FIoTFuzzer: Response-Based Black-Box fuzzing for IoT Devices

Zelin Xu
State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing,China
eddiexu@cuc.edu.cn

Wei Huang
State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing,China
huangwei.me@cuc.edu.cn

Wenqing Fan
State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing,China
fanwenqing@cuc.edu.cn

Yixuan Cheng *
State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing,China
cucfitz@gmail.com

2022 IEEE/ACIS 22nd International Conference on Computer and Information Science (ICIS) | 978-1-6654-9463-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICIS54925.2022.9882418

***Abstract*—To prevent IoT devices from being exploited, it is particularly important to detect vulnerabilities as many as possible during the device development process. The black-box fuzzing test is widely used in vulnerability detection for IoT devices for several reasons. First of all, the source code of the firmware is rarely provided in public, device response messages are a valuable source of device status. In legacy black-box fuzzing tests, there was a lack of checks on network protocols, message formats and encodings. Byte-to-byte mutation without these checks produced a large amount of garbage input data, which could not reach the deep-level function code. The efficiency and accuracy of fuzzing testing were negatively impacted accordingly. Secondly, communication protocol specification of firmware is rarely provided in public too, and it is difficult for existing grammar-based fuzzing strategies to distinguish the meaning of each field of the message. To solve the above issues, this paper proposes a response-based black-box fuzzing method, named FIoTFuzzer. We set up a message adapt-er to identify the protocol, format, encoding and other information of original communication packets. To improve the syntax inference capability, FIoTFuzzer divides the message segment based on the response, avoiding blind mutation of the content. This method of using mutation strategy based on message segment under the premise of format specification can reach deep functional components of smart devices. This fuzzing method has lightweight dependencies and does not require reverse engineering. Our tests were evaluated on 12 IoT devices, which included routers, smart bulbs and IP cameras. The results show that: (1) FIoFuzzer is able to detect real-world vulnerabilities in IoT devices; (2) In our benchmark comparison tests with Boofuzz and Sulley, FIoTFuzzer detected 9 vulnerabilities while Boofuzz detected only 5 and Sulley detected only 4 among these 9 vulnerabilities.**

Keywords—IoT device, blackbox fuzzing, response-based mutation, message fragment mutation

I. INTRODUCTION

In recent years, with the outbreak pandemic of the Covid-19 virus, the digital transformation of many industries has become an urgent need. The large-scale promotion of hybrid work and hybrid teaching models are the best proof, which promotes the continuous development of network-related industries. IoT devices also entered a period of rapid development. But with the proliferation of IoT devices, a more complex IoT ecosystem is shaping, along with security-related issues. According to a recent report, following the onset of COVID-19, 85% of CISOs overlooked cybersecurity to quickly enable remote working [1]. In another research, when asked if they are confident that they have visibility of the IoT devices connected to their organization's network, 97% of IT decision-makers responded with an affirmative answer that they were [2]. However, according to Palo Alto Networks research, which examined 135,000 security cameras in March, 54% of the examined cameras had at least one vulnerability. Such vulnerabilities make it possible for cameras to be hijacked and subsequently weaponized by cybercriminals, setting up these devices as springboards to perpetrate attacks and access broader corporate networks. In September 2020, IBM X-Force reported that IoT attacks we observed from October 2019 through June 2020 rose 400% when compared to the combined number of IoT attacks in the previous two years [3]. IoT devices have become part of everyday consumer and business activities, and threats in that space are driven by the availability of devices, most of which are poorly secured, and by prominent IoT botnets, such as Mozi and Mirai. It's important to remember that source code from various IoT malware can be reused in other variants that may be detected as the same malware.

There are common logic defects and security vulnerabilities in IoT device firmware. To save development costs and improve efficiency, some developers directly use third-party components but do not pay attention to whether there are security risks or defects in the introduced components, which results in a large number of vulnerabilities in the firmware. They can be easily exploited by attackers. Almost all software contains third-party libraries. According to Gartner statistics, in 2018, the amount of code using third-party libraries in software accounted for 80% of the total, and the proportion of self-developed code is getting lower. By default, software security should be guaranteed by the component provider, the upstream of the supply chain. But, through the analysis of some famous open source projects, it is found that these projects hardly do any security verification [4].

