IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 July 2022, accepted 12 September 2022, date of publication 20 September 2022, date of current version 30 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3208146

RESEARCH ARTICLE

FuzzDocs: An Automated Security Evaluation Framework for IoT

MYOUNGSUNG YOU[1], YEONKEUN KIM[2], JAEHAN KIM[1], MINJAE SEO[3], SOOEL SON[2], (Member, IEEE), SEUNGWON SHIN[1], (Member, IEEE), AND SEUNGSOO LEE[4]

[1]School of Electrical Engineering, KAIST, Yeonsu-gu, Daejeon 34141, Republic of Korea
[2]School of Computing, KAIST, Yeonsu-gu, Daejeon 34141, Republic of Korea
[3]National Security Research Institute, Yeonsu-gu, Daejeon 34044, Republic of Korea
[4]Department of Computer Science and Engineering, Incheon National University, Yeonsu-gu, Incheon 22012, Republic of Korea

Corresponding author: Seungsoo Lee (seungsoo@inu.ac.kr)

This work was supported by Incheon National University (International Cooperative) Research Grant in 2022.

ABSTRACT As Internet of Things (IoT) devices have rooted themselves in the daily life of billions of people, security threats targeting IoT devices are emerging rapidly. Thus, IoT vendors have employed security testing frameworks to examine IoT devices before releasing them. However, existing frameworks have difficulty providing automated testing, as they require a lot of manual effort to support new devices due to the lack of information about the input formats of the new devices. To address this challenge, we introduce FuzzDocs, a document-based black-box IoT testing framework designed to automatically analyze publicly accessible API documents about target IoT devices and extract information, including valid inputs used to call each functionality of the target devices. Based on the extracted information, it generates valid-enough test inputs that are not easily rejected by target devices but can trigger vulnerabilities deep inside them. This document-based input generation allows FuzzDocs to support new devices without manual work, as well as provide effective security testing. To prove its feasibility, we evaluated FuzzDocs in a real-world IoT environment, and the results showed that FuzzDocs extracted input formats with 93% accuracy from hundreds of pages of documents. Also, it outperformed the existing frameworks in testing coverage and found 35 potential vulnerabilities, including two unexpected system failures in five popular IoT devices.

INDEX TERMS IoT security, IoT security scanning, fuzz testing, document-based fuzzing.

I. INTRODUCTION

Currently, IoT (Internet of Things) devices are getting smarter and being actively deployed in diverse networking environments, such as factory management systems and home automation services. With the dramatic advances in hardware and software technology, complicated functions can be operated using a small chip or device and fully optimized. Thus, IoT devices are now capable of more advanced and complicated functions than in their early stages, and they have become an essential component of people's daily lives.

Now that IoT devices can manage and store much more data, including some sensitive information, IoT security has become a major issue. We can easily see many real-world cases showing how IoT devices are vulnerable [1], [2], [3]. Worse, the overheated IoT market encourages IoT vendors to aggressively release new devices, but a lack of budget discourages them from conducting sufficient security checks. For the same reason, even if vulnerabilities are found in already-released devices, IoT vendors are unlikely to have the chance or capability to fix the vulnerabilities properly.

One practical solution to address this problem is to identify vulnerabilities in IoT devices using automated tools and take appropriate security precautions. Several researchers have proposed a series of solutions to test the security of IoT devices, ranging from fuzzing device firmware through emulation [4], [5], [6], [7] to solutions that test IoT devices directly over the network [8], [9], [10], [11], [12], [13].

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Masucci.

Firmware fuzzing can provide comprehensive and accurate security testing by emulating and analyzing firmware through software fuzzing techniques. However, obtaining firmware is challenging even for vendors because of the highly fragmented IoT industry, which involves many small integration and distribution vendors (e.g., OEMs) [10]. Moreover, analyzing firmware requires a lot of manual effort due to the diversity of platforms used by IoT devices [9], [11]. Another line of research finds vulnerabilities without firmware by sending randomized inputs to IoT devices over the network (i.e., network-based fuzzing) [8], [9], [10], [11], [12], [13]. Unfortunately, IoT devices reject invalid inputs that violate their input format, rather than processing them. Thus, network-based fuzzers rely on additional software analysis [9], [11] or well-refined seeds [8], [10], [12], [13] (or manually created message formats) to create inputs for testing. However, this prevents automated testing and limits test coverage. As a result, due to the difficulty of obtaining and analyzing firmware and the limitation of seed inputs, existing solutions fail to provide effective and automated security testing for newly emerging IoT devices.

