An Efficient Feedback-enhanced Fuzzing Scheme for Linux-based IoT Firmwares

Qidi Yin
National University of Defense Technology
Changsha, China
yinqidi@nudt.edu.cn

Xu Zhou
National University of Defense Technology
Changsha, China
zhouxu@nudt.edu.cn

Hangwei Zhang
National University of Defense Technology
Changsha, China
zhanghangwei@nudt.edu.cn

***Abstract*—As the number of IoT devices grows at an exponential rate, the security issues of these devices are having a huge impact on people's lives. Fuzzing, a dynamic testing method that can be automated at scale, is becoming more and more extensively utilized on Io devices in order to find the vulnerabilities in these devices. In this work, we present an efficient feedback-enhanced fuzzing scheme for Linux-based IoT firmwares. It uses a two-level scheduler and a feedback-enhanced monitor to collect operating data for seed selection and ranking. We develop a prototype system and evaluate it by emulating and testing five different firmwares. The result shows that our system is able to effectively discover more known vulnerabilities than the state-of-the-art IoT fuzzer and eventually discovered 5 unknown vulnerabilities.**

***Index Terms*—IoT; Security; Gre-box fuzzing; Feedback**

I. Introduction

With the advancement of Internet of Things (IoT), IoT devices are no longer restricted to certain commercial applications; they've made their way into our homes and have become an integral part of our daily lives. According to estimates [1], there will be over 75 billion IoT connected devices in use by 2025, which is a roughly threefold growth over the installed base of IoT in 2019. Linux-based systems are commonly employed in IoT devices, as they are open source and manufacturers can customize a simple Linux-based system to meet their own requirements. Because IoT devices have limited computational resources, system performance optimization and application functionality are prioritized. As a result, IoT devices are more vulnerable when hacked and it's critical to assess the IoT devices' dependability and security.

Fuzzing [2], which is effective in finding vulnerabilities in both software and systems, is the most promising dynamic testing technique applied in IoT devices. Fuzzing identifies vulnerabilities by supplying random data to the target and looking for anomalies in the program or system. Fuzzing towards IoT devices often sends inputs in protocol format to IoT devices and detects the abnormal behaviors like hangs or no responds.

Emulation technologies are widely used in IoT fuzzing [3]–[9] as the target firmware can be simply downloaded from the internet, extracted [10], and emulated on desktop computers. Firm-AFL [6] combines user-level and system-level emulation to deliver high throughput fuzzing on targets. FirmFuzz [7] and EWVHunter [8] design a crawler to browse the WEB interface of target IoT devices, then display multiple mutations on collected messages and replay them as inputs to target IoT devices. On top of crawler-based fuzzers, SIoTFuzzer [9] analyzes intra-dependencies between messages and generates stateful inputs for targets.

1) Challenges: Despite the fact that crawler-based grey-box fuzzers such as FirmFuzz and SIoTFuzzer use cutting-edge fuzzing technologies, they are still ineffective in detecting real-world vulnerabilities for the following reasons. First, in these fuzzers, all test cases are generated by mutating seed from an initial seed set. The initial seed set is collected by capturing real-word messages in communication in an initial phase. During fuzzing, they apply mutations on these messages field by field to generate well-structured inputs. However, simply modifying a field each time is insufficient, i.e., the generated test cases are unlikely to uncover deeper flaws. For example, the body of a POST message usually contains multiple parameters, as shown in Fig.1(a). Once mutation is applied to just one field at a time, inputs will be generated as shown in Fig.1(b) or Fig.1(c). However, in order to exploit the buffer overflow (BO) vulnerability, we have to build the body as shown in figure Fig.1(d), which cannot be done by existing approaches.

```
Type=t&Addr=192.168.1.1
(a)

Type=XXX&Addr=192.168.1.1
(b)

Type=t&Addr=XXX
(c)

Type=p&Addr=A*100
(d)
```

Fig. 1. The body of a POST message

Second, there is a lack of seed scheduling, which slows the discovery of vulnerabilities. FirmFuzz, for example, uses a crawler to traverse the web interface and apply a series of message mutations after collecting a POST message. Not only does it ignore the waste generated by fuzzing duplicate

978-1-6654-3205-4/21/$31.00 ©2021 IEEE

messages, but it also ignores the priority of various types of messages. Regardless of the fact that SIoTFuzzer captures all messages while traversing the web interface and filters out duplicates, it still lacks seed scheduling.

