

Article

FirmHunter: State-Aware and Introspection-Driven Grey-Box Fuzzing towards IoT Firmware

Qidi Yin , Xu Zhou * and Hangwei Zhang

College of Computer, National University of Defense Technology, Changsha 410073, China; yinqidi@nudt.edu.cn (Q.Y.); zhanghangwei@nudt.edu.cn (H.Z.)

* Correspondence: zhouxu@nudt.edu.cn

Abstract: IoT devices are exponentially increasing in all aspects of our lives. Via the web interfaces of IoT devices, attackers can control IoT devices by exploiting their vulnerabilities. In order to guarantee IoT security, testing these IoT devices to detect vulnerabilities is very important. In this work, we present FirmHunter, an automated state-aware and introspection-driven grey-box fuzzer towards Linux-based firmware images on the basis of emulation. It employs a message-state queue to overcome the dependency problem in test cases. Furthermore, it implements a scheduler collecting execution information from system introspection to drive fuzzing towards more interesting test cases, which speeds up vulnerability discovery. We evaluate FirmHunter by emulating and fuzzing eight firmware images including seven routers and one IP camera with a state-of-the-art IoT fuzzer FirmFuzz and a web application scanner ZAP. Our evaluation results show that (1) the message-state queue enables FirmHunter to parse the dependencies in test cases and find real-world vulnerabilities that other fuzzers cannot detect; (2) our scheduler accelerates the discovery of vulnerabilities by an average of 42%; and (3) FirmHunter is able to find unknown vulnerabilities.



Citation: Yin, Q.; Zhou, X.; Zhanag, H. FirmHunter: State-Aware and Introspection-Driven Grey-Box Fuzzing towards IoT Firmware. *Appl. Sci.* **2021**, *11*, 9094. <https://doi.org/10.3390/app11199094>

Academic Editor: Luis Javier Garcia Villalba

Received: 17 August 2021

Accepted: 26 September 2021

Published: 29 September 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: IoT; firmware; fuzzing; vulnerability

1. Introduction

With the development of the Internet of Things (IoT), more and more embedded devices have begun to enter people's lives. According to Statista [1], at the end of 2018, there were an estimated 22 billion internet of things (IoT) connected devices in use around the world, and by 2025, there will be 38.6 billion IoT connected devices. Together with the exponential growth of use, the security issue of IoT devices should also be considered.

The web interface is an important way for users to interact with IoT devices as well as making configurations. A search on Shadon [2] showed that among 1,442,722 routers, 790,515 (54.8%) devices provide web interfaces. With the wide use of web interfaces for service implementation and configuration, web interfaces have also become a big attack surface. According to a survey [3], each web application had an average of 33 vulnerabilities, with six of them being of high severity. Therefore, testing the reliability and security of network interfaces is essential.

There are two main vulnerability detection techniques for IoT devices: static analysis and dynamic testing.

Static analysis can be placed on both the front-end and the back-end of IoT devices. For front-end analysis, a typical method is to use RIPS [4] to analyze PHP language. However, the diversity of front-end languages restricts the development of general analysis engines, and the high false positive rate caused by static analysis is also not optimistic. For the back-end static analysis, the common process is extracting binaries from the IoT firmware and analyzing them. In order to analyze the binaries in firmware, time-consuming reverse engineering needs to be applied, which also introduces false positive alerts.

Dynamic testing, independent of a particular server language, is able to detect vulnerabilities through interaction with IoT network services. The most promising technique of

dynamic testing is fuzzing [5], which is effective at finding vulnerabilities in both software and systems. Fuzzing finds vulnerabilities by sending random input to the test target and detecting abnormalities in the program or system.

Owing to the low false positive rate, large scalability, and high automation, fuzzing has been applied in recent studies. With the wide use of web interface, many researchers focus on fuzzing IoT devices via web interfaces. Some work [6–9] relies on manually formulated templates to generate test cases. However, writing templates requires full knowledge of the design of protocols. To this end, some works implement crawlers [10–15] to traverse the web application and capture the traffic messages. After that, customized strategies are applied to mutate these captured messages to generate test cases.

However, in practice, the approaches often fail to detect vulnerabilities effectively due to the following two challenges.

One challenge is the state dependency in test cases, without which exploiting a vulnerability is impossible. As for a fuzzer, who needs to interact with IoT devices via web interfaces, there are three types of dependencies through the interaction: authentication dependency, intra-message dependency, and inter-message dependency. As most of the vulnerabilities can be triggered by sending malicious messages only after authentication, it is essential to maintain the authentication state which means authentication dependency. Intra-message dependency indicates some values in a message that are determined by other fields within the message, and we focus on the content length field in the HTTP protocol in this paper. Furthermore, inter-message dependency refers to the fact that the trigger of a vulnerability after authentication may require multiple messages. The latter message always contains a field that is rendered from the previous message. Without the full knowledge of the protocol implementation, a writer can hardly satisfy the three dependencies and generate lots of invalid test cases. Although some works focus on stateful fuzzing, they have bad performance on more complex IoT devices. FirmFuzz [10] and EWVHunter [14] implement web crawlers to maintain authenticated state, but intra-message and inter-message dependencies are often neglected. SIOTFuzzer [16] extracts messages between page responds to gain a stateful seed while not analyzing the intra-message dependencies in these messages.

Another challenge is the lack of test case scheduling. Test case scheduling means ranking test cases based on their features like execution time and giving priority to testing those test cases with higher scores. At present, most IoT fuzzers firstly capture traffic packets as initial test cases and generate new test cases by mutating them. Furthermore, they have not proposed a proper test case scheduling method for better vulnerability detection. For FirmFuzz [10], test cases are sorted in the time sequence in which they were crawled. Furthermore, modern web scanners [11–13] store test cases based on the value of their URL fields. After that, these tools choose one message in order and change the values of different fields in it to generate new test cases. However, applying all mutation strategies to fuzz one message consumes a considerable amount of time due to low throughput. Assuming the valuable test case, which can generate valuable inputs to a target device, is crawled at the end and therefore the last to be tested. Lots of computer resources will be used on fuzzing meaningless test cases and the vulnerability detection process is greatly delayed.

