applied sciences

MDPI

Article

SIoTFuzzer: Fuzzing Web Interface in IoT Firmware via Stateful Message Generation

Hangwei Zhang, Kai Lu, Xu Zhou *, Qidi Yin, Pengfei Wang and Tai Yue

College of Computer, National University of Defense Technology, Changsha 410073, China; zhanghangwei@nudt.edu.cn (H.Z.); kailu@nudt.edu.cn (K.L.); yinqidi@nudt.edu.cn (Q.Y.); pfwang@nudt.edu.cn (P.W.); yuetai17@nudt.edu.cn (T.Y.)

* Correspondence: zhouxu@nudt.edu.cn

Abstract: Cyber attacks against the web management interface of Internet of Things (IoT) devices often have serious consequences. Current research uses fuzzing technologies to test the web interfaces of IoT devices. These IoT fuzzers generate messages (a test case sent from the client to the server to test its functionality) without considering their dependency, which is unlikely to bypass the early check of the server. These invalid test cases significantly reduce the efficiency of fuzzing. To overcome this problem, we propose a stateful message generation (SMG) mechanism for IoT web fuzzing. SMG addresses two problems in IoT fuzzing. First, we retrieve the message dependency by using web front-end analysis and status analysis. These dependent messages, which can easily bypass the server check, are used as a valid seed. Second, we adopt a multi-message seed format to preserve the dependency of the messages when mutating the seed to get a valid test case, so that the test case can bypass the state check of the server to make a valid test. Message dependency preservation is implemented by our proposed parameter mutation and structural mutation methods. We implement SMG in our IoT fuzzer, SIoTFuzzer, which applies IoT firmware on the latest Linux-based simulation tool, FirmAE. We test nine IoT devices including a router and an IP camera and adopt a vulnerability detection mechanism. Our evaluation results show that (1) SIoTFuzzer is capable of finding real-world vulnerabilities in IoT devices; (2) our SMG is effective as it enables Boofuzz (a popular protocol fuzzer) to find command injection and cross-site scripting (XSS) vulnerabilities; and (3) compared to FirmFuzz, SIoTFuzzer found all the vulnerabilities in our benchmarks, while FirmFuzz found only four—the efficiency of our tool increased by 20.57% on average.

Keywords: IoT device; web management interface; stateful message generation (SMG); message dependency; front-end analysis; multi-message seed format

check for updates

Citation: 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. https://doi.org/10.3390/app11073120

Academic Editor: Subhas Mukhopadhyay

Received: 6 February 2021
Accepted: 24 March 2021
Published: 1 April 2021

1. Introduction

With the rapid development of the Internet of Things (IoT), the number of global IoT connections continues to grow exponentially and will reach 27 billion by 2025 [1]. Since the rise of the smart home, more and more smart devices are widely used, such as routers, and IP cameras. A large number of vulnerabilities in IoT devices have been disclosed in recent years. For example, at the 2013 Black Hat Conference, Heffner [2] demonstrated the overflow, hard-coded password, and command injection vulnerabilities of a variety of web cameras, involving D-Link, TP-Link, Linksys, and Trendnet equipment vendors. Attackers can use these vulnerabilities to log in without authorization and hijack the real-time video of the camera. In addition, real security incidents caused by security vulnerabilities are frequently emerging. In 2019, most areas in Venezuela, including the capital Caracas, experienced a continuous power outage over more than 24 h [3]. The power outage made the Caracas subway inoperable and caused large-scale traffic congestion, and the Internet could not be used normally. Due to the long service-life of IoT devices, there are a large number of devices in the network that have not been maintained by vendors. In the same year, a D-link product found an unauthenticated remote code execution vulnerability [4], which affected more than 10 products of related models. Although the product has been discontinued, D-link vendors did not release related patches, and the vulnerability has not been fixed. In cyber attacks, the user interface of devices is one of the main attack surfaces. This means that once user interfaces are exposed, these routers and IP cameras are likely to become a zombie host and be used in attacks such as DDoS. As a result, IoT security is increasingly becoming a topic of concern to researchers. It is an important research field to detect the vulnerabilities of IoT devices quickly.

Due to huge differences in the hardware and software of IoT devices from different vendors, it is difficult to build IoT vulnerability analysis models and establish a unified dynamic simulation environment. However, centralized testing for certain devices is inefficient. Testers prefer to use automated approaches to test all kinds of devices. The automated approaches to detect IoT vulnerability are generally divided into static methods [5,6] and dynamic methods [7–9]. Firmware images are usually published in the vendor's online service and testers can obtain images easily. For static methods, testers unpack images by Binwalk [10]. The unpacked files can be used for analysis with a reverse tool, such as IDA. Dynamic methods mainly stand by firmware simulation. This method requires that images contain an operating system, including a general operating system (e.g., Linux) or a specific embedded operating system (e.g., VxWorks). Qemu [11], a processor emulator based on Linux, is able to provide fast system-mode emulation. Therefore, IoT devices that are Linux-based could be emulated by Qemu and tested more easily.

Vendors usually use web and application to provide users with operating interfaces. These interfaces can directly operate a device, and their design standards evolve according to the actual operation of the device. When the web and application obtain a user's input, they will send operation messages to the device. After receiving these requests, the device performs further procedures according to the request parameter and executes the targeted program. The status of the device will change with this process. If an implementation flaw exists in the parameter parsing and processing program, a vulnerability will be exploited. Therefore, an IoT device with user interfaces can be treated as a blackbox. Through feeding these interfaces with mutated messages, a fuzzer may trigger potential vulnerabilities. The testers do not need knowledge of the underlying architecture. The fuzzing could send a large number of messages per unit of time. However, the blackbox fuzzing receives a limit on the throughput of the device processor and will generate many more invalid test cases without feedback. These invalid test cases are not helpful for fuzzing and they take more time. At the same time, if the device does not receive the stateful message and is not in a state to accept messages, the device will refuse service or interrupt the connection. As a result, it is ineffective to continue sending mutated messages. Furthermore, some internal parameters depend on the previous message. When these parameters are mutated, these messages will also be rejected. According to these issues, detecting vulnerabilities through blackbox fuzzing is low in efficiency and effectiveness.

Motivated by the above description, this paper leverages generation-based fuzzing technology to perform blackbox testing automatically. In our work, we propose a stateful message generation (SMG) mechanism to generate more valid test cases, which can improve the efficiency and effectiveness of fuzzing. SMG addresses two challenges, including the maintenance status of a device and the mutation of parameter dependency messages. Since the devices have many I/O operations, the status of a device consists of a physical state and a protocol state. When the physical state is abnormal, the device needs to be maintained manually. In a message sequence, if a message has an inappropriate protocol state, the device will not be able to accept the subsequent messages in this sequence. Our work mainly solves the state problem of the protocol. We analyze the front-end of an IoT device's web interface to build initial seeds and generate test cases. Due to the difficulty of firmware operation monitoring, we can analyze operating interfaces to obtain prior knowledge. This knowledge will help us test devices more comprehensively. We adopt a multi-message seed format, and every seed contains a complete sequence of operations. Based on Boofuzz [12] (a popular protocol fuzzer), we design a blackbox fuzzing tool called SIoTFuzzer which can detect IoT device vulnerability. By building a simulation environment, it is more suitable for analyzing the web management interface and constructing the input of an IoT device. Finally, vulnerabilities can be discovered through device monitoring deployed in the system or built in the simulator.