A. OUR APPROACH

In this paper, we propose a document-based black-box IoT testing framework, named FuzzDocs, to achieve automatic and effective security testing for IoT devices. The key idea is based on the observation that most IoT vendors make API documents of IoT devices publicly available so third-party developers (or users) can build various IoT systems. These API documents contain information about valid inputs for calling IoT devices' functionality (i.e., API). Thus, FuzzDocs can use API documents as guidelines for test input generation. To this end, it automatically analyzes human-readable API documents of various types, using heuristics and NLP-based[1] methods to extract input formats for IoT devices. Based on the input formats, FuzzDocs generates valid-enough inputs for testing that are not rejected by IoT devices at early stages but can trigger security bugs deep inside them. After transmitting valid-enough inputs, it detects vulnerabilities by analyzing the responses of the IoT devices. This design allows FuzzDocs to test all device functionalities listed in API documents automatically and effectively without software analysis (e.g., firmware) or manually provided seeds. That is, to support a new device, FuzzDocs requires only API documents for the device, rather than manual effort by security experts.

We implement a prototype of FuzzDocs and compare it with existing solutions in real-world IoT environments, including Hue [14] and Shelly [15] devices. As a result, FuzzDocs extracted correct input formats with 93% accuracy from 134 APIs (i.e., functionalities) for the five devices used in the experiment. FuzzDocs also outperforms other solutions in terms of test coverage and has newly discovered 35 potential vulnerabilities in five popular IoT devices, including two DoS vulnerabilities.

[1]NLP refers to Natural Language Processing.

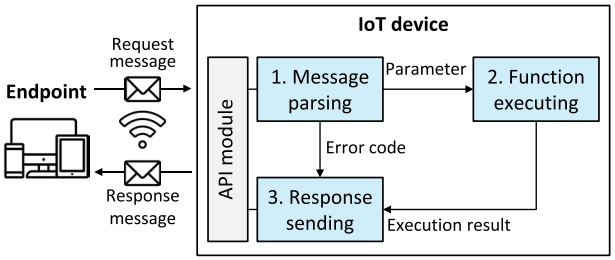


FIGURE 1. General communication model of modern IoT devices. IoT devices receive a request message (input) from endpoints (users or other devices) and then transmit a response message that contains the execution result.

In summary, our contributions are as follows:

- We present a test input generation model leveraging API documents for automatically identifying potential vulnerabilities in target IoT devices.
- We present the design and implementation of a new document-based testing framework for IoT, called FuzzDocs, which is capable of automatically generating the test inputs from the API documents and detecting the vulnerabilities in the target devices.
- We evaluate FuzzDocs in real-world environments, demonstrating that the effectiveness of FuzzDocs is superior to existing solutions and discovering 35 potential vulnerabilities in five IoT devices.

B. OUTLINE

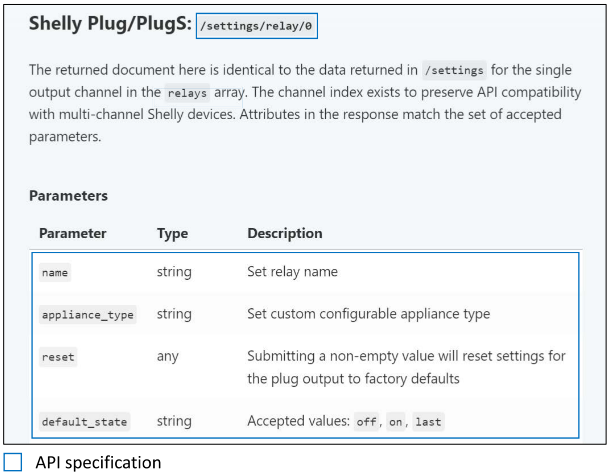
The rest of this paper is organized as follows. Section II gives the required background and our insights of FuzzDocs with a running example. Section III reviews the previous studies and their limitations. The overall system design is presented in Section IV. Section V and VI detail two key modules of FuzzDocs, respectively. The evaluation results are summarized in Section VII, and Section VIII discusses the limitations of the current design. Finally, we conclude this paper in Section IX.

II. BACKGROUND AND MOTIVATION

A. COMMUNICATION MODEL OF IoT DEVICES

Unlike traditional embedded devices, most IoT devices interact with other endpoints (i.e., devices or users) over the network. As IoT devices become more diverse and complex, interoperability between these devices gains importance. Thus, most IoT vendors open application programming interfaces (APIs) to effectively interact with their IoT devices. The APIs allow developers to build various IoT ecosystems (e.g., smart homes and smart factories) by connecting multiple IoT devices, regardless of vendor.