The complexity of the format, diversity of the architecture and the difficulty of the simulation operation are the main differences between the firmware program of the embedded device and an ordinary binary program. Because of these differences, the common fuzzing methods and tools cannot be directly used for IoT devices. Since there is no unified firmware format standard for embedded devices, most manufacturers will not provide firmware publicly, and the public firmware may be encrypted by a custom encryption format to prevent users from decrypting. Therefore, black-box fuzzing has become one of the main methods for researchers to conduct security research.

978-1-6654-9463-2/22/$31.00 ©2022 IEEE
ICIS2022, June 26-28, 2022, Zhuhai, China

Feng et al. [5] propose a black-box fuzzing, Snipuzz, which analyzes response codes, using them as indicators to identify sensitive bytes of the inputs (snippets) that trigger different paths in the target due to the difficulty of knowing the internal details of the firmware. While too strict message fragment classification in Snipuzz may result in too fragmented response types. Also, Snipuzz's approach cannot produce under-constrained yet well-structured(i.e., accepted by the IoT device) fuzzing inputs, which can reach deeper code locations, uncovering more vulnerabilities.

Our Approach. In this paper, we propose and implement a black-box fuzzing framework based on mutated message fragment inference. To overcome the limitations in Snipuzz, we improved the mechanism of message snippet inference to reduce message fragmentation. Consider that IoT devices use numerous network protocols (such as HTTP, Zigbee) and data exchange formats (such as JSON, SOAP, XML) and extremely strict syntax checking. The traditional method of byte-by-byte mutation can easily break the grammar rules. Test cases that are rejected just because of format errors are difficult to trigger deep-level vulnerabilities. At the same time, some fields may be encoded. Directly mutating the encoded part will also lead to IoT device firmware can't handle it. We will identify the data format in the data message. In the case of not destroying the data format, perform operations such as mutation on message fragments outside the format to improve code coverage as much as possible.

Contributions. We summarize the contributions of the paper as follows:

- **New technique.** We propose a message adapter to more accurately adjust the relationship between IoT device logic verification and data encoding and format. This adapter can control the generation and delivery of messages which enables fuzzers to implement effective IoT device fuzzing during execution.
- **New framework.** We introduce a framework for handling information formats and data encoding during mutation. To overcome the limitations of Snipuzz, FIoTFuzzer analyzes the protocol, format, and coding in the process of processing original data, mutating the message segments outside the format, and then fully encoding the mutated data. They are well-structured so that they won't be immediately discarded by the fuzzed IoT device. To prevent message segments from being too fragmented, we set up undulators to control the inefficiency caused by a large number of response categories.
- **Implementation and findings.** The group implemented FIoTFuzzer with Python and compared it with other well-known black-box fuzzing tools. In experiments, we found 9 known vulnerabilities in 8 devices, which shows the efficiency of our method.

Roadmap. In the remainder of this article, Section 2 provides a review of the background and related work of smart device fuzzing and fuzzers. Section 3 presents a detailed design of FIoTFuzzer and Section 4 covers its implementation details and summarized evaluation results. Section 5 discusses some limitations of the current design and points out the future work. Finally, Section 6 concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Fuzzing's purpose and method

Fuzz testing is a method to find faults in software by providing invalid, unexpected or random data as inputs to the program and monitoring the anomalies in the output such as crashes, failing built-in code assertions, or potential memory leaks. Applying fuzzing test technology to the security testing of IoT devices is one of the important research directions in the field of network security. It is of great practical significance to protect the privacy and security of users and enterprises.

The program accepts specified structure which obeys protocol or some format and distinguishes valid from invalid input. An effective fuzzer generates semi-valid inputs that are "valid enough" in that they are not directly rejected by the parser of the program, but do create unexpected behaviours deeper in the program and are "invalid enough" to expose corner cases that have not been properly dealt with.

B. Fuzzers for IoT

Existing fuzzing tools can be divided into two types, one is the test carried out from the network protocol level in the presence of physical equipment, and the other is the test based on the simulation environment, and the test objects include application programs, operating systems and network protocols. interface. Table 1 lists several fuzzers that have been used for fuzzing IoT devices in recent years.