In order to validate and evaluate this blackbox fuzzing, SIoTFuzzer was designed and implemented for discovering vulnerabilities in IoT devices automatically. The test objectives are mainly routers and IP cameras. These devices are based on MIPS or ARM architecture, and they also have a web interface. To verify the improvement of our seed generation and mutation strategy, we set up a control group to prove that our optimization is effective. Compared with FirmFuzz [13], the latest device blackbox fuzzing test tool, SIoTFuzzer has a greater vulnerability discovery capability.

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

1. For addressing two difficulties in detecting vulnerabilities in IoT devices, including the maintenance status of devices and keeping parameter dependency between messages, we adopt stateful message generation (SMG). In addition, we adopt a multi-message seed format and deploy a corresponding mutation strategy to guide fuzzing;
2. We design and implement a blackbox fuzzer called SIoTFuzzer (with open-source code available at https://github.com/yinfeidi/SIoTFuzzer (accessed on 27 March 2021)) for fuzzing IoT devices. Through analysis of the device web interface, we can obtain prior knowledge of web elements. SIoTFuzzer traverses the device web pages and obtains normal communication messages. These messages will be used for fuzzing;
3. We evaluated SIoTFuzzer on 9 IoT devices and 12 known vulnerabilities were found. At the same time, we deployed our two optimizations on Boofuzz to conduct a controlled experiment, and results show they improve the detection speed by almost 61.99%. Compared with FirmFuzz, SIoTFuzzer could indeed detect known vulnerabilities much faster than FirmFuzz, and the vulnerability detection time is reduced by about 20.57% on average.

2. Related Work

With an increasing number of IoT security issues, fuzzing techniques are proposed to find IoT device vulnerabilities in an automatic manner. There are two methods in device vulnerability analysis: static analysis and dynamic analysis. At the early stages, part of the firmware source code is open and the file system is not encrypted. The tester can audit the source code and extract the target program for analysis quickly. Costin et al. [5] adopted static analysis tools to unpack the file system of firmware and performed large-scale firmware analysis. Furthermore, based on symbolic execution engines and program slicing, Firmalice [6] proposed a binary analysis scheme to detect vulnerabilities in embedded device firmware. These studies are instructive, but they did not find various vulnerabilities. Due to a lack of implementation, they cannot judge whether a device still has related vulnerabilities. Therefore, dynamic analysis and dynamic execution clearly have more advantages than static analysis. Device input is usually a major source of vulnerabilities. Fuzzing can generate abnormal data through mutation, which is suitable for vulnerability mining of various objects. Since different communication processes will generate diverse protocol messages, the greatest difficulty in testing is usually the processing of protocol templates. The testers can also choose to avoid this process and inject the mutated data into the user interface before messages are generated. IoT fuzzing mainly includes mutation-based fuzzing and generation-based fuzzing.

2.1. Mutation-Based Fuzzing

For mutation-based fuzzing in IoT devices, the fuzzer needs a more effective mutation strategy to lead vulnerability detection. Wang et al. [14] presented WMIFuzzer, a mutation-based blackbox fuzzer targeting the web management interface in commercial off-the-shelf

(COTS) IoT devices; WMIFuzzer parses HTTP messages into a weighted message parse tree. This method can generate structurally valid messages. However, when fuzzing for a test real device, WMIFuzzer does not have a high throughput because the connection of the test is often interrupted. In addition, WMIFuzzer does not have the support for local monitoring. Therefore, WMIFuzzer imposes overhead on the device's startup and reboots for each fuzzing session. In order to improve this problem, through firmware simulation, rebooting the device from a snapshot could reduce the overhead. Zheng et al. [15] presented *Firm-AFL*, a mutation-based greybox fuzzing platform for IoT firmware. *Firm-AFL* adopts augmented process emulation to minimize each fuzzing iteration overhead. It achieves high throughput fuzzing by running the target program in a user-mode emulator and switching to a full-system emulator when the target program invokes a system call that has specific hardware dependencies. This work resolved the performance bottlenecks. However, *Firm-AFL* focuses on the coverage of a single program and does not consider the communication process. The increase in the coverage of a single program makes it difficult to trigger the inter-program vulnerability.

2.2. Generation-Based Fuzzing

In other research, Chen et al. [9] presented *IoTFuzzer*, which analyzes Android apps to detect memory-related vulnerabilities in IoT devices. *IoTFuzzer* adoptes a taint-based approach and mutates the data flow that is used to generate the protocol message. Therefore, *IoTFuzzer* does not need a protocol template. Through mutating the data flow in application, *IoTFuzzer* skips the protocol analysis. In addition, the mutation strategy can not only trigger memory corruption but also logic corruption. Due to the difficulty in device monitoring, *IoTFuzzer* is used to find obvious firmware crashes. More recently, the research work into IoT devices has mainly concentrated on the web interface. Yu et al. [16] implemented IoTHunter that completes the fuzz testing of stateful network protocols. IoTHunter uses multi-level message generation mechanism to test the some standard protocols in IoT firmware, such as SNMP, FTP, BGP. Besides, Wang et al. [17] presented *EWVHunter*, a grey-box fuzzer for embedded IoT devices. *EWVHunter* constructed a prior knowledge base about the web front-end and filled data at the input source. In device monitoring, *EWVHunter* uses firmware information extraction to obtain execution information that can guide further fuzzing. *EWVHunter* is, like *IoTFuzzer*, tested on real devices. As described in Section 2.1, testing on real devices will impose overhead.

Based on firmware simulation, Prashast et al. [13] presented *FirmFuzz*, a fuzz testing of embedded firmware images. *FirmFuzz* detects IoT device vulnerabilities via the web interface. It is a generational fuzzer for syntactically legal input generation that leverages static analysis to aid fuzzing of the emulated firmware images while monitoring the firmware runtime. *FirmFuzz* mutates communication messages by collecting payloads that can trigger vulnerabilities. However, it does not consider mutation strategies, and hence the chance of detecting a vulnerability is relatively low.

3. Background and Motivation

In this section, we introduce the background knowledge and motivation about discovering vulnerabilities via a fuzzing web management interface. For fuzzing IoT devices, we need to pay attention to the following issues. 1. In test preparation, how we can obtain more prior knowledge from the web page and whether the method is applicable to devices of different design specifications. 2. Based on issue 1, we need to maintain the connection between the fuzzer and the device, and ensure that mutated messages are received by the device. These two issues will be explained in Section 3.4 below.

3.1. Web Interface in IoT Devices

Vendors usually provide users with a network interface for self-management. Although there is no standard on how to implement this interface, many vendors prefer to use web technology because of its flexibility and simplicity [18]. A web server is mainly used for message transmission between the front-end and the device program processor, called a pagehandler. The main workflow is shown in Figure 1. First, the front-end obtains the user inputs. Then, the front-end packages these user inputs into messages. Next, after decoding the messages, the web server passes the parameters to the pagehandler. After this, the pagehandler returns the processing results, which obtain the HTTP status code. Finally, the front-end receives the results and displays them on the page.

