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.); zhouxu@nudt.edu.cn(X.Z.);
yinqidi@nudt.edu.cn(Q.Y.); pfwang@nudt.edu.cn(P.W.); yuetai17@nudt.edu.cn(T.Y.);

Abstract: Cyber attacks against the web management interface of the IoT devices often cause serious consequences. Current researches use fuzzing technologies to test the web interfaces of IoT devices. These IoT fuzzers generate the 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. The SMG addresses two problems in IoT fuzzing. First, we retrieve the messages 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-messages 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 firmwares on the latest Linux-based simulation tool FirmAE. We test 9 IoT devices including router and IP camera and adopt a vulnerability detection mechanism. Our evaluation results show that (1) SIoTFuzzer is capable of finding real-world vulnerabilities in IoT device; (2) our SMG is effective as it enables Boofuzz (a popular protocol fuzzer) to find command injection and 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);Messages Dependency;Front-end Analysis;Multi-messages Seed Format;

Citation: Zhang, H.; L, Kai.; Zhou, X.; Wang, P.; Yue, T.; Yin, Q. . *Journal Not Specified* **2021**, *1*, 0. https://doi.org/

Received:
Accepted:
Published:

1. Introduction

With the rapid development of the Internet of Things(IoT), more and more smart devices are widely used, such as smart homes, routers, and IP cameras. The number of global IoT connections continues to grow exponentially and will reach 25 billion by 2025. A large number of vulnerabilities in IoT devices have been disclosed in recent years. For example, at the 2013 Black Hat Conference, Heffner [1] 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. Besides, real security incidents caused by security vulnerabilities are also emerging in endlessly. In 2019, most areas in Venezuela including the capital Caracas experienced a continuous power outage more than 24 hours [2]. 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 [3], which affected more than 10 products of related models, but the product has been discontinued, D-link vendor did not release related patches, and the vulnerability has not been fixed. This means that once this device is exposed, it is very 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 in time.

However, due to the huge differences in 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. The approaches to detect IoT vulnerability is divided into static methods [4–6] and dynamic methods [7–9]. There are three steps in the workflow. Firstly, testers need to collect firmware images from public channels, such as online support service [10]. Secondly, these images are processed by unpacking tools, such as Binwalk [11]. Thirdly, static methods or dynamic methods are deployed to detect flaws in these unpacked files. These approaches suffer from known drawbacks. For static methods, different IoT devices usually use different chipsets that have customized features (e.g., instruction sets, memory layouts, and so on), so it is difficult to analyse firmware binary due to the diversity of underlying architectures. And for dynamic methods, on the basis of ensuring the correct operation of the device, it is complex to monitor device, and the monitor imposes the overhead of vulnerability analysis.

Vendors usually use web and APP to provide users with operating interfaces. These interfaces can directly operate the device, and their design standards evolve according to the actual operation of the device. When the web and APP obtain user's input, they will send operation messages to device. After receiving messages, the device does more further procedures according to the message content (e.g., executing a targeted program) and the status of device will change with this process. If there is an implementation flaw in the message parsing or the further procedure, a vulnerability may be exploited. Therefore, an IoT device that has the web interface can be treated as a blackbox, and feeding this box with malformed messages could trigger potential vulnerabilities of it. Additionally, this blackbox fuzzing does not require the knowledge of underlying architecture about the targeted device and there is no need to device monitor timely, the fuzzing could keep a high throughput. However, blackbox fuzzing will generate much more invalid test cases without feedback. At the same time, if the device does not receive the stateful message and is not in a state of accepting messages, device will refuse service or interrupt the connection. As a result, it is ineffective to continue sending mutated messages. Furthermore, some message 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 the 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. For improving the efficiency and effectiveness of fuzzing, we propose a stateful message generation (SMG) mechanism, SMG addresses two challenges including the status maintenance of device and the mutation of parameter dependency messages. We analyse the front-end of IoT device's web interface to build initial seeds and generate test cases. Due to the difficulty of firmware operation monitoring, we can analyse operating interfaces to obtain prior knowledge. This knowledge will help us test device more comprehensively. We adopt a multi-messages seed format, and every seed contains a complete sequence of operations. Based on Boofuzz [12] (a popular protocol fuzzer) we design a black-box fuzzing tool called SIoTFuzzer which could detect IoT device vulnerability. By building a simulation environment, it is more suitable for analyzing the web management interface and constructing the input of 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. 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 of IoT device including the status maintenance of device and keeping parameter dependency between messages, we adopt the stateful messages generation (SMG). In addition, we adopt a multi-messages seed format and deploy a corresponding mutation strategy to guide fuzzing;
2. We design and implement a blackbox fuzzer SIoTFuzzer for fuzzing IoT device. Through analysis of device web interface, we can obtain prior knowledge of web elements. SIoTFuzzer traverses the device web pages and gets the normal communication messages. These messages will be used to 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 vulnerability detection time is reduced by about 20.57% on average.