In this paper, we present FirmHunter, a state-aware introspection-driven grey-box fuzzer towards Linux-based firmware images. To overcome the above-mentioned challenges, we propose the message-state queue and the introspection-driven scheduling. By analyzing both the behavior of the crawler and the features of captured messages, we extract the authentication, intra-message, and inter-message dependencies from messages and construct message-state queues. In order to schedule test cases, we emulate target IoT firmware and obtain two granularities of execution feedback: system calls and code coverage from system introspection. Based on collected execution feedback, we rank test cases and give priority to testing more valuable test cases.

To evaluate our tool, we tested it on a set of real-world IoT firmware images with a state-of-the-art IoT fuzzer and a web scanner. The evaluation results showed that (1) the message-state queue enables FirmHunter to parse the dependencies in test cases and to find real-world vulnerabilities in IoT firmware images; (2) our scheduler accelerates the detection of vulnerabilities by an average of 42%; and (3) FirmHunter finds known vulnerabilities much faster than some state-of-the-art IoT fuzzers are able to find unknown vulnerabilities.

In summary, we make the following contributions in this paper:

- We proposed the message-state queue suited for IoT fuzz test cases, which can satisfy authentication dependency, intra-message dependency, and inter-message dependency.
- We developed the scheduling algorithm making use of introspection on system emulation to rank the test cases which speeds up vulnerability detection with an average of 42%.
- We designed and implemented FirmHunter, a state-aware introspection-driven grey-box fuzzer towards Linux-based firmware images via web interfaces. Our evaluation showed that FirmHunter outperforms other tools in IoT vulnerability detection and found 3 unknown vulnerabilities.
- The source code of FirmHunter can be found at <https://github.com/StrongerQQ/FirmHunter> (accessed on 15 August 2021).

2. Background and Motivation

2.1. Firmware Emulation

As firmware emulation does not require real devices and can provide web interfaces for dynamic analysis, thus offering the possibility for automated, large-scale IoT fuzzing. In the emulation, the emulated system is called the guest system, and the system providing software support is called the host system. Firmware emulation is divided into user-level emulation, hybrid emulation, and system-level emulation.

User-level emulation only emulates a single program in the firmware. Using user-level emulation, fuzzing can obtain the execution information of the program to guide the generation of test cases and improve the throughput in testing. For example, AFL [17] uses QEMU [18], which translates and executes the instructions of the target program with the granularity of the code block level, to achieve the purpose of efficient execution. However, emulating a program often fails because of the complicated interactions with other binaries or hardware, such as peripherals.

To overcome the execution problems in user-level emulation, Avatar [19] first proposed hybrid emulation. By connecting the user-level QEMU-based emulation with real devices, it is possible to realize the translation and execution of firmware instructions on the desktop operating system while directing I/O operations to the real devices to solve the running problems. However, the introduction of symbolic execution [20], as well as frequent switching between the emulation environment and physical equipment, will greatly reduce the performance of the fuzz testing.

System-level emulation treats the IoT system as a whole. By emulating the kernel and peripherals, system-level emulation realizes the whole system emulation. On the basis of QEMU, Costin et al. [21] developed an extensible dynamic analysis framework to analyze the web interface of IoT devices. Gustafson et al. [22] emulated the operation of memory-mapped I/O (MMIO) in peripheral communication to successfully run the firmware.

2.2. Web Interface in IoT

Most IoT devices have access to a web Interface [2], which provides an effective way for IoT dynamic testing.

Web interface communication usually consists of three parts as shown in Figure 1: front-end, server, and back-end [23]. The front-end is the browser used by the user, on which static resources such as HTML, CSS, and JS are running. The server and the back-end are both running in the IoT device, the server is a binary program such as httpd or mini_httpd, which is responsible for receiving and responding to the front-end and

communicating with the back-end; the back-end is composed of some executable files like cgi, php, and asp files. The flow of interaction between the user and the web interface provided by IoT is shown in Figure 1. ① User operations are performed, such as filling the form in the browser, clicking buttons, and ajax events. ② The front-end browser sends the corresponding request to the IoT device. ③ The server component in an IoT device parses the request and obtains parameters from the message. Then the server chooses the corresponding back-end executable file, passes the parameters to it and executes it. ④ Result of the back-end execution is sent back to the server, and ⑤ the server packs up the feedback result and sends it to the front-end.

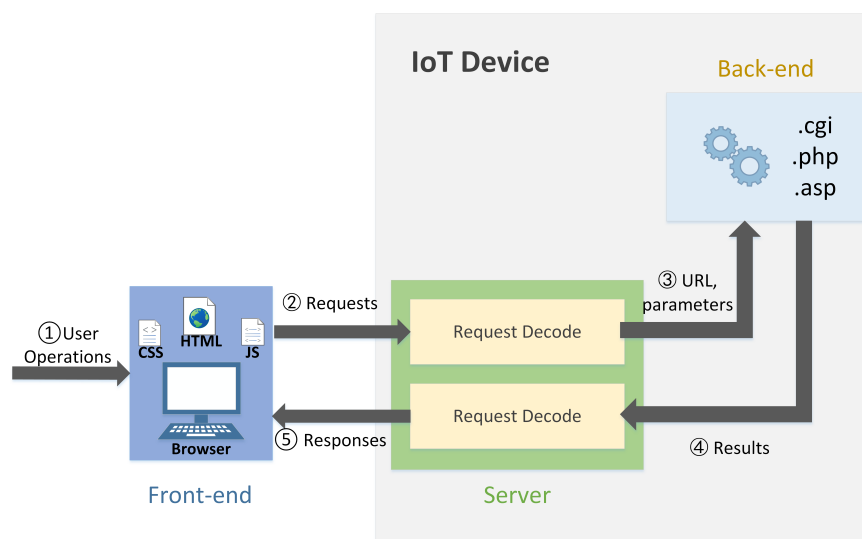


Figure 1. Communication via web interface.