Since the front-end is directly accessible, it is easier to analyze the front-end than the web server or pagehandler. The front-end is composed of HTML code, JavaScript code, CSS code, and other static resources. All we need to analyze are the HTML and JavaScript codes, then we can obtain page elements and function parameters. CSS codes and other static resources mainly affect page layout and appearance. These codes are useless for message generation. For IoT devices, the front-end generates the message sequence and transmits commands to the server. By using a variety of inputs, it may cause vulnerabilities in the device.

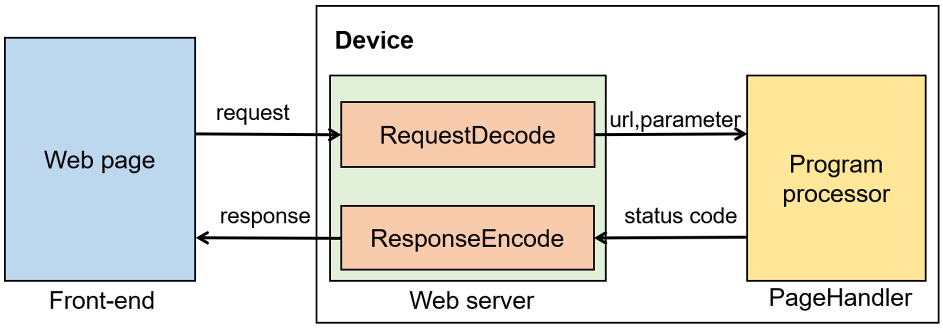


Figure 1. Workflow of web interface.

3.2. Firmware Simulation

The previous research mainly adopted three methods for the operation of IoT devices: 1. physical objects; 2. semi-simulation (e.g., *AVATAR* [19]); 3. full system simulation (e.g., *Firmadyne* [7]). In the testing of real devices, *IoTfuzzer* [9] detects whether the device is online by sending a heartbeat message. And after every ten messages, server will send a heartbeat message. This method is suitable for detecting obvious crashes, and the next time for testing after a crash requires the device to restart. This time cost for restarting is unacceptable. As a study by Muench et al. [20] pointed out, because IoT devices are slower than desktop workstations or servers, a complete system simulation can produce the highest throughput. For fuzzing, higher throughput means greater efficiency. At the same time, it is convenient to monitor the simulation process. And when the device crashes, it can be quickly restored by the snapshot.

Firmadyne is an automated and scalable system for performing emulation and dynamic analysis of Linux-based embedded firmware. It uses a modified kernel to support MIPS and ARM architecture firmware for simulation. *Firmadyne* also has an extractor to extract a filesystem and kernel from downloaded firmware and can perform a basic automated analysis to detect vulnerabilities. This script tests for the presence of 60 known vulnerabilities using exploits from *Metasploit*. But in nearly 2000 firmwares tested, only 16.28% can be correctly simulated. Since our fuzzing test requires the network service of the device, a low simulation success rate cannot result in better runtime environment support. The subsequent improvement work, *FirmAE* [21] proposes arbitrated emulation to apply failure handling heuristics to the emulation environment. *FirmAE* significantly increases the emulation success rate (from *Firmadyne*'s 16.28% to 79.36%). Through *FirmAE*, we can simulate most of the collected firmware.

3.3. Fuzzing Technology

Fuzzing is a software testing technique that can provide a random input to programs and has been proven to be effective in finding vulnerabilities in real programs. As fuzzing is gradually used more in other fields, people hope to use this method to test more complex objects, such as embedded devices, library functions, and file systems. For these targets, the first focus is obtaining a stable operating environment, and the second is establishing appropriate inputs for the target. In Table 1, we make a comparison with five IoT firmware testing tools.

Table 1. Comparison of Internet of Things (IoT) firmware testing tools.

Fuzzer	**Boofuzz** [12]	**IoTFuzzer** [9]	**WMIFuzzer** [14]	**FirmFuzz** [13]	**Firm-AFL** [15]
Fuzzing Technique	Blackbox	Blackbox	Blackbox	Blackbox	Greybox
Hardware Support	All	Real	Real	Emulation	Emulation
Protocol Support	Needs template	None	HTTP	HTTP	HTTP
Message Dependency	None	None	None	None	None

As described in Sections 3.1 and 3.2, because of the difficulty of firmware analysis and the accessibility of the front-end, most IoT tests adopt blackbox fuzzing. *Boofuzz* is a protocol fuzzing tool based on the Python language, and it requires protocol templates. Writing protocol templates could create a large workload, but *Boofuzz* is strongly extensible for many kinds of scenarios.

3.4. Motivation

In Section 3.1, a web interface is used to accept the user inputs and translate inputs into communication messages. These messages result in the change of device state. When the pagehandler accepts the error messages, it may cause the device to crash. Generating mutated device messages is a major concern for a tester. Due to the different standards established by vendors in the protocol communication process between the front-end and web server, the method of injecting mutated data into a web page is often used. However, with the application scenarios of IoT devices shifting from LAN to WAN, vendors are improving the security of their web interfaces, such as adding some kind of security validation to the input field. From the code in Figure 2, lines 1–6 show the input validation for the web page, including XSS, special character, and invalid address checks. Inputs that cannot pass validation will not be received by device. As a result, the method of direct injection does not apply. Therefore, we can only use a proxy server to grab the normal messages. We need web crawlers to visit all pages of the device. Through front-end analysis, input simulation, and click-on page elements, we obtain the normal device messages.

```
 1 if(!isSafeforXSS(str)){...}
 2 //XSS check
 3 if (isValidCfgStr('', str, 256) == false || (str == '')) {...}
 4 //special character check
 5 if (!is_valid_ip(str,0)) {...}
 6 //invalid address check
 7 $.get("/get_sessionKey.asp", function(sessionKey){
 8   page_val.sessionKey = sessionKey;
 9   page_val.Addr = str;
10   setTimeout('$.post("/page", page_val, function(){getTestInfo();});',
    300);});
11 //sesssionKey check
```

Figure 2. Security validation of web page.

In previous work, *FirmFuzz* [13] grabs the first message after a click operation and mutates all of the message's data fields. *FirmFuzz* will generate hundreds of test cases and send these requests to the server one at a time. As an operation always contains a message sequence, a single message is just a part of the operation. Some messages are used for device state transition. As shown in Figure 2 lines 7–10, this example shows that the front-end needs to ask for a *sessionKey* of the current session to perform parameters. The *sessionKey* is unique in every connection, and a single message without the key to the parameter transmission will be rejected by the web server. When generating a test case, the fuzzer needs to request the server for a unique key first and then add it into the message. In addition, this session has a timeout so that we need to request the server in every test case. If we ignore device status and stateful messages, this will lead to two issues:

1. During the generation phase, if a test case lacks stateful messages, it will not bypass the early check of the server;
2. During the mutation phase, only mutating all data fields of the message could break parameter dependency between messages.

The forced mutation strategy will lead to too many invalid messages being generated, and most of these mutated messages will be rejected by server. In general, for improving the efficiency of fuzzing, we prefer to send more test cases in a period of time. However, when most test cases are invalid, testing cannot trigger vulnerabilities.