2) Our Solution: To solve the above problems, we proposed a feedback-enhanced fuzzing scheme for Linux-based firmware to address the aforementioned problems. To update the seed set, we trace the coverage of the emulated platform each time we mutate a seed to generate an input and send it to the target IoT device. For input that brings coverage increasement, we put it into the seed set and afterwards generating more inputs based on that new seed. To compensate for lack of scheduling, we do a first run on all of the seeds and rank them using a two-level scheduling mechanism. And each time we find a coverage-increasing input, its score will be used to prioritized it in the seed set.

To evaluate our system, we implement it on the top of SIoTFuzzer [9]. We collect five firmwares including four routers and an IP camera for our evaluation and emulate these firmwares based on FirmAE [11]. Following that, we evaluate the performance of our system by comparing the number of vulnerabilities discovered and the time used with SIoTFuzzer. Our evaluation demonstrates that (1) our system can find more vulnerabilities than SIoTFuzzer in shorter time and (2) our system is able to find unknown unknown vulnerabilities.

3) Contributions: In summary, we make the following contributions in our paper.

- We design and develop a feedback-enhanced fuzzing prototype for IoT firmwares. Using feedback in emulated IoT devices, our system can update the seed set for finding deep vulnerabilities and schedule seeds for finding vulnerabilities faster.
- The evaluation on real-world IoT firmwares shows that our approach could efficiently find more vulnerabilities in these firmwares than the state-of-the-art tool and is able to find unknown vulnerabilities.

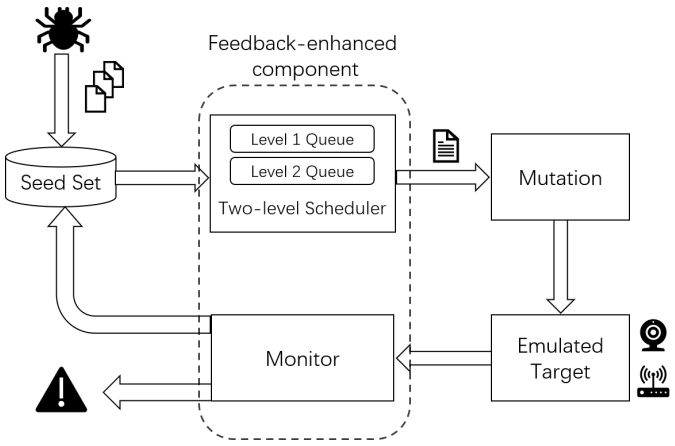

Fig. 2. The main components of our system

II. Overview

To address the challenges raised in the preceding section, we present a feedback-enhanced fuzzing scheme for IoT firmwares. Fig.2 depicts a high-level overview of its main components. A feedback-enhanced fuzzing loop is included in our method to aid in the development of new inputs from the most recently added seed, which is aimed for deeper vulnerabilities. Furthermore, our system analyses the attributes of each seed in the seed set and ranks them using a two-level scheduling based on feedback from the emulation target.

A. Workflow

The workflow of our system is shown in Algorithm1.

To begin, we use a crawler to capture messages in communication. We put each message in the seed set and send it to the emulated target for a first run, as illustrated in lines 3-4. To evaluate this seed, our feedback-enhanced monitor (II-B) will collect operating data such as system calls and basic block coverage. Once completed, this seed will be inserted into our two-level scheduler (II-C) based on its privilege and score. Then there's our fuzzing loop, which is depicted in lines 8-19. Every time, a more privileged seed is selected from the queue and mutated to generate more inputs using various mutation policies (II-D). We send each input to the target emulated device and capture its operating data. If it adds new block coverage, we preserve it as a new seed in the seed set, as indicated in lines 12-14. And once a vulnerability is discovered (II-B), we will save it into our vulnerability report.

Algorithm 1 Workfow of our system.