2. Background and Motivation

In this section, we introduce the background knowledge and motivation about discovering vulnerabilities via fuzzing web management interface. For fuzzing IoT device, we need to pay more attention to the following issues: 1. in the test preparation, how can we get 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 keep 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 2.4 below.

2.1. 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 [14]. The web server is mainly used for message transmission between the front-end and the device program processor called pagehandler. The main workflow is shown in Figure 1. Firstly the front-end gets the user's inputs. Then the front-end packages these inputs into messages. Secondly after decoding the message, web server passes the parameters to pagehandler. Thirdly pagehandler returns the processing results which obtain the HTTP Status Code. Finally, front-end receive the results and display them on the page.

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

2.2. Firmware Simulation

The previous research mainly adopted three methods for the operation of the IoT devices: 1. physical objects; 2. semi-simulation(e.g., *AVATAR* [15]); 3. full system simulation (e.g., *Firmadyne* [8]). In the test of real devices, *IoTfuzzer* [16] detects whether

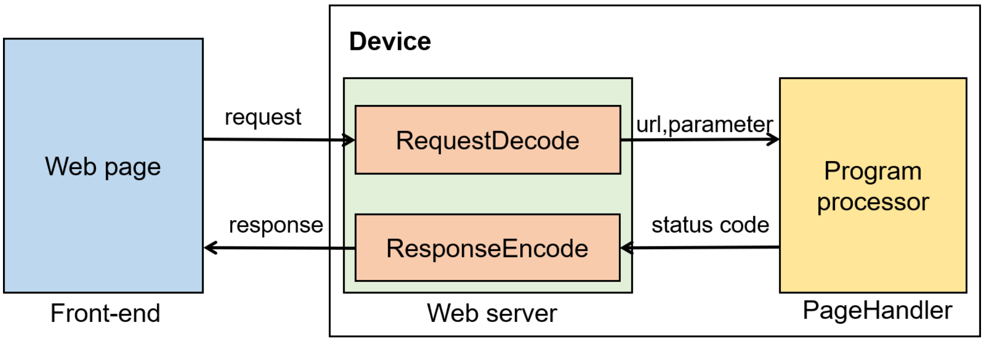


Figure 1. Workflow of web interface

the device is online by sending a heartbeat message, and every ten messages obtain a heartbeat message. This method is suitable for detecting obvious crashes, and the next time for test after crash requires device to restart, this time cost for restart is unacceptable. As a study by Muench et al. [17] pointed out, because the 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 device crash, 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 a basic automated analyse to detect vulnerability. This script tests for the presence of 60 known vulnerabilities using exploits from *Metasploit*. But in nearly 2,000 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 bring better runtime environment support. The subsequent improvement work, *FirmAE* [18] 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.