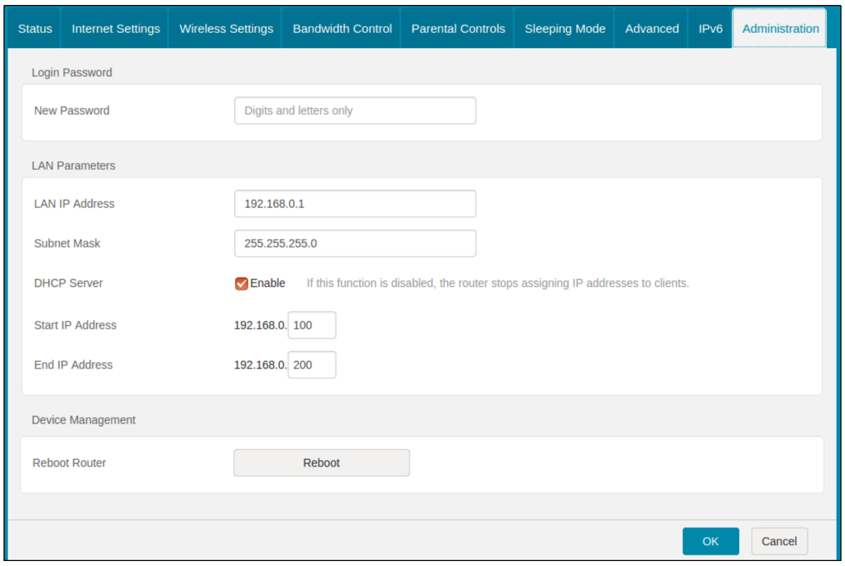
Through the above problems, the fuzzer needs to keep the connection status and mutated data field that have no dependency to make sure that web server receives all mutated message. In Table 1, we list five fuzzing tools that can test IoT devices; none of them can deal with these two problems. In this work, we propose a stateful message generation (SMG) mechanism to maintain the device status and connection, which is described in Section 4.

4. Stateful Message Generation (SMG)

In Section 4, we will introduce our pre-analysis process for the IoT device web management interface, which is used to generate stateful messages to solve the device status problem in Section 3. This process is divided into three parts: front-end analysis, state analysis, and seed generation.

4.1. Front-End Analysis

The front-end of the IoT device usually adopts the single-page mode. Each sub-page of the page contains device information and corresponding settings, which are filled in and submitted by users of IoT devices. As shown in Figure 3, this is an administration sub-page in the router management interface. The input elements on this page include the device's new password, IP address, subnet mask, and address fields. The click elements include three buttons.



Login Password

New Password	Digits and letters only

LAN Parameters

LAN IP Address	192.168.0.1
Subnet Mask	255.255.255.0
DHCP Server	✓ Enable If this function is disabled, the router stops assigning IP addresses to clients.
Start IP Address	192.168.0. 100
End IP Address	192.168.0. 200

Device Management

Reboot Router	Reboot

Figure 3. The administration settings of a router.

The elements that affect page changes mainly include link and button elements. The link element only needs to be clicked to trigger the server response. The button element may need the corresponding form content to trigger. The current analysis tools for device webpages mainly crawl the links on the page and then enter the page under the link for further operation. However, the web page still has many pop-up windows or implicit links that need to be triggered through clicking, which cannot be obtained by simple page analysis. At this point, our work improved on the page crawler. The link elements are classified as click elements. By identifying all click elements, all page jump actions are triggered by clicking instead of jumping through links. Before the page jump occurs, it is necessary to identify all input and click elements on the page and fill the input elements. For every page, we maintain a clicked queue to make sure all operations are triggered.

The front-end analysis is divided into three steps:

1. Determine whether to enter a new page that has never been visited; we need to identify the current page elements and create lists of input and click elements. These element lists will not be released until the end of the analysis.
2. Fill in the input elements and create a dictionary library to match the element names with certain rules. The code in Figure 4 corresponds to the device page in Figure 3, where lines 1–20 are the input elements on the page. Our page element filling uses certain rules to match the type of data, including address, character, and number, and select data from the dictionary to fill it. The element *oldPwd* in lines 3–6 is not a form element, so it will not appear in the generated message parameters; if the server lacks verification of such parameters when they are added to the message, the server may crash. We call this type of input element a non-form input, and we need to record the ID and type information to add these parameters to the mutation.
3. Click on the link or button while recording the page status. Each click may cause a change to the page. At the same time, we need to use an agent to record the data sequence corresponding to this change.

```
 1 <form class="form-horizontal" id="loginPwd">
 2    <h2 class="legend">Login Password</h2><fieldset>
 3    <div class="form-group none" id="oldPwdWrap" style="display: none;">
 4        <label for="oldPwd">Old Password</label>
 5        <input type="password" id="oldPwd" name="oldPwd"></div></div>
 6    <div class="form-group">
 7        <input type="password" id="newPwd" name="newPwd"></div></fieldset>
 8 </form>
 9 <form class="form-horizontal" id="lanParame">
10 <div class="form-group">
11    <input type="text" id="lanIP" name="lanIP" >
12    <input type="text" id="lanMask" name="lanMask">
13    <span class="ipNet">192.168.0.</span>
14    <input type="text" id="lanDhcpStartIP" name="lanDhcpStartIP">
15    <span class="ipNet">192.168.0.</span>
16    <input type="text" id="lanDhcpEndIP" name="lanDhcpEndIP"></div>
17 </form>
18 <div class="form-horizontal" id="deviceManage">
19    <form name="rebootfrm" method="post" action="http://192.168.0.1/goform/
  sysReboot">
20        <div class="form-group">
21           <button type="button" name="reboot" id="reboot">Reboot</button></
  div></form>
22 </div>
23 <button id="submit">OK</button><button id="cancel">Cancel</button>
```

Figure 4. The code of the administration settings web page.

4.2. State Analysis

In order to keep the connection between the server and the fuzzing process, it is necessary to maintain the state of the device to receive the mutated message. As shown in Figure 5, the states mainly include authorize, wait, and action. When the web server receives parameters, the device needs to be authorized, and then the front-end can send messages until timeout.

In state analysis, we should first make the device status change from waiting to authorization. We need to capture the authorization messages and replay these messages to the device. Secondly, the web server sends the operation messages. A page operation may include the interaction of multiple messages. The traditional fuzzing tool uses a single message to construct a test case. This method cannot handle the vulnerabilities that may be caused by the complex message process. The fuzzer will generate a large number of invalid test cases that are rejected by the server. To solve this problem, each time we analyze the device state, the operation sends a message sequence that corresponds to a page operation. The message sequence from the waiting to the end of the operation is what we need to obtain. After the input elements in the subpage are filled, when each clicked element is clicked, the message sequence starts to be obtained until the operation finishes.