Require: Captured messages, $msgs$;
Ensure: Discovered Vulnerabilities, $vuls$;

```
1:  Initialize seed set seeds, block coverage cov, secondary
    queue queue and discovered vulnerabilities vuls;
2:  for each msg in msgs do
3:      seeds.append(msg)
4:      privilege, score, cov_inc, bug=Monitor_run(msg)
5:      cov.append(cov_inc)
6:      queue.insert(privilege, score)
7:  end for
8:  while queue is not NULL do
9:      seed=Choose(queue)
10:     input=Mutate(seed)
11:     privilege, score, cov_inc, bug=Monitor_run(input)
12:     if cov_inc not in cov then
13:         seeds.append(input)
14:         queue.insert(privilege, score)
15:     end if
16:     if bug then
17:         vuls.append(bug)
18:     end if
19: end while
20: return vuls;
```

B. Feedback-enhanced Monitor

With the help of emulation technology, feedback-enhanced monitor is able to collect operating information with the emulated targets. It mainly collects two types of information: system calls and executed basic block (BB) coverage. As for the system calls, the monitor modifies the driver in Linux kernel, which will hook the system calls we want to collect. And for BB coverage, we utilize the QEMU plugin to trace the ids of the executed basic blocks.

Each time a seed or new-generated input is send to the target emulated devices, this Monitor is invoked to trace the operating information on it. Providing the detected information, feedback-enhanced monitor is responsible for the three following functions:

First and foremost, it gathers executed BB coverage in order to update the seed set. After a newly-generated test case brings new BB coverage, we will add it to the seed set and further push it into the two-level scheduler.

Second, it checks the emulated device's status. Monitor drawback the status of the emulated device to its initial state by using snapshots once there is no response or the target's service is closed.

Last but not least, monitor identifies the vulnerabilities and generates a report. Buffer Overflow (BO), Command Injection (CI), and Cross-Site Scripting (XSS) are the three types of vulnerabilities it can identify (XSS). To detect BO, Monitor examines the log information created by the Emulator, which records the memory error. Monitor detects the file-open system calls for CI, which creates a "ci file" if the attacking payload is successfully injected. Monitor also examines Emulator's response messages for injected scripts, which indicates that a XSS vulnerability has been triggered.

C. Two-level Scheduler

Given the operating information, the two-level scheduler will push a seed or an input to queues based on the following rule:

a) : For each input we instrument each field with a random string, after sending this new input to the target, we utilize monitor to obtain information on the system call do_execve, as well as its argument strings. Once we discover that the instrumented random strings are in the do_execve argument strings, we presume that the privilege of this input is 1, indicating that the input is subject to less stringent scrutiny and is more likely to cause vulnerabilities.

b) : In the meantime, the monitor will track the BB coverage as an input is executed. If the input is inside the initial seeds, regardless of whether it increases BB coverage, it will be assigned to two-level queues based on its privilege. And for generated input, only when it increases BB coverage, it will be added to the seed set as a new seed and pushed into two-level queues based on its privilege.

c) : If the seed has privilege 1, it will be added to Level 1 queue. Otherwise, it will be added to the Level 2 queue. We believe that the more BB coverage traversed, the more probable it is that the seed will cause vulnerabilities. And, as the number of fields in the seed decreases, so does the amount of time spent in the fuzzing loop. As a result, the seeds are ranked according to the following rule:

$$Score = \frac{inc\_kernel + inc\_program}{field\_num} \quad (1)$$

where *inc_kernel* denotes the increased kernel BB coverage, *inc_program* denotes the increased program BB coverage and *field_num* denotes the number of fields in the body this message.

D. Fuzzing Policy

The higher scored seeds are selected first from the Level 1 queue during the fuzzing process. When the Level 1 line is empty, the better scoring seeds are selected from the Level 2 queue. After a seed is selected, we will apply the following fuzzing strategies to each field in the seed:

a) Changing string lengths and number numerical values of numbers for Buffer Overflow (BO) access: To trigger potential BO vulnerabilities, we add a thousand characters "A" to the string or change the original number to a very big amount.

b) Injecting attacking payloads for Cross-Site Scripting (XSS) and Command Injection (CI) access: To trigger potential CI or XSS vulnerabilities, we used attacker payloads such as *;/test*, *<script>alert("XSS");</script>*.

c) Modifying the value for more complex operational logic: We modify the value of the string field to "Yes", "No", or other spliced letters, and the value of the numeric field to "0", "1", and minus. This method is used to generate inputs that will result in more BB coverage being executed.