2.3. Fuzzing Technology

Fuzzing is a software testing technique that can provide 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 IoT firmware testing tools

Fuzzer	**Boofuzz** [12]	**IoTFuzzer** [16]	**WMIFuzzer** [19]	**FirmFuzz** [13]	**Firm-AFL** [20]
Fuzzing Technique	Blackbox	Blackbox	Blackbox	Blackbox	Greybox
Hardware Support	All	Real	Real	Emulation	Emulation
Protocol Support	Need Template	None	HTTP	HTTP	HTTP
Message Dependency	None	None	None	None	None

As described in Section 2.1 and 2.2, because of the difficulty of firmware analysis and the accessibility of front-end, most IoT tests adopt blackbox fuzzing. *Boofuzz* is a protocol fuzzing tool based on Python language, it requires protocol templates. Writing protocol templates could bring a large workload, but *Boofuzz* is strongly extensible for many kinds of scenarios.

2.4. Motivation

In Section 2.1, the web interface is used to accept the user's inputs and translate inputs into communication messages. These messages result in the change of device state. When pagehandler accepts the error messages, it may cause the device to crash. Generating mutated device messages is a major concern of 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 web page is often used. However, with the application scenarios of IoT device 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 of web page, including XSS, special character, and invalid address check. Every input which 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 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 field. *FirmFuzz* will generate hundreds of test cases and send these requests to server at a time. As an operation always contains a message sequence, a single message is just a part of the operation. And 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 web server. When generating a test case, fuzzer need to request server for a unique key first, and then add it in message. Besides, this session has a timeout so that we need to request server in every test case. If we ignore device status and stateful messages, it will lead to two matters:

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 that too many invalid messages are 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, test cannot trigger vulnerability on the contrary.

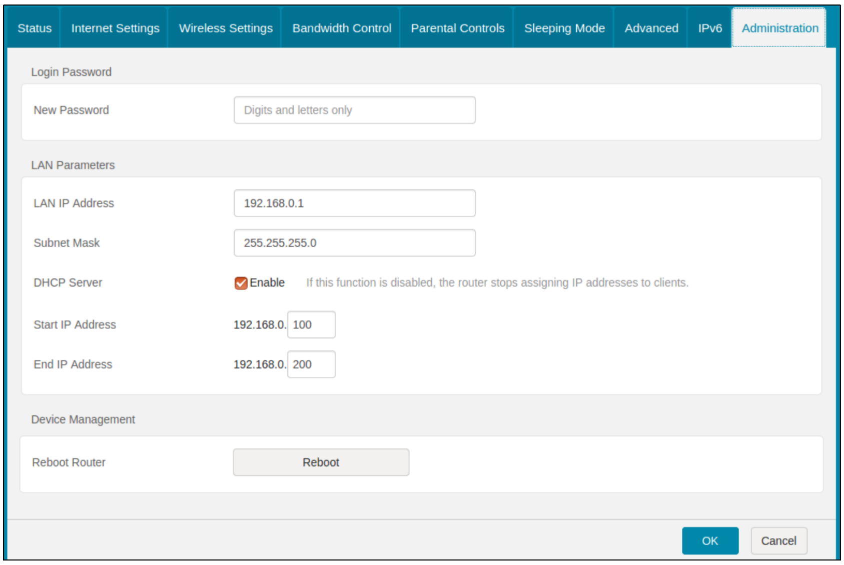
Through the above problems, fuzzer need keep the connection status and mutate data field which has no dependency to make sure that web server receives all mutated message. In Table 1, we list five fuzzing tools which can test IoT device, None of them can deal with these two problems. In this work, we propose a stateful message generation (SMG) mechanism to keep the device status and connection which is described in Section 3.

3. Stateful Message Generation (SMG)

In Section 3, we will introduce our pre-analysis process of the IoT device web management interface, which is used to generate stateful messages to solve the device status problem in Section 2. This process is divided into three parts: front-end analysis, state analysis, and seed generation.

3.1. 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 to 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. And 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

Figure 3. The administration setting 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 trigger through click, 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 elements and click elements on the page and fill the input elements. For every page, we maintain a clicked queue to make sure trigger all operations.

