Hindawi
Security and Communication Networks
Volume 2019, Article ID 5076324, 19 pages
https://doi.org/10.1155/2019/5076324

Research Article

Discovering Vulnerabilities in COTS IoT Devices through Blackbox Fuzzing Web Management Interface

Dong Wang,[1] **Xiaosong Zhang**,[2] **Ting Chen**,[2] **and Jingwei Li**[3]

[1]*University of Electronic Science and Technology of China, ADLab of Venustech, Chengdu, China*
[2]*University of Electronic Science and Technology of China, Chengdu, China*
[3]*University of Electronic Science and Technology of China, State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Chengdu, China*

Correspondence should be addressed to Xiaosong Zhang; johnsonzxs@uestc.edu.cn

Received 16 March 2019; Revised 31 August 2019; Accepted 10 September 2019; Published 4 November 2019

Academic Editor: Prosanta Gope

A novel approach for discovering vulnerability in commercial off-the-shelf (COTS) IoT devices is proposed in this paper, which will revolutionize the area. Unlike previous work, the web management interface in IoT was used to detect vulnerabilities by leveraging fuzzing technology. To validate and evaluate this scheme, a tool named WMIFuzzer was designed and implemented. There were also two challenges: (1) due to the diversity of web interface implementations, there were no existing seed messages for fuzzing this interface and it was inefficient while taking random messages to launch the fuzzing and (2) because of the highly structured seed message, fuzzing with byte-level mutation could conduce to be rejected by the device at an early stage. To address these challenges, a brute-force UI automation was designed to drive the web interface to generate initial seed messages automatically, as well as a weighted message parse tree (WMPT) was proposed to guide the mutation to generate mostly structure-valid messages. The extensive experimental results show that WMIFuzzer could achieve expected result while 10 vulnerabilities including 6 zero-days in 7 COTS IoT devices were discovered.

1. Introduction

With the rapid progress of Internet of things (IoT) technologies, more and more devices have been deployed in our daily lives, such as smart home routers and IP cameras. According to a recent report [1], the number of IoT devices will reach 20+ billion in 2020. These network-enabled devices bring new ability to customers and make their lives easier, while they also attract attacks to compromise them [2–4]. Some studies show that many IoT devices are protection-less, and the security vulnerabilities in them are usually easy to be exploited [5–9]. For example, a security researcher in F-Secure found 18 zero-days in a Foscam IP camera, including insecure default credentials, command injection, stack-based buffer overflow, and so on [10]. Security vulnerabilities in IoT devices can cause a big impact because of the huge quantity of these devices. An example is the Mirai that launched a massive DDoS, which made several leading online services inaccessible including Twitter, PayPal, and Netflix [11]. Therefore, it is crucial to discover the vulnerabilities in IoT devices before deploying them in practice.

A straightforward approach for discovering vulnerabilities is analyzing implementation flaws in the firmware of targeted IoT device, as the firmware provides the low-level control of the device hardware. This is called firmware-based approach that usually contains three steps. Firstly, firmware images are collected from public channels, such as online support service [12]. Secondly, these images are processed by unpacking tools, such as Binwalk, BAT, and FRAK [13–15]. Thirdly, static methods [16–18] or dynamic methods [6, 15, 19, 20] are deployed to detect flaws in these unpacked files. However, firmware-based approaches suffer from known drawbacks. The first one is the availability of firmware images since many vendors do not release them publicly. The second one is the difficulty of unpacking

images, as most vendors prefer to pack them with a compression algorithm to reduce the consumption of device storage. Existing tools can only unpack some images because they work upon known algorithms and file formats. Those images packed with a private file format or encrypted with a private key cannot be unpacked by these tools. For example, a prior work [19] collected 23,035 images, and it only succeeded in unpacking 8,617 images of them via existing tools. The third one is the difficulty of binary analysis due to the diversity of underlying architectures. Different IoT devices usually use different chipsets that have customized features (e.g., instruction sets, memory layouts, and so on). It is still challenging to analyze the unpacked binary files without the knowledge of underlying architecture.

Due to the lightweight and resource constrain, many IoT devices usually provide network interfaces to allow users to interact with them instead of a screen and a keyboard. Web management interface that is designed for device administration is a popular instance of these network interfaces. This interface packs user inputs into a message and sends it to the device. After receiving this message, the device parses the message and does more further procedure according to the message content (e.g., executing a targeted program). If there is an implementation flaw in the message parsing or the further procedure, a vulnerability may be exploited. So, 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.

Motivated by this, this paper leverages mutation-based fuzzing technology to perform blackbox testing automatically. Since the web management interface is accessed via network, it is fully independent of the device firmware images and can be used to test COTS IoT devices that do not release their firmware images publicly. Additionally, this blackbox testing does not require the knowledge of underlying architecture about the targeted device. Two challenges detailed in Section 3 are needed to be addressed, including the generation of initial seed messages and the mutation of highly structured message.

To validate and evaluate this blackbox fuzzing, a tool named WMIFuzzer was designed and implemented for discovering vulnerabilities in COTS IoT devices automatically. WMIFuzzer was tested on 7 devices, and it discovered 10 vulnerabilities including 7 zero-days. National Internet Emergency Center in China (CNCERT/CC) is a coordination organization [21] that is responsible for analyzing and reducing cyber threats, vulnerabilities, disseminating cyber threat warning information, and coordinating incident response activities. All vulnerabilities found by WMIFuzzer have been reported to the CNCERT/CC, and all of them have been fixed now. WMIFuzzer was also compared to the state-of-the-art fuzzers, AFL and Sully [22, 23], and the result showed that it is better than both of them in efficiency and effectiveness.

In short, this paper makes the following major contributions:

(i) To the best of our knowledge, WMIFuzzer is the *first* blackbox fuzzer designed for fuzzing the web management interface in COTS IoT devices automatically

(ii) Two challenges about fuzzing this interface were addressed including the generation of initial seed messages and the mutation of highly structured message

(iii) We implemented the blackbox fuzzing in WMIFuzzer and evaluated it on 7 COTS IoT devices: 10 vulnerabilities including 6 zero-days were found

The remainder of this article is structured as follows. In Section 2, the background knowledge of IoT vulnerability discovery is introduced. Then, the design of WMIFuzzer is presented in Section 3. In Section 4, the implementation and evaluation are presented. The limitations are discussed in Section 5. The survey of related work in Section 6 is followed by the conclusion in Section 7.

2. Background

In this section, the background knowledge about discovering vulnerabilities via fuzzing web management interface is introduced.

2.1. Typical IoT Network Architecture. In a typical IoT environment (like the smart home shown in Figure 1), several devices are deployed for different purposes. There are two types of IoT nodes: a *gateway node* and multiple *sensor nodes*. The *sensor nodes* work by (1) collecting external information and pushing them to the remote user and (2) receiving commands from the remote user and executing them. Some *sensor nodes* are coupled with Internet capability, such as the camera connected to the *homeGate* via WiFi. While some cannot access the Internet directly, such as the wristband connected to a mobile phone via Bluetooth and the light connected to a hub via Zigbee. The *mobile phone* and the *Zigbee hub* can make a connection with remote user via Internet, and both of them contain a proxy module that works as a bridge between the Bluetooth (Zigbee) and the Internet. The *gateway node*, which is usually a wireless home router, provides the access point of Internet. The *camera*, *mobile phone*, and *Zigbee hub* are connected to this node. It plays an important role since the insider and outsider of the whole home network are separated by this *gateway node*.

Unlike the traditional information system, there is usually a lot of sensitive data in the IoT environment (e.g., sleeping patterns, health information, human activity, and child privacy). Since these data are shared through the Internet that is an open network, security and privacy are very critical. Due to the resource constrain, some lightweight authentication protocols are proposed [24, 25] for IoT devices. Another recent work proposed a privacy-preserving scheme, called PrivHome, which supports authentication, secure data storage, and query for sensitive data [26]. Since PrivHome is lightweight, it is practical for smart devices which have limited resources.

Most IoT nodes do not provide a screen and a keyboard, and they prefer to provide a network interface to facilitate

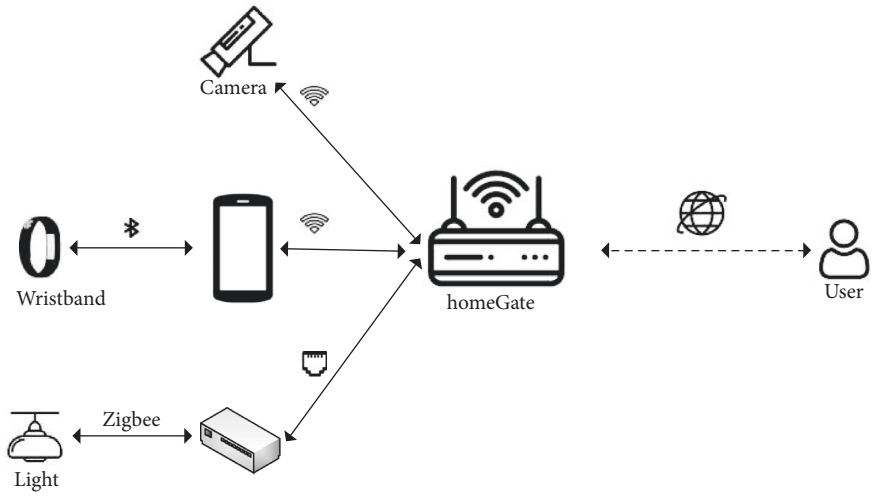

Figure 1: Smarthome network architecture.

the user's operations on device management. For example, the customer uses the browser to login to his camera management interface and update video parameters. If the implementation of this network interface contains security vulnerabilities, an attacker can apply them to launch a crime. A prior study shows that security is an important factor for IoT users [27] because people want the devices to be under their control rather than being fully automated. However, vulnerabilities can make people lose control of these devices.

In this paper, we focus on discovering vulnerabilities in IoT devices that have the capability of Internet since an attacker can do more serious harm by compromising them. For example, vulnerabilities in IP cameras had been used to launch the well-known DDoS. Moreover, even if the camera in Figure 1 does not contain any vulnerability, an attacker can also utilize vulnerabilities in the gateway node to redirect the network flow of the camera into an evil Internet node to extract privacy via machine learning [28, 29].

2.2. Web Management Interface in IoT Devices. As stated previously, an IoT device usually provides a network interface for users to manage itself. Although there are no standards about how to implement this interface, many vendors prefer to utilize web technology because of its flexibility and simplification [6, 30]. However, it is well known that making secure web applications is not a trivial task [31].

