limited time.
- We implement a prototype system of Fw-fuzz and evaluate the system on practical devices, discovering five unknown vulnerabilities.

The rest of the article is organized as follows. Section 2 takes a motivating example to point out the challenges of firmware protocol fuzzing. Section 3 provides an overview of Fw-fuzz design. Section 4 describes the novel dynamic firmware instrumentation. And Section 5 introduces our genetic algorithm model. We present the experimental result in Section 6. Related work is discussed in Section 7. We conclude and discuss directions for future work in Section 8.

2 | MOTIVATION

We reverse the program httpd in Netgear router and select a function in it as a motivating example to indicate the challenges of fuzzing firmware.

2.1 | Motivating example

The motivating example represents HTTP request processing and it is simplified for clarity and space reasons in Listing 1.

```
1  function http_handle(http_request, netaddress)
2  {
3        url = http_request;
4        s1 = http_request;
5        s = char[1024];
6        accept_language = strstr(http_request, "Accept_language:");
7        host_name = strstr(http_request, "Host:");
8        user_agent = strstr(http_request, "User_agent:");
```

```
9         if (host_name){
10              if strisstr(host_name, "www.routerlogin.net"){
11                    func1();
12              }
13        }
14        if (user_agent){
15              if strstr(user_agent, "Firefox"){
16                    func2();
17              }
18        }
19        if(!strcmp(s1, "GET")){
20              sprintf(s,url);
21              if(strcmp("index.html", url_name)){
22                    func3();
23                    if((strstr(accept_language, "zh")){
24                          func4();
25                    }
26              }
27        }
28        else if (!strcmp(s1, "POST")){
29              func6();
30        }
31  }
```

Listing 1: A motivating example for the HTTP protocol processing in httpd

The example only handles three kinds HTTP request header, which is far from the HTTP RFC standard. In this case, using the blackbox protocol fuzzers such as Boofuzz, Peach consumes a lot of time to traverse all request headers to mutate because the template for the test cannot be provided for these format-based protocol fuzzing without knowing the internal structure of httpd.

Then consider the checks from line 19 to line 23. The greybox fuzzer such as AFL fails to bypass the three round of magic number guard. Driller[34] uses symbolic execution to generate constraint for branch. However, strstr() function returns the pointer to the first character of the substring, which makes constraints are not strict enough. T-fuzz[21] addresses the motivating example by using program transformation to modify the origin program to remove the complicated magic number. Redqueen[35] observes the input-to-state correspondence of the program to addresses magic number and nested checksum. However, these greybox fuzzing relies on the instrumentation, which is not applicable to RISC architecture firmware such as PPC, MIPS.

Principled searching used by Angora[36] can also solve the motivating example. The whitebox fuzzers need the source code to be implemented, while firmware is closed source.

2.2 | Challenges

- ***Inefficiency of test case generation*:** The protocol processing on firmware is not strictly implemented according to the RFC standard while the fields of the protocol are in a fixed format. Testing all protocol fields results in inefficient fuzzing. On the other hand, if the mutation of the test case is fixed, the fuzzing will fails to reach the deep inside of the program. The ideal input generation of firmware protocol fuzzing is not only based on a certain format specification but also should be adjusted dynamically based on the information obtained.
- ***Crossarchitecture instrumentation*:** The diverse architecture of the embedded system invalidates the traditional insertion mechanism[17,37] that is used in nowadays greybox fuzzing. In order to construct efficient input cases, the ability to crossarchitecture instrumentation is needed in the absence of firmware source code.
- ***Fault detection*:** As Marius mentioned in Reference [23], protection measurements are rarely present on firmware, resulting in the difficulties to observe crashes in embedded devices directly. The additional liveness check is needed to deploy to capture the exception during fuzzing.

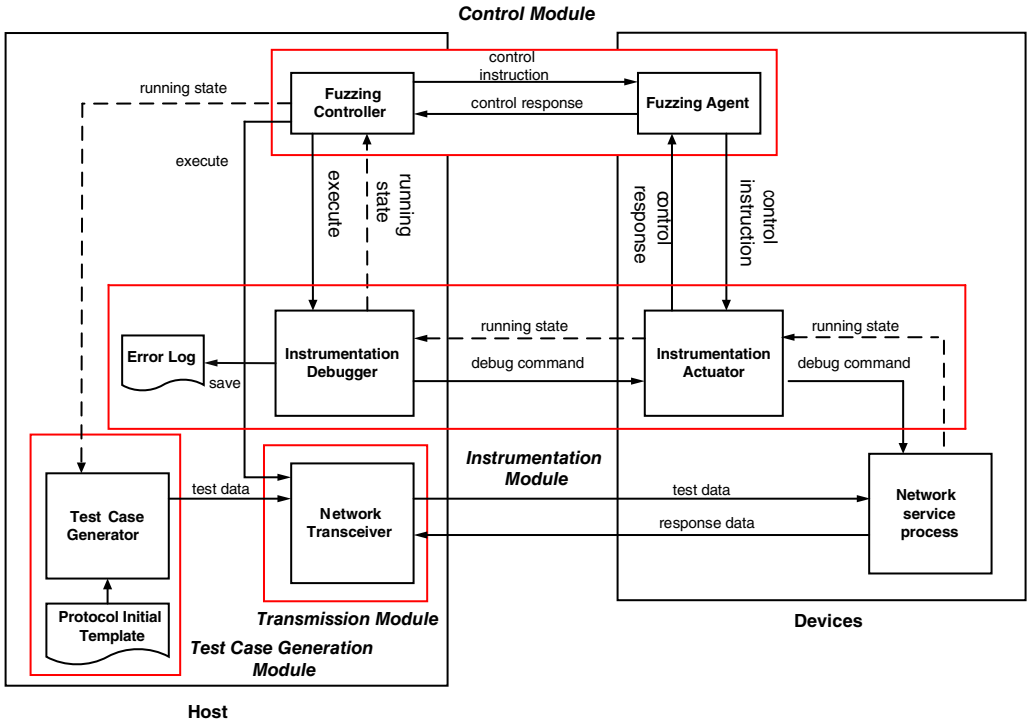


FIGURE 1 The components of Fw-fuzz framework and how the framework works

3 | DESIGN

To address the challenges mentioned above, we develop a code coverage-guided fuzzing framework Fw-fuzz, a vulnerability detection technique that is focused on testing the network service on firmware. The Fw-fuzz retains the efficiency of greybox fuzzing because the generation of inputs uses a genetic algorithm model based on code coverage. Fw-fuzz has the crossarchitecture capability to use for different kinds of IoT devices by its dynamic instrumentation approach.