2.3. IoT Fuzzing via Web Interface

Therefore, there exists three dimensions of fuzzing towards the web interface in IoT devices: front-end fuzzing, server fuzzing, and back-end fuzzing:

In the front-end fuzzing, the inputs are user operations, including GET requests, filling forms, and JS events. EWVHunter [14] implements web crawlers to interact with network applications which crawl the static resources from the front-end and explore input fields like URLs and forms. After filling randomly generated values in fields, EWVHunter clicks buttons to send mutated test cases and then checks the response message as well as the connection status to detect potential vulnerabilities. The drawback is that many network applications in front-end set up security rules which limit the string length, special characters, transcoding to filter sensitive values.

To fuzz the server in IoT, the inputs are messages in protocol format. Among the tools for server testing, Peach [6], Sully [7], and boofuzz [8] are widely used in the industry, which relies on manually formulated templates. The automatic fuzz server, FirmFuzz [10] uses crawlers to interact with network applications provided by emulated IoT devices. After capturing a message, it randomly mutates the content of the message. At the same time, it observes the log and connection of the emulated device after sending each malicious message.

In the fuzzing of back-end programs, the inputs are the parameters needed to run these executable files. FIRM-AFL [24] puts forward the concept of augmented emulation, which combines user-level emulation of a single program and system-level emulation of the firmware, to fuzz binaries in firmware. The disadvantage of this type of tools is that we need to select a specific binary program and fuzz it which not only introduces heavy overhead in static analysis but also limits the automated testing.

2.4. Motivations

Given the status-quo of IoT fuzzing techniques, we choose to implement a server-oriented fuzzing tool towards IoT firmware, which is based on system emulation. The reason is two-fold. On the one hand, unlike front-end fuzzing that is restricted by filtering rules or back-end fuzzing is dependent on static analysis of binaries as mentioned in Section 2.3, server fuzzing can achieve both goals of effective test case generation and low pre-analysis overhead. On the other hand, easy accessibility of firmware on the internet and the development of firmware emulation enable the possibility for large-scale testing of IoT firmware images [25,26].

Equipped with system emulation and robust crawlers, IoT fuzzers can achieve automated testing on firmware images. However, there still exist two obstacles hindering fuzzing from effectively testing IoT via web interfaces.

One obstacle is the state dependency in test cases. Take CVE-2019-7298 in the D-link DSL-3782 as an example, this vulnerability is located in the input box on the page Diagnostics.asp as shown in Figure 2. After filling this blank with a random IP address and clicking the “Run test” button, two requests will be sent from the front-end to the server in the target router, which is shown in Figure 3. The former is a GET request which asks the server to get a sessionKey. After receiving a sessionKey from the server, the front-end will append sessionKey as a form data to the form in the latter request. What an attacker does is just capturing these two request messages, modifying the value in the “Addr” field of the latter request with a long string, and replaying these two requests. Then the target router will crash due to buffer overflow.

Figure 2. Vulnerable web page of DSL-3782.

```
GET /cgi-bin/new_GUI/get_sessionKey.asp?_=1617523610704 HTTP/1.1
Host: 192.168.1.1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101 Firefox/87.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: close
Referer: http://192.168.1.1/cgi-bin/New_GUI/Diagnostics.asp

POST /cgi-bin/New_GUI/Set/Diagnostics.asp HTTP/1.1
Host: 192.168.1.1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101 Firefox/87.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 45
Origin: http://192.168.1.1
Connection: close
Referer: http://192.168.1.1/cgi-bin/New_GUI/Diagnostics.asp
Type=ping&sessionKey=1804289383&Addr=192.168.1.1
```

Figure 3. Inter-message dependency in requests.

However, state-of-the-art IoT fuzzers have not proposed approaches to detect such kind of challenge–response mechanism. FirmFuzz [10] captures the first message after each

button-clicking action and mutates this message, and the most popular web application scanners just choose one message from crawled database to mutate. They both neglect this inter-message dependency in IoT testing and thus cannot find this vulnerability.

Another obstacle is the scheduling of test cases to mutate. Suppose current IoT fuzzing tools can recognize the inter-message dependency within these two messages in Figure 2. The vulnerability detection time is determined by when the fuzzer turns to mutate the above two messages. For FirmFuzz, you can implement the depth-first-search (DFS) or breadth-first-search (BFS) algorithm on the crawler to traverse web applications. Furthermore, fuzzing one test case is time-consuming as the fuzzer needs to wait for IoT devices to receive messages, execute back-end binaries and send messages back. Once the vulnerable page is located deeply, it will waste lots of time for FirmFuzz to fuzz the previously found pages.

3. Our Approaches

3.1. Message-State Queue

To satisfy authentication, intra-message, and inter-message dependencies in fuzzing, we construct a message-state queue as shown in Figure 4. The construction of this message-state queue is divided into two phases: the collecting phase and the parsing phase.

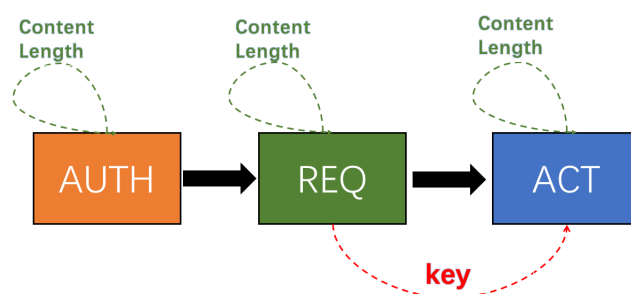


Figure 4. Message-state queue.

In the collecting phase, we implement a crawler traversing the web interface and a middle proxy capturing traffic messages. Every time the crawl fills a form, clicks a button, or triggers a javascript event, it will record the corresponding action with a time stamp. Furthermore, the middle proxy will capture the messages and store them according to the time sequence.