The overview of the web management interface is illustrated in Figure 2. There are 3 parts: *FrontEnd*, *Webserver*, and *PageHandler*. The *FrontEnd* is composed of HTML codes, JavaScript codes, CSS codes, and other static resources. The *FrontEnd* is run in a browser, and it has 2 goals. The first one is to be presented as a graphical user interface (GUI) to receive user inputs and clicking, and the second one is packing user inputs into a request message and sending it to the *Webserver*. The *Webserver* and *PageHandler* are both run in the IoT device; the former focuses on message encoding/decoding and the latter focuses on business logic. In detail, the *Webserver* firstly decodes the message to URL and other parameters, then it locates the corresponding *PageHandler* and passes parameters to the handler for further procedure, and finally it encodes the result from the *PageHandler* into a response message and sends it back to the *FrontEnd*. The implementations of *PageHandler* are diverse, such as PHP, Perl, Lua, and CGI-bin [32–34]. If there are flaws in the implementation of the *Webserver* or the *PageHandler*, sending malformed messages to this interface could trigger them.

Additionally, an IoT device is usually designed for a specific purpose, such as the IP camera that focuses on the transmission of captured video data via TCP/IP network. Vendors usually prefer to pay more attention on the improvement of data transmission and video quality, while they just pay little resource on the web management interface including the security. A prior work [5] found a lot of embedded devices are misconfigured in the web management interface by static analysis. Therefore, the web management interface is a good surface to discover vulnerabilities in IoT devices.

2.3. Fuzzing Technology. Fuzzing is an automated random testing technology that was first developed by Takanen et al. [35] to understand the reliability of UNIX tools. The core of fuzzing is to fill targeted tools with random testing data in the goal of triggering flaws. Fuzzing has become widely popular in software testing, and many serious vulnerabilities have been found by this technology [23].

Based on how to generate the random testing data, there are two major categories [36]: *generation-based fuzzing* generates random data by following a data model written manually, while *mutation-based fuzzing* generates random data by mutating an initial data seeds. *Generation-based fuzzing* can generate complicated data fields, such as a field that represents the checksum of a set of bytes. SPIKE, Sulley, and Peach are the state-of-the-art fuzzers built on *generation-based fuzzing*, and all of them are popular in protocol testing [22, 37, 38]. However, it is challenging to deploy generation-based fuzzing when the data model is not available, since writing data model is laborious, time-consuming, and error-prone. While *mutation-based fuzzing*

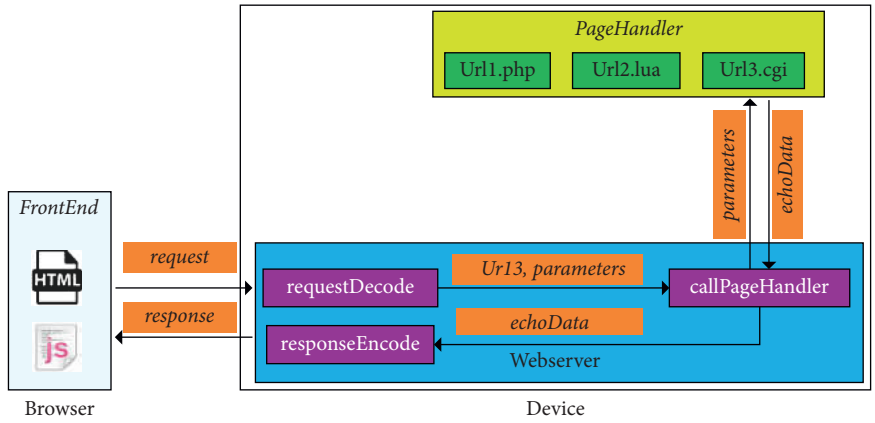

Figure 2: Web management interface.

performs a small mutation on a set of well-formed seeds and then feeds the targeted program with the mutated seeds to monitor unexpected behaviors. For testing COTS devices that do not have documents about its web management interface, it is unpractical to write the data model for every targeted device. In contrast, collecting a set of seeds for every device to start mutation-based fuzzing is much easier.

Based on the complexity, there are three major categories of mutation-based fuzzing [39]: *blackbox fuzzing* just feeds the target with mutated data and monitors the program status in the goal of detecting abnormal behaviors; *greybox fuzzing* employs some feedback information (e.g., code coverage) to improve efficiency; and *whitebox fuzzing* usually performs heavy program analysis (e.g., symbolic execution) to retrieve more internal status to guide the fuzzing. *Whitebox fuzzing* has a well-known drawback of scalability about targeted program [40], so it is challenging for testing real-world programs. *Greybox fuzzing* usually needs to make modification on the original program (source code or binary code) [41], and it cannot work for those programs that cannot be modified. For the web management interface of COTS IoT devices, the *Webserver* and *PageHandler* are embedded inside the device. This constraint makes *whitebox fuzzing* and *greybox fuzzing* to not work. In contrast, *blackbox fuzzing* is independent of program size and it does not require program modification. So, it is practical to discover vulnerabilities in the web management interface by blackbox fuzzing.

3. WMIFuzzer

In this section, we firstly introduce the challenges in WMIFuzzer via an example in Section 3.1, secondly we present the system architecture in Section 3.2, thirdly we present the detailed design about seed generation in Section 3.3 followed by the WMPT for representation of the highly structured message in Section 3.4, and finally we present our fuzzing upon the WMPT in Section 3.5.

3.1. Challenges in WMIFuzzer

3.1.1. An Example. To better understand the problem, we use an enterprise gateway as the example. This device has a web management interface for administration, and its firmware is available publicly. However, the firmware may be encrypted with a private key or be compressed with a private format since it cannot be unpacked via existing tools. So, it is unable to discover vulnerabilities in this device via firmware analysis.

As stated previously, the *FrontEnd* of web management interface is run as a set of GUIs in a browser. Figure 3 illustrates one GUI designed for configuring the *TCP/UDP session limitations* and the corresponding message generated by this GUI. As can be seen from it, this message is highly structured. It contains two parts: *header* and *contents*, and both of them are following the HTTP protocol specification [42]. In the *header* part, *conn_limit.cgi* represents the targeted *PageHandler*. If the *Webserver* makes an incorrect assumption about the length of *PageHandler*, a memory overflow flaw may be triggered by mutating it with a long string. In the *contents* part, *COMN_LIMIT_NUM = 1000* represents the parameter about *TCP session limitation*. If *PageHandler* assumes that it is a number, an unknown bug may be triggered by mutating it with *COMN_LIMIT_NUM = ABCD*. To discover vulnerabilities in this device automatically, one possible approach is mutating every message generated by all GUIs and sending them to the targeted device. However, two challenges are required to be solved in order to launch this mutation-based fuzzing.

3.1.2. Challenge 1: Generation of Initial Message Seeds. Mutation-based fuzzing works upon some well-formed seeds [23], or it will generate a lot of invalid test cases to make the fuzzing inefficient. For fuzzing traditional file format processing software (e.g., BMPViewers), a BMP file can be used to test different BMPViewers since all BMPViewers follow the same format specification. In contrast, an existing web management interface message cannot be used for testing device devices as there is no unique specification for the implementation this interface. For example, the field named *CONN_LIMIT_NUM* in Figure 3 is specific for this example device, and it will be invalid for another device. So, an automatic approach is needed for generating the initial valid message seeds when testing different COTS IoT devices.

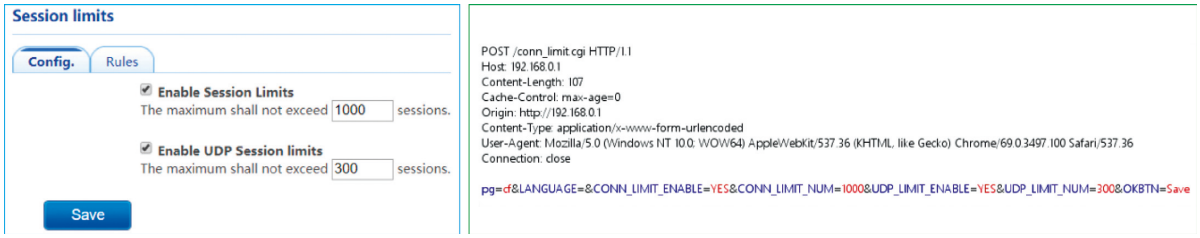

Figure 3: Example of web management interface GUI and message.

3.1.3. Challenge 2: Mutation of Highly Structured Message. Traditional mutation-based fuzzing mainly focuses on unstructured data, most fields of which are at fixed location and have a fixed length. For example, the field about the BMP file length is located in offset 2 with 4 bytes, and its value can be from 0 to max integer number. No matter how the byte-level random mutation changes this field, all BMPViewers still treat this 4 bytes field as an integer number and parse it. In contrast, most fields in the highly structured web management interface messages are not at fixed location and their length is also not fixed. Byte-level mutation on these messages usually corrupts their structure to cause mutated messages being rejected by the device at an early stage. For example, the *POST* field in Figure 3 is the HTTP method, and mutating its byte *S* to the blank ASCII char will cause the remaining *ST* to be a new field named URL followed by the old URL. The *Webserver* will discard this mutated message to stop more deep procedure since it contains two URL fields. Therefore, an approach is required for handling the mutation of highly structured messages.

3.1.4. Solutions. Fortunately, this paper obtained the following approaches to address the above challenges:

(i) *Generating Message Seeds via UI Automation.* UI automation [43] is an automatic method to emulate user operation such as inputting data to UI elements and clicking buttons to submit the inputting. Observed on this, UI automation can be applied to input data and click a button to drive the *FrontEnd* to generate a web management interface message. Since the network of a browser can be intercepted, the generated message can be captured as a seed. By trying all GUIs via UI automation, sufficient messages can be generated as the fuzzing seeds.

(ii) *Mutating a Message Seed Based on Its WMPT.* Abstract grammar tree (AST) [44] is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural, content-related details. Inspired by the AST and HTTP protocol, dissecting a message to a tree and modifying the content of a node can keep the structural information and perform mutation simultaneously. In this way, the weighted message parse tree (WMPT) is proposed to perform the mutation to generate mostly structure-valid messages. This will bring the blackbox fuzzing the ability to detect deep vulnerabilities in IoT devices, such as the *PageHandler*.

3.2. System Architecture of WMIFuzzer. The architecture of our WMIFuzzer is illustrated in Figure 4. At a high level, there are 3 parts: seed generation, WMPT parsing, and fuzzing upon WMPT. Their goals are presented as follows:

(i) *Seed Generation*. This is the first part that utilizes UI automation to generate seed messages as much as it can. In detail, it firstly tries to explore every GUI via a browser and then triggers GUI events to drive the browser generating original messages. A HTTP proxy is deployed to intercept these messages sent by the browser and save them as the initial seeds. All of these steps are fully automatic.