3.1 | Main components

The overview of Fw-fuzz is shown in Figure 1, which consists of the following several parts:

The ***Control Module*** is responsible for scheduling other modules. Controller and agent work as a Client-Server mode to transmit control commands between the host and the device.

The ***Instrumentation Module*** includes instrumentation debugger and instrumentation actuator, which is responsible for instrumentation on the physical device. It feeds back the running state of the target process to the host by dynamic firmware instrumentation.

The ***Transmission Module*** is a network transceiver for sending and receiving network packets to the target program. It sends packets filled with fuzzing test cases under the scheduling of control module.

The ***Test Case Generation Module*** is located on the host, including the protocol template and the test case generator. The protocol template continually mutates under the guidance of the genetic algorithm model based on code coverage to construct new test cases.

3.2 | Work flow

The work flow of Fw-fuzz is shown below:

Step 1: Users define the protocol template file as the initial fuzzing input.

Step 2: The controller sends control instruction to the agent to notify the instrumentation actuator to attach the process of the target program on an embedded device.

Step 3: The controller starts the instrumentation debugger to remote connect with the Instrumentation actuator .

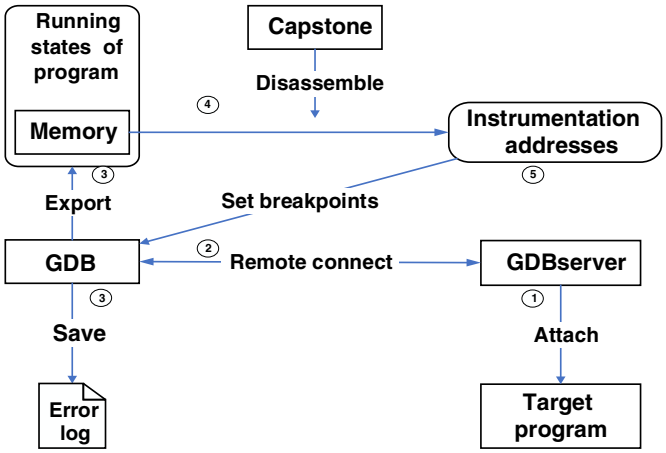


FIGURE 2 Schematic diagram of dynamic firmware instrumentation

Step 4: The Fw-fuzz starts a round of fuzzing by utilizing the transmission module to send the packet to target program.

Step 5: During the fuzzing, the instrumentation module monitors the target program and feeds back running information of the target program to the host.

Step 6: The crash or exception occurred during fuzzing will be caught by the instrumentation module. The error information is saved to log file in the host.

Step 7: The test case generation module mutates the template under the guiding of a genetic algorithm based on code coverage calculated from the running information in step 5.

Step 8: The Fw-fuzz is reset for the next fuzzing round and repeat step 5 to step 7.

4 | DYNAMIC FIRMWARE INSTRUMENTATION

The design of dynamic firmware instrumentation should meet the requirement to address the challenges in Section 2. So, our instrumentation needs to be suitable for various architectures firmware. Our instrumentation also has the capability to detect the fault occurring during the execution of the program. In addition, we design an optimization algorithm to mitigate the overhead of the instrumentation.

4.1 | Cross-architecture

We leverage GDB[38] and Capstone[39] to achieve cross-architecture. GDB is a powerful tool to debug applications running in linux system. By cross-compilation, it can run in server mode to remotely debug the program in different architectures. Capstone is a disassembly framework for binary analysis and reversing. Capstone supports multiarchitecture and is special support for embedding into firmware or OS kernel.

The workflow of the dynamic firmware instrumentation is shown in Figure 2. First, the target program is attached by GDBserver. Next, a connection is built between GDB and GDBserver in remote mode, where the program can be debugged with the inside functions of GDB. The program will pause by triggering the initial breakpoint offered by the user. In this case, the running state of the target program such as memory information, the value of registers, and the structure of stack and heap can be obtained with GDB during the debug. Then the Capstone disassembles the memory of the program to the assembly instructions. After theses above operations, Fw-fuzz sets breakpoints at some addresses and keeps the program continue to run. Once the program reaches the instrumented address, GDB could export the running information of the program again.

Our novel dynamic firmware instrumentation is easy to be expanded to firmware with a specific architecture by compiling the gdbserver on the target IoT device or using cross-compilation technology.

4.2 | Fault detection

GDB helps to poke around inside the programs while they are executing and allows seeing what exactly happens when the program crashes. By presetting the signal mechanism provided with GDB, Fw-fuzz automatically detects the fault occurred during fuzzing. The fault information is saved as the core dump file by GDB. The complicated fault and inexplicable errors can be further used for analysis or replay. It overcomes the challenges mentioned in Section 2.2 that crashes in firmware are hard to be observed.

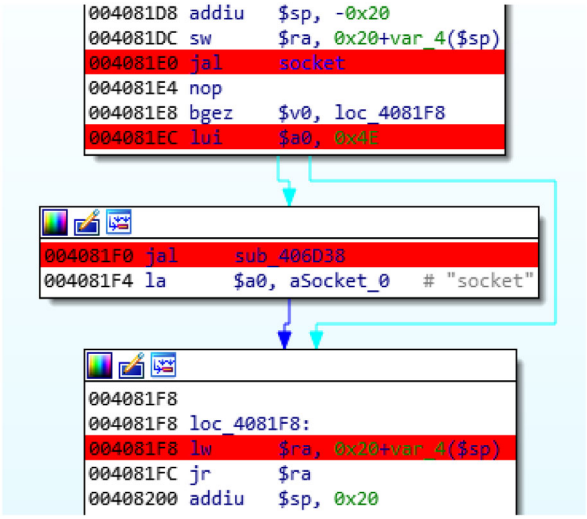


FIGURE 3 Instrumentation example

4.3 | Optimized instrumentation algorithm

An optimization algorithm is designed for minimizing the number of instrumentation, taking into account that instrumentation reduces the efficiency of fuzzing.