Generate-based fuzzing tools such as Sulley [6], Peach [7], and Boofuzz [8], generate test cases by modelling the input syntax format and behavioral state. This method works well for software programs that deal with highly structured input, but at present, the overhead of manually constructing grammar and state models is high. In addition, it is difficult to deal with unknown grammar conventions. Mutation-based fuzzing tools such as AFL [9], LibFuzzer [10], and HonggFuzz [11] generate new test cases by applying a set of mutation operations to pre-provided seed cases to modify them. The method is concise, easy to implement, can handle large-scale programs, and is widely used in practice.

For IoT firmware, RPFuzzer [12] is a black-box fuzzing tool that detects vulnerabilities in routing protocols in IoT devices by sending a large number of packets to IoT devices, monitoring device CPU usage, and checking system logs. IoTFuzzer [13] implements an app-based black-box fuzzing tool, which sends testing messages to the device through the App, observes the

Tab. 1. Fuzzers used for IoT devices

Fuzzer Name	Fuzz Layer	Hardware Support	Fuzzing Type
SRFuzzerr	Administration layer	SOHO router	Grey-box
Boofuzz	Internet interface	Bare-metal	Black-box
FIRM-AFL	Application	Emulation	Grey-box
IoTFuzzer	Internet interface	Bare-metal	Black-box
RPFuzzer	SNMP protocol	Router	Black-box

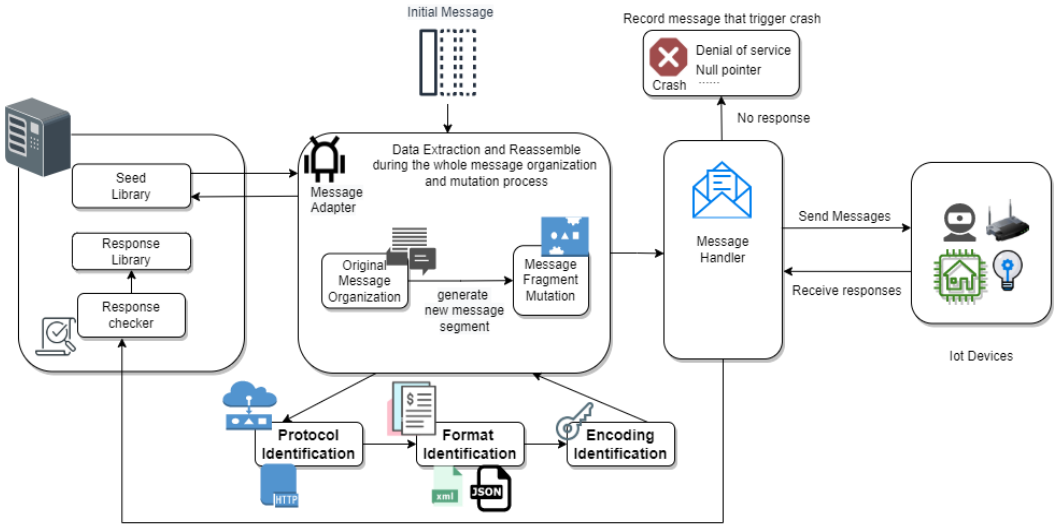

Fig. 1. Overview of FIoTFuzzer

system status based on liveness analysis and realizes the detection of memory-related errors in IoT devices.

III. DESIGN

In this section, we will introduce the detailed design of FIoTFuzzer. Figure 1 shows the overview of FIoTFuzzer.

A. Origin Message Acquisition

Effective original message seed can trigger the vulnerability of the deep-seated operation of the code. At the same time, IoT devices have restricted requirements for the format of input content. If it does not comply with relevant protocols (such as HTTP, UPnP) or does not have well-structured message seeds, it cannot be understood or executed by the firmware syntax parser. Therefore, FIoTFuzzer attempts to obtain the original message sequence as the initial message seed from the first-or third party API documents of the IoT device or use relevant applications to communicate with the target IoT device. For example, when the humidifier is turned on by the program, there is some confirmation of the device status. When similar information is transmitted in the LAN, we can capture it and store it in the seed library.

B. Message Adapter

At this stage, the message adapter will first identify the network protocol of the original message seed sequence. There are different processing methods for different protocols. After that, it will identify the format of the data segment in the message (such as JSON, XML, Key-value, etc.), and extract the content of the non-formatted part. The encoding recognizer will identify the encoding type. The role of the message adapter is not only data extraction, but also data reassemble (restore mutated data to its original format and encoding), which will appear in the entire process of fuzzing.