(ii) *WMPT Parsing*. This part is converting an original message to its WMPT detailed in Section 3.4. The goal of the WMPT is to store the message structure into a tree and store the content of message fields into the tree nodes. In this way, mutation on the nodes of a WMPT will mostly generate structure-valid messages.

(iii) *Fuzzing upon WMPT*. The last part is performing fuzzing schedule on the WMPT generated previously. It performs random mutation to generate structure-valid messages and sends them to the device under testing. To infer whether a vulnerability is triggered, several monitoring strategies are deployed to monitor the device running status.

3.3. Seed Generation via UI Automation. As stated previously, WMIFuzzer is built on mutation-based fuzzing that needs a set of valid seeds to start the fuzzing. This section describes how to generate valid messages as the initial seeds by UI automation.

Definition 1. Traditional UI automation widely used by software developers is a 5-tuple $T_{ui} = (I, S, O, E, M)$, where

(i) I is the set of GUI elements used to receive user inputs in which $\alpha \in I$ is called input control (IC)

(ii) S is the set of GUI elements used to receive user clicks in which $\beta \in S$ is called submission control (SC)

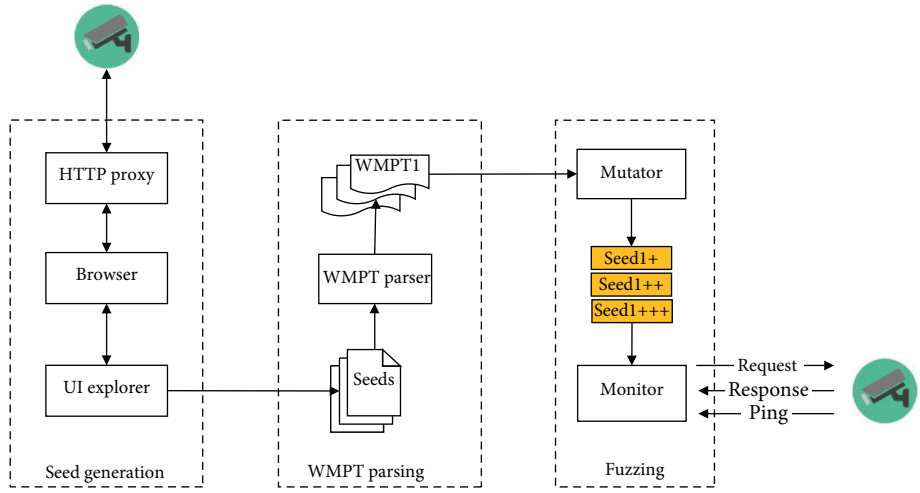

Figure 4: Overview of WMIFuzzer.

(iii) O is the set of GUI elements that cannot receive user inputs and click in which $\delta \in O$ is called other control (OC)

(iv) E is the set of all GUI elements in which $E = I \cup S \cup O$, $I \cap S = \phi$, $I \cap O = \phi$, and $S \cap O = \phi$

(v) M is the set of all mappings: $\{M_j \mid I_j \longrightarrow \gamma_j\}$ in which $I_j \subseteq I$ and $\gamma_j \in S$

The procedure of UI automation is enumerating every $\gamma_j \in S$, inputting data to I_j, and clicking γ_j. Figure 5 is one example web management interface GUI that has two SCs named *Save* and *Check again*. This GUI has two mappings: (1) the *Save* is mapped to fours ICs named *Enable, Disable, 02,* and *00* and (2) the *Check again* is mapped to none of these ICs. With these mappings, UI automation firstly inputs four values to these four ICs and then clicks the *Save* or directly clicks the *Check again* without inputting.

In practice, the mapping M in Definition 1 is generated by software testing engineers by hand. However, an automatic approach for building the M is required since WMIFuzzer is designed to fuzz different IoT devices automatically. A previous work [45] that utilizes UI automation to detect IoT account vulnerabilities builds this mapping by the observation that many IoT control apps share the same style for the account management GUI, and most of these apps use the default ICs and SCs. In contrast, the GUI of web management interface in IoT devices is diverse, as different types of devices are usually designed for different purposes. Even for the same type of devices, most vendors prefer a new GUI style to make themselves different from other vendors. So, WMIFuzzer cannot depend on the GUI style, and it has to analyze the codes of *FrontEnd* to identify I, S, and M.

In the development domain of *FrontEnd*, the browser renders the GUI based on three parts: HTML codes, JavaScript codes, and CSS codes. HTML is the standard markup language for creating web pages, and HTML elements are the building blocks of a web page. The JavaScript codes process the business logic (e.g., submitting user inputs), while the CSS codes describe the style of E. This means that every $\kappa \in E$ is declared as an HTML element in the HTML code. According to the HTML specification [46], different types of HTML elements have different purposes: *input* and *textarea* elements are defined as ICs and *input-with-submit* and *button* elements are defined as SCs. Therefore, WMIFuzzer should identify I and S from all elements in the HTML code. Unfortunately, it usually fails because of the flexibility of HTML elements. In Pseudocode 1 we present the core HTML codes of the GUI presented in Figure 5.

From the above codes, we can get the following information:

(i) *Check again* can be identified as a SC since it is a button element in code line 3.

(ii) *Enable* and *Disable* cannot be identified as ICs since they are *img* elements in code lines 5-6, but the *FrontEnd* utilizes two *img* elements with boolean-style value attribute to work as a checkbox-style IC. So, these two *img* elements are not ICs, but they have the same functionality of ICs.

(iii) *02* and *00* cannot be identified as ICs because they have the read-only attribute in code lines 9–12, but their parent element has a click event that can dynamically modify the read-only attribute.

(iv) Lines 15-16 contain two useless ICs that have hidden attributes to make themselves invisible in the GUI.

(v) *Save* also cannot be identified as a SC since it is a *div* element in code line 17, but this element has been binded with a click event handler in code line 19. The event handler collects data from two read-only elements and saves data to a hidden element hosted in a form in code lines 20-21. At last, the event handler triggers the form submitting a request in code line 22. So, this *div* element has the same functionality of a SC.

In short, every HTML element can be an IC or a SC because it can use value attribute and event handler to receive user inputs and button clicks. This is the flexibility of

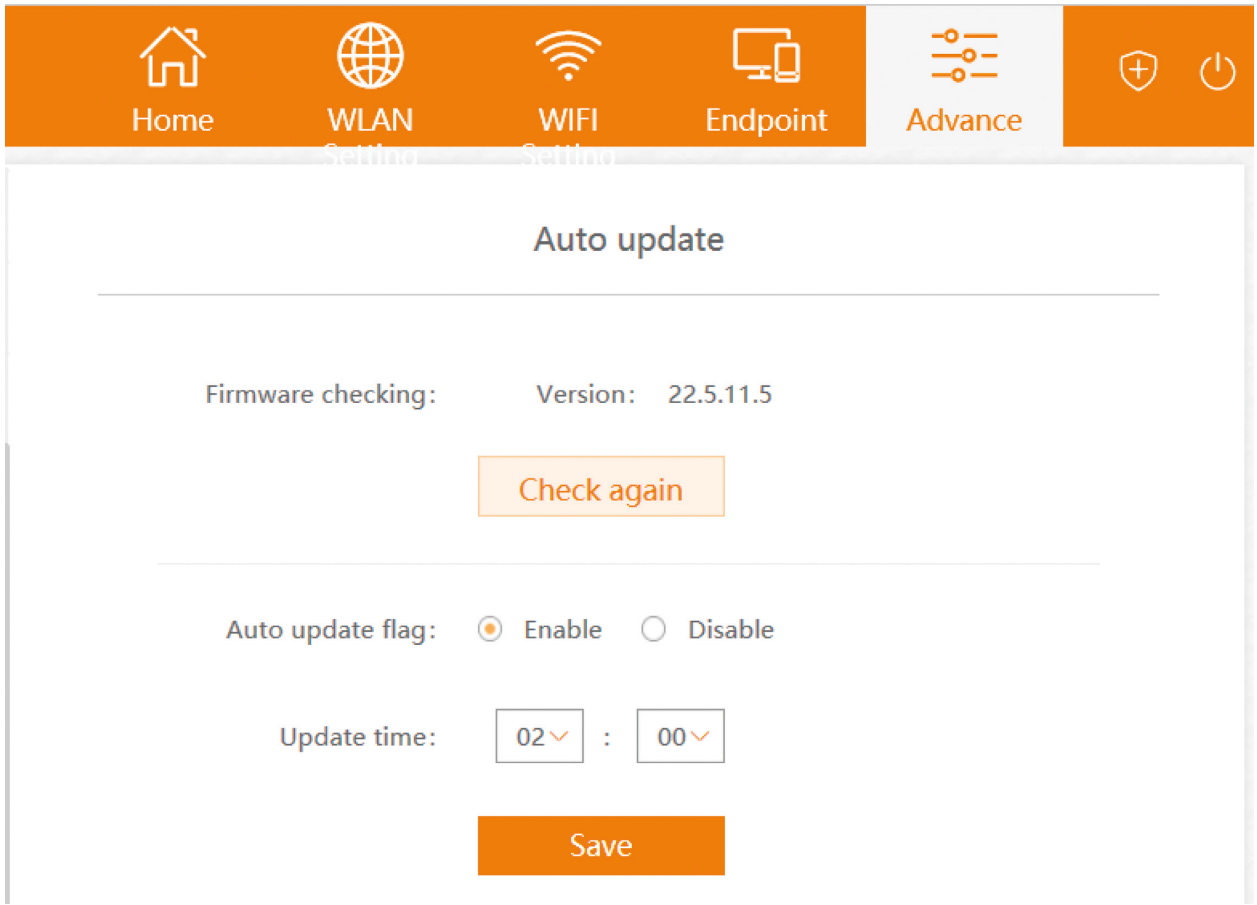

Figure 5: GUI of the AutoUpate.

```
(1) <div><span>Auto update</span>
(2) <div><span>Firmware checking: Version: 22.5.11.5</span>
(3) <button id = "recheck_btn">Check again</button></div>
(4) <div><span>Auto update flag: </span>
(5) <img src = "/radio_on.png" value = "1"><span>Enable</span>
(6) <img src = "/radio_of f.png" value = "0"><span>Disable</span>
(7) <span>Update time: </span>
(8) <div onclick = "dropdown_click (this)" type = "button">
(9) <input id = "uptimehour" Type = "text" read only = "read only"
(10) value = "02"></div>
(11) <div onclick = "dropdown_click (this)" type = "button">
(12) <input id = "uptimemin" Type = "text" read only = "read only"
(13) value = "00"></div>
(14) <form action = "/autoupgrade/save" method = "post">
(15) <input type = "hidden" value = "1">
(16) <input name = "autoUpTime" type = "hidden" value = "·"></form>
(17) <div id = "AutoUpSave" type = "button">Save</div>
(18) </div></div>
(19) $("#AutoUpSave").click (function (·){
(20)     $ ("#autoUpTime").val ($ ("#uptimehour").val (·)+":" +
(21)     $ ("#uptimemin").val (·));
(22) setTime out (function (·) {document.save.submit (·);}, 1000);
(23) });
```

Pseudocode 1: AutoUpdate.html.

web technology, and it makes identifying the ICs and SCs difficult. Program analysis (e.g., symbolic execution and taint tracking) can be utilized to explore whether a HTML element can collect data or submit data to help identify the ICs, SCs, and their mappings, but their known limitations make them inefficient for the web interface.