As Algorithm 1 shows, the function *Disassemble* first disassemble the binary code from the addresses of *pc* register (line 1) and get the assembly instruction set *I*. Then the loop (lines 2-19) is aimed to find the nearest branch to the current *pc* by walking through the instruction set *I*. When fuzzing the specific program, the user may not be interested in some area in the program. So, an omission mechanism is implemented to skip these *ommit_address* (lines 3-5). The function *is_branch* is used to determine whether the instruction *i* is the branch (line 6). If *i* is a branch, we first get the jump address of the branch *i* (line 7). Then the algorithm omits the jump addresses that are library functions (lines 8 and 9) since the fuzzing process is concentrated with the target program instead of the function from the dynamic link library. For a branch instruction, we could not predict whether it will jump or not before the program does not reach the branch address. Thus, algorithm instruments the next address of the branch instruction (line 11) and the jump address (lines 12-15). In addition, the algorithm builds a CFG (control flow graph), which uses the next jump address as the root node by function *CFGnode* (line 12). The algorithm outputs all the nodes in CFG and then instrument at the addresses of these nodes (lines 13-15). Preinstrumentation takes place here is used for reducing the times of loop to discover next jump address in the future. Once the algorithm finds the branch address and takes instrumentation, it promotes the program run by function *Gdbrun* until the *pc* of the program reaches the next address of instruction *i+1* (lines 20-22).

We use an example in Figure 3 to better explain our instrumentation method. The example is a function fragment showed in IDA. The entry address of the function is 0x4081DB. When the program is executed to the entry address, our method pauses the program and starts to use the Algorithm 1. It first reads the memory information with the size of 4 kb from the *pc*. By using Capstone to disassemble information to the assembly code, we can determine the next branch address is at 0x4081E0. However, the jump address of 0x4081E0 is lib function *socket*. Thus, algorithm skips it and continues to find the next branch 0x4081E8. The jump addresses of 0x4081E8 may be the 0x4081EC or the 0x4081F8. So, our method instruments at theses addresses and keeps the program run. In the actual execution of program, it will trigger the breakpoints at the 0x4081EC. Then as observing the perhaps execution addresses of the next branch (0x4081F0) are 0x4081F4 and *sub_406D38*, the static analysis takes place to preinstrument at the entry of each basic blocks in *sub_406D38*.

5 | GENETIC ALGORITHM GENERATION MODEL

A genetic algorithm model based on code coverage is proposed in this article to deal with the challenge of the inefficiency to generate input on firmware protocol fuzzing. The model produces test cases that not only satisfy protocol specifications but also execute in-depth procedures in the program.

As illustrated in Figure 4, the test case generation can be divided into two stages: shallow detection and depth testing.

Shallow detection stage: The shallow detection stage starts with an initial protocol template provided by the user. Each protocol field in the template is sent to the target network process in turn. Through the feedback on the code coverage, it can be determined that which fields are not processed in target program. The unprocessed fields are prune out to reduce the space of the fuzzing. At the end of the shallow detection stage, a deep protocol template set is formed. The set includes templates correspond to a protocol field to be tested.

Algorithm 1. An optimized instrumentation

Input: target program, *program*; the addresses defined by users, *ommitaddress*;the addresses of lib functions, *libaddress*; the address stored in pc register, *pc*;
Output: the address stored in pc register, *pc*;

```
I ← Disassemble(program, pc)
for i ∈ I do
    if i ∈ ommitaddress then
        continue
    end if
    if isbranch(i) then
        jmpaddress ← Getjumpaddress(i)
        if jmpaddress ∈ libaddress then
            continue
        else
            Instrumentation(next(i))
            Nodes ← CFGnode(jmpaddress)
            for j ∈ Nodes do
                Instrumentation(j)
            end for
            break
        end if
    end if
end for
while pc ≠ (i + 1) do
    Gdbrun()
end while
```

Depth testing stage: In depth testing stage, a template is selected from the protocol template to mutate to form the initial test cases. These test cases are stored in a queue and are sequentially taken from the queue for testing. The code coverage of each test case is obtained with the dynamic instrumentation in Section 4. After calculating each test case in the fitness function, the queue updates under a schedule strategy to remove test cases of low fitness scores. The cases of high fitness score are pushed into the queue for further mutation. The above process keeps running until the code coverage is no longer improved, indicating that the test of the specified field has been completed. Then a new template is selected from the deep protocol template set for testing. Once all templates in the set are completed, the fuzzing ends automatically.