Protocol Identification. This part is focusing on the application layer protocol. For the application layer, there are standard network protocols that disclose detailed standards (such as HTTP, Zigbee, Z-Wave, etc.) and proprietary protocols without disclosing protocol standards (such as IGRP, EIGRP, etc. used by Cisco devices). After obtaining the original message, we use a third-party library to parse its format. When parsing succeeds, the message adapter will extract semantic features for the corresponding standard network protocol. For example, for the HTTP protocol, the message adapter can locate variable domain information such as terminal information (IP, port), user parameters, length domain, cookies, etc. Traditional fuzzing tools will waste a lot of time when sending random data that doesn't conform to network protocol specifications to IoT devices; when the protocol type cannot be identified, the message adapter will mark it as an unknown protocol, and then use the complete protocol data segment as the object of mutation.

Format Identification. Messages that don't give the expected format will be discarded immediately by the device, so the deep-level functional components can not be reached. To expand the code coverage, the message adapter identifies the format of the data carried during the communication process. Algorithm 1 shows the main thought of JSON format recognition. After the format recognition is completed, the non-

Algorithm 1: Scan JSON object

Input: *string*
Output: *jsonstruct*

```
for nextchar in string do
    if nextchar = ''' then
        parse the string
    if nextchar = '{' then
        parse the string object
    if nextchar = '[' then
        parse the array
    if nextchar = 'n' and next word is 'null' then
        return None
return jsonstruct
```

```
POST /cgi-bin HTTP/1.1
Host: 192.168.2.2
Content-Length: 69
Content-Type: application/json
Authorization: Basic
YWRtaW46YCB3Z2V0IGh0dHA6Ly8xOTIuMTY4LjIuMTo4MDAwL3Rvb2xzL21zZiAtTyAvbXN
mXG5jaG1vZCA3NzcgL21zZlxuL21zZmA=

{"profile": {"name": "Lee","age": 30}}
```

Fig. 2. A packet with base64 data encoded content

formatted part of the data will be extracted for the next encoding detection step.

Encoding Identification. We employ a heuristic encoding recognizer, which has multiple encodings built-in. Some common encodings such as base64 or URL encoding, appear in some request headers or bodies. Figure 2 shows a packet with base64 data encoded content. Random mutation of the encoded data will make the original content meaningless and may be discarded by the device in the initial syntax parser. The heuristic encoding recognizer will confirm the encoding for the value of each field separated by the format confirmation in the previous step. When the encoding format is recognized, it will decode it, and then hand it over to the message segment organization in the next stage for processing.

C. Original Message Organization

A scheme similar to Snipuzz is used here to determine the snippets in each message. FIoTFuzzer divides the non-formatted information into some initial fragments. The core idea of the division is to delete bytes one by one based on the initial detection message p_i to generate new probe messages, which are fragmented by classifying the responses r_i for each probe message p_i. To compare whether the two responses are of the same category, we use the following method: "Edit distance" [14] is an indicator used to measure the similarity of two sequences, from which the similarity score of the two sequences can be calculated.

$$s_{ij} = 1 - \frac{edit\_distance(r_i, r_j)}{\max\_len(r_i, r_j)} \qquad (1)$$

We first send the same message fragment twice in a second, calculate the self-similarity score of the message fragment, and use it to determine whether the two responses are of the same category. Specifically, for two different responses r_i and r_j, the degree of similarity s_{ij} between the two can be calculated. In Snipuzz, when $s_{ij} \geqslant s_{ii}$ or $s_{ij} \geqslant s_{jj}$, the two responses are considered to be in the same category. Different from this method, we consider that too strict self-similarity comparison will lead to using different parameters may cause differences in the response. The response similarity is bound to be smaller than its self-similarity, thus resulting in a large number of different message segment types. In extreme cases, there may be more types of responses than the number of responses, which increases the degree of message fragmentation. To solve this issue, we set up an undulator, whose function is to set a certain threshold when comparing s_{ij} with s_{ii}, s_{jj}, reduce the degree of message fragmentation, and enhance the generalization ability of the operation.