The front-end analysis is divided into three steps:

```
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 setting web page

1. Determine whether to enter a new page that has never visited; We need to identify the current page elements and create the lists of input and click elements. These element lists will not be released until the end of analysis.
2. Fill in the input elements and create a dictionary library to match the element names with certain rules. The code in 4 corresponds to the device page in Figure 3, where in lines 1-20 are the input elements on the page. Our page elements filling uses certain rules to match the type of data, including address, character, and number, 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 non-form input, and we need to record these 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

3.2. State Analysis

In order to keep the connection between the server and the fuzz 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, firstly, we should make the device status change from waiting to an authorization. we need to capture the authorization messages and replay these message to device. Secondly, the web server sends the operation messages. A page operation may include the interaction of multiple messages. The traditional fuzzing tool is used a single message to construct a test case. This method cannot handle the vulnerabilities that may be caused by the complex message process. Fuzzer will generate a large number of invalid test cases that are rejected by the server. To solve this problem, each time we analyse the device state, the operation sends a message sequence that corresponds to a page operation. The message sequence from the wait 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 finish.

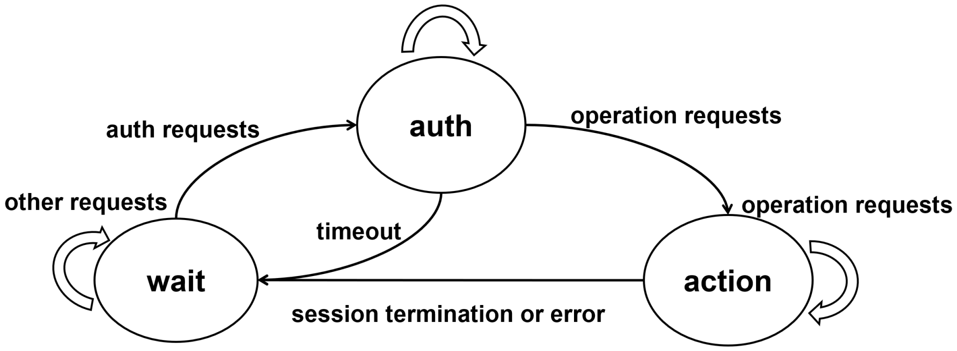
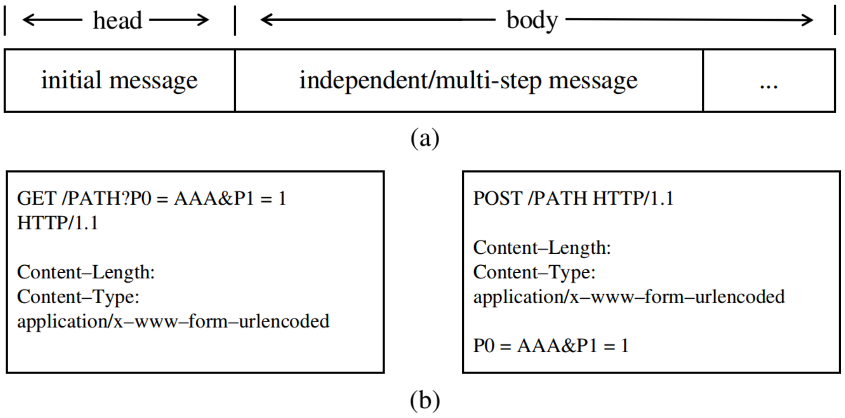


Figure 5. State transition of the device.

3.3. Seed generation

Through state analysis, we get the sequence of messages corresponding to an operation, filter the useful messages, and reconstruct the seeds. First, we need to filter the messages, only keep the GET and POST requests in the HTTP request, and remove the GET request for web resources, in Figure 6b is 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**: it is used to authorize the device so that subsequent messages can be accepted by the web server;
2. **Independent reference message**: it 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.



```
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 the multi-messages seed.(**b**)The format of messages consisting seed.