In the parsing phase, we use the above-mentioned information to construct a message-state queue. As the web application firstly comes to the logging page when visiting the web interface, we find the first form filling action as the logging action and denote the triggered request as AUTH to satisfy the authentication dependency. After that, we check the num of requests between every two actions.

If there exist more than one request, we parse them further to identify the potential inter-message actions. The detailed steps are

1. Extracting the values of all fields in both the header and the body of these requests;
2. Performing a match between the extracted values and the body contents of responses to those requests;
3. Once matched, the request containing the matched value is denoted as ACT and the request B, whose response contains that value, is denoted as REQ;
4. We append REQ to AUTH and denote the response of REQ as the key sender;
5. Then we push ACT to the queue and denote the field of that matched value as the key receiver.

As a result, the pass of inter-message dependency is accomplished as the red dotted arrow in Figure 4.

If there exists only one request, we denote this request as ACT and directly connect AUTH and ACT to construct a queue. Furthermore, if there is no request between two

actions, we skip to the next two actions. Last but not least, we constrained each request with their content length fields to satisfy the intra-message dependency, which is shown as the green dotted arrow in Figure 4.

3.2. Introspection-Driven Scheduling

In order to detect vulnerabilities faster, we need to give priority to those valuable test cases, which can generate valuable inputs. In order to rank the queues after the construction of message-state queues mentioned above, we implement a scheduler to score these queues.

We send a message-state queue to the target emulated device, wait for its execution, collecting execution information using system introspection, and score them using the collected information. The system introspection seeks two kinds of execution information: coarse-grained system calls and fine-grained coverage.

For system calls, the six system calls listed in Table 1 are preferred for analysis for the following reasons. When valuable inputs, e.g., a series of network messages causing buffer-overflow are sent to the target device, this device will parse these messages and invoke several processes to handle them. Then some operations, like file creations, file deletions, binary executions, memory comparisons, and memory copies, are carried out. Thus, test cases triggering more system calls in Table 1 are more valuable. To capture the system calls triggered by a test case, we modify the Linux kernel to apply the kernel dynamic probes (kprobes) framework, which can intercept the system calls of the kernel.

Table 1. Hooked System Calls.

Type	Name	Func
Network	inet_accept	Acceptation of messages
System	do_mount	Mounting of file system
Write	vfs_unlink	Deletion of files
Read	do_sys_open	Opening of file descriptors
Execution	mmap_region	Memory mapping
	do_execve	Execution of programs

For coverage, the more program logics are executed by a test case, the more likely this test case is to contain vulnerabilities. Therefore test cases improving more program coverage are more valuable. In order to obtain coverage information, we implement a QEMU plugin to collect the ids of executed basic blocks (BBs).

For each message queue q_i in $\{q_1, q_2, \dots, q_n\}$, we send it to the emulated device, wait for several seconds and count the triggered system calls $S(q_i)$ and the ids of executed BBs $C(q_i)$. After sending all of the message queues we can score each queue using the following algorithm:

$$Score = C(q_i)^k \quad (1)$$

$$k = \frac{S(q_i) - \min S}{\max S - \min S} \quad (2)$$

where $\min S$ selects the minimum in $S(q_1), S(q_2), \dots, S(q_n)$ and $\max S$ selects the maximum.

4. Tool Design

The architecture of FirmHunter is shown in Figure 5. The workflow is that we first emulate the target firmware and provide its web interfaces. Once running, our Traffic Collector applies a crawler to interact with web applications, captures the messages, and delivers the collected messages formatting in pcap files to Message Parser. Then Message Parser builds state-message queues which satisfy authentication dependency, intra-message dependency, and inter-message dependency. After that, Queue Scheduler sends these message queues to the emulated device and scores them. After a more interesting queue is chosen, Fuzzer establishes socket communication with the emulated device and sends test cases generated

by applying mutation on the chosen queue; Monitor detects the abnormal behaviors of the emulated device and generates reports.

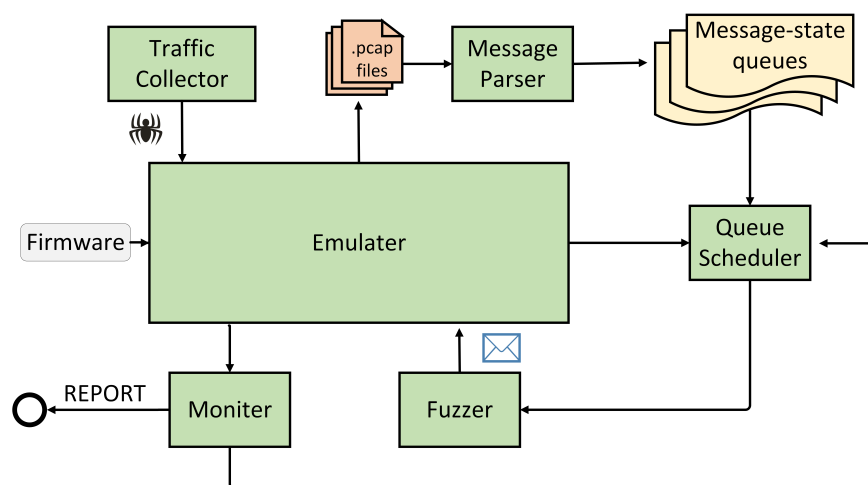


Figure 5. Architecture of FirmHunter.

Emulator: For the collected firmware images, we implement a full-system emulation platform to emulate them.

In order to capture system calls, we modify the Linux kernel to hook sensitive system calls in Table 1. At the same time, to obtain the coverage information of the whole system, we write a QEMU plugin to gain the executed basic block ids.

Furthermore, after we extracted the file system from firmware, we injected a CI (Command Injection) file into it, which helps us find command injection vulnerabilities. When a CI file is executed, a system call for open “ci_file” will be triggered and we assume a OS command is successfully injected once we capture this specified system call.