For example, in Table 2, the parameter of the packet is [652,"smooth",300], when '2' and 's' are deleted from the parameters in turn, the corresponding similarity scores are S_{12}=0.939, S_{13}=1.000, S_{12} will be lower than the original self-similarity score S_{11}= 1.000, but they are essentially the same type. The difference is that the transformed RGB color values are distinct. It is speculated that this part of the code has undergone the same code execution process inside the firmware. After being processed by the undulator, they are grouped into the same category.

To further refine the message fragments, FIoTFuzzer adopts the agglomerative hierarchical clusters method in Snipuzz to extract the message in the process of coacervating clusters ranked. One cluster remains after the process, two most similar at the present stage continuously merge, and finish.

The core idea of continuous condensing is to vectorize the existing coacervating clusters and extract the representative features in the message segment. Similar coacervating clusters are merged until only one cluster remains.

All the new generated message segments together with the initial message will be used in message fragment mutation in the next stage.

D. Message Fragment Mutation

Combining 5 of the 22 mutation strategies proposed by Offutt [15] can make the mutation test obtain the maximum branch coverage in a short period. Non-format part of the message segment will be mutated, and the mutation is based on the following strategies:

Flip Bit. This is a challenge for the parser in the firmware, FIoTFuzzer will flip all bytes in the message fragment in bit units.

Bytes Delete. FIoTFuzzer will randomly delete the target segment (which may be empty). When the firmware does not properly check the data field, this may cause a crash.

Special Character. Insert and replace format strings of common encodings (such as ASCII, Unicode, UTF 8, GBK, etc.), and truncate characters. When the syntax parser processes the message field content, there may be problems such as buffer overflow, and string truncation.

Set Dictionary. FIoTFuzzer has established a dictionary with interesting values, which can replace message segments with the content in the dictionary, such as replacing the number C with 0, 1, C-1, C+1. A good dictionary also plays an important role in the success rate of fuzzing.

Bytes copy. Repeating a lot of content may cause memory leaks in firmware, FIoTFuzzer will repeat a message fragment many times (like 1000 times).

Tab. 2. Different parameters cause different response content and similarity score

Message	Response	Similarity Score	Type
{"id":1,"method":"set_rgb","params":[652,"smooth",300]}	{"method":"props","params":{"hue":239,"rgb":652}}	S_{11}=1.000	1
{"id":1,"method":"set_rgb","params":[65,"smooth",300]}	{"method":"props","params":{"hue":240,"rgb":65}}	S_{12}=0.939	1
{"id":1,"method":"set_rgb","params":[652,"mooth",300]}	{"method":"props","params":{"hue":239,"rgb":652}}	S_{13}=1.000	1

E. Message Handler

The message handler is used to receive responses, send messages, detect crashes, and record them. There may be multiple ways of communication between the original test suite and IoT devices, such as TCP, UDP at the transport layer, and HTTP protocol at the application layer. Therefore, to make the function of sending and receiving information more widely available, we use sockets to communicate. When the specific signal is received or no response is received for multiple consecutive requests, a crash may have occurred, FIoTFuzzer will record the crash and the corresponding packet.

IV. IMPLEMENTATION AND EVALUATION G

A. Framework Implementation

FIoTFuzzer prototype is written in about 1,500 lines of python code. FIoTFuzzer supports fuzzing on bare-metal and simulated smart devices at the same time. In the initial message acquisition phase, we use Wireshark to get the communication between the test suite and the IoT device Communication data, these raw data will be used for subsequent fuzzing. The message adapter extracts and reassembles data during the whole message organization and mutation process. For example, after the non-formatted part of the message is divided into message segments and mutated, the message adapter restores the origin encoding format and then hands it over to the message handler for processing, which will send the message to the device. Send messages and receive responses, record the data messages that triggered the crash, and compare them with the corresponding seed messages to locate the mutated fields and further confirm the reason for vulnerabilities.

B. Experiment Setup

Environment setup. FIoTFuzzer runs on a Windows10 PC with Intel Core i5 - 8265U × 1.60 GHz CPU and 8 G RAM. For the fuzzing experiment environment, we configured the IoT devices under testing on a fully-controlled local Wi-Fi network set up by us, which avoids the interference of irrelevant traffic and unwanted package filtration of the firewall. After the device initialization, all the IoT devices are connected to the same wireless LAN. To verify FIoTFuzzer's performance in finding crashes, we used Boofuzz and Sulley as benchmarks. We built a data set containing 14 vulnerabilities in 12 devices. Device types include cameras, Wi-Fi routers, and smart bulbs. Table 3 lists details of some target devices.