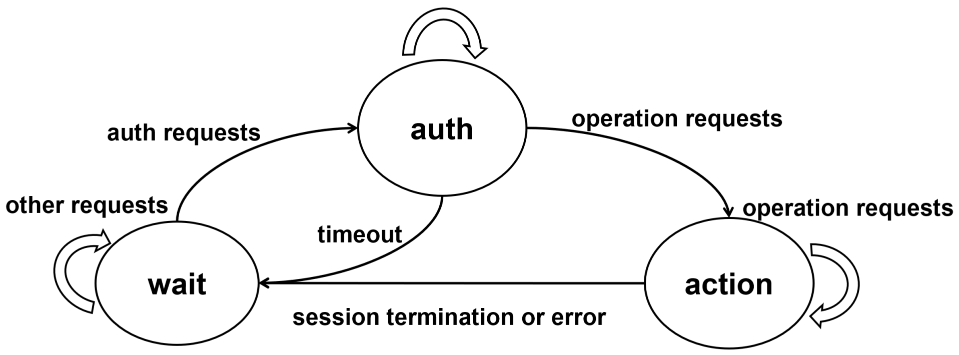


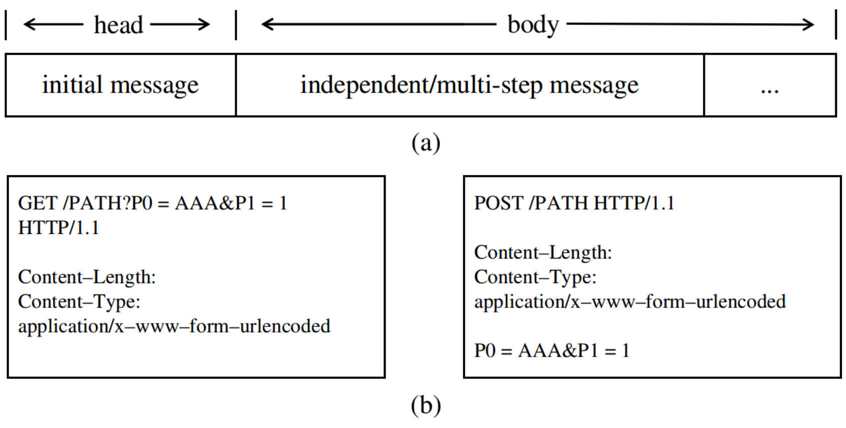
Figure 5. State transition of the device.

4.3. Seed Generation

Through state analysis, we obtain the sequence of messages corresponding to an operation, filter the useful messages, and reconstruct the seeds. First, we need to filter the messages. We only keep the GET and POST requests in the HTTP request, and remove the GET request for web resources. Figure 6b shows the specific format of the seed message. Second, we need to combine the filtered messages to form a seed. We divide the messages that make up the seed into four categories:

1. **Authorization message**: this is used to authorize the device so that subsequent messages can be accepted by the web server;
2. **Independent reference message**: this is a single message used to transmit parameters to the server;
3. **Multi-step reference message**: according to the device rules, the client may need to initiate a verification request before transmitting parameters to the server, so a multi-step reference message consists of multiple messages containing verification information;
4. **Payload message**: in our research, the trigger link of some vulnerabilities is inaccessible, so we collected some payload messages about vulnerabilities in IoT devices to trigger certain vulnerabilities that cannot be accessed from the page. Note that the payload message is mainly used for mutation and does not constitute the initial seed.

As shown in Figure 6a, each initial seed can be divided into two parts: head and body. The head must be the initial message, and the body can contain several independent messages and multi-step messages. We then put the generated seed into the seed pool and wait for seeds to be selected and mutated.



```
GET /PATH?P0 = AAA&P1 = 1
HTTP/1.1

Content–Length:
Content–Type:
application/x–www–form–urlencoded
```

```
POST /PATH HTTP/1.1

Content–Length:
Content–Type:
application/x–www–form–urlencoded

P0 = AAA&P1 = 1
```

Figure 6. (**a**) The structure of multi-message seed. (**b**) The format of messages consisting of seeds.

5. Framework of SIoTFuzzer

The Framework of SIoTFuzzer is described from two main aspects in Section 5. As shown in Figure 7, after simulating the IoT device, SIoTFuzzer selects seeds in order from the seed queue that was generated in Section 4. Next, SIoTFuzzer mutates seed with parameter mutation and structural mutation. Then we generate test cases and perform vulnerability testing on the server. At the same time, SIoTFuzzer monitors the status of the device. We will describe the process in detail in Sections 5.1 and 5.2.

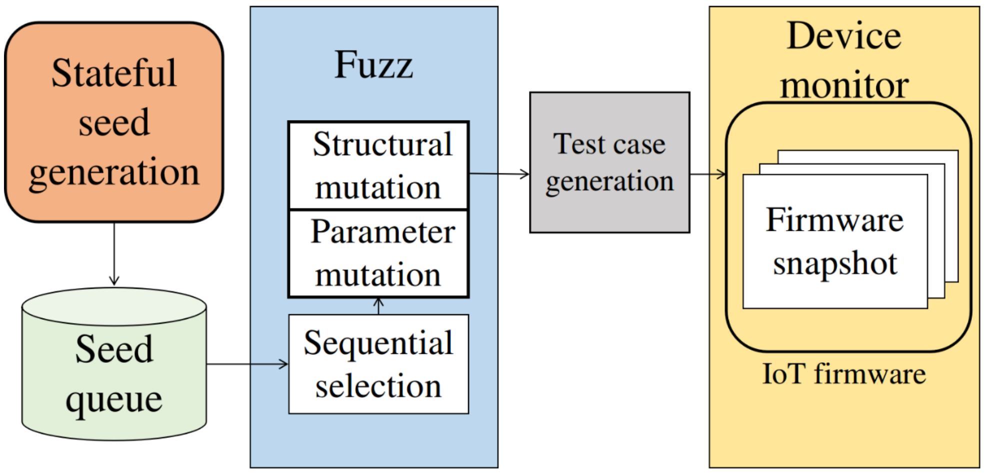


Figure 7. Framework of SIoTFuzzer.

5.1. Mutation Strategy

According to the seed format used in Section 4, there are multiple messages in a seed, and it is unknown which message with mutated content can trigger a vulnerability. As shown in Algorithm 1, the mutation strategy is proposed to perform the fuzzing. There are two phases, including a determined phase and a random phase. The determined phase is divided into two stages: parameter mutation and structural mutation.

5.1.1. Parameter Mutation

Parameter mutation is used to trigger memory-related vulnerabilities and command injection vulnerabilities. To ensure that the message sequence is completely accepted and data are transmitted to the device server, we mainly mutate the message parameter. For protocol messages, a parameter usually contains nodes and values. Therefore, parameter mutation includes parameter node mutations and value mutations. Before the mutation proceeds, the parameters in the message need to be parsed with a parameter dictionary. In particular, when multi-step messages are mutated, we mark the verification message and verification field and do not mutate this part. We adopt node mutation first and then value mutation.

For node mutation, we randomly select a parameter position, and perform the following operations on this parameter node:

- **N1**: delete this node;
- **N2**: repeat this node. The purpose of this step is to test whether the server will generate an error if a parameter is assigned multiple times in a statement;
- **N3**: select one parameter from the non-form parameter library and insert it at this position. If the server lacks verification of the non-form parameter and an illegal value is passed in, the device will crash.