III. Implementation and Evaluation

In this section, we present the evaluation of our approach and the results from our experiment. To evaluate the effectiveness of finding vulnerabilities, we implement our system and compare it with a state-of-the-art IoT fuzzer SIoTFuzzer on a set of five Linux-based firmwares.

A. System implementation

We have implemented our system using python and C languages. In message preparing phrase, we employ mitmproxy [12] and selenium [13] to traverse and collecting communicating messages. In the feedback-enhanced monitor, we patch the Linux kernel [14] to hook the system calls and develop a PANDA [15] plugin to trace the running BB coverage information in the QEMU. In the fuzzing phrase, we build our system on the top of SIoTFuzzer [9].

TABLE I
SUMMARY OF FIRMWARES UNDER TESTING

Type	Vender	Model	Version
Router	Trendnet	tew-652BRP	3.04b01
	Dlink	DSL-3782	EU1.01
	DLink	DAP-2555	REVA1.20
	Netgear	WNDR3700	v2-1.0.1.14
IP Camera	Trendnet	IP110wn	v2-1.2.2.68

TABLE II
STATISTICS ON FUZZING – DISCOVERED VULNERABILITIES AND TEST TIME

Exploited ID	Type	Vendor	Model	SIoTFuzzer	Our system
CVE-2018-19240	BO	Trendnet	IP110wn	3h23min	1h41min
CVE-2019-11400	BO	Trendnet	tew-652BRP	18min52s	1h52min
CVE-2019-7298	BO	DLink	DSL-3782	17h41min	6min38s
Unknown1	BO	DLink	DSL-3782	NA	3h16min
CVE-2018-17990	CI	DLink	DSL-3782	9h20min	5h52min
CVE-2019-11399	CI	Trendnet	tew-652BRP	7h11min	1h2min
Unknown2	CI	Netgear	WNDR3700	NA	3min13s
Unknown3	CI	Netgear	WNDR3700	NA	4h8min
CVE-2018-17989	XSS	DLink	DSL-3782	9h51min	6h23min
CVE-2021-31655	XSS	Trendnet	IP110wn	NA	1min3s
Unknown4	XSS	DLink	DAP-2555	NA	3min39s
Unknown5	XSS	DLink	DAP-2555	NA	5h8min

B. Experiment setup

Target: We collected five Linux-based IoT firmwares including four routers and an IP camera and utilize FirmAE [11] to configure these firmwares. The basic information of testing firmwares is shown in TableI.

Testing Environment: We conducted our evaluation in a virtual machine with an Intel Core i9 quad-core 3.6 GHz CPU and 8G RAM. The operating system is Ubuntu 18.04.

Before the fuzzing test, we emulated the testing firmwares and utilize a crawler to traverse their web applications. After the crawling, we get the communication messages and delete the duplicated messages, which are used as initial seeds.

Then we feed these seeds to our system as well as SIoTFuzzer and run them within 24 hours.

C. Fuzzing Results

After 24-hour running, the discovered vulnerabilities and their times of discovered are listed in TableII.

Number of vulnerabilities: From the result, we can see that our system discovered 12 vulnerabilities, including 4 BO, 4 CI, and 4 XSS vulnerabilities, whereas SIoTFuzzer only discovered 6. With the help of the feedback-enhanced monitor, which updates the seed set to explore deeper logics of emulated devices, our system was able to find more 1-day vulnerabilities.

Time for finding: Our system can find a vulnerability faster than SIoTFuzzer except for CVE-2019-11400. In this case, as SIoTFuzzer fuzzes the initial seeds in the order of the messages' alphabetical order and it fuzzes the valuable seed at firstly by coincidence. In other cases, our system takes a significant lead over SIoTFuzzer with the help of two-level scheduler.

0-day vulnerabilities: We discovered 6 0-day vulnerabilities within 6 hours, with a minimum time of less than four minutes. All of these vulnerabilities were reported to the manufacturers, one of which was assigned a CVE ID (CVE-2021-31655), and three of which were mentioned in the security announcement [16].