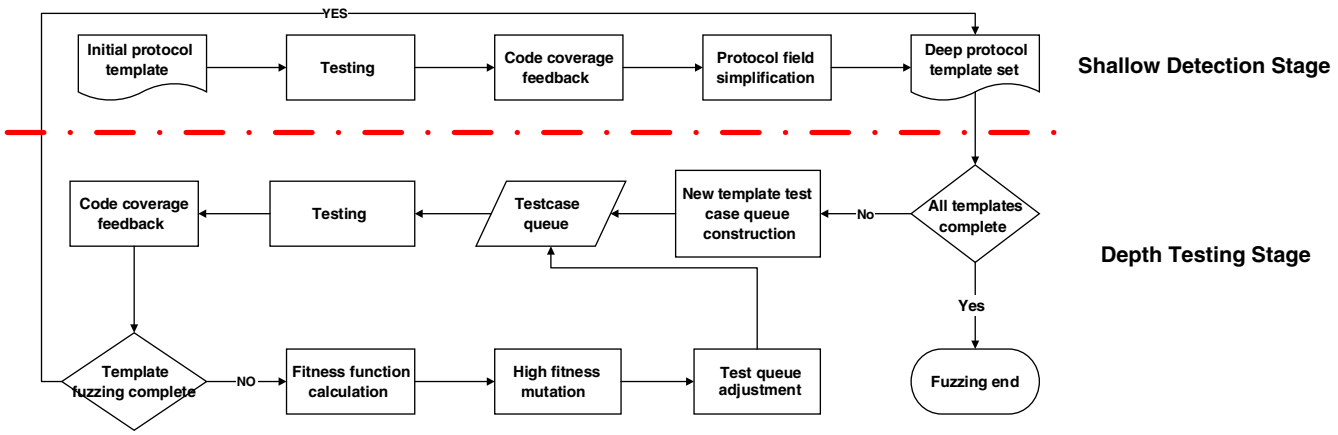


FIGURE 4 The workflow of genetic algorithm model

The *fitness function* determines whether fuzzing could generate test cases with high code coverage. Its design follows three principles in order to maximize the code path discovered by the generation of input.

Principle I: The more new paths a test case finds, the more likely the generation of its mutation will discover more new paths.

Principle II: The more times a newly discovered path is executed by a test case, the path is more valuable.

Principle III: A test case finds the new path earlier, the more likely the generation of its mutation will discover new paths earlier.

According to the above three principles, the specific fitness function is expressed as follows in formula (1):

$$Fitness\_score(input) = \frac{\sum_{i \in New\_branch} Find\_time(i) \times Hit\_num(i)}{Total\_branch}, \tag{1}$$

Total_branch indicates the total number of branches found before the test case *input* executed. The set *New_branch* concludes all branches that the *input* arrives. The size of the *New_branch* represents the **Principle I**. *Hit_num* is the collision parameter, indicating the number of times the branch *i* is reached during the test, which reflects the **Principle II**. *Find_time* refers to the first time that branch *i* arrives, corresponding to the **Principle III**.

The value of the collision parameter *Hit_num* is *k* as follows in formula (2). The actual number of executions *Actual_hit_num* is between the base of *k* and the *k+1* power of base two. The purpose of selecting the *k* value is to avoid the *Actual_hit_num* being too large when processing the loop structure.

$$2^k < Actual\_hit\_num < 2^k + 1. \tag{2}$$

The value of the time parameter *Find_time* is as follows in formula (3). The value is the ratio of the number of branches executed when branch *i* is first found in test to the total number of branches *Execute_branch* executed by this round of tests. The ratio is used to indirectly measure the time when the test case finds a new path. When the ratio is larger, it means that the time to discover a new branch is earlier.

$$Find\_time(i) = \frac{First\_new\_branch(i)}{Execute\_branch(i)}. \tag{3}$$

The *mutation* of Fw-fuzz inherits the method of Sulley.[40] It breaks the a protocol request into individual primitives. Each primitive mutates depends on its type. The basic type is defined into four categories: string, data value, separator, and static. Finally, these mutated primitives splice together to form new case data.

The *scheduling strategy* is used to adjust the test cases in the queue. As it is shown in the formula (4), the *remove_rate* represents the proportion of test case removal generated in the *n*th round. *m* indicates the number of rounds after the *n*th mutation. The *remove_rate* decreases by an exponential function, which means the scheduling strategy removes the test cases of low scores at a higher ratio when these cases are just generated. In addition, the strategy does not delete all the lower score cases generated in early rounds, avoiding the problem of genetic algorithm fitness function converge too early to discover fewer paths in the program.

$$remove\_rate(n) = \frac{1}{2^m}. \tag{4}$$

As for the time complexity of the genetic algorithm, it is O(g(nm + nm + n)) with g being the number of generations, n being the population size, and m being the size of the individuals. Therefore, the complexity is on the order of O(gnm)).

In order to reveal the above two stages more intuitively, the example in Section 2.1 is used to illustrate the workflow of the generation model. In Figure 5, a tree structure on the left represents the initial template according to the Http RFC standard. The right tree represents the deep protocol template after the process of pruning out useless branches. Since "Accept-language," "Host," and "User-agent" are processed in the example, the code coverage only increases when constructing the above three fields. This allows model to trim out the extra fields in initial template to reduce the fuzzing space.

Then in the first step of depth testing stage, a template for "Accept-language" is selected to construct the initial test cases. These cases are pushed into the queue as Figure 6 shows. The case which "Accept language" filed is "zh" gets a higher code coverage because it can enter the func4() and discover more paths to the program. According to the meaning of fitness function, the higher code coverage test case gets the higher score. So, the case of "zh" is retained in the new queue for next round mutation.

6 | IMPLEMENTATION AND EVALUATION

We implemented prototype system of Fw-fuzz in Python based on a set of open source tools: the instrumentation module was built on GDB[38] and Capstone,[39] the generation module was implemented based on Sulley,[40] and the control module and the transmission module were implemented to client-server architecture.