Algorithm 1 SeedMutation($Seed, N\text{-}Mutation, V\text{-}Mutation, S\text{-}Mutation$)

```
Input: the set of seed messages, $Seed$;
    the set of node mutation method, $N\text{-}Mutation\{N1, N2, N3\}$;
    the set of value mutation method, $V\text{-}Mutation\{V1, V2, V3, V4\}$;
    the set of structural mutation method, $S\text{-}Mutation\{S1, S2, S3, S4\}$;
    //determined phase
    $seed_i$ = random($Seed$) // randomly select a seed
    split $Seed_i$ to messages set $\{M_1, M_2, ..., M_n\}$
    for each $M_i$ ! = $M_1$ and $M_i \in M$ do
        $P$ = message-parameter($M_i$) // get the set of parameters from message
        $P_i$ = random($P$) // randomly select a parameter
        for each $Mutation \in \{N\text{-}Mutation, V\text{-}Mutation\}$ do
            $Mu_i$ = random($Mutation$) // randomly select a mutation method
            $P_i$ = mutation($Mu_i, P_i$) //mutate the message parameters
        end for
        $S_i$ = random($S\text{-}Mutation$)
        $seed_i$ = mutation($S_i, seed_i$) //mutate the structure of the seed
    end for
    $testcase$ = script-generated($Seed_i$)
    $result$ = sending-detection($testcase$)
    if interesting($result$) then
        alert($result$)
    end if
    //random phase
    for $M_i \in M$ do
        $operation$= random($N\text{-}Mutation, V\text{-}Mutation, S\text{-}Mutation$)
        $testcase$ = mutation($operation, Seed_i$)
    end for
    $testcase$ = script-generated($Seed_i$)
    $result$ = sending-detection($testcase$)
    if interesting($result$) then
        alert($result$)
    end if
```

For value mutation, we randomly select a parameter position, and perform the following operations on the value of this parameter node:

- **V1**: extend the data content. This step includes two methods. The first is to increase the data length for the data content in the form of characters. This step generally uses multiple copies of the original string or directly fills a character to the maximum length to trigger the buffer overflow vulnerability. The second is to add execution commands after the data, including ping, reboot, or execute a script. Before the device simulation runs, we will execute the script into its file system.
- **V2**: clear the data content. If the web server lacks non-empty verification, this operation will trigger related vulnerabilities;

- **V3**: replace digital data in the boundary integer. This operation might trigger possible data verification errors. Since the HTTP protocol is text-based, we use heuristic rules to judge the field style.
- **V4**: change the content type. This operation might trigger vulnerabilities about assumptions on the data type. The content type of the replacement value has triggered type assumptions. For example, replace the type of digital data with the data in the form of ASCII code. It may cause a crash when the data is processed as a number type.

5.1.2. Structural Mutation

Structural mutation is to mutate the structure of multi-message seed. For the determined phase, we only randomly select a body message to ensure that the authorization messages remain unchanged. The following four mutation strategies are used:

- **S1**: exchange the message adjacent to this position;
- **S2**: repeat the message at this position;
- **S3**: delete the message at this position;
- **S4**: add the payload message after the position.

Table 2 summarizes the seed mutation algorithms supported by the determined phase with examples. The determined phase assigns each algorithm a specific weight at runtime. We empirically set structural mutations with low priority, as the wrong structures generally lead to rejection by the server.

Table 2. Examples of the mutation algorithms.

#	Operation	Before	After
Node Mutation	**N1**	P0 = AAA&P1 = 0	P0 = AAA
	N2	P0 = AAA&P1 = 0	P0 = AAA&P1 = 0&P1 = 0
	N3	P0 = AAA&P1 = 0	P0 = AAA&P1 = 0&P3 = 1
Value Mutation	**V1**	P0 = AAA&P1 = 0	P0 = AAAAAA./test.py&P1 = 0
	V2	P0 = AAA&P1 = 0	P0 = &P1 = 0
	V3	P0 = AAA&P1 = 0	P0 = AAA&P1 = 2^i
	V4	P0 = AAA&P1 = 0	P0 = AAA&P1 = AAA
Structural Mutation	**S1**	M1; M2; M3	M1; M3; M2
	S2	M1; M2; M3	M1; M2; M3; M3
	S3	M1; M2; M3	M1; M2
	S4	M1; M2; M3	M1; M2; M3; Payload

In the random phase, from all the mutation strategies described above, we randomly select multiple mutations, mutate the seeds in the order of selection, at the same time adding the initial message to the mutation sequence.

5.2. Vulnerability Detection

In vulnerability detection, we can monitor the firmware from two aspects: 1. the response from the server and 2. the status of the firmware simulation. For memory-related vulnerability detection, the detection mechanism based on server feedback is faster than the status monitor. By the HTTP status code in response, we can roughly judge whether the device has obvious errors. When an exception occurs to the device, the server's response may include: 1. normal responses; 2. error responses; 3. no response. For error responses, if the crash causes the connection to be interrupted, the user will not access the server. At the same time, the simulation will also make obvious mistakes. For normal response and no response, we can further monitor the process status through instrumentation.

For command injection, it is more difficult to monitor for command injection attacks on real devices. For firmware simulation, the specified executable file is placed in the firmware file system before the simulation. We run the command of the file and check whether the command injection is successful or not by checking whether the file is executed.

6. Implementation and Evaluation

We present the prototype implementation of SIoTFuzzer in Section 6.1 and the evaluation in Section 6.2.

6.1. Implementation of SIoTFuzzer

SIoTFuzzer was implemented with around 5000 lines of Python code in total. In addition, several open-source projects (e.g., *Chrome*, *Boofuzz* [12], *Mitmproxy* [22], *Pyppeteer* [23]) were integrated into this fuzzer to avoid reinventing the wheel.

In the seed generation phase, the front-end analysis was built based on *Chrome* and its *Pyppeteer* driver. Python code was written to use the *Pyppeteer* driver to control the *Chrome* behavior, such as opening a URL, inputting data, and clicking a button. The *mitmproxy* project, an HTTP proxy written in Python code, was extended to filter useless messages and generate initial seeds.

In the fuzzing phase, Python code was written to schedule the fuzzing and convert the seed to the *Boofuzz* test script, and we modified the mutation code of *Boofuzz*. The response message is analyzed to obtain parameter dependency and whether the device will crash.

6.2. Evaluation of SIoTFuzzer

6.2.1. Testing Devices

We crawled firmware images through the official websites of various vendors for simulation, and crawled more than 30 device images, including 9 devices that have web interfaces and can be successfully simulated. The detailed specifications of these images and whether they can be successfully simulated by *Firmadyne* and *FirmAE* are described in Table 3.

Table 3. Summary of IoT devices with firmware simulation.

Type	Vender	Device	Firmadyne	FirmAE
Router	D-Link	DSL-3782	Yes	Yes
	D-Link	DIR-822	Yes	Yes
	D-Link	DIR-823G	Yes	Yes
	D-Link	DIR-865L	Yes	Yes
	D-Link	DAP-2695	Yes	Yes
	TP-Link	WR940N	Yes	Yes
	Netgear	WNAP320	No	Yes
	Trendnet	TEW-652BRP	No	Yes
IP Camera	Trendnet	TV-IP110WN	No	Yes

6.2.2. Testing Environment

The SIoTFuzzer and the other two fuzzers ran in separate virtual machines that host Ubuntu 18.04 with an Intel Core i9 quad-core 3.6 GHz CPU and 8G RAM. Each virtual machine built a *FirmAE* simulation platform. For our seed generator, it was only deployed on our tool, and the generated seed file could be directly transferred to the tool on other virtual machines.

We deployed *FirmFuzz* and *Boofuzz* respectively on the other two virtual machines. For *FirmFuzz*, we did not make any changes and maintained its normal operation. For *Boofuzz*, we extended our function of seed generation and monitoring strategy to create two versions: $Boofuzz_S$ and $Boofuzz_M$.