As shown in Figure 6a, for each initial seed, it can be divided into two parts, head, and body. The head must be the initial message, and the body can contain several

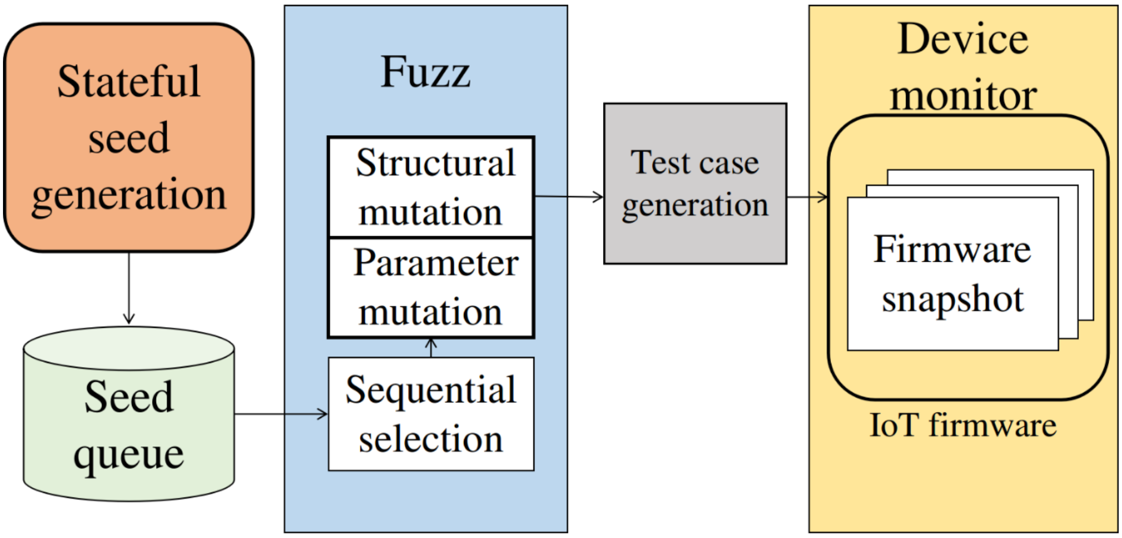


Figure 7. Framework of SIoTFuzzer

independent messages and multi-step messages. Then put the generated seed into the seed pool and wait for seeds to be selected and mutated.

4. Framework of SIoTFuzzer

The Framework of SIoTFuzzer is described from two main aspects in Section 4. As shown in Figure 7, after simulating the IoT device, SIoTFuzzer selects a seed in order from the seed queue which has been generated in Section 3. 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 Section 4.1 and 4.2.

4.1. Mutation Strategy

According to the seed format used in Section 3, 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 determined phase and random phase. The determined phase is divided into two stages: parameter mutation and structural mutation.

4.1.1. 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 is 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.

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; 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. The HTTP protocol is a text-based protocol, so we use regular matching to determine whether the parameter may be a digital type. The digital data will be replaced with classic boundary integer numbers: 2^i, $2^i - 1$, and $2^i + 1$, where $0 \leqslant i \leqslant 32$.
- **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.

4.1.2. Structural Mutation

Structural mutation is to mutate the structure of multi-messages seed. For the determined phase, we only randomly select a body massage 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: The 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;

Table 2 summarizes the seed mutation algorithms supported by determined phase with examples. 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.

In 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 add the initial message to the mutation sequence.