As it is a challenge to identify ICs, SCs, and their mappings, a brute-force UI automation that makes a trade-off between the accuracy and the practice is proposed.

Definition 2. Brute-force UI automation is a 2-tuple $T_{\text{ui}} = (E, M)$, where

(i) E is the set of all GUI elements in which $\alpha \in E$ is an IC and a SC at the same time

(ii) M is the set of all mappings: $\{M_j \mid E_j \longrightarrow \gamma_j\}$ in which $E_j = E - \gamma_j$ and $\gamma_j \in E$

This means (1) every element is an SC, (2) every element is an IC, (3) and every SC is mapped to all ICs. So, M is very simple, and its size is a constant that matches the element count of E. It seems that this brute-force UI automation works well at most times, since the event handler of most SC ignores those ICs that are not really mapped to this SC. In this way, WMIFuzzer can use UI automation to drive the *FrontEnd* to generate messages automatically by the following steps:

(i) *Initialization*. In this step, the UI automation firstly starts a browser, which is run under control, to open a welcome page that guides the user to input the IP address of the targeted IoT device. Then, the browser opens the login page of the web management interface to guide the user inputting his credentials to login to the interface. This is the only step that requires a little of manual efforts because this interface usually denies accesses without successful login. On the contrary, this is also the standard step for managing an IoT device via web management interface. The whole step does not require any knowledge about software testing and vulnerability discovering.

(ii) *Crawling*. The web management interface is defined and implemented by the device vendor, so there are usually not any documents about how many web GUIs are there in a device. For a customer, he/she accesses the root GUI via the IP address of the device and then switches to other GUIs by exploring the root GUI. As all GUIs are accessed by URLs, WMIFuzzer performs crawling from the root GUI to get all GUI-related URLs. A GUI may (1) not send a network message as it is a static GUI, (2) or send one network message as it is a simple GUI, (3) or send multiple network messages as it is a complex GUI with multiple business logics. So, heuristic strategies are deployed to exclude mostly useless URLs: (1) URLs that do not start with the IP address of the IoT device since the messages with these URLs will not be sent to the device and (2) URLs that have a filename extension about known names (e.g., png, ico, bmp, jpg, CSS, xml, and js) since they are not GUIs but pure static resources.

(iii) *Inputting and Clicking*. For every URL collected in the crawling step, firstly its HTML codes are analyzed to identify all elements, then an initial value is brute-force assigned to every identified element, and finally an element is chosen to be clicked by injecting a brute-force click. WMIFuzzer chooses the click element by the elements' sequential orders in the HTML codes, and the values of all elements are reset every time before WMIFuzzer chooses the click element. Moreover, all HTML elements of a URL are just identified once and the result is saved. This feature enables directly locating all elements and inputting values to them when this URL is reloaded.

(iv) *Capturing Messages*. Previous clicking may trigger the browser generating a network message sent to the device, and the message can be built in the JavaScript codes (e.g., posting data via Ajax) or in the standard form submission. To capture these messages, API hooking [47] can be utilized to intercept the network-related APIs in the browser, but API hooking is not a robust approach, so another approach named *browser network proxy* that is supported by most browsers natively is utilized in WMIFuzzer. By installing a proxy and configuring the browser with this proxy, all messages will be transparently sent to the proxy instead of the real device.

In a word, WMIFuzzer overcomes the challenge of seed generation by utilizing the brute-force UI automation.

3.4. Weighted Message Parse Tree. Traditional mutation-based fuzzing works by randomly selecting a set of bytes in a seed, modifying them and feeding the target with this mutated seed. However, it cannot work well for the web management interface messages that are highly structured, as pure byte-level random mutation mostly generates structure-invalid messages to make the fuzzing inefficient. Inspired by the abstract syntax tree [44, 48], WMIFuzzer applies the WMPT to represent the seed message and perform random mutation on the WMPT.

A WMPT is an ordered rooted tree with typed nodes where each node has a weight scope, and this tree has two types of nodes: leaf nodes and internal nodes. A leaf node contains an atomic part of message content while an internal node concatenates its child nodes. The weight of a node means how much time should be assigned to the mutation of this node. Figure 6(a) is an example message about web management interface and Figure 6(b) is the corresponding WMPT. All green nodes are the leaf nodes that store the atomic content of a seed message, and others are internal nodes that store the structure of their child nodes. Obviously, the mutation of a leaf node will not change the structure of the WMPT at most times. Moreover, inserting or

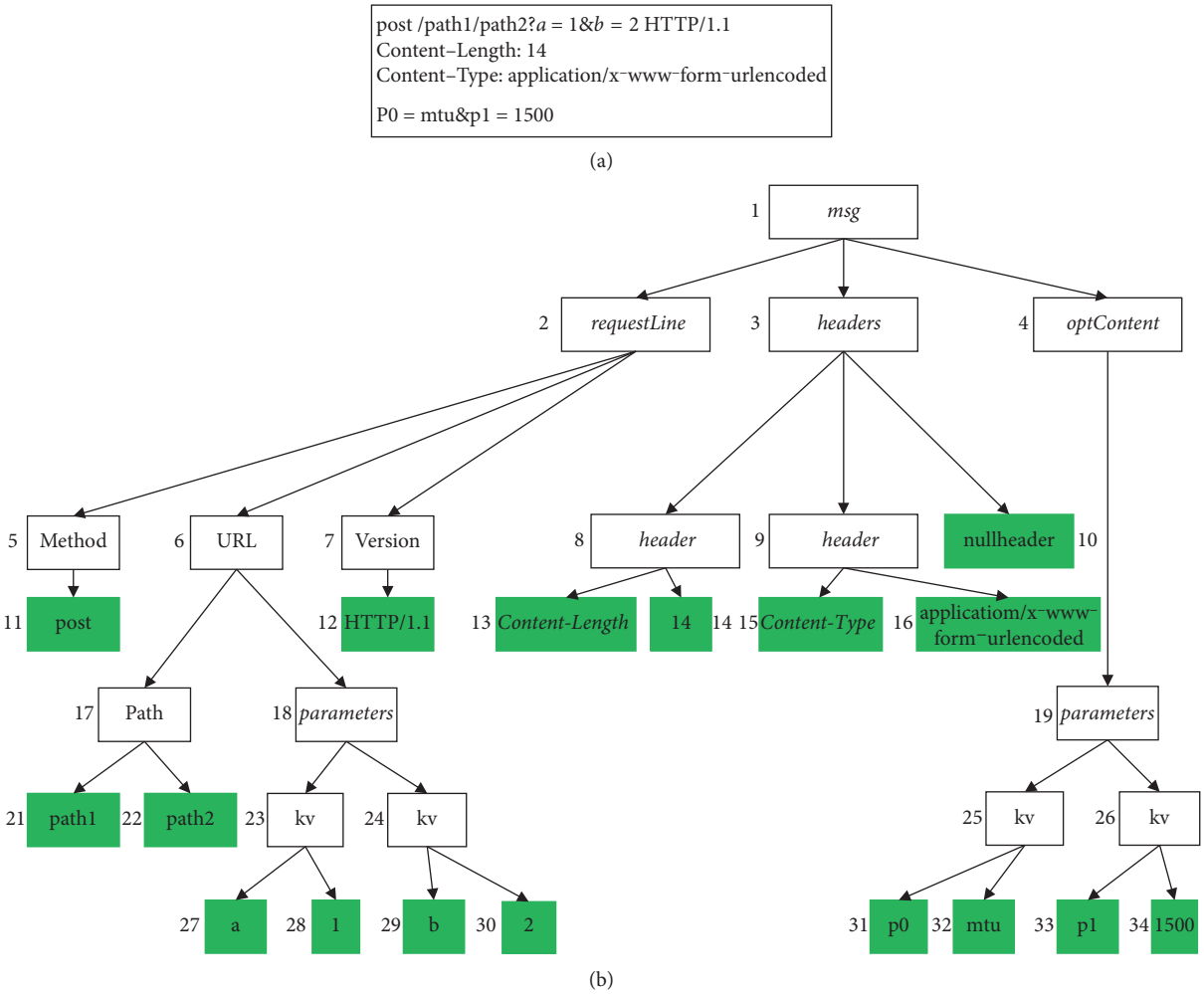

Figure 6: Weighted message parse tree. (a) A sample message. (b) The WMPT of the sample message.

deleting some internal nodes can generate mutated messages that have a big difference from the original seeds. For a mutated tree, collecting the content of its nodes and contacting them will output a message.

Based on the mutation of leaf nodes and internal nodes in WMPT, WMIFuzzer can generate both structure-valid and structure-invalid messages on demand. So, it is capable to detect shallow and deep vulnerabilities. There are three operations on the WMPT: *deserialization* is converting a seed message to be a WPMT and *mutation* is modifying the WMPT while *serialization* is converting the mutated WMPT to a new message.

3.4.1. Deserialization Operation. As the web management interface messages are built on HTTP protocol, WMIFuzzer follows the protocol specification to dissect a seed to its WMPT. Algorithm 1 details the procedure of seed deserialization. For every seed message $\alpha \in M$, WMIFuzzer needs to perform deserialization for converting it to a WMPT $t \in T$. According to the HTTP protocol specification [46], the web interface message is an ACSII string composed of 3 parts: *requestLine*, *headers*, *and optContent*. The *requestLine* is the first line of the message, the *headers* are the following lines that end with a special empty line, and the *optContent* is the whole remaining data determined by some special line in the *headers*. So, Algorithm 1 firstly splits a message into these three parts (nodes 2, 3, and 4 in Figure 6) since this is the mandatory constraint about HTTP message.

The *requestLine* contains three parts: a request *method* that is a predefined ASCII string (e.g., GET, POST, and so on), a targeted URL that is composed of *pagePath* and optional *parameters*, and a predefined protocol *version* (nodes 5, 6, and 7 in Figure 6). The two parts of a target URL are concatenated via the symbol "?" (nodes 17 and 18 in Figure 6). The *pagePath* is separated by the symbol "/" (nodes 21 and 22 in Figure 6), while the *parameters* are encoded as k-v blocks concatenated via "&" (nodes 23 and 24 in Figure 6) and k-v is concatenated via the symbol "=" (nodes 27 and 28 in Figure 6). In this way, Algorithm 1 generates the *requestNode* and inserts it to the WMPT *t*. Since the targeted URL is the most customized part, a double weight scope was assigned on its child nodes to perform more mutation.