Figure 1 shows an overview of the communication model of modern IoT devices. APIs and the API module enable the endpoints to invoke the IoT device's functionality by sending request messages [9], [10], [11], [16]. For example, the endpoints transmit a request message to the device through the network (e.g., using Wi-Fi). The device then parses the


Shelly Plug/PlugS: /settings/relay/0

The returned document here is identical to the data returned in /settings for the single output channel in the relays array. The channel index exists to preserve API compatibility with multi-channel Shelly devices. Attributes in the response match the set of accepted parameters.

Parameters

Parameter	Type	Description
name	string	Set relay name
appliance_type	string	Set custom configurable appliance type
reset	any	Submitting a non-empty value will reset settings for the plug output to factory defaults
default_state	string	Accepted values: off, on, last

API specification

FIGURE 2. Example of partial API document for a popular IoT device [15]. The API document is publicly accessible and provides detailed information (i.e., API specifications) needed to create a valid request message to call the API.

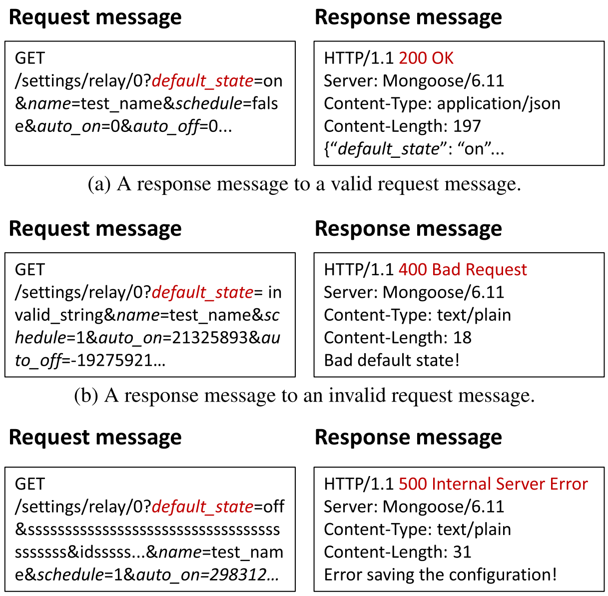
message to extract its parameters and do the job requested (e.g., turn on the light). Most IoT devices have their own message format for efficient communications. Thus, if the device finds any violations during the parsing process, it immediately rejects the request and transmits an error message to the sender (i.e., the endpoint), notifying them that the message has been rejected. Also, even if the request message complies with the format, an error message could be generated if the parameter values in the request violate the expected scope. Otherwise, the device executes the corresponding function using the extracted parameters and sends a response message with the execution result.

There are many ways to implement the APIs in the IoT devices, but most IoT vendors adopt HTTP-based APIs (e.g., REST API) due to their flexibility, scalability, and simplicity [17], [18], [19]. Therefore, we focus on HTTP-based APIs in this work for the sake of simplicity.

B. API DOCUMENTS FOR IoT DEVICES

Most IoT vendors provide an API document to help developers understand how to use their device APIs. Figure 2 depicts the partial API document for the popular IoT device Shelly PlugS [20]. This API document is publicly accessible on the Internet [21] and semi-structured and formatted in HTML pages, like most API documents. An API document typically starts with the details of the API specifications. API specifications are information (e.g., URL, HTTP method and parameters) required to create a request message to call a specific API of the device. Followed by the API specifications, some concrete examples such as a JSON object for message parameters and CURL commands [22] are often provided to demonstrate the utilization of such APIs.

Given the purpose of the API document, it is reasonable for the vendors to make human readability a priority. For example, an API document describes the IoT device's APIs on a separate page, one by one, according to each URL and HTTP method. On the same page, the document explains each API



(a) A response message to a valid request message.

(b) A response message to an invalid request message.

(c) A response message indicating an internal error of the IoT device.

FIGURE 3. Examples of response messages received from the IoT device [20]. The device would transmit different messages resulting from processing the input message.

specification by splitting it into sections. The tables or lists in each section are employed to describe API specifications that contain multiple descriptions (e.g., parameters).

C. MOTIVATING EXAMPLE

As discussed earlier, the API document guides the user through the process of creating an valid request message to execute a specific function (API) of the IoT device. For example, when calling the '`settings`' API, we must set the '`default_state`' parameter to one of the three predefined strings ('`off`', '`on`', or '`last`') as described in the 'Parameters' section in Figure 2. When we send a valid request message created according to the requirements, the device sends the response message containing the API execution results, as shown in Figure 3a.