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
        $P = message\text{-}parameter(M_i)$ // 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 structural of seed
    end for
    $Testcase = Script\text{-}generated(Seed_i)$
    $result = sending\text{-}detection(Testcase)$
    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
    $Testcase = Script\text{-}generated(Seed_i)$
    $result = sending\text{-}detection(Testcase)$
    if $interesting(result)$ then
        $alert(result)$
    end if
```

4.2. Vulnerability Detection

In vulnerability detection, we can monitor the firmware from two aspects: 1. The response from the server. 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 response; 2. error response; 3. no response. For error response, if the crash causes the connection 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 be monitored by command injection attacks for real devices. For firmware simulation, the specified executable file is placed in the firmware file system before the simulation. Run the command of the file and check whether the command injection is successful or not by checking whether the file is executed.

5. Implementation and Evaluation

We present the prototype implementation of SIoTFuzzer in Section 5.1 and the evaluation in Section 5.2.

5.1. Implementation of SIoTFuzzer

SIoTFuzzer was implemented with around 5,000 Python lines of code in total. Also, several open-source projects (e.g., *Chrome*, *Boofuzz* [12], *Mitmproxy* [21], *Pyppeteer* [22]) are 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, convert the seed to the *Boofuzz* test script and we modified the mutation code of *Boofuzz*. The response message is analysed to get parameter dependency and whether the device crash.

5.2. Evaluation of SIoTFuzzer

5.2.1. 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

5.2.2. Testing Environment

The SIoTFuzzer and the other two fuzzers run 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 builds a *FirmAE* simulation platform. For our seed generator, it is only deployed on our tool, and the generated seed file can be directly transferred to the tool on other virtual machines.

We deploy *FirmFuzz* and *Boofuzz* respectively on the other two virtual machines. For *FirmFuzz*, we do not make any changes and maintain its normal operation. For

$Boofuzz$, we extend our function of seed generation and monitoring strategy on it to create two versions: $Boofuzz_S$ and $Boofuzz_M$.

5.2.3. Research Questions

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

- **Q1**: how effective is SIoTFuzzer in finding real vulnerabilities in IoT firmware?
- **Q2**: how about 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 **??** lists the vulnerabilities discovered by SIoTFuzzer. For each device under test, SIoTFuzzer uses SMG to automatically generate initial seeds within 1 hour, and next start fuzzing within 24 hours. 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 decive 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 use the original messages which were analysed through the front-end as the initial seed to test the device; for $Boofuzz_S$, add the SMG to test, and for $Boofuzz_M$, add the mutation strategy. The experiment time is 24 hours. 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, but it is unable to cause crashes which 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. But without device monitor, command injection cannot be detected. These results show that our optimization can help us to find more

Table 5: Control experiment of fuzzing tools

Exploit ID	**Vulnerability**	**Boofuzz**	**Boofuzz**$_S$	**Boofuzz**$_M$	**SIoTFuzzer**
CVE-2019-7298	Buffer overflow	N/A	1h14m	1h05m	1h19m
CVE-2019-6258	Buffer overflow	N/A	1h35m	1h26m	1h39m
CVE-2019-11400	Buffer overflow	3h47m	1h26m	1h23m	1h42m
CVE-2017-13772	Buffer overflow	3h34m	1h14m	56m	1h01m
CVE-2016-1555	Buffer overflow	1h14m	43m	36m	39m
CVE-2016-1558	Buffer overflow	1h31m	45m	37m	41m
CVE-2018-19240	Buffer overflow	N/A	1h02m	49m	52m
CVE-2019-11399	Command injection	N/A	N/A	N/A	2h45m
CVE-2018-17990	Command injection	N/A	N/A	N/A	2h21m
CVE-2019-7297	Command injection	N/A	N/A	N/A	3h01m
CVE-2018-17989	XSS	N/A	N/A	2h40m	3h11m
CVE-2018-6529	XSS	N/A	N/A	2h33m	3h05m

[1] $Boofuzz_S$: $Boofuzz$ with comprehensive seed;
[2] $Boofuzz_M$: $Boofuzz_S$ with mutation strategies;
[3] $SIoTFuzzer$: $Boofuzz_M$ with device monitor;

vulnerabilities. And compared $Boofuzz_M$ with $Boofuzz$, 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 need to read the simulation log and find the possible vulnerability. These operations will cause the time consumption.