Tab. 3. Evaluated Devices

Device ID	Vendors	Models	Device Types	Firmware Versions
1	Tenda	AC9	Wi-Fi router	V15.03.05.19（6318）
2	Tenda	AC15	Wi-Fi router	V15.03.05.19(6318)_CN
3	TP-link	TL-WR941	Wi-Fi router	V6_150312
4	TP-link	TP-Link WR841N	Wi-Fi router	(EU)_V14_180319
5	TP-link	TL-IPC42C-4	Wireless camera	1.0.6 Build 201126 Rel.62706n
6	D-Link	DIR-822	Wi-Fi router	FW v1.03
7	MikroTik	CHR	Wi-Fi router	6.40.5
8	Xiaomi	MJDP003	Smart bulb	1.4.4_0031
9	Mercury	MIPC372-4	Camera	1.0.1
10	Amcrest	IP2M841	Camera	V2.800.0000000.1R

Effectiveness of Vulnerability Detection. To compare the capability of vulnerability detection, we let FIoTFuzzer run for 24 hours. When a crash occurs, we automatically reset the device to resume testing. Finally, it found 9 vulnerabilities: 3 buffer overflows, and 6 DOS. These results show that FIoTFuzzer can automatically detect device vulnerabilities based on our message adapter mechanism. Table 4 lists the known vulnerabilities discovered by FIoTFuzzer.

Effectiveness of the Optimizations. To compare the efficacy of FIoTFuzzer, we reconfigured BooFuzz. When detecting buffer overflow vulnerabilities, Boofuzz can them detect independently. However, it is unable to cause crashes that are triggered by dependency messages. In addition, as shown in Table 5, regarding the degree of message fragmentation, one of the segmentation results for each device is selected. On average, FIoTFuzzer obtains 3, 7, and 11 response segments for one device, when setting the similarity threshold to 0.6, 0.3 and 0, respectively. This proves that setting the threshold can effectively control the degree of message fragmentation.

V. DISCUSSION AND FUTURE WORK

Although our framework can discover memory corruptions in IoT devices efficiently, there are still some challenges for future improvements. In this section, the limitations existing in the current design are discussed and how they could be handled in the future is taken into exploration.

Tab. 4. List of discovered known vulnerabilities

Exploit ID	Vulnerability Type	FIoTFuzzer	Boofuzz	Sulley
CVE-2020-8423	Buffer Overflow	150s	384s	286s
CVE-2019-6258	Buffer Overflow	145s	264s	N/A
CNVD-2021-30168	Buffer Overflow	63s	55s	N/A
CNVD-2021-35879	DOS	113s	N/A	N/A
CNVD-2021-81533	DOS	66s	N/A	N/A
CNVD-2021-35790	DOS	71s	136s	N/A
CNVD-2021-24948	DOS	134s	N/A	156s
CNVD-2022-16280	DOS	59s	N/A	74s
CNVD-2020-67555	DOS	34s	78s	52s

Tab. 5. Statistics of message fragmentation

Device ID	Message fragmentation		
	Threshold=0.6	Threshold=0.3	Threshold=0
1	3	5	8
2	1	7	15
3	5	8	9
4	3	4	4
5	6	8	12
6	2	6	13
7	2	7	19
8	1	1	1
9	3	5	8
10	4	9	12
11	6	11	20
12	4	8	13
Average	3	7	11
Total	40	79	134

Special Content. The capability of recognizing special fields needs to be improved. There are some specific fields in some packets. The IoT device will detect the values of special fields (such as MAC address, IP address and timestamp). When the format is incorrect, the mutation is invalid.

Data Encryption. FIoTFuzzer has limitations in the processing of encrypted data. Some devices encrypt the transmitted content, and the cloud server participates in the transmission process. The cloud server will act as an intermediary to transmit users' instructions to the IoT devices or feedback on the response of the IoT devices to users. In this case, FIoTFuzzer is difficult to analyze the encrypted fields.