IV. RELATED WORK

A. Dynamic testing on IoT devices

Chen et al. [17] proposed Firmadyne, a large-scale full-system emulation platform for Linux-based firmware, to address the hardware dependence problem of dynamic testing such as fuzzing. By altering the kernel and drivers to enable hardware support, full-system emulation can be achieved. 9486 firmwares were decompressed and 74 exploit scripts were performed on the emulated firmwares in this research. Costin et al. [18] built a system on Firmadyne to evaluate the web interfaces of IoT devices and discovered 45 undiscovered vulnerabilities. Zaddach et al. developed Avatar [19] to route I/O accesses from Emulator to physical devices in order to obtain running data in physical devices.

B. Firmware fuzzing

Chen et al. [4] combines APPs of target IoT devices and the fuzzing technology. Rich protocol information such as URLs, commands, and encryption techniques will be evaluated when reversing an APP, which will assist develop valid test cases to find flaws. Finally, it discovered 15 different forms of memory corruption vulnerabilities in eight different devices. However, this research was limited to IoT devices running APPs and was hampered by physical device throughput. To combine the advantages of system emulation and the user emulation, Firm-AFL [6] proposed augmented process emulation, which greatly accelerates the discovery of vulnerabilities in system emulation. Prashast Srivastava et al. implemented FirmFuzz [7], an automated fuzzing framework which detects vulnerabilities in target firmwares on the base of system introspection. SIoTFuzzer [9] focus on stateful seed generation by analyzing the initial inputs.

V. CONCLUSION

In this paper, we present an efficient feedback-enhanced fuzzing scheme for Linux-based firmwares. To overcome problem of the fixed seed set that prevent fuzzing from discovering deep vulnerabilities, we design a feedback-enhanced monitor to collect operating data and select new seeds for further fuzzing. In addition. we propose a two-level scheduler ranking

seeds based on their running properties to speed up vulnerabilities finding. In our evaluation, our system could find more vulnerabilities than the state-of-the-art IoT fuzzer SIoTFuzzer and eventually discovered 5 unknown vulnerabilities.

REFERENCES

[1] R. Statista, "Internet of things-number of connected devices worldwide 2015-2025," *Statista Research Department. statista. com/statistics/471264/iot-numberof-connected-devices-worldwide*, 2019.

[2] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, pp. 118–137, 2018.

[3] B. Feng, A. Mera, and L. Lu, "P^{2}im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling (extended version)," 2019.

[4] J. Chen, W. Diao, Q. Zhao, C. Zuo, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *Network and Distributed System Security Symposium*, 2018.

[5] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing iot firmware via multi-stage message generation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2525–2527.

[6] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation," in *USENIX Security Symposium*, 2019.

[7] Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "Firmfuzz: Automated iot firmware introspection and analysis," *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, 2019.

[8] E. Wang, B. Wang, W. Xie, Z. Wang, and T. Yue, "Ewvhunter: Grey-box fuzzing with knowledge guide on embedded web front-ends," *Applied Sciences*, vol. 10, no. 11, p. 4015, 2020.

[9] H. Zhang, K. Lu, X. Zhou, Q. Yin, P. Wang, and T. Yue, "Siotfuzzer: Fuzzing web interface in iot firmware via stateful message generation," *Applied Sciences*, vol. 11, no. 7, p. 3120, 2021.

[10] Binwalk, "binwalk: Firmware analysis tool," https://github.com/ReFirmLabs/binwalk/, 2015.

[11] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," *Annual Computer Security Applications Conference*, 2020.

[12] Mitmproxy, "mitmproxy - an interactive https proxy," https://mitmproxy.org/, 2021.

[13] C. Mcmahon, "History of a large test automation project using selenium," in *Agile Conference*, 2009.

[14] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, "Probing the guts of kprobes," in *Linux Symposium*, vol. 6, 2006, p. 5.

[15] B. F. Dolangavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering for the greater good with panda," *Computer Science*, 2014.

[16] D-Link, "Support announcements," https://supportannouncement.us.dlink.com/announcement/publication.aspx?name=SAP10232, 2021.

[17] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *NDSS*, 2016.

[18] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[19] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, vol. 18, 2018, pp. 1–11.