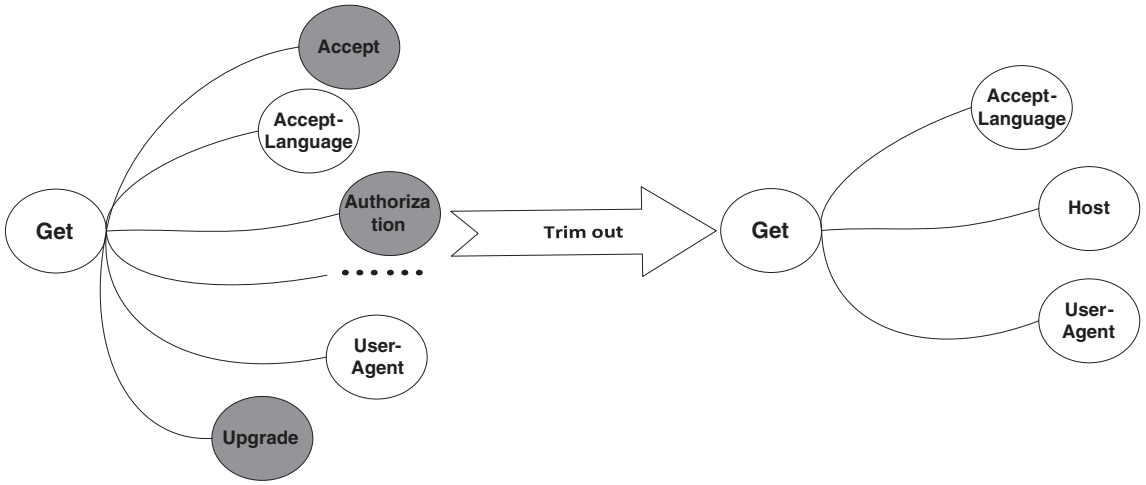


FIGURE 5 The process of protocol pruning

FIGURE 6 The process of updating of queue

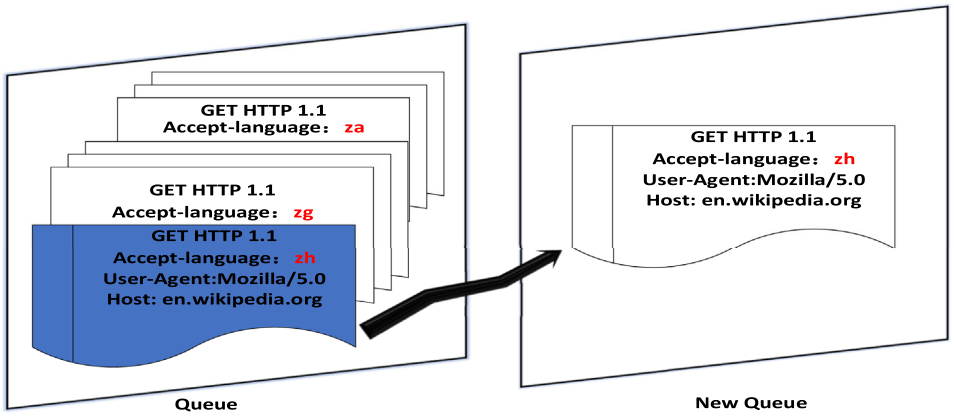


TABLE 1 Dataset used for evaluation

Architecture	Device	Program	Protocol
PPC	TASKalfa 356ci Printer	jnetprint	HTTP
MIPS	MiWiFi Pro	SSH	SSH
ARM64	ROCK960 Development Board	Oftpd	FTP
ARM32	Samsung Smart Hub	hubcore	HTTP
ARM32	Netgear r6400	httpd	HTTP
ARM32	Dahua IPC Camera	Sonia	RTSP

To determine the effectiveness of Fw-fuzz, we present evaluations on the datasets in Table 1, which contains multiple architecture network service programs. In order to outperform the advantages of Fw-fuzz, we answer the following four research questions:

1 Can Fw-fuzz fuzz different architecture firmware?
2 How does Fw-fuzz perform compared to other protocol fuzzers?
3 Can Fw-fuzz discover vulnerabilities in the real-world IoT devices?
4 What is the overhead of Fw-fuzz?

The experiments were run on the computer that was running Ubuntu 16.04 LTS and equipped with an Intel i7-8550K processor and 32 GB of memory. And the parameter settings of the genetic algorithm are the number of the initial seed is less than 10; the termination iterations depends

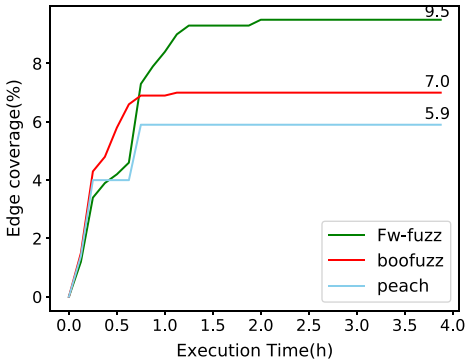


FIGURE 7 The coverage of jnetprint

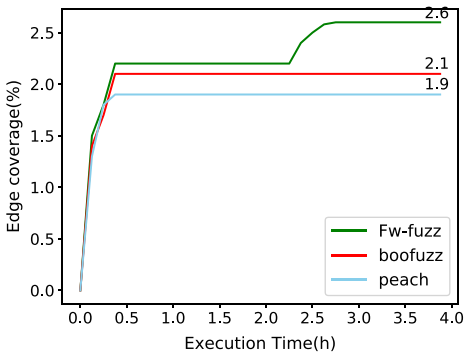


FIGURE 8 The coverage of SSH

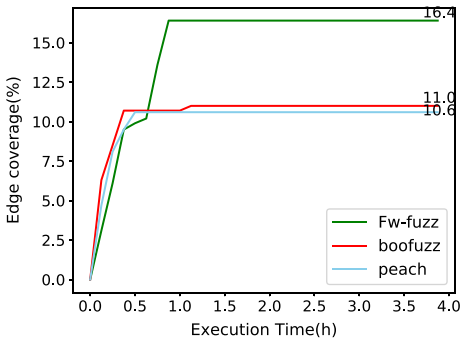


FIGURE 9 The coverage of oftpd

on the fuzzing time; the mutation probability always equals to 1, and the crossover probability equals to 0 because there are actually only mutations and no crossovers in Fw-fuzz procedure.

6.1 | Edge coverage

For questions 1 and 2, we compare the edge coverage of the Fw-fuzz with protocol fuzzers Boofuzz, Peach on our dataset. The edge coverage indicates the numbers of transmission path between two different basic blocks, which is a metric to measure the performance of fuzzing. Compared with basic block coverage and function coverage, edge coverage reflects the control flow of the program more accurately. Note the programs jnetprint, hubcore, httpd, and sonia are integrated web interfaces that contain processing of handling different protocols. So, their edge coverage is measured by the paths associated with the target protocol processing in Table 1.

Since the feedback mechanism of code coverage is not implemented in Boofuzz and Peach, the dynamic instrumentation method proposed in this article is used to collect their execution paths. The experiment is carried out for 4 hours on each program in the dataset. The trends of the edge coverage are shown in Figures 7-12.