Traffic Collector: To avoid manually writing protocol templates, we obtain templates by analyzing traffic packets. This component consists of two parts, one is a crawler, which traverses web applications, and the other is a middle proxy used to process traffic packets.

First, we write a crawler for the web application of IoT devices. This crawler logs in on the homepage according to the IP address provided by the emulation platform, recognizes the input box and clicks on the button, fills in the username and password, and traverses the web application. At the same time, it records these actions with time stamps.

While the crawler crawls the pages, the middle proxy processes the generated requests. First, it de-duplicates the messages according to the URL in the message header and filters out the useless requests, which are related to static resources like Javascript files and images. Then this component delivers the filtered messages to the Message Parser.

Message Parser: Message Parser analyzes the collected messages to construct message-state queues according to the rules in Section 3.1.

Queue Scheduler: Queue Scheduler sends the constructed state-aware queues to the Emulator and collects execution information after 10 s separately. Then this component ranks these queues according to the algorithm in Section 3.2 and sends queues to Fuzzer in order.

Fuzzer: For one message queue, Fuzzer performs the following mutations on the queue:

1. Select the Auth of the message queue for fuzzing.
2. Keep the Auth unchanged and mutate the REQ.
3. Keep the Auth and the REQ unchanged and mutate the ACT.

For each message, Fuzzer identifies the types of different values in both the header and the body. Then Fuzzer performs mutation strategies, which are shown in Table 2, according to the value Type.

Table 2. Mutating strategies.

Type	Name	Description
String	Null Value	Change the value to null to trigger potential Null Pointer Dereference (NPD) vulnerabilities.
	Long String	Extending the string with a thousand character “A” to trigger Buffer Overflow (BO) vulnerabilities.
	CI Attack	Using command injection statements to trigger potential Command Injection (CI) vulnerabilities.
	XSS Attack	Using XSS injection statements to trigger potential Cross-Site Scripting (XSS) vulnerabilities.
Number	Large Number	Mutate original number value to very large number for integer overflow to trigger potential BO vulnerabilities.
	Boundary Number	Mutate original number value to off-by-one value for out-of-bound access to trigger potential BO vulnerabilities.

Monitor: Monitor is responsible for detecting abnormal behaviors of IoT devices, identifying types of vulnerabilities, and maintaining the status of emulated IoT devices.

First of all, Monitor detects the aliveness of the web service provided by Emulator according to the socket connection, and also checks the responses from the Emulator to detects some internal crashes, such as the response code is 504.

As for bug detection, Monitor detects four kinds of vulnerabilities: Buffer Overflow (BO), Null Pointer Dereference (NPD), Command Injection (CI), and Reflected Cross-Site Scripting (XSS). Monitor reads the log information generated by the Emulator, which records the memory error, to detect BO and NPD. For CI, Monitor detects the file-open system opening “ci_file” which will be generated if the command is successfully injected. Monitor also checks the response messages from Emulator to find whether there exist injected scripts, which indicates a reflected XSS bug is triggered.

Last but not least, Monitor needs to maintain the stability of the IoT device. Once the IoT device responds with a 504 status code which means there is an internal error, or has no response for a long time, Monitor will backtrack the initial snapshot of the Emulator to restore the IoT device to its initial state. Typically, this timeout is set by the user as 5 to 10 s.

5. Evaluation

5.1. Implementation of FirmHunter

FirmHunter is implemented with around 3000 python lines of code and 1500 C lines of code in total. Furthermore, several open-source projects (e.g., Selenium [27], FirmAE [26], mitmproxy [28], and Panda [29]) are integrated into this fuzzer.

In Emulator, we build our emulation on the basis of the state-of-the-art system emulation tool FirmAE [26]. To capture the system calls within the kernel, we added a probes driver, which is written by C code, to the modified Linux kernel. Furthermore, we leverage the QEMU plugin of Panda [29] to trace the executed BB. In Traffic Collector, the crawler which coordinates with a Selenium driver [27] to traverse the web interface, is written in python code. Furthermore, the middle proxy that intercepts the messages is written in python as an application of mitmproxypackage. In Queue Scheduler, python code is written to read the output log of system calls as well as the executed BB ids and to calculate the score of each queue according to the rule in Section 3.2. In Fuzzer, we extend boofuzz [8], a popular protocol fuzzing tool with state connections in message-state queues and mutation strategies, which is written in python code. Furthermore, in Monitor, python code was written to detect potential vulnerabilities as well as outputting the result.

5.2. Evaluation of FirmHunter

In order to evaluate the effectiveness and efficiency of FirmHunter, we tested it on eight IoT firmware images and compared it with a state-of-the-art IoT fuzzer FirmFuzz [10] and a web application scanner ZAP [11]. The purpose of this section is to test whether

FirmHunter can detect vulnerabilities in real-world IoT firmware images effectively with the help of our state-message queue and introspection-driven scheduling, together with whether FirmHunter outperforms other work in vulnerability detection. In short, we would answer the following questions:

- **Q1:** Do our message-state queue and introspection-driven scheduling help to detect vulnerabilities?
- **Q2:** How effective is FirmHunter in detecting real vulnerabilities compared to other popular tools?

Testing Images: We collected more than 500 IoT firmware images and successfully emulated no more than 30 unique firmware images with accessible WEB interfaces. Within them, FirmHunter and benchmarks found vulnerabilities in eight firmware images, including seven routers and an IP camera. The detailed specification of these images is described in Table 3.

Testing Environment: Experiments are conducted on an 8-core Intel(R) Core(TM) i9-990KF 3.60 GHz CPU machine with 62.8 GB of RAM. The operating system is Ubuntu 18.04 LTS.

Table 3. Summary of firmware images under testing.