6.2.3. Research Questions

Using the previous experimental setup, we would like to answer the following questions:

- **Q1**: how effective is SIoTFuzzer in finding real vulnerabilities in IoT firmware?
- **Q2**: how are the suitability and effectiveness of our seed generation function and fuzzing scheduling?
- **Q3**: can SIoTFuzzer outperform the IoT fuzzing tool FirmFuzz in detecting vulnerabilities?

Effectiveness of Vulnerability Detection (Q1): Table 4 lists the vulnerabilities discovered by SIoTFuzzer. For each device under test, SIoTFuzzer used SMG to automatically generate initial seeds within 1 h, and then started fuzzing within 24 h. Finally, it found 12 vulnerabilities: 7 buffer overflows, 3 command injections, and 2 XSSs. These results show that SIoTFuzzer can automatically detect device vulnerabilities based on our SMG mechanism and device monitor.

Table 4. List of discovered known vulnerabilities.

Vulnerability	Device	Exploit ID
Buffer Overflow	D-Link DSL-3782	CVE-2019-7298
	D-Link DIR-822	CVE-2019-6258
	Trendnet TEW-652BRP	CVE-2019-11400
	TP-Link WR940N	CVE-2017-13772
	Netgear WNAP320	CVE-2016-1555
	D-Link DAP-2695	CVE-2016-1558
	Trendnet TV-IP110WN	CVE-2018-19240
Command Injection	Trendnet TEW-652BRP	CVE-2019-11399
	D-Link DSL-3782	CVE-2018-17990
	D-Link DIR-823G	CVE-2019-7297
XSS	D-Link DSL-3782	CVE-2018-17989
	D-Link DIR-865L	CVE-2018-6529

Effectiveness of the Optimizations (Q2): In order to evaluate the effectiveness of our optimizations, we set up three control groups. The specific settings are as follows: for the original $Boofuzz$, we used the original messages that were analyzed through the front-end as the initial seed to test the device; for $Boofuzz_S$, we added the SMG to test, and for $Boofuzz_M$, we added the mutation strategy. The experiment time was 24 h. The results are shown in Table 5.

We performed a further manual analysis and found the following:

1. For comparing $Boofuzz$ with $Boofuzz_S$, when detecting buffer overflow vulnerabilities, $Boofuzz$ is able to detect independently. However, it is unable to cause crashes that are triggered by dependency messages.
2. For comparing $Boofuzz_S$ with $Boofuzz_M$, through adding the mutation strategies, we can cause command injection and XSS. However, without the device monitor, command injection cannot be detected. These results show that our optimization can help us to find more vulnerabilities. In Figure 8, comparing $Boofuzz_M$(grey) with $Boofuzz$(blue), the stateful message and mutation strategy could improve the detection speed by 61.99%.
3. SIoTFuzzer takes more time than $Boofuzz_M$ to find vulnerabilities. The discovery time was increased by about 11.42%. Due to our device monitor, for every test case, we needed to read the simulation log and find the possible vulnerability. These operations will cause time consumption.

Table 5. Control experiment of fuzzing tools.

Exploit ID	Vulnerability	Boofuzz	Boofuzz$_S$	Boofuzz$_M$	SIoTFuzzer
CVE-2019-7298	Buffer overflow	N/A	1 h 14 min	1 h 05 min	1 h 19 min
CVE-2019-6258	Buffer overflow	N/A	1 h 35 min	1 h 26 min	1 h 39 min
CVE-2019-11400	Buffer overflow	3 h 47 min	1 h 26 min	1 h 23 min	1 h 42 min
CVE-2017-13772	Buffer overflow	3 h 34 min	1 h 14 min	56 min	1 h 01 min
CVE-2016-1555	Buffer overflow	1 h 14 min	43 min	36 min	39 min
CVE-2016-1558	Buffer overflow	1 h 31 min	45 min	37 min	41 min
CVE-2018-19240	Buffer overflow	N/A	1 h 02 min	49 min	52 min
CVE-2019-11399	Command injection	N/A	N/A	N/A	2 h 45 min
CVE-2018-17990	Command injection	N/A	N/A	N/A	2 h 21 min
CVE-2019-7297	Command injection	N/A	N/A	N/A	3 h 01 min
CVE-2018-17989	XSS	N/A	N/A	2 h 40 min	3 h11 min
CVE-2018-6529	XSS	N/A	N/A	2 h 33 min	3 h 05 min

$Boofuzz_S$: $Boofuzz$ with comprehensive seed; $Boofuzz_M$: $Boofuzz_S$ with mutation strategies; $SIoTFuzzer$: $Boofuzz_M$ with device monitor.

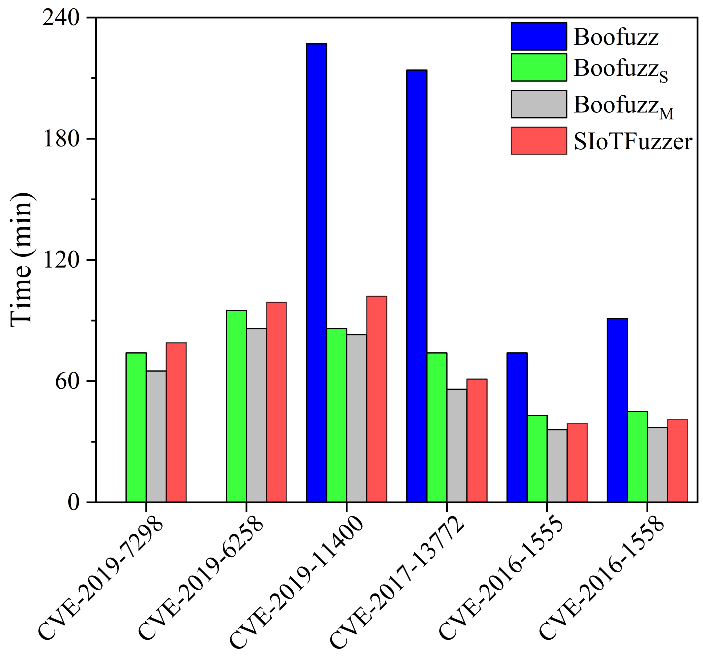


Figure 8. Efficiency comparison of fuzzing tools.

Compare with the FirmFuzz (Q3): In order to evaluate the efficiency and the effectiveness of SIoTFuzzer, we compared it with *FirmFuzz*. Every tool ran within 24 h.

Table 6 lists the efficiency of vulnerability detection by *FirmFuzz* and SIoTFuzzer. We performed a further manual analysis and found the following:

1. *FirmFuzz* can only find four vulnerabilities, and the most common vulnerability found is buffer overflow.
2. In the total execution time, SIoTFuzzer is 17.64% to 23.53% faster than *FirmFuzz*. These results indicate that our work can find more vulnerabilities and detection time is reduced by about 20.57% on average.

Table 6. Statistics on vulnerability detection.