The *hea de rs* are a set of header lines, that is, a k-v style string, and the separator is a colon. There is no limitation of

Input: the set of seed messages, M;
Output: the set of generated WMPT, T;
(1) initial $T = \varnothing$
(2) **for** each $\alpha \in M$ **do**
(3) initialize an empty WMPT $t = \varnothing$
(4) split α to 3 parts {*requestLine*, *headers*, *optContent*}
(5) generate *requestNode* from *requestLine* and insert it to t
(6) initialize an empty *headersNode* = $\varnothing$
(7) **for** echo $\beta \in$ *headers* **do**
(8) generate header from β and insert it to *headersNode*
(9) **end for**
(10) insert *headersNode* to t
(11) if *optContent* is not empty **then**
(12) generate *contentNode* from *optContent* and insert it to t
(13) **end if**
(14) **end for**
(15) **return** T

Algorithm 1: Seed dissection algorithm.

the quantity of header lines, and the HTTP protocol applies an empty line as the last header line. There are some standard header lines with known keys generated by the browser, such as *Content-Length, Content-Type, Cookie, User-Agent*, while other customer header lines are generated by the *FrontEnd*. To parse each line, we also follow the k-v style to decode the line. Because the data format of the value content in a k-v header line is not known, heuristic strategies are applied to parse the content into basic elements by scanning special chars including blank, comma, semicolon, and "&". In this way, Algorithm 1 generates the *headersNode* and inserts it to the WMPT t.

The *optContent* is an optional part, and there is no limitation of its content. The content length is declared in a well-known header named *Content-Length* in *headers*, and it is strongly recommended to declare the content type in another well-known header named *Content-Type*. In the web development domain, there are 3 popular types: application/x-www-form-urlencoded, multipart/form-data, and application/json. For these three types, WMIFuzzer deserializes the content following their specifications, while for other types, the content is deserialized as a bytes stream. With this, Algorithm 1 generates the *contentNode* and inserts it to the WMPT t. Finally, a message is dissected to its WMPT.

3.4.2. Mutation Operation. As stated previously, the mutation is performed on the WMPT nodes to generate mostly structure-valid messages. For a leaf node, WMIFuzzer mutates its content with random data directly. Since the HTTP message is built on text string that ends with a NUL character (NUL has all bits zero), NUL is imported into a mutated message before the tail will make the message to be truncated. For example, node 7 in Figure 6 is the protocol *version*, and the *headers* and *optContent* will be truncated by the *Webserver* in IoT devices when a char of this node content is mutated to NUL. So, NUL is filtered when performing mutation. In detail, when mutation will change a byte to NUL, WMIFuzzer dynamically generates a random number, that is, 1100 and checks whether it is bigger than a threshold (now is 97). If the result is TRUE, WMIFuzzer accepts this mutation, or it just ignores this mutation. For an internal node, WMIFuzzer deletes the node or deletes a child node randomly to the mutation on a leaf node, this mutation will bring a big difference.

3.4.3. Serialization Operation. Serialization is converting a WPMT to its corresponding message, and it is very straightforward: traversing the WPMT depth-first and contacting the content of every leaf node. The contacting symbol is determined by the parent node of every node. Especially, the serialization will keep the parent node symbol when a leaf node is mutated to an empty value. For example, node 30 in Figure 6 is a parameter value in URL, and the serialization of its parent (node 24) will be "b=": In contrast, serialization will not keep the parent node symbol when a leaf node is mutated to be deleted. For the previous example, the serialization of node 24 will be *b*.

3.5. Fuzzing upon WMPTs. Based on the generated WMPTs, the *Mutator* in Figure 4 can be scheduled to generate structure-valid mutated messages. Then, these mutated messages are sent to IoT devices in the goal of triggering vulnerabilities detected by the *Monitor*.

3.5.1. Fuzzing Scheduling. There are multiple nodes in a WMPT, and it is unknown that which node with mutated content can trigger a vulnerability. So, scheduling as shown in Algorithm 2 is proposed to perform the fuzzing. There are two phases of fuzzing: determined phase and random phase.

The determined phase is performing limited mutation on a single leaf node to trigger vulnerabilities as fast as it can. Every seed message $\alpha \in M$ will be processed once in this phase. For a leaf node $\beta \in \alpha$, *sequentia_mutating* tries following modifications including

```
Input: the set of generated WMPT, M;
(1)  for each α ∈ M do
(2)    for each node β ∈ α do
(3)      if β is a leaf node then
(4)        tm = sequentia_mutating(β)
(5)        result = sending_monitoring(tm)
(6)        if interesting(result) then
(7)          alert(tm)
(8)        end if
(9)      end if
(10)   end for
(11) end for
(12) loop
(13)   for each α ∈ M do
(14)     indexList = randomIndexList(0, len(α))
(15)     for each λ ∈ indexList do
(16)       tm = random_mutating(λ)
(17)     end for
(18)     result = sending_monitoring(tm)
(19)     if interesting(result) then
(20)       alert(tm)
(21)     end if
(22)   end for
(23) end loop
```

Algorithm 2: WMIFuzzer fuzz scheduling algorithm.

(i) Extending content in the goal of triggering vulnerabilities about buffer overflow. WMIFuzzer just appends a number of printable ASCII chars to the content for simplification. Given the limitation of memory resource in IoT devices, the length of extended content is configured to classic values: 2^i, where $0 \le i \le 12$.

(ii) Emptying the content in the goal to trigger vulnerabilities about assumption of nonempty content. For example, the serialization of the URL parameters will be $a =$ and $b = 2$ when node 28 in Figure 6 is mutated to empty content. As the content owned by the key a is empty now, an implementation flaw may be triggered if the *Webserver* or the *PageHandler* makes an incorrect assumption.

(iii) Replacing the numerical-style content with typical integer number in the goal to trigger vulnerabilities about integer overflow. Because the HTTP message is text based and there is no standard approach to identify whether the node content is a string or a number, WMIFuzzer just applies regular express to match whether the content looks like an integer number. If the matching result is FALSE, it will be treated as a string, or the content will be replaced with classic boundary integer numbers: 2^i, $2^i - 1$, and $2^i + 1$, where $0 \le i \le 32$.

(iv) Replacing the string-style content with special strings in the goal to trigger various vulnerabilities. FuzzDB [49] is an open-source project that collects a lot of popular strings used in multiple security tools, and WMIFuzzer also uses its string database to replace string-style content.

(v) Changing the content type in the goal to trigger vulnerabilities about assumptions on the data type. For the node that matches an integer number, it will be replaced with an ASCII string, or the node content is treated as a string and will replaced with an integer number. For example, node 28 in Figure 6 is an integer string, and the *Webserver* will convert it to an integer number. However, the conversion may fail when the integer string is mutated to a nonnumerical string, and a crash may occur when the *Webserver* does not check the result of conversion.

In contrast, the random phase is an endless loop that performs random mutation on all seeds repeatedly. When fuzzing a WMPT, only a subset of its nodes will be mutated at a time. This is because the generated message will be useless and rejected at an early stage at most times if most nodes are mutated. The *randomIndexList* in Algorithm 2 is firstly used to generate a list of index values that identify which nodes should be mutated in this iteration. And the *random_mutating* is used to perform mutation as shown in following:

(i) For a leaf node, both the modifications in the determined phase and random byte-level are fully supported. In contrast to the determined phase, the major difference is that every mutated message in this phase may contain modification on multiple nodes.

(ii) For an internal node, deleting itself and deleting a child node are both supported. As there are multiple mutations on different nodes via *indexList*, deleting a node from the tree will make the *indexList* invalid and the remaining mutations may trigger a crash. Inspired by the *copy-on-write* [50] that is popular in modern operation system (e.g., Windows and Linux), WMIFuzzer does not really remove the node from the tree, and it just marks the node 'deleted' to notify the serialization of WMPT ignoring this node.

The messages generated in these two phases will be sent to the IoT devices under testing. If an interesting event in the following *Remote monitoring* is detected, an alert will be generated with the current mutated message to help further manual inspection.

3.5.2. Remote monitoring. Monitoring running status is very important because it is the only way to infer whether a vulnerability is triggered in a COTS device. However, it is challenging to monitor the running status since these devices usually do not support local monitoring to make a lot of information unavailable (e.g., running processes, threads, CPU overhead, and memory statistics). So, the main surface for device monitoring is the network communication. If a vulnerability causes the device to crash, the device network will be unavailable and this can be detected in the outside of the device. But not every vulnerability will cause the system crash, such as the *Command Injection* and *Interface Leak*. If *Command Injection* executes a network-related system command in the device, the network status of device will also

be infected and this can be monitored outside. So, WMIFuzzer uses two commands, *reboot* and *ping* to infect the device network; the former can make the device to go offline and the latter can send *ICMP* message. The root cause of *Interface Leak* is incorrect authentication, and WMIFuzzer uses a simple strategy to monitor this type of vulnerabilities. If a mutated message contains modification in the authentication data or in the URL but the response status is successful, a leak vulnerability is triggered. Given the device has two phases to consume the mutated messages, *Webserver* and *PageHandler*, we analyze the potential network status when a flaw is triggered in each phase:

(i) *Webserver Phase*. If a message just crashes the *Webserver*, there is no response message after the mutated message is sent to the device, and the *Webserver* may become unavailable if it is running in a single-process model while the device low-layer network is available. Moreover, if the crash also infects the system at the same time, the device may reboot due to the following reasons: 1) no response , 2) Webserver unavailable, and 3) low-layer network unavailable. Otherwise, the *Webserver* will reply a response message, and both the low-layer network and *Webserver* are reachable.

(ii) *PageHandler Phase*. If a message crashes the *PageHandler*, the fuzzer will not receive the response message while the *Webserver* is available. Moreover, if the crash also infects the system, the network status is the same as the crash in the *Webserver* and they cannot be distinguished from each other.

In a word, WMIFuzzer monitors the status of the low-layer network, *Webserver*, and *PageHandler* to detect multiple types of vulnerability including crash, *Interface Leak*, and *Command Injection*.

4. Implementation and Evaluation

We present the prototype implementation of WMIFuzzer in Section 4.1 and the evaluation in Section 4.2 and then briefly study some vulnerabilities found by WMIFuzzer in Section 4.3.

4.1. Implementation of WMIFuzzer. WMIFuzzer was implemented with around 3,000 Python lines of code and 2,000 C lines of code in total. Also, several open-source projects (e.g., Chrome, Selenium, mitmproxy, and fuzzdb [49, 51, 52]) are integrated into this fuzzer to avoid reinventing the wheel.