From the results in Figures 7-12, the execution paths of programs in the dataset are all successfully traced and expressed as the trends of edge coverage over time, proving the firmware dynamic instrumentation can be applied for multiple architectures. Since the state-of-art greybox-fuzzing

FIGURE 10 The coverage of hubcore

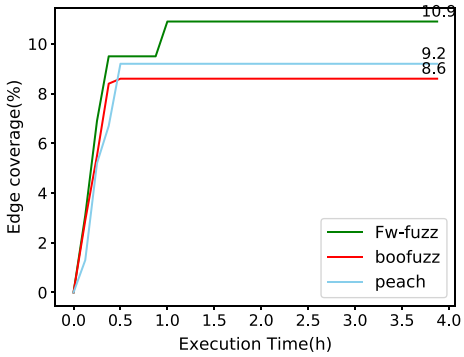


FIGURE 11 The coverage of httpd

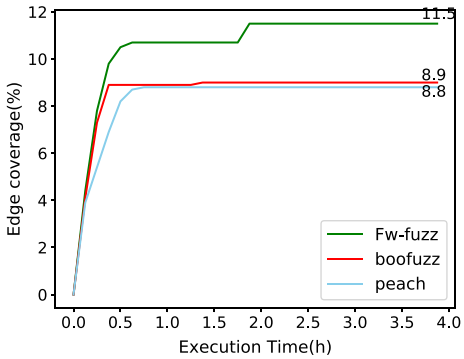
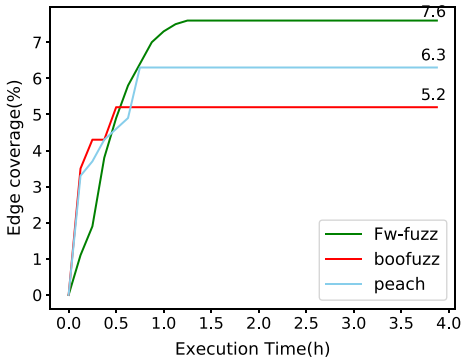


FIGURE 12 The coverage of sonia



and firmware fuzzing mentioned in Section 7 cannot fuzz the program jnetprint and hubcore, the feedback of the internal running state of the different architectures programs highlight the crossarchitecture capability of Fw-fuzz.

The ultimate edge coverage of Fw-fuzz is 38.4% higher than Peach and 33.7% than Boofuzz in six programs on average. The higher edge coverage means that Fw-fuzz can better cover programs, which proves our genetic model improves the efficiency of test case. In addition, two phenomena in the experiment caused our attention. One is the edge code coverage of Boofuzz and Peach is higher than Fw-fuzz in the early time at the case of oftpd, sonia, and jnetprint. The reason to explain it is Fw-fuzz is at the shallow detection stage during the first half hour. The purpose of shallow detection is not to go deep into the program, resulting in a certain probability that is weaker than random fuzzers. And once Fw-fuzz enters the deep detection stage, it discovers much more new branches. The other phenomenon is that all types of coverage of three fuzzers on SSH are very low: even the Fw-fuzz just reaches about 3.1%. The SSH protocol involves many encryption processes, which is complex for fuzzers to generate the proper test case.

6.2 | Vulnerabilities

For question 3, we tested the capability of Fw-fuzz to discover vulnerabilities on the dataset.

We first test Fw-fuzz by reproducing verified vulnerabilities on Samsung Smart Hub. There are series of overflow vulnerabilities on Samsung Smart Hub's Web interface program hubcore with the firmware version 0.20.17. Within 12 hours, Fw-fuzz successfully triggers the vulnerabilities

TABLE 2 Verified CVE ID on Samsung samrthub

CVE ID	Vulnerability cause	Could Boofuzz or Peach discover
CVE-2018-3905	Buffer overflow on field "state"	Yes
CVE-2018-3904	Buffer overflow on field "update"	Yes
CVE-2018-3902	Buffer overflow on field "replace"	Yes
CVE-2018-3867	Buffer overflow of a series of Http requests	Yes
CVE-2018-3895	Buffer overflow on field "endTime"	Yes
CVE-2018-3894	Buffer overflow on field "startTime"	Yes

TABLE 3 0-day vulnerabilities discovered by Fw-fuzz

Vulnerability ID	Type	Could Boofuzz or Peach discover
PSV-2019-0145	Stack overflow	Yes
PSV-2019-0416	Stack overflow	No
PSV-2019-0417	Stack overflow	No
PSV-2019-0418	Stack overflow	No
PSV-2019-0118	Out-of-bounds read and write	No

shown in Table 2. Then we perform our prototype system on Netgear r6400 and Dahua IPC Camera for 3 days, successfully discovering five 0-day vulnerabilities in Table 3. In addition, the four of 0-day vulnerabilities cannot be discovered by Boofuzz or Peach. The evaluation results demonstrate our framework can be used for real cases of IoT devices and has better performance than the state-of-art protocol fuzzers.

6.3 | Overhead

The meaning of measuring overhead is to evaluate the impact of our fuzzing method on IoT devices. We need to evaluate whether using Fw-fuzz will affect the performance of the device. So, we count the memory usage and CPU usage of the dataset programs in three modes: disable fuzzing, enable fuzzing without instrumentation, and enable both fuzzing and instrumentation. We perform each mode on every program in the dataset for 12 hours and calculate the average CPU consuming and memory consuming.

The evaluation results in Table 4 show that memory usage increases by an average of 5% when fuzzing with instrumentation. While deploying fuzzing without instrumentation is nearly consistent with disable fuzzing. It reveals that utilizing GDB in server mode to implement the instrumentation consumes the memory. The maximum of memory growth is on Sonia about 10.2% by fuzzing with instrumentation, which we think the additional memory overhead is acceptable. The CPU usage of three modes on different programs is close, indicating our fuzzing framework not occupy the CPU heavily. The overheads of Boofuzz and Peach on the devices are actually closed to the results of mode B because Boofuzz and Peach do not use dynamic firmware instrumentation. Our method has an additional increase in device memory compared with the state-of-art protocol fuzzers, but within acceptable limits.