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

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 vulnerability and detection time is reduced by about 20.57% on average.

6. Discussion and Limitations

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

6.1. Scope of Test Targeted

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

Table 6: Statistics on vulnerability detection

Exploit ID	FirmFuzz	SIoTFuzzer	improvement
CVE-2019-7298	N/A	1h19m	N/A
CVE-2019-6258	N/A	1h39m	N/A
CVE-2019-11400	N/A	1h42m	N/A
CVE-2017-13772	1h15m	1h01m	18.67%
CVE-2016-1555	51m	39m	23.53%
CVE-2016-1558	53m	41m	22.64%
CVE-2018-19240	1h03m	52m	17.64%
CVE-2019-11399	N/A	2h45m	N/A
CVE-2018-17990	N/A	2h21m	N/A
CVE-2019-7297	N/A	3h01m	N/A
CVE-2018-17989	N/A	3h11m	N/A
CVE-2018-6529	N/A	3h05m	N/A

6.2. Fuzzing Strategy Optimization

We generation more comprehensive seeds to obtain better test results, but we use 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 seed will be evaluated before the fuzzing test. After the pre-run, the seed which can call more processing functions will be selected first.

6.3. Timely Firmware Monitoring

After the web server transmits the parameters, it has taken a long time for the device to process the parameters. While the device is processing wrong, 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.

7. Related Work

With the increasing number of IoT security issues, fuzzing techniques are proposed to find the IoT devices vulnerabilities in an automatic manner, including mutation-based fuzzing and generation-based fuzzing.

7.1. Mutation-based fuzzing

Since most network-enabled devices will communicate with an external entity, some works are presented to fuzz these communication protocols for vulnerability discovery. *RPFuzzer* [23] is a blackbox fuzzing framework to detect vulnerabilities in Cisco routers, and it used a predefined data model to generate seeds for mutation-based fuzzing. The main challenge is that it requires a security expert to write the data model, so it cannot be leveraged to test other devices automatically. Wang et al.[19] presented WMIFuzzer, a mutation-based blackbox fuzzer targeting the web management interface in COTS IoT devices; a weighted message parse tree (WMPT) was proposed to guide the mutation to generate mostly structure-valid messages. However, WMIFuzzer does not have the support for local monitoring. And fuzzing for test real device, WMIFuzzer do not have a high throughput because the connection of the test is often interrupted. However, these works impose overhead on the device's startup and rebooting for each fuzzing session. In order to improve this problem, through the firmware simulation, rebooting device from snapshot could reduce overhead. Zheng et al.[20] presented *Firm-AFL*, the mutation-based greybox fuzzing platform for IoT firmware which aims to minimize each fuzzing iteration overhead so that the fuzzer can test more test cases in the same unit of time. It proposed a novel technique, augmented process emulation to achieve high throughput fuzzing by running the target program in a user-mode emulator and switch 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 care about the communication process, so the increase in the coverage of a single program is difficult to trigger inter-program vulnerability.

7.2. Generation-based fuzzing

Chen et al. [16] presented *IoTFuzzer* that performs a protocol-guarded fuzzing on COTS devices; its key idea is that many IoT devices can be controlled through their official mobile apps. So, they firstly adopted a taint-based approach to track the atomic data that are used to construct the network message; then, they mutated these atomic data dynamically to reuse the original code of message building. However, not all IoT devices have an official control app, and *IoTFuzzer* can just detect memory corruption. After that, Costin et al. [14] presented an automated framework to discover vulnerabilities in web interfaces of embedded devices; it works by integrating Qemu to run the web service and testing the web service via existing web penetration tools. Although it used some heuristic techniques to run chroot and init to launch the web service, it may fail because of the side effects of forced emulation, diversity of web server environment, and limitations of Qemu. Based on this automated framework, Prashast et al. [13] presented *FirmFuzz*, a fuzz testing of embedded firmware images. Closest to our work, *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 care about mutation strategy, and hence the chance of detecting a vulnerability is relatively low. Compared with above blackbox fuzzing works, SIoTFuzzer pays more attention on communication process. Through the front-end analysis and state analysis, SIoTFuzzer generates comprehensive seed messages targeting different web interfaces. And more pertinency mutation strategies can trigger more vulnerabilities.