In the seed generation phase, the brute-force UI automation was built based on Chrome and its Selenium driver. Python code was written to use the Selenium driver to control the Chrome behavior, such as opening a URL, inputting data, and clicking a button. The mitmproxy project, a HTTP proxy written in Python code, was extended to intercept the web management interface messages.

In the WMPT parsing phase, C code was written to implement the deserialization, mutation, and serialization. HTTP protocol specification was used to parse the URL, headers, and content. To reduce redundant parsing of the content type of a node, the data type is calculated once when converting a message to its WMPT and saved into the node. Inspired by the copy-on-write strategy, the mark-deleted flag was applied on the WMPT nodes to reduce unnecessary memory operations. The C code was compiled to a dynamic library that can be loaded by Python code.

In the Fuzzing phase, Python code was written to schedule the fuzzing. After a mutated message has been sent to the device, the reachability of *Webserver*, *PageHander*, and the low-layer network is checked to detect crashes and command injection in the targeted device. In addition, the response message was analyzed to detect interface leak.

4.2. Evaluation of WMIFuzzer. In order to evaluate the effectiveness and efficiency of WMIFuzzer, we tested it on 7 IoT devices and compared it with two state-of-the-art fuzzers: American Fuzzy Lop (AFL) [23] and Sulley [22].

4.2.1. Testing Devices. We bought 7 popular IoT devices, and all of them contain the web management interface. After manually checking their firmware images, 4 of them have been released publicly while others were not. For the public firmware images, popular tools were deployed to unpack them: Binwalk [14] and BAT [15]. And just one of these images can be unpacked, but it fails on being deployed in an emulator presented in a previous work [19]. The detailed specifications of these devices are described in Table 1, where FirmwareAva means firmware is released publicly, FirmwareDec means firmware can be unpacked, and FirmwareRun means firmware can be run in an emulator environment.

4.2.2. Testing Environment. WMIFuzzer and the other two fuzzers are run in a separate virtual machine that hosts an Ubuntu 16.04 with Intel Core i7 quad-core 3.6 GHz CPU with 4G RAM. The original branch of AFL cannot work on the web management interface because it works by testing targeted program locally. So, its code was patched to create two versions: AFL^1 and AFL^2. AFL^1 replaced its code about target program execution with sending mutated data and waiting for a response. AFL^2 extended the AFL^1 with remote monitoring strategies used in WMIFuzzer. However, the fuzzing schedule of AFL was never changed. Sulley is another state-of-the-art fuzzer that is popular for protocol fuzzing, but it needs a data model to start fuzzing. As different seeds have different fields, every seed needs a data model in Sulley. Since it is laborious, time-consuming, and error-prone to write data models by manual analysis of every seed, a data model generation that treats every message as a set of tokens separated by CR-LR or blank was used in this experimentation. In detail, parsing every seed to tokens once can generate its data model that can be integrated into the $Sulley^1$. $Sulley^2$ extended $Sulley^1$ with remote monitoring strategies used in WMIFuzzer. Anyway, the seed messages generated by WMIFUzzer are used as the initial seeds for AFL and Sulley.

Table 1: Summary of devices under testing.

Type	Vendor	Device	FirmwareAva	FirmwareDec	FirmwareRun
SOHO router	Phicomm	K2-A6	Yes	Yes	No
	JieXi	AC836M	Yes	No	No
Enterprise gateway	FeiYuXing	VE602W+	Yes	No	No
	RuiJie	NBR1300G	Yes	No	No
IP camera	RIWYTH	RW-950S	No	No	No
	NEO	NIP-25SY	No	No	No
	ZTE	C520P	Yes	Yes	No

4.2.3. Research Questions. Using the previous experiment setup, we aim to answer the following research questions:

(i) *RQ1*. Can WMIFuzzer discover vulnerabilities in COTS IoT devices based on its seed generation and fuzzing of WMPT?

(ii) *RQ2*. Can WMIFuzzer outperform the state-of-the-art fuzzers by performing mutation-based fuzzing on the same initial seeds?

4.2.4. Effectiveness of Vulnerability Detection (RQ1). Table 2 lists the unique vulnerabilities found by WMIFuzzer. For each device under testing, WMIFuzzer firstly applies the UI automation in Section 3.3 to generate seed messages in 1 hour automatically. Then, it converts the seed messages to WMPTs and starts mutation-based fuzzing in 23 hours. At last, it discovered 10 vulnerabilities: 3 command injections, 4 interface leaks, and 3 crashes.

All of them have been reported to the CNCERT/CC [21] that cooperates with many vendors to fix vulnerabilities in their products, and 6 CNVD-IDs are assigned as they are zero-days. The two crashes (Vul-ID-003 and Vul-ID-009) have been reported by other researchers independently before this paper did, so they are not been assigned with a CNVD-ID. The crash (Vul-ID-006) has been reviewed by the vendor, and they inferred that it is not security related, so it was not assigned with a CNVD-ID. Vul-ID-004 and Vul-ID-005 are different vulnerabilities in a same device, but they have been combined to be assigned with one CNVD-ID.

In addition, most of these vulnerabilities are of high impacts because they can be exploited remotely. An attacker can utilize them to compromise the targeted device, such as installing malware into a router for further cybercrimes and rebooting a camera for bypassing surveillance. At the time of this writing, patches have been released to fix these vulnerabilities.

These results indicate that WMIFuzzer can discover vulnerabilities in the web management interface of COTS IoT devices automatically based on its seed generation and mutation-based fuzzing.

4.2.5. Efficiency of Vulnerability Detection (RQ2). We measure the efficiency in terms of vulnerabilities discovered over quantity and time. Table 3 lists the results by comparing WMIFuzzer with AFL and Sulley; it shows that WMIFUzzer can detect more vulnerabilities than the other fuzzers. In detail, WMIFUzzer detected 10 vulnerabilities including *Command Injection, Interface Leak*, *and Crash* while the other two fuzzers just detected a subset. Moreover, AFL and Sulley take more time than WMIFuzzer for detecting the same vulnerability.

We performed a further manual analysis and found the following: (1) *Command Injection and Interface Leak* vulnerabilities cannot be detected by both AFL[1] and Sulley[1] because these two fuzzers can both receive response messages from the devices where no crashes are triggered. However, AFL[2] and Sulley[2] can detect them because of the integration with the remote monitoring strategies used in WMIFuzzer. (2) Sulley[2] can detect more vulnerabilities than AFL[2], and Sulley[2] also takes less time on the same vulnerability. After manual checks, we get the result that byte-level mutation in AFL[2] generated a lot of structure-invalid messages rejected by the device. In contrast, the data model in Sulley[2] can guide the mutation to generate structure-valid messages. Actually, the data model in Sulley[2] can be considered as a simple version of WMPT, where mutation on internal nodes is not permitted, every leaf node has the same weight scope, and every generated message just contains one mutated leaf node.

These results indicate that WMIFuzzer is competitive compared to the state-of-the-art fuzzers: AFL and Sulley, in terms of quantity and time about vulnerability detection in the web management interface of COTS IoT devices.

4.3. Case Study

4.3.1. Feiyuxing Enterprise Gateway. *VE602W+* is a wireless gateway designed for small office and company, and it contains web management interface. The firmware of this device has been released publicly, but it is encrypted or packed with a private format, and it cannot be unpacked by existing tools. Therefore, WMIFuzzer can be used to detect vulnerabilities in this device by blackbox fuzzing. After 30 minutes, WMIFuzzer reported a potential interface leak vulnerability as a message with mutated URL can get a successful response. This mutated URL is "http://192.168.0.1/.htpasswd," and the response content is "admin:$1$$FQzEy2IhIMAL60u9OrHLp1." After manually analyzing the content, it can be decrypted by John the Ripper password cracker [53]. The result is "admin:147258369," which is just the credential of the web management interface. So, anyone who can access the router work can get the password of this device no matter how the administrator changed his/her password. After another 4 hours, WMIFuzzer reported a potential *Command Injection* vulnerability as another message triggered the device sending a *ICMP* request to

Table 2: Summary of discovered vulnerabilities.

Device	Vul-ID	Vulnerability	Remotely exploitable	CNVD-ID	Addition
Phicomm K2-A6	001	Command injection	True	CNVD-2017-25289	Fixed
	002	Interface leak	True	CNVD-2017-20666	Fixed
JieXi AC836M	003	Crash	True	*N*-day	Fixed
FeiYuXing VE602W+	004	Interface leak	True	CNVD-2017-35720	Fixed
	005	Command injection	True		Fixed
RuiJie NBR1300G	006	Crash	False	Just-a-Dos	Fixed
	007	Command injection	True	CNVD-2018-22138	Fixed
RIWYTH RW-950S	008	Interface leak	True	CNVD-2017-37032	Fixed
NEO NIP-25SY	009	Crash	False	*N*-day	Fixed
ZTE C520P	010	Interface leak	True	CNVD-2018-21990	Fixed

Table 3: Statistics on vulnerability detection.

Vulnerability type	Device	WMIFuzzer	AFL[1]	Sulley[1]	AFL[2]	Sulley[2]
Command injection	Phicomm K2-A6	3 h 29 m	NA	NA	19 h 7 m	NA
	FeiYuXing VE602W+	5 h 37 m	NA	NA	NA	11 h 29 m
	RuiJie NBR1300G	7 h 12 m	NA	NA	NA	NA
Interface leak	Phicomm K2-A6	2 h 5 m	NA	NA	NA	40 m
	FeiYuXing VE602W+	3 h 49 m	NA	NA	18 h 52 m	10 h 21 m
	RIWYTH RW-950S	5 h 24 m	NA	NA	NA	13 h 52 m
	ZTE C520P	52 m	NA	NA	21 h 27 m	8 h 3 m
Crash	NEO NIP-25SY	18 m	2h21	25m	5 h 9 m	1 h 22 m
	JieXi AC836M	2 h 11 m	NA	NA	NA	5 h 36 m
	RuiJie NBR1300G	4 h 31 m	NA	NA	NA	NA

the fuzzer. We reviewed the mutated message content in Pseudocode 2 and found the difference that "PING_HOSTIP = 11" has been mutated to "PING_HOSTIP = 11 | ping 192.168.0.11."

Further inspection showed that the module named *System Diagnosis* is responsible for this message. This module executes *ping* command inside the device, and the parameter of *ping* is referenced to the user-supplied *PING_HOSTIP*. It seems that the developer must have forgotten to sanitize this content. So, evil content can execute additional commands.