7 | RELATED WORKS

- ***Code coverage guided fuzzing:*** AFL[17] is the first fuzzer to use the heuristic algorithm to guide the generation of the test cases based on code coverage. AFLFast[18] and AFLGo[19] improve the scheduling strategy in the AFL, further increasing the code coverage of the fuzzing. Driller[34] utilize symbolic execution to enhance the AFL, partly solving the problem of encoded number in the program. VUzzer[41] extract the control flow and data flow information of the program based on the dynamic and static analysis to guide the generation of test cases corresponding to the execution deeper path. Hawkeye[42] evaluates seeds based on static information and execution traces to improve the seed selection process to cause program crashes faster. Profuzz[43] automatically analyzes input fields that are critical for vulnerability discovery and intelligently adjusts mutation strategies to increase the probability of discovering vulnerabilities. However, these advanced fuzzing not support to fuzz the program on the IoT device. We are inspired by these advanced fuzzing to build a genetic algorithm model to guide the generation of fuzzing input based on code coverage.

TABLE 4 The comparison of CPU and memory usage in three modes

Program	Mode	Memory usage (%)	CPU usage (%)
jnetprint	A	11.0	7.2
	B	11.5	7.3
	C	16.2	7.3
SSH	A	9.8	3.5
	B	10.7	3.3
	C	12.2	3.5
Oftpd	A	18.8	10.4
	B	19.5	11.2
	C	25.1	11.3
httpd	A	11.9	3.2
	B	12.1	4.1
	C	17.5	4.1
Sonia	A	13.6	6.2
	B	15.5	6.3
	C	23.8	6.3

[a]*Note:* The modes A, B, and C represent disable fuzzing, enable fuzzing without instrumentation, and enable both fuzzing and instrumentation

- ***Firmware fuzzing:*** Some researchers fuzz firmware in full emulation or partial emulation environment these years. However, the evaluation of Daming[25] shows only a quarter of firmware can run the network service; in Andrei[24] experiment, none of the firmware with PPC architecture is supported to run Web server; SURROGATES[26] admits that their model does not ensure the correctness of the data flow during fuzzing. The reason for the above results is the emulation relies on the Qemu[27] while there is a variety of architecture firmware that Qemu not supports. IoTFuzzer[44] fuzz the IoT physical devices through the client applications. However, their method could not be extended to the IoT devices without apps.
- ***Protocol fuzzing:*** Most protocol fuzzing[28,40,45] can be classified into blackbox fuzzing. Their generation of test cases is based on grammar. Recently, protocol fuzzing concentrates on solving the transmission of the state in protocol communication[32] and the security protocol.[30,31] SpFuzz[46] uses hierarchical scheduling to improve the test cases of protocol based on the feedback information. However, the feedback of SpFuzz relies on the AFL instrumentation, which is not suitable for close sourced firmware.

8 | CONCLUSION

In this article, the main challenges of fuzzing protocol on firmware are attributed to the inefficiency of test case generation, crossarchitecture instrumentation, and fault detection.

Fw-fuzz is proposed to conquer these challenges to a genetic algorithm model based on code coverage and dynamic firmware instrumentation. The dynamic firmware instrumentation not only addresses the difficulty in the feedback information on the target program on devices but also supports to directly capture the faults when crashes occur in the firmware program. The genetic algorithm model uses heuristic principles with code coverage to guide the mutation of fuzzing inputs to explore more paths. And in the genetic model, the initial template based on grammar avoids mutating the fixed field of the protocol.

The prototype system of the framework has undergone actual tests. The evaluation results in the edge coverage highlight the efficiency of Fw-fuzz. And the dynamic firmware instrumentation is proved to be suitable for different architectures programs. In addition, we successfully trigger known vulnerabilities and discover five 0-day vulnerabilities in the devices.

Future works include using machine learning methods to generate the initial template automatically and improving the mutation to cover more paths of the program by combining static analysis technique.

ACKNOWLEDGMENT

This project was supported by the National Natural Science Foundation of China (No. 61802431).