8. Conclusion

We present SIoTFuzzer, an automated framework to fuzz the web interface of IoT device based on whole-system emulation. We adopt the function of stateful message generation (SMG). These messages consisting seed could basically cover all page operations and make the device normal state transition. Then we design a multi-messages seed format to improve the probability of being received mutated messages by devices. At the same time, Our mutation strategy could contain the 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 test 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 the error warning in time. Compared with Firmfuzz, the results showed that SIoTFuzzer could indeed detect known vulnerabilities much faster than $FirmFuzz$, and 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 is supported by National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), NSF 61902405, the HUNAN Province Science Foundation 2017RS3045.

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
DDoS	Distributed Denial-of-service
HTML	Hyper Text Markup Language
CSS	Cascading Style Sheets
CVE	Common vulnerabilities and exposures
XSS	Cross-site Scripting
HTTP	HyperText Transfer Protocol
FTP	File Transfer Protocol
SSH	Secure Shell
COTS	Commercial Off-the-shelf
CPU	Central Processing Unit

References

1. Exploiting Network Surveillance Cameras Like a Hollywood Hacker. https://media.blackhat.com/us-13/US-13-Heffner-Exploit-ing-Network-Surveillance-Cameras-Like-A-Hollywood-Hacker-WP.pdf.
2. Venezuela Denounces US Participation in Electric Sabotage. https://www.telesurenglish.net/news/Venezuela-Denounces-US-Participation-in-Electric-Sabotage-20190308-0021.html.
3. Fortinet Discovers D-Link DIR-866L Unauthenticated RCE Vulnerability. https://fortiguard.com/zeroday/FG-VD-19-117.
4. Cui, A.; Stolfo, S.J. A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan. Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010, 2010.
5. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Cross-Architecture Bug Search in Binary Executables. IEEE Symposium on Security and Privacy, 2015, pp. 709–724.
6. Shoshitaishvili, Y.; Wang, R.; Hauser, C.; Kruegel, C.; Vigna, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. Network and Distributed System Security Symposium, 2015.
7. Hemel, A.; Kalleberg, K.T.; Vermaas, R.; Dolstra, E. Finding software license violations through binary code clone detection **2011**. p. 63.
8. Chen, D.D.; Woo, M.; Brumley, D.; Egele, M. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. NDSS, 2016.
9. Zaddach, J.; Bruno, L.; Balzarotti, D.; Francillon, A. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares **2014**.
10. DLink. DLink firmware FTP server. http://ftp.dlink.ru/pub/, 2016.
11. Craig, H. Binwalk: firmware analysis tool. https://code.google.com/p/binwalk/, 2010.
12. J, P. boofuzz. https://github.com/jtpereyda/boofuzz, 2016.
13. Srivastava.; Peng, H.; Li, J.; Okhravi, H.; Shrobe, H.; Payer, M. FirmFuzz: Automated IoT Firmware Introspection and Analysis. *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things* **2019**.
14. Costin, A.; Zarras, A.; Francillon, A. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* **2016**.

15. Liu, K.; Koyuncu, A.; Kim, D.; Bissyandé, T.F. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* **2019**, pp. 1–12.
16. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. Network and Distributed System Security Symposium, 2018.
17. Muench, M.; Stijohann, J.; Kargl, F.; Francillon, A.; Balzarotti, D. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. NDSS, 2018.
18. Kim, M.; Kim, D.; Kim, E.; Kim, S.; Jang, Y.; Kim, Y. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. *Annual Computer Security Applications Conference* **2020**.
19. Wang, D.; Zhang, X.; Chen, T.; Li, J. Discovering Vulnerabilities in COTS IoT Devices through Blackbox Fuzzing Web Management Interface. *Security and Communication Networks* **2019**, *2019*, 1–19.
20. 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. USENIX Security Symposium, 2019.
21. mitmproxy. https://github.com/mitmproxy/mitmproxy, 2016.
22. pyppeteer. https://github.com/pyppeteer/pyppeteer, 2018.
23. Wang, Z.; Zhang, Y.; Liu, Q. RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing. *Ksii Transactions on Internet and Information Systems* **2013**, *7*, 1989–2009.