Then, the *nmap* [54] is deployed to scan this device, and two special open ports are found: 23, that is, the *telnet* service, and 10089, that is, the SSH service. At last, the upper vulnerabilities were used to crack the device system credentials, and we succeeded in logging into the SSH service with the cracked credentials. Moreover, by manually checking the web interface, we found that there is no entry for closing these two services. This means that an attacker can utilize these vulnerabilities to control the gateway device independent of the web interface authentication.

4.3.2. Phicomm Smart Router. K2 is a smart home router that can be managed via a browser or its official app. The firmware of this device is released on the vendor website, Binwalk can unpack it, and its web management interface is implemented by Lua [33] scripts. We could not find a tool that is designed to detect vulnerabilities in Lua scripts, and we failed running the firmware in the emulator [19] because of some hardware features unavailable in the emulator. Then, WMIFuzzer was deployed to perform the blackbox fuzzing, and a crash was reported after 4 hours. Manual analyzing the crash showed that the root cause is the URL about time reboot module. This module is designed for the administrator to configure the router rebooting at a special time automatically, and the main part of the message is listed in Pseudocode 3.

Compared to the original seed message, the additional content is the "—reboot." By reviewing all unpacked Lua scripts, this crash was hosted in the script file that contains the core codes in Pseudocode 4.

This script firstly gets the request parameters named *timeRebootEnablestatus and timeRebootrange*, and then combines them to construct the parameters of *luci.sys.call* without any sanitization. So, malicious parameter in this message can execute extra commands via *luci.sys.call*.

By this *Command Injection* vulnerability, we detected the underlying architecture and found that it contains *Wget* that can download a file from a remote server. Then, *Wget* was applied to download the corresponding *Dropbear*, a popular SSH server for embedded devices, and install it into the device. Finally, we got the full control of this router independent of the web management interface authentication.

4.3.3. NEO IP Camera. In the manual, customers are suggested to manage this device via its official control app while the web management interface is also enabled. The vendor does not release the firmware publicly, and we also found

```
POST/diag.cgi HTTP/1.1
Host: 192.168.0.1
Content-Length: 82
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWeb Kit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/
537.36
Accept: text/html, application/xhtml + xml,application/xml;
q = 0.9,image/webp, image/apng,∗/∗; q = 0.8
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh; q = 0.9
Cookie: username = %%; hash_key = 5122420728838914,session_id = 8492544218643274
Connection: close
ENABLE_EXTERNAL_PING = YES_PING_HOSTIP = 11 | ping 192.168.0.11_PING_COUNT = 3
&pg = ping&LANGUAGE = &OKBTN=Start
```

Pseudocode 2

```
—WebKitFormBoundarysrLKRlLVLplqRfAu
Content-Disposition: form-data; name = “timeRebootEnablestatus”
on | reboot
—WebKitFormBoundarysrLKRlLVLplqRfAu
Content-Disposition: form-data; name = “timeRebootrange”
00 : 00
—WebKitFormBoundarysrLKRlLVLplqRfAu
Content-Disposition: form-data; name = “cururl”
http://192.168.2.1/cgi-bin/luci/;stok=b458f583573f2e98181cf1d9ce0f8443/admin/index
—WebKitFormBoundarysrLKRlLVLplqRfAu–
```

Pseudocode 3

```
function do_timereboot()
local uci_t = require “luci.model.uci”.cursor()
local rebootenable = luci.http.formvalue(“timeRebootEnablestatus”)
local timerange = luci.http.formvalue(“timeRebootrange”)
local cururl = luci.http.formvalue(“cururl”)
luci.sys.call(“uci set timereboot.timereboot.enable = %s>/dev/null” % rebootenable)
luci.sys.call(“uci set timereboot.timereboot.time = %s>/dev/null” % timerange)
luci.sys.call(“uci commit timereboot.timereboot>/dev/null”)
. . .//other codes
end
```

Pseudocode 4

nothing from online search engines, such as Google and Baidu. So, WMIFuzzer was deployed to fuzz its web management interface automatically. After 18 minutes, a crash was reported and the corresponding mutated message was captured in Pseudocode 5.

The content of authentication header has been appended with a number of ASCII char *A*. In our whole experimentation, both AFL and Sulley can trigger this crash, but they take more time than WMIFuzzer. We manually analyzed the mutated messages in these two fuzzers and found that the crash will not be triggered if the mutation changed the token *Basic*. As AFL cannot treat *Basic* and the *YWRtaW46YWRtaW4=* as two atomic fields, its mutation rarely generates a message that keeps the first field unchanged while the second field is modified to a long string. In contrast, both Sulley and WMIFuzzer can identify these two atomic fields, so they can detect the vulnerability quickly.

5. Discussion and Limitations

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

5.1. Scope of Targeted Devices. This paper focuses on COTS IoT devices that have a web management interface. Although

```
GET/HTTP/1.1
Host: 192.168.100.100
Authorization: Basic YWRtaW46YWRtaW4 = AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/
537.36
Accept: text/html, application/xhtml + xml,application/xml; q = 0.9,image/webp, image/apng,
*/*; q = 0.8
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh; q = 0.9
Connection: close
```

Pseudocode 5

this interface is very popular, not every device has it since web interface requires an underlying network that has a high bandwith (e.g., TCP/IP stack). For example, a micro temperature sensor that has a limited power resource usually applies Bluetooth Low Energy (BLE) as its network stack. WMIFuzzer cannot be used to detect vulnerabilities in these devices. However, WMPT is not dependent on web technology and any highly structured message can apply the WMPT to dissect itself for mutation-based fuzzing.

5.2. Interface Protection in Device. To detect vulnerabilities as fast as it can, WMIFuzzer sends a lot of messages to the device at a high speed. But if there is protection about message speed in the device firmware, most messages will be blocked to make WMIFuzzer inefficient. Message context is another protection, such as the CSRF token [55], which is designed to prevent CSRF attack. If this web interface contains CSRF protection, every message sent to the device must contain an unequal token extracted from the previous response. This means that a message will be invalid if it had been sent to the device once. This will make WMIFuzzer to not work since mutated messages from a seed have the same invalid CSRF token.

5.3. Limitations of Message Seed Generation. WMIFuzzer generates the initial message seeds by crawling the web pages of an IoT device. HTTP and HTTPS are very popular in transferring the web page content of IoT devices, and WMIFuzzer works well for these devices now. However, if a security scheme is deployed for this transmission (e.g., PrivHome [26]), WMIFuzzer will not work directly because of failure on decrypting captured web messages. One possible approach is implementing an internal interceptor of this new scheme and using it to capture the messages before encrypting them. A HTTP client containing this scheme is also needed for encrypting mutated messages and sending them to targeted devices. The WMPT and strategy of mutation are independent of this scheme, and they can remain unchanged.

5.4. Limitations of Remote Monitoring. As COTS IoT devices usually do not have the support for local monitoring, WMIFuzzer has to infer whether a vulnerability is triggered by monitoring the device network. But vulnerabilities in the targeted device may not alter the network status. For example, a memory overflow vulnerability may not cause a system crash if the device does not integrate MMU [56] to its memory management policy. Moreover, WMIFuzzer detects *Command Injection* dependent on *ping* and *reboot* commands since they can modify the device network status. However, if they are not available in the targeted COTS device, the device network cannot be changed by command injection vulnerabilities. More channels are required to monitor the running status of the targeted device, and one possible approach is monitoring signals from the hardware interface.

6. Related Work

As firmware images are been packed to reduce storage usage, existing tools (e.g., BinWalk, BAT, and FRAK [13, 14, 15]) are presented to unpack them. Costin et al. [5] presented the first public, large-scale analysis of firmware by analyzing the unpacked files statically. By collecting thousands of firmware images and analyzing them, the authors showed that account passwords and self-signed SSL certificates together with their corresponding private RSA keys, outdated software, building images as root and web servers configuration make many firmware images vulnerable. In another study, Costin et al. [6] used RIPS to scan PHP files in the device webroot to detect vulnerabilities, but the results showed that only 8% of embedded firmware images contain PHP code. Yan et al. [18] utilized a novel model of authentication bypass flaws to analyze binary files in firmware images, and finally they discovered several vulnerabilities in three firmware. It seems easy for static approaches to test IoT devices, as they do not need to run the firmware and they are independent of the real hardware. However, there are well-known limitations for static analysis techniques. The first one is *false negatives* (FNs), since static analysis mainly works upon some known patterns. For those vulnerabilities that have a new pattern,

static approaches usually fail. The second one is *false positives* (FPs), since every detected vulnerability by static approaches is just a potential vulnerability. A lot of manual efforts are required for further analysis to confirm them. So, some researches look forward to dynamic techniques.

Costin et al. [6] 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*. Chen et al. [19] presented an automated dynamic analysis system for Linux-based embedded firmware; it works on building a full system emulator to run the firmware with a stocked kernel and a private library of *nonvolatile memories* (NVRAM). It overcomes the challenges of userland emulation, such as *chroot* and *init*. However, the boot and configuration information in different devices are diverse, and it is very difficult to emulate fully. So, it still requires a lot of manual efforts to model its operation of NVRAM when testing a new device. Davidson et al. [57] presented a platform FIE for discovering implementation flaws in MSP430 family of microcontrollers, and FIE works by utilizing KLEE [58] symbolic execution engine to verify security properties in open-source firmware programs. FIE can perform a complete analysis of firmware images because symbolic execution can explore all possible execution paths in a program. However, it is limited to analyzing small firmware that is open-source and written in C Zaddach et al. [20] presented another platform, Avatar, supporting dynamic analysis of firmware; it is a hybrid approach that contains both the device and an emulator integrated with the S2E [59] engine. Avatar utilizes S2E to perform selective symbolic execution in the emulator and forwards I/O operations from the emulator to the real device. Anyway, both static analysis and dynamic analysis based on firmware are dependent on the availability of device firmware images. However, firmware images of many COTS IoT devices are not available publicly, or they have been encrypted with a private key, or they have been compressed with an unknown file format.

Some other research studies applied online scanning techniques to detect embedded devices' vulnerabilities in the Internet scale. Cui et al. [60] and Cui and Stolfo [16] utilized Nmap [54] to scan a wide range of open embedded devices in the Internet and attempted to log into them using well-known default credentials. The results showed that a large number of devices ranging from enterprise equipment to office equipment are vulnerable. In another work [5], ZMap [61] was used to search vulnerable devices in the Internet based on vulnerable SSL certificates, and around 30K online affected devices were identified. Heninger et al. [62] studied vulnerable keys in embedded devices at Internet scale, and they found that 0.75% of TLS certificates share keys due to insufficient entropy during key generation. SHODAN [63] is a powerful engine to search vulnerable devices in the global network, and it works by matching open service fingerprints in online devices. However, online scanning mainly focuses on discovering known vulnerabilities, as it works on a number of known signatures. So, it usually cannot discover zero-days. On the contrary, it is also ethically questionable, even illegal, to scan online devices without authorization.