ORCID

Zicong Gao https://orcid.org/0000-0003-2789-1731

REFERENCES

1. Bilal M. A review of internet of things architecture, technologies and analysis smartphone-based attacks against 3D printers; 2017. arXiv preprint arXiv:1708.04560.
2. Wang W, Zhao M, Gao Z, et al. Constructing features for detecting android malicious applications: issues taxonomy and directions. *IEEE Access*. 2019;7:67602-67631.
3. Qiu J, Zhang J, Luo W, et al. A3CM: automatic capability annotation for android malware. *IEEE Access*. 2019;7:147156-147168.
4. D'Orazio CJ, Choo KKR, Yang LT. Data exfiltration from Internet of Things devices: IOS devices as case studies. *IEEE IoT J*. 2016;4(2):524-535.
5. Cimitile A, Martinelli F, Mercaldo F. Machine learning meets ios malware: identifying malicious applications on apple environment. *ICISSP*. Porto, Portugal: SciTePress; 2017:487-492.
6. Meng W, Li W, Wang Y, Au MH. Detecting insider attacks in medical cyber–physical networks based on behavioral profiling. *Future Generat Comput Syst*. 2018.
7. Wang Y, Meng W, Li W, Liu Z, Liu Y, Xue H. Adaptive machine learning-based alarm reduction via edge computing for distributed intrusion detection systems. *Concurr Comput Pract Exp*. 2019;31(19):e5101.
8. Raza S, Wallgren L, Voigt T. SVELTE: real-time intrusion detection in the Internet of Things. *Ad Hoc Netw*. 2013;11(8):2661-2674.
9. Ye N, Zhu Y, Rc W, Malekian R, Qiao-Min L. An efficient authentication and access control scheme for perception layer of Internet of Things. *Appl Math Inf Sci*. 2014;8(4):1617.
10. Aman MN, Chua KC, Sikdar B. Position paper: physical unclonable functions for iot security. Paper presented at: Proceedings of the 2nd ACM international workshop on IoT privacy, trust, and security; 2016:10-13; ACM.
11. Meng W, Jiang L, Choo KKR, Wang Y, Jiang C. Towards detection of juice filming charging attacks via supervised CPU usage analysis on smartphones. *Comput Electr Eng*. 2019;78:230-241.
12. Davidson D, Moench B, Ristenpart T, Jha S. {FIE} on firmware: finding vulnerabilities in embedded systems using symbolic execution. Paper presented at: Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13); 2013:463-478.
13. Cheng K, Li Q, Wang L, et al. DTaint: detecting the taint-style vulnerability in embedded device firmware. Paper presented at: Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN); 2018:430-441; IEEE.
14. Lin J, Jiang L, Wang Y, Dong W. A value set analysis refinement approach based on conditional merging and lazy constraint solving. *IEEE Access*. 2019;7:114593-114606.
15. Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. *Commun ACM*. 1990;33(12):32-44.
16. Manes VJM, Han H, Han C, et al. Fuzzing: art, science, and engineering; 2018.
17. Zalewski M. AFL; 2015. http://lcamtuf.coredump.cx/afl.
18. Böhme M, Pham V, Roychoudhury A. Coverage-based greybox fuzzing as Markov Chain. *IEEE Trans Softw Eng*. 2019;45(5):489-506. https://doi.org/10.1109/TSE.2017.2785841.
19. Böhme M, Pham VT, Nguyen MD, Roychoudhury A. Directed greybox fuzzing. Paper presented at: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications SecurityCCS'17; 2017:2329-2344; ACM, New York, NY.
20. Gan S, Zhang C, Qin X, et al. Collafl: path sensitive fuzzing. Paper presented at: Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP); 2018:679-696; IEEE.
21. Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: fuzzing by program transformation. Paper presented at: Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP); 2018:697-710; IEEE.
22. Lemieux C, Sen K. Fairfuzz: targeting rare branches to rapidly increase greybox fuzz testing coverage; 2017. arXiv preprint arXiv:1709.07101.
23. Muench M, Stijohann J, Kargl F, Francillon A, Balzarotti D. What you corrupt is not what you crash: challenges in fuzzing embedded devices. *NDSS*. San Diego, California: The Internet Society; 2018.
24. Costin A, Zarras A, Francillon A. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. Paper presented at: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security; 2016:437-448; ACM.
25. Chen DD, Woo M, Brumley D, Egele M. Towards automated dynamic analysis for linux-based embedded firmware. *NDSS*. San Diego, California: The Internet Society; 2016:1-16.
26. Koscher K, Kohno T, Molnar D. {SURROGATES}: enabling near-real-time dynamic analyses of embedded systems. Paper presented at: Proceedings of the 9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15); 2015.
27. QEMU: the FAST! processor emulator. https://www.qemu.org; 2019.
28. Peach: discover unknown vulnerabilities. https://www.peach.tech.
29. Boofuzz: a fork and successor of the Sulley fuzzing framework; 2015. https://github.com/jtpereyda/boofuzz.
30. Lee S, Yoon C, Lee C, Shin S, Yegneswaran V, Porras PA. DELTA: a security assessment framework for software-defined networks. *NDSS*. San Diego, California: The Internet Society; 2017.
31. Tsankov P, Dashti MT, Basin D. SECFUZZ: fuzz-testing security protocols. Paper presented at: Proceedings of the 7th International Workshop on Automation of Software Test; 2012:1-7; IEEE Press.
32. De Ruiter J, Poll E. Protocol state fuzzing of {TLS} implementations. Paper presented at: Proceedings of the 24th {USENIX} Security Symposium ({USENIX} Security 15); 2015:193-206.
33. Cui A, Costello M, Stolfo SJ. When firmware modifications attack: a case study of embedded exploitation. *NDSS*. San Diego, California: The Internet Society; 2013;.
34. Stephens N, Grosen J, Salls C, et al. Driller: augmenting fuzzing through selective symbolic execution. *NDSS*. San Diego, California: The Internet Society; 2016.
35. Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T. REDQUEEN: fuzzing with input-to-state correspondence. *NDSS*. San Diego, California: The Internet Society; 2019.

36. Chen P, Chen H. Angora: efficient fuzzing by principled search. Paper presented at: 2018 IEEE Symposium on Security and Privacy (SP); 2018:711-725; IEEE.
37. Luk CK, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not*. 2005;40(6):190-200. https://doi.org/10.1145/1064978.1065034.
38. GDB: the GNU project debugger; 2019. https://www.gnu.org/software/gdb.
39. Capstone: the ultimate disassembler; 2015. https://github.com/aquynh/capstone
40. Pedram AAP, Sears R. sulley. https://code.google.com/archive/p/clusterfuzz.
41. Rawat S, Jain V, Kumar AJS, Cojocar L, Giuffrida C, Bos H. VUzzer: application-aware evolutionary fuzzing. *NDSS*. San Diego, California: The Internet Society; 2017.
42. Chen H, Xue Y, Li Y, et al. Hawkeye: towards a desired directed grey-box fuzzer. Paper presented at: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security; 2018:2095-2108; ACM.
43. You W, Wang X, Ma S, et al. Profuzzer: on-the-fly input type probing for better zero-day vulnerability discovery. *ProFuzzer: On-the-Fly Input Type Probing for Better Zero-Day Vulnerability Discovery*. New York, NY: IEEE; 2019.
44. Chen J, Diao W, Zhao Q, et al. IoTFuzzer: discovering memory corruptions in iot through app-based fuzzing. *NDSS*. San Diego, California: The Internet Society; 2018.
45. Banks G, Cova M, Felmetsger V, Almeroth K, Kemmerer R, Vigna G. SNOOZE: toward a stateful network protocol fuzzer. In: Katsikas SK, López J, Backes M, Gritzalis S, Preneel B, eds. *Information Security*. Berlin, Heidelberg / Germany: Springer; 2006:343-358.
46. Song C, Yu B, Zhou X, Yang Q. SPFuzz: a hierarchical scheduling framework for stateful network protocol fuzzing. *IEEE Access*. 2019;7:18490-18499.

How to cite this article: Gao Z, Dong W, Chang R, Wang Y. Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware. *Concurrency Computat Pract Exper*. 2020;1–15. https://doi.org/10.1002/cpe.5756