Type	Vender	Device	Version
Router	Trendnet	tew-652brp	3.04b01
	D-Link	DSL3782	1.01a01
	D-Link	DIR-865L	1.08b01
	D-Link	DAP-2695	1.11
	TP-Link	WR940	3.16.9v04
	Netgear	WNAP320	2.0.3
Ip Camera	Trendnet	ip110wn	1.2.2

5.2.1. Effectiveness of Message-State Queue and Introspection-Driven Scheduling (Q2)

Table 4 lists the vulnerabilities detected by FirmHunter. For each image under testing, FirmHunter firstly emulates it. Then FirmHunter leverages a crawler in Traffic Collector to traverse the web interface for 1 h automatically. After that, the remaining parsing, scheduling, and fuzzing rounds are performed on those collected messages within 24 h. At last, FirmHunter found 13 vulnerabilities: 7 BOs, 4 CIs, and 2 XSSs, including three unknown vulnerabilities.

FirmHunter uses state-message queues to maintain the state for the fuzzer and leverages introspection-driven scheduling to speed up the vulnerability detection. To evaluate their effectiveness, we conducted a comparison testing between FirmHunter-NUL, FirmHunter-MSQ, and FirmHunter, where

- FirmHunter-NUL directly fuzzes the messages captured by Traffic Collector;
- FirmHunter-MSQ construct a full message-state queue that satisfies all of authentication, intra-message, and inter-message dependencies;
- FirmHunter not only constructs message-state queues but also applies introspection-driven scheduling.

For FirmHunter-NUL, it does not analyze the dependencies in messages and directly fuzzes the captured messages. It finally detected three unauthorized BO vulnerabilities.

For FirmHunter-MSQ, its message-state queue satisfies authentication, inter-message, and intra-message dependencies, which helps it detect more vulnerabilities but introduces some overhead in sending REQ before ACT. Thus, FirmHunter-MSQ is slower than FirmHunter-NUL on detecting the above-mentioned 3 BO vulnerabilities except for CVE-2016-1558 in which the vulnerability lies in Auth node.

For FirmHunter, it leverages introspection-driven scheduling to speed up vulnerabilities detection by an average of 42%, comparing to FirmHunter-MSQ. One exception is

CVE-2016-1558 that FirmHunter-MSQ and FirmHunter fuzz Auth node firstly and detect this vulnerability in similar time.

The result shows that both message-state queue and introspection-driven scheduling contribute to vulnerability detection.

Table 4. Statistic on vulnerability detection against FirmHunter-NULL and FirmHunter-MSQ.

Exploited ID	Vendor	Device	Type	FirmHunter-NULL	FirmHunter-MSQ	FirmHunter	Improvement
CVE-2016-1555	Netgear	WNAP320	BO	8 min 10 s	21 min 11 s	9 min 42 s	54%
CVE-2016-1558	DLink	DAP2695	BO	51 min 43 s	6 min 39 s	6 min 43 s	−1%
CVE-2017-13772	TPLink	WR940N	BO	NA	16 min 2 s	11 min 59 s	25%
CVE-2018-19240	Trendnet	IP110wn	BO	1 h 15 min	2 h 39 min	1 h 21 min	49%
CVE-2019-11400	Trendnet	tew-652BRP	BO	NA	1 h 56 min	1 h 3 min	46%
CVE-2019-7298	DLink	DSL-3782	BO	NA	18 min 10 s	12 min 25 s	32%
CVE-2021-40284	DLink	DSL-3782	BO	NA	35 min 3 s	15 min 27 s	56%
CVE-2018-17990	DLink	DSL-3782	CI	NA	10 h 22 min	6 h 19 min	39%
CVE-2019-11399	Trendnet	tew-652BRP	CI	NA	3 h 34 min	2 h 12 min	38%
Unknown2	Netgear	WNDR3700	CI	NA	9 h 24 min	2 h 22 min	75%
Unknown3	Netgear	WNDR3700	CI	NA	14 h 51 min	5 h 27 min	63%
CVE-2018-6529	DLink	DIR-865L	XSS	NA	18 min 31 s	13 min 48 s	25%
CVE-2018-17989	DLink	DSL-3782	XSS	NA	10 h 20 min	6 h 18 min	39%

5.2.2. Comparison with FirmFuzz and ZAP(Q2)

In order to evaluate the efficiency of FirmHunter in detecting real-world vulnerabilities, we compare it with a state-of-the-art IoT fuzzer Firmfuzz and a popular web application scanner ZAP.

To configure the other two tools, we manually modified the crawler in FirmFuzz so that it can automatically traverse the web application. Furthermore, for ZAP, we set username and password in credentials for authentication, used the automated scan with both traditional spider and ajax spider. Likewise, FirmFuzz applied the crawler traversing the WEB interface to obtain initial messages and ZAP collected websites in automated scan mode within 1 h.

The discovered vulnerabilities and used time are listed in Table 5.

Table 5. Statistic on vulnerability detection against FirmFuzz and ZAP.

Exploited ID	Vendor	Device	Type	FirmHunter	FirmFuzz	ZAP
CVE-2016-1555	Netgear	WNAP320	BO	9 min 42 s	12 min 55 s	NA
CVE-2016-1558	DLink	DAP2695	BO	6 min 43 s	NA	NA
CVE-2017-13772	TPLink	WR940N	BO	11 min 59 s	NA	NA
CVE-2018-19240	Trendnet	IP110wn	BO	1 h 21 min	2 h 12 min	NA
CVE-2019-11400	Trendnet	tew-652BRP	BO	1 h 3 min	1 h 32 min	NA
CVE-2019-7298	DLink	DSL-3782	BO	12 min 25 s	NA	NA
CVE-2021-40284	DLink	DSL-3782	BO	15 min 27 s	NA	NA
CVE-2018-17990	DLink	DSL-3782	CI	6 h 19 min	NA	NA
CVE-2019-11399	Trendnet	tew-652BRP	CI	2 h 12 min	3 h 15 min	NA
Unknown2	Netgear	WNDR3700	CI	2 h 22 min	NA	NA
Unknown3	Netgear	WNDR3700	CI	5 h 27 min	NA	NA
CVE-2018-6529	DLink	DIR-865L	XSS	13 min 48 s	36 min 5 s	45 min 56 s
CVE-2018-17989	DLink	DSL-3782	XSS	6 h 18 min	NA	5 h 56 min

For FirmFuzz, it only detected five vulnerabilities due to the neglect of dependencies between messages. Furthermore, for ZAP, the front-end web application scanner, it could just find 2 XSS vulnerabilities.