Assign Reason. The cause of the vulnerability cannot be directly reported to the users, since the sign of vulnerability successfully triggered by FIoTFuzzer is that the router device sends a crash response. Though FIoTFuzzer can record the message of triggering the vulnerability, manual efforts are still needed in the determination of specific problem causes.

VI. CONCLUSION

In this paper, we have presented a black-box fuzzing framework FIoTFuzzer designed for improving the effectiveness of discovering vulnerabilities hiding in IoT devices. Different from other black-box fuzzing tests, FIoTFuzzer uses a message adapter to extract and reassemble data to overcome highly-structured formats requirement by IoT devices. In addition, FIoTFuzzer improves the method in Snipuzz that uses the response messages returned by the device to establish a feedback mechanism for guiding the fuzzing mutation process. The setting of a threshold can further reduce the fragmentation of message segments and improve mutation efficiency. By conducting experiments in the data set we prepared, FIoTFuzzer found more vulnerabilities than Boofuzz and Sulley, which shows the effectiveness of FIoTFuzzer.

ACKNOWLEDGMENT

This work was supported by the National Key Research and Development Program of China(No.2021ZD0111405).

REFERENCES

[1] A. Lukehart, A. A. Lukehart, and Author: "2022 cyber attack statistics, data, and Trends," Parachute, 25-Mar-2022. [Online]. Available: https://parachute.cloud/2022-cyber-attack-statistics-data-and-trends.

[2] "The Connected Enterprise: IOT Security Report 2021," Palo Alto Networks. [Online]. Available: https://www.paloaltonetworks.com/resources/research/connected-enterprise-iot-security-report-2021.

[3] D. McMillen, co-authored by Wei Gao, Dave McMillen Senior Threat Researcher, D. McMillen, S. T. Researcher, and Dave brings over 25 years of network security knowledge to IBM. "A new botnet attack just mozied into town," Security Intelligence, 17-Sep-2020. [Online]. Available: https://securityintelligence.com/posts/botnet-attack-mozi-mozied-into-town/.

[4] "2020 IoT Endpoint Security White Paper.," 2020 IoT Terminal Security White Paper-Fujian Chaoruichuangyuan Information Technology Co., Ltd. [Online]. Available: http://www.superrisc.com/xingyexinwen/33.html.

[5] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of IOT firmware via message snippet inference," Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021.

[6] OpenRCE, "OpenRCE/sulley: A pure-python fully automated and unattended fuzzing framework.," GitHub. [Online]. Available: https://github.com/OpenRCE/sulley.

[7] "Integrating security into your DevOps lifecycle," Integrating security into your DevOps Lifecycle. [Online]. Available: https://www.peach.tech/.

[8] Jtpereyda, "JTPEREYDA/Boofuzz: A fork and successor of the Sulley Fuzzing Framework," GitHub. [Online]. Available: https://github.com/jtpereyda/boofuzz.

[9] Google, "Google/AFL: American fuzzy lop - a security-oriented fuzzer," GitHub. [Online]. Available: https://github.com/google/AFL.

[10] "LibFuzzer – a library for coverage-guided Fuzz Testing.," libFuzzer – a library for coverage-guided fuzz testing. - LLVM 15.0.0git documentation. [Online]. Available: https://www.llvm.org/docs/LibFuzzer.html.

[11] Google, "Google/honggfuzz: Security oriented software fuzzer. supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based)," GitHub. [Online]. Available: https://github.com/google/honggfuzz.

[12] "RPFuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing," KSII Transactions on Internet and Information Systems, vol. 7, no. 8, pp. 1989–2009, 2013.

[13] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. F. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering memory corruptions in IOT through APP-based fuzzing," Proceedings 2018 Network and Distributed System Security Symposium, 2018.

[14] "Edit distance," Wikipedia, 06-Jan-2022. [Online]. Available: https://en.wikipedia.org/wiki/Edit_distance.

[15] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," ACM Transactions on Software Engineering and Methodology, vol. 5, no. 2, pp. 99–118, 1996.

[16] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for IOT devices," 2021 IEEE Symposium on Security and Privacy (SP), 2021.

[17] 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.

[18] Y. Cheng, W. Fan, W. Huang, G. Yu, Y. Han, H. Dong, and W. Liu, "PDFuzzerGen: Policy-driven black-box fuzzer generation for smart devices," Security and Communication Networks, vol. 2022, pp. 1–14, 2022