Exploit ID	FirmFuzz	SIoTFuzzer	Improvement
CVE-2019-7298	N/A	1 h 19 min	N/A
CVE-2019-6258	N/A	1 h 39 min	N/A
CVE-2019-11400	N/A	1 h 42 min	N/A
CVE-2017-13772	1 h 15 min	1 h 01 min	18.67%
CVE-2016-1555	51 min	39 min	23.53%
CVE-2016-1558	53 min	41 min	22.64%
CVE-2018-19240	1 h 03 min	52 min	17.64%
CVE-2019-11399	N/A	2 h 45 min	N/A
CVE-2018-17990	N/A	2 h 21 min	N/A
CVE-2019-7297	N/A	3 h 01 min	N/A
CVE-2018-17989	N/A	3 h 11 min	N/A
CVE-2018-6529	N/A	3 h 05 min	N/A

Compared with *FirmFuzz*, SIoTFuzzer pays more attention to the communication process. Through the front-end analysis and state analysis, SIoTFuzzer generates comprehensive seed messages targeting different web interfaces. More pertinent mutation strategies can trigger more vulnerabilities. Through these optimizations, it is easier to reach the path that triggers the vulnerability.

7. Discussion and Limitations

Although SIoTFuzzer can discover vulnerabilities in IoT devices efficiently, there are still some avenues for future improvements.

7.1. Scope of Test Targeted

There are limitations in not only the firmware simulation but also the testing protocols. Although *FirmAE* brings great improvements to the simulation success rate, there are still many devices that cannot be simulated for the different architectures, filesystems, or other reasons. To solve this problem, semi-simulation is promising. SIoTFuzzer or other IoT fuzzing tools mainly focus on HTTP protocols, but some protocols like FTP, SSH, or Telnet lack fuzzing strategies. Combining with machine learning and protocol identification may be the solution to this issue.

7.2. Fuzzing Strategy Optimization

We generated more comprehensive seeds to obtain better test results, but we used a random method for seed selection in each test case. The probability of selecting seeds is the same without distinguishing the priority of the seeds. We will follow up using the coverage guide method. Through the analysis of the simulated firmware process, the priority of the seeds will be evaluated before the fuzzing test. After the pre-run, the seed that can call more processing functions will be selected first.

7.3. Timely Firmware Monitoring

After the web server transmits the parameters, a device takes a long time to process them. While the device is processing incorrectly, the fuzzer has sent some new messages during this time, so the message that triggers the vulnerability needs to be manually located. The testers need to determine the cause of the vulnerability. In order to better locate the error message in the follow-up, a fine-grained monitoring method will be implemented through firmware instrumentation, which makes it easier to find the vulnerability.

8. Conclusions

We have presented SIoTFuzzer, an automated framework to fuzz the web interface of IoT devices based on whole-system emulation. We adopted the function of stateful message generation (SMG). Messages consisting of seeds can basically cover all page operations and make the device normal state transition. We also designed a multi-message seed format to improve the probability of mutated messages being received by devices. At the same time, our mutation strategy can contain parameter dependency between messages.

We used SIoTFuzzer to test for three types of vulnerabilities in the firmware images that we studied: buffer overflow, command injection, and XSS. To evaluate the effectiveness and the efficiency of the SIoTFuzzer, we tested 9 IoT devices and finally found 12 vulnerabilities. Through control experiments, we proved our optimizations are efficient. The stateful message and mutation strategy could improve the detection speed by 61.99%, and our device monitor could issue an error warning in time. Compared with Firmfuzz, the results showed that SIoTFuzzer can indeed detect known vulnerabilities much faster than $FirmFuzz$, and the vulnerability detection time is reduced by about 20.57% on average.

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

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

Acknowledgments: We would like to sincerely thank all the reviewers for your time and expertise on this paper. Your insightful comments help us improve this work.

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

Abbreviations

The following abbreviations are used in this manuscript:

IoT	Internet of Things
SMG	Stateful message generation
ARM	Acorn RISC Machine
MIPS	Microprocessor without Interlocked Piped Stages
I/O	Input/Output
IDA	Interactive Disassembler
LAN	Local Area Network
WAN	Wide Area Network
DDoS	Distributed denial-of-service
HTML	Hypertext markup language
CSS	Cascading style sheets
CVE	Common vulnerabilities and exposures
XSS	Cross-site scripting
HTTP	Hypertext transfer protocol
FTP	File transfer protocol
SNMP	Simple Network Management Protocol
BGP	Border Gateway Protocol
SSH	Secure shell
COTS	Commercial off-the-shelf
CPU	Central processing unit

References

1. Gartner. IoT Security Primer: Challenges and Emerging Practices. 2020. Available online: https://www.gartner.com/en/doc/iot-security-primer-challenges-and-emerging-practices (accessed on 27 March 2021).
2. Exploiting Network Surveillance Cameras Like a Hollywood Hacker. Available online: https://privacy-pc.com/articles/exploiting-network-surveillance-cameras-like-a-hollywood-hacker.html (accessed on 27 March 2021).
3. Venezuela Denounces US Participation in Electric Sabotage. Available online: https://www.telesurenglish.net/news/Venezuela-Denounces-US-Participation-in-Electric-Sabotage-20190308-0021.html (accessed on 27 March 2021).

4. Fortinet Discovers D-Link DIR-866L Unauthenticated RCE Vulnerability. Available online: https://fortiguard.com/zeroday/FG-VD-19-117 (accessed on 27 March 2021).
5. Costin, A.; Zaddach, J.; Francillon, A.; Balzarotti, D. A Large-Scale Analysis of the Security of Embedded Firmwares. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 95–110.
6. Shoshitaishvili, Y.; Wang, R.; Hauser, C.; Kruegel, C.; Vigna, G. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 8–11 February 2015.
7. Chen, D.D.; Woo, M.; Brumley, D.; Egele, M. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016.
8. Zaddach, J.; Bruno, L.; Balzarotti, D.; Francillon, A. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2014.
9. 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, San Diego, CA, USA, 18–21 February 2018.
10. Craig, H. Binwalk: Firmware Analysis Tool. 2010. Available online: https://code.google.com/p/binwalk/ (accessed on 27 March 2021).
11. Alimi, V.; Vernois, S.; Rosenberger, C. Analysis of Embedded Applications By Evolutionary Fuzzing. In Proceedings of the Workshop on Security and High Performance Computing Systems (SHPCS), the IEEE International Conference on High Performance Computing and Simulation (HPCS), Bologna, Italy, 21–25 July 2014.
12. Pereyda, J. A Fork and Successor of the Sulley Fuzzing Framework. 2020. Available online: https://github.com/jtpereyda/boofuzz (accessed on 27 March 2021).
13. 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, 15 November 2019.
14. 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*, 1–19. [CrossRef]
15. 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; pp. 2525–2527
16. 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, CCS 2019, London, UK, 11–15 November 2019.
17. Wang, E.; Wang, B.; Xie, W.; Wang, Z.; Luo, Z.; Yue, T. EWVHunter: Grey-Box Fuzzing with Knowledge Guide on Embedded Web Front-Ends. *Appl. Sci.* **2020**, *10*, 4015. [CrossRef]
18. 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, Xi'an, China, 30 May–3 June 2016.
19. Liu, K.; Koyuncu, A.; Kim, D.; Bissyandé, T.F. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 1–12.
20. Muench, M.; Stijohann, J.; Kargl, F.; Francillon, A.; Balzarotti, D. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In Proceedings of the NDSS, San Diego, CA, USA, 18–21 February 2018.
21. 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, Austin, TX, USA, 7–11 December 2020.
22. Mitmproxy. 2016. Available online: https://github.com/mitmproxy/mitmproxy (accessed on 27 March 2021).
23. Pyppeteer. 2018. Available online: https://github.com/pyppeteer/pyppeteer (accessed on 27 March 2021).