The results show that FirmHunter not only finds more vulnerabilities than other tools, but also takes less time to find them in almost all cases.

6. Related Work

Since IoT programs are commercially available, few manufacturers disclose source code and documents. We can only obtain firmware images or purchase physical devices to analyze IoT devices. As firmware contains more information and lends access for large-scale analysis, when the firmware is available, performing analysis on firmware is a better choice. To obtain firmware, one way is to extract firmware from real devices through the debugging interface or a flash chip [30]; another way is to download it from the manufacturer's official websites. After that, Binwalk [31] and other firmware extractors are used to extract all files and programs within firmware for further static analysis or dynamic testing.

6.1. Static Analysis on Firmware

The homology analysis, which is based on the similarity analysis on file granularity, was first applied on IoT firmware by Costin et al. [21]. They found 38 unknown vulnerabilities in 693 firmware images. Through similar file associations, they found that some vulnerabilities infected 123 different products, affecting 140,000 physical devices in cyberspace. However, only the coarse-grained file similarity comparison was performed in this work, which resulted in a low accuracy rate. Thomas et al. [32] proposed a technique using static data analysis to find hardcoded vulnerabilities. This work identified data comparison functions by modeling function features and ranked these functions based on the uniqueness of their comparing data. Through further analysis of the important functions, the hard-coded authentication backdoor vulnerability was detected. However, in the applications of backdoor vulnerability detection and text-based protocol instruction recovery, it could not promise a high accuracy rate. Hence, the applicability of the program in large-scale testing is not good.

6.2. Dynamic Testing on Firmware

In order to overcome the dependence of dynamic testing such as fuzzing on the hardware, Chen et al. [25] proposed Firmadyne, a large-scale full-system emulation platform for Linux-based firmware. By modifying the kernel and drivers to add hardware support to realize the full-system emulation. This work completed the decompression of 9486 firmware and tests of 74 exploit scripts. On this basis, Costin et al. [21] conducted testing on the web interfaces of IoT devices and found 45 unknown vulnerabilities. Chen et al. [33] resorted to APPs of IoT devices. Through reversing an APP, rich protocol information like URLs, commands, encryption algorithms will be analyzed, which helps generate valid test cases to detect vulnerabilities. Furthermore, it finally detected 15 types of memory corruption vulnerabilities in eight devices. However, this work is limited to IoT devices with APPs and suffered from the low throughput of physical devices.

7. Discussion

Although FirmHunter could resolve the dependency problem using our message-state queue and accelerate the vulnerability detection process with the help of introspection-driven scheduling, there still exist some limitations in our framework.

7.1. Limitation on IoT Firmware Emulation

The emulation platform in FirmHunter is implemented on the basis of FirmAE which is limited by the CPU architecture and operating system of a firmware image. For now, it only supports three CPU architectures: mipsel, mipseb, and armel. Furthermore, it can only run a POSIX-compatible OS (e.g., Linux). This limitation stems from the system emulation tool and we need to enlarge the range of applicable IoT firmware images. We will improve the emulation capability to support more CPU architectures as well as non-POSIX programs for future work.

7.2. Limitation on Protocol Types for Fuzzing

For now, we only fuzz IoT devices via their web interfaces using the HTTP protocol. However, IoT devices not only use HTTP protocols but also provide services via SOAP, HNAP, ftp, or other protocols. We expect that FirmHunter can fuzz IoT through multiple protocols in the future.

8. Conclusions

We have proposed an automated stateful and introspection-driven grey-box fuzzer towards Linux-based firmware images. It utilizes the message-state queue to satisfy authentication, intra-message, and inter-message dependencies; it implements introspection-driven scheduling to collect system execution information and thus ranks test cases based on that information. The evaluation on eight firmware images indicates that message-state queue can help FirmHunter detect vulnerabilities other tools cannot and introspection-driven scheduling can provide an average 42% acceleration. FirmHunter outperforms other tools on vulnerability detection has the capability to find unknown vulnerabilities.

Author Contributions: Conceptualization, Q.Y. and X.Z.; methodology, Q.Y.; software, Q.Y.; validation, Q.Y., X.Z. and H.Z.; formal analysis, H.Z.; investigation, Q.Y.; resources, X.Z.; data curation, Q.Y.; writing—original draft preparation, Q.Y.; writing—review and editing, H.Z.; visualization, Q.Y.; supervision, X.Z.; project administration, Q.Y.; funding acquisition, X.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research is funded by Natural Science Foundation of China(61902412), Research Project of National University of Defense Technology(ZK20-17), National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), NSF 61902405 and the HUNAN Province Science Foundation 2017RS3045.

Data Availability Statement: Not applicable.

Acknowledgments: We thank our shepherd Xu Zhou for his care and the anonymous reviewers for their insightful comments on our work.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

IoT	Internet of things
XSS	Cross-site Scripting
BO	Buffer Overflow
CI	Command Injection
HTTP	HyperText Transfer Protocol
CVE	Common vulnerabilities and exposures

References

1. Lionel Sujay Vailshery. IoT Connected Devices Worldwide 2030. Available online: <https://www.Statista.Com/statistics/802690/worldwide-Connect> (accessed on 15 August 2021).
2. Shadon. The Search Engine for Refrigerat. 2021. Available online: <https://www.shodan.io/> (accessed on 15 August 2021).
3. Technology, P. Web Application Vulnerabilities: Statistics for 2018. 2019. Available online: <https://www.ptsecurity.com/ww-en/analytics/web-application-vulnerabilities-statistics-2019/> (accessed on 15 August 2021).
4. RIPS. RIPS—A Static Source Code Analyser for Vulnerabilities in PHP. 2018. scripts. Available online: <http://rips-scanner.sourceforge.net/> (accessed on 15 August 2021).
5. Chen, C.; Cui, B.; Ma, J.; Wu, R.; Guo, J.; Liu, W. A systematic review of fuzzing techniques. *Comput. Secur.* **2018**, *75*, 118–137. [CrossRef]
6. Peach. Peach Fuzzing Platform. 2011. Available online: <https://www.peach.tech/> (accessed on 15 August 2021).
7. OpenRCE. A pure-Python Fully Automated and Unattended Fuzzing Framework. 2012. Available online: <https://github.com/OpenRCE/sulley> (accessed on 15 August 2021).
8. Boofuzz. Boofuzz: Network Protocol Fuzzing for Humans. 2021. Available online: <https://boofuzz.readthedocs.io/en/stable/index.html/> (accessed on 15 August 2021).