Since most network-enabled devices will communicate with an external entity, some works are presented to fuzz these communication protocols for vulnerability discovery. RPFuzzer [64] 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. Chen et al. [65] 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. Our work is similar to IOTFUZZER; both of them perform mutation on atomic elements and do not require a predefined data model. However, not all IoT devices have an official control app, and IOTFUZZER can just detect memory corruption. In addition, the taint-based approach is much heavier than our blackbox approach. AFL and Sulley are two state-of-the-art fuzzers that are widely used to discover vulnerabilities in many software, but additional patches are needed to support fuzzing COTS IoT devices.

7. Conclusion

We have proposed the first blackbox fuzzer targeting the web management interface in COTS IoT devices; it utilizes the mutation-based fuzzing technology to discover vulnerabilities automatically. To improve the efficiency of fuzzing, a set of techniques including the seed generation, the WMPT, and the remote device monitoring are designed. The experimentation on 7 COTS devices successfully identified 10 vulnerabilities including 6 zero-days. All vulnerabilities have been reported to CNCERT/CC to help the vendor to fix them, and all of them have been fixed now.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request. The security vulnerabilities found in this paper can be accessed in the CNVD (http://www.cnvd.org.cn).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This study was supported in part by the National Key R&D Program of China (2017YFB0802300 and 2017YFB0802900), National Natural Science Foundation of China (61602092, 61972073, and 61572115), and Open Research Project of the

State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences (2019-MS-05).

References

[1] Gartner, "Internet of things (IoT) market," 2017, https://www.gartner.com/newsroom/id/3598917.

[2] B. Douglas, "Eight crazy hacks: the worst and weirdest data breaches," 2015, https://securityintelligence.com/eight-crazy-hacks-the-worst-and-weirdest-data-breaches-of-2015/.

[3] T. Dunlap, "The 5 worst examples of IoT hacking and vulnerabilities in recorded history," 2017, https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities/.

[4] S. Ravipati, "University hackers attacked 5,000 IoT devices on campus," 2017, https://campustechnology.com/articles/2017/02/13/university-hackers-attacked-5000-iot-devices-on-campus.aspx.

[5] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the 23rd USENIX Security Symposium (SEC)*, pp. 95–110, Berkeley, CA, USA, August 2014.

[6] A. Costin, A. Zarras, and A. Francillon, "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*, pp. 437–448, Xi'an, China, May 2016.

[7] D. Wang, H. Cheng, P. Wang, X. Huang, and G. Jian, "Zipf's law in passwords," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 11, pp. 2776–2791, 2017.

[8] D. Wang, W. Li, and P. Wang, "Measuring two-factor authentication schemes for real-time data access in industrial wireless sensor networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 9, pp. 4081–4092, 2018.

[9] D. Wang and P. Wang, "Two birds with one stone: two-factor authentication with security beyond conventional bound," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 4, pp. 708–722, 2018.

[10] F-Secure, "Vulnerabilities in Foscam IP cameras enable root and remote control," December 2015, http://images.news.f-secure.com/Web/FSecure/%7B43df9e0d-20a8-404a-86d0-70dcca00b6e5%7D_vulnerabilities-in-foscam-IP-cameras_report.pdf.

[11] D. Roland and S. Bjarnason, "Mirai IoT botnet description and DDoS attack mitigation," 2016, https://asert.arbornetworks.com/mirai-iot-botnet-description-ddos-attack-mitigation/.

[12] DLink, "DLink firmware FTP server," 2016, http://ftp.dlink.ru/pub/.

[13] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: a case study of embedded exploitation," in *Proceedings of the NDSS Symposium*, San Diego, CA, USA, February 2013.

[14] H. Craig, "Binwalk: firmware analysis tool," 2010, https://code.google.com/p/binwalk/.

[15] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 63–72, Honolulu, HI, USA, May 2011.

[16] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan," in *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 97–106, Austin, TX, USA, December 2010.

[17] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 709–724, San Jose, CA, USA, May 2015.

[18] S. Yan, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2015.

[19] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proceedings 2016 Network and Distributed System Security Symposium (NDSS)*, pp. 1–16, San Diego, CA, USA, February 2016.

[20] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: a framework to support dynamic security analysis of embedded systems' firmwares," in *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2014.

[21] CNCERT/CC, *National Computer Network Emergency Response Technical Team/Coordination Center of China*, CNCERT/CC, Beijing, China, 2018, http://www.cert.org.cn/.

[22] OpenRCE, "Sulley," 2012, https://github.com/OpenRCE/sulley.

[23] M. Zalewski, "American fuzzy lop," 2014.

[24] C. Doukas, I. Maglogiannis, V. Koufi, F. Malamateniou, and G. Vassilacopoulos, "Enabling data protection through PKI encryption in IoT m-health devices," in *2012 IEEE 12th International Conference on Bioinformatics & Bioengineering (BIBE)*, Larnaca, Cyprus, November 2012.

[25] J.-Y. Lee, W.-C. Lin, and Y.-H. Huang, "A lightweight authentication protocol for internet of things," in *Proceedings of the International Symposium on Next-Generation Electronics (ISNE)*, Tao-Yuan, Taiwan, May 2014.

[26] G. S. Poh, P. Gope, and J. Ning, "PrivHome: privacy-preserving authenticated communication in smart home environment," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, 2019.

[27] H. Yang, W. Lee, and H. Lee, "IoT smart home adoption: the importance of proper level automation," *Journal of Sensors*, vol. 2018, Article ID 6464036, 11 pages, 2018.

[28] J.-H. Lee and H. Kim, "Security and privacy challenges in the internet of things [security and privacy matters]," *IEEE Consumer Electronics Magazine*, vol. 6, no. 3, pp. 134–136, 2017.

[29] L. Xiao, X. Wan, X. Lu, Y. Zhang, and D. Wu, "IoT security techniques based on machine learning," 2018, http://arxiv.org/abs/1801.06275.

[30] M. Muench, S. Jan, K. Frank, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: challenges in fuzzing embedded devices," in *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2018.

[31] S. Chong, J. Liu, A. C. Myers et al., "Secure web applications via automatic partitioning," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 31–44, 2007.

[32] S. Guelich, S. Gundavaram, and G. Birznieks, *CGI Programming with Perl: Creating Dynamic Web*, O'Reilly Media, Inc., Newton, MA, USA, 2000.

[33] A. Hester, R. Borges, and R. Ierusalimschy, "Building flexible and extensible web applications with Lua," *Journal of Universal Computer Science*, vol. 4, no. 9, pp. 748–762, 1998.

[34] X. Yu and Y. Cai, "Design and implementation of the website based on PHP & MYSQL," in *Proceedings of the 2010*

International Conference on E-Product E-Service and E-Entertainment, pp. 1–4, Henan, China, November 2010.

[35] A. Takanen, J. D Demott, and C. Miller, "Fuzzing for software security testing and quality assurance," 2008.

[36] C. Miller and Z. N. J. Peterson, "Analysis of mutation and generation-based fuzzing," Tech. Rep 4, Independent Security Evaluators, Baltimore, MA, USA, 2007.

[37] D. Aitel, "An introduction to SPIKE, the fuzzer creation kit," in *Proceedings of the Black Hat*, USA, August 2002.

[38] M. Eddington, "Peach fuzzing platform," 2011, https://www.peach.tech/.

[39] M. Ehmer and F. Khan, "A comparative study of white box, black box and grey box testing techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 6, 2012.

[40] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[41] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[42] F. Roy, Jim Gettys, J. Mogul et al., "Hypertext transfer protocol–HTTP/1.1," Technical Report, 1999, https://www.w3.org/Protocols/rfc2616/rfc2616.html.

[43] H. Shuai, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, "PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 204–217, Bretton Woods, NH, USA, 2014.

[44] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[45] D. Wang, M. Jiang, T. Chen, X. Zhang, and C. Wang, "Cracking IoT device user account via brute-force attack to SMS authentication code," in *Proceedings of the First Workshop on Radical and Experiential Security*, pp. 57–60, New York, NY, USA, June 2018.

[46] W3C 2018, "HTML specification," 2018, https://www.w3.org/TR/html/.

[47] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cws and box," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[48] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 359–368, Orlando, FL, USA, December 2012.

[49] FuzzDB, "Dictionary of attack patterns and primitives for black-box application fault injection and resource discovery," 2016, https://github.com/fuzzdb-project/fuzzdb.

[50] T. F. Javier and J. D. Guttman, "Copy on write," 1995.

[51] C. McMahon, "History of a large test automation project using selenium," in *Proceedings of the 2009 Agile Conference*, pp. 363–368, Chicago, IL, USA, August 2009.

[52] mitmproxy, "An interactive TLS-Capable intercepting HTTP proxy," 2016, https://github.com/mitmproxy/mitmproxy.

[53] Solar Designer, "John the ripper password cracker," 2006, https://www.openwall.com/john/.

[54] L. Gordon, "Nmap–free security scanner for network exploration & security audits," 2009.

[55] B. Adam, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 75–88, Alexandria, VA, USA, October 2008.

[56] G. Rose, "Using the microprocessor MMU for software protection in real-time systems," 2006, http://www.lynx.com/using-the-microprocessor-mmu-for-software-protection-inreal-time-systems/.

[57] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22nd USENIX Security Symposium (SEC)*, pp. 463–478, Washington, DC, USA, August 2013.

[58] C. Cadar, D. Dunbar, D. R. Engler et al., "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the OSDI*, pp. 209–224, San Diego, CA, USA, December 2008.

[59] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[60] A. Cui, Y. Song, P. V. Prabhu, and S. J. Stolfo, "Brave new world: pervasive insecurity of embedded network devices," in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*, pp. 378–380, Lecture Notes in Computer Science, Saint-Malo, France, September 2009.

[61] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: fast internet-wide scanning and its security applications," in *Proceedings of the 22nd USENIX Security Symposium (SEC)*, pp. 47–53, Washington, DC, USA, August 2013.

[62] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: detection of widespread weak keys in network devices," in *Proceedings of the USENIX 21st Security Symposium*, Bellevue, WA, USA, August 2012.

[63] B. Genge and C. Enăchescu, "ShoVAT: Shodan-based vulnerability assessment tool for Internet-facing services," *Security and Communication Networks*, vol. 9, no. 15, pp. 2696–2714, 2016.

[64] Z. Wang, Y. Zhang, and Q. Liu, "RPFuzzer: a framework for discovering router protocols vulnerabilities based on fuzzing," *KSII Transactions on Internet and Information System*, vol. 7, no. 8, pp. 1989–2009, 2013.

[65] J. Chen, W. Diao, Q. Zhao et al., "Iotfuzzer: discovering memory corruptions in iot through app-based fuzzing," in *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2018.