9. Song, C.; Yu, B.; Zhou, X.; Yang, Q. SPFuzz: A Hierarchical Scheduling Framework for Stateful Network Protocol Fuzzing. *IEEE Access* **2019**, *7*, 18490–1849. [CrossRef]
10. Srivastava, P.; Peng, H.; Li, J.; Okhravi, H.; Shrobe, H.; Payer, M. FirmFuzz: Automated IoT Firmware Introspection and Analysis. In Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, London, UK, 11–15 November 2019.
11. OWASP. OWASP Zed Attack Proxy(ZAP). 2021. Available online: <https://www.zaproxy.org/> (accessed on 15 August 2021).
12. Arachni. Home-Arachni-Web Application Security Scanner Framework. 2017. Available online: <https://www.arachni-scanner.com/> (accessed on 15 August 2021).
13. w3af. w3af—Open Source Web Application Security Scanner. 2018. Available online: <http://w3af.org/> (accessed on 15 August 2021).
14. Wang, E.; Wang, B.; Xie, W.; Wang, Z.; Yue, T. EWHunter: Grey-Box Fuzzing with Knowledge Guide on Embedded Web Front-Ends. *Appl. Sci.* **2020**, *10*, 4015. [CrossRef]
15. Yu, B.; Wang, P.; Yue, T.; Tang, Y. Poster: Fuzzing iot firmware via multi-stage message generation. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 2525–2527.
16. Zhang, H.; Lu, K.; Zhou, X.; Yin, Q.; Wang, P.; Yue, T. SIOtFuzzer: Fuzzing Web Interface in IoT Firmware via Stateful Message Generation. *Appl. Sci.* **2021**, *11*, 3120. [CrossRef]
17. Zalewski, M. American Fuzzy Lop. 2014. Available online: <https://lcamtuf.coredump.cx/afl/> (accessed on 15 August 2021).
18. Bellard, F. QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference, Freenix Track, Monterey, CA, USA, 10–15 June 2002; Volume 41, p. 46.
19. Muench, M.; Nisi, D.; Francillon, A.; Balzarotti, D. Avatar 2: A multi-target orchestration platform. In Proceedings of the Workshop Binary Anal. Res. (Colocated NDSS Symp.), Nara, Japan, 4 December 2018; Volume 18, pp. 1–11.
20. King, J.C. Symbolic execution and program testing. *Commun. ACM* **1976**, *19*, 385–394. [CrossRef]
21. Costin, A.; Zarras, A.; Francillon, A. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, Vienna, Austria, 24 October 2016.
22. Gustafson, E.; Muench, M.; Spensky, C.; Redini, N.; Machiry, A.; Fratantonio, Y.; Balzarotti, D.; Francillon, A.; Choe, Y.R.; Kruegel, C.; et al. Toward the analysis of embedded firmware through automated re-hosting. In Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019), Beijing, China, 23–25 September 2019; pp. 135–150.
23. Wang, D.; Zhang, X.; Chen, T.; Li, J. Discovering Vulnerabilities in COTS IoT Devices through Blackbox Fuzzing Web Management Interface. *Secur. Commun. Netw.* **2019**, *2019*, 5076324. [CrossRef]
24. Zheng, Y.; Davanian, A.; Yin, H.; Song, C.; Zhu, H.; Sun, L. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In Proceedings of the USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019.
25. Chen, D.D.; Woo, M.; Brumley, D.; Egele, M. *Towards Automated Dynamic Analysis for Linux-based Embedded Firmware*; NDSS: New York, NY, USA, 2016.
26. Kim, M.; Kim, D.; Kim, E.; Kim, S.; Jang, Y.; Kim, Y. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In Proceedings of the Annual Computer Security Applications Conference, Online, 7–11 December 2020.
27. McMahon, C. History of a Large Test Automation Project Using Selenium. In Proceedings of the Agile Conference, Hannover, Germany, 2–5 June 2009.
28. Mitmproxy. Mitmproxy—An Interactive HTTPS Proxy. 2021. Available online: <https://mitmproxy.org/> (accessed on 15 August 2021).
29. Dolangavitt, B.F.; Hodosh, J.; Hulin, P.; Leek, T.; Whelan, R. Repeatable Reverse Engineering for the Greater Good with PANDA. In Proceedings of Workshop on Program Protection and Reverse Engineering (PPREW), Los Angeles, CA, USA, 8 December 2015. [CrossRef]
30. Vasile, S.; Oswald, D.; Chothia, T. Breaking all the things—A systematic survey of firmware extraction techniques for IoT devices. In *International Conference on Smart Card Research and Advanced Applications*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 171–185.
31. Binwalk. Binwalk: Firmware Analysis Tool. 2015. Available online: <https://github.com/ReFirmLabs/binwalk/> (accessed on 15 August 2021).
32. Thomas, S.L.; Chothia, T.; Garcia, F.D. Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality. In Proceedings of the European Symposium on Research in Computer Security, Oslo, Norway, 11–15 September 2017.
33. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In Proceedings of the Network and Distributed System Security Symposium, Diego, CA, USA, 18–21 February 2018.