Let us consider an example in which a user intentionally builds an invalid request message with parameters that violate the requirements from the API document in Figure 2. For instance, we set the '`default_state`' parameter to a random string rather than the predefined values. If the device appropriately handles the invalid request message, it sends the response message indicating that the message includes invalid parameters (e.g., HTTP 400 bad requests), as shown in Figure 3b. However, some invalid request messages cause unexpected errors that the device cannot handle. If we set '`default_state`' to a very long random string, unexpected errors such as buffer overflows may occur when the device processes the parameter. In this case, the device sends response messages indicating the unhandled errors (e.g., HTTP 500 internal errors), which could lead to severe security vulnerabilities. Figure 3c shows the result of this example, which makes the device unable to respond to any further request messages until it is rebooted (i.e., DoS).

The API document offers information about the majority of APIs supported by the IoT device. Therefore, we can discover potential security vulnerabilities in most of the device's functions by sending invalid request messages, as demonstrated in Figure 3. The biggest challenge is that it requires manual effort to analyze the document and generate invalid but high-quality request messages that are effective enough to cause internal errors. This limitation motivates us to design a document-based testing framework for IoT devices that is automated.

III. RELATED WORK

Fuzz testing (i.e., fuzzing) is one of the dominant solutions for testing IoT devices as well as software. Several solutions [4], [5], [6], [7], [8], [9], [10], [11], [16], [23], [24], [25], [26], [27], [28], [29] have emerged to discover security bugs in IoT devices, and most of them adopt black-box fuzzing due to the closed source of target devices and the ease of testing. These previous black-box fuzzing techniques for IoT devices can be classified into three main categories based on the way they work: firmware-based fuzzing [4], [5], [6], [7], [23], [24], network-based fuzzing [8], [10], [25], [26], [27], [28], and companion-app-based fuzzing [9], [11], [16]. However, these prior studies have two limitations: (i) the amount of manual effort required (i.e., seeds or settings), and (ii) restricted testing coverage. These limitations make these prior studies hard to automate and also reduce their efficiency. We summarize the prior studies in Table 1 and discuss their approach and limitations in detail.

A. FIRMWARE-BASED FUZZING

The key purpose of firmware-based fuzzing is to emulate the target IoT device firmware, which is dedicated software for the devices to discover vulnerabilities in them. Although this approach may be effective enough to test, it has some critical limitations from an automation point of view. First, the acquisition of the device firmware is a challenging task itself. As demonstrated by several previous works [4], [5], [6], [7], [23], [24], [29], IoT vendors tend not to release their device firmware to the public due to concerns about loss prevention and general security. Thus, without the published firmware, security experts with hardware knowledge could extract firmware directly from the device through debugging ports (e.g., UART [30]). However, most IoT vendors ship their IoT devices without debugging ports to prevent hardware-based firmware leakage [9].

Second, even if we somehow obtain the firmware, emulation testing is difficult because there is no unified firmware analysis method due to the diversity of IoT device architectures (e.g., MIPS [31], RISC-V [32]). For instance, we have to set up various configurations such as firmware unpacking (or decryption) and NVRAM parameter settings [5] to set up the made-to-order emulation environments, using a lot of manual effort. This shows that the firmware-based fuzzing method is not suitable for an automated testing system.

B. NETWORK-BASED FUZZING

In order to test IoT devices without firmware, researchers have developed network-based fuzzing [10], [25], [26], [27], [28] that tests the security of IoT devices by sending mutated messages directly to the devices over networks. However, most IoT devices have strict message formats in network communication and reject invalid messages. That is, input messages created in a brute-force manner are easily rejected by the devices, reducing the effectiveness of fuzzing tests [9], [11]. Thus, network-based fuzzing first takes a set of seed messages (or grammar) from security experts and mutates the seed messages to create input messages for testing. After this method sends the mutated input messages to the IoT devices, it analyzes the response messages to detect potential vulnerabilities. Some researchers [10] use the response messages as feedback to create subsequent input messages, increasing the possibility of triggering more unusual responses.

Network-based fuzzing does not require firmware analysis, but well-formed seed messages are essential, which means its success depends strongly on the quality of the seed messages. Also, if the seed messages cannot invoke a specific API of the target devices, the API cannot be tested by network-based fuzzing, which significantly reduces the coverage of testing. Moreover, creating high-quality seed messages requires a lot of manual effort by security experts, which is not suitable for automated testing.

C. COMPANION-APP-BASED FUZZING

In a parallel line of work, researchers [9], [11], [16] leverage a companion application, which is a mobile app for IoT devices instead of firmware. This method creates input messages by analyzing companion apps. Technically, it retrieves message creation functions of companion apps using reverse engineering. Then, it mutates the retrieved functions' parameters at runtime to send mutated input messages to IoT devices.

However, this method suffers from two fundamental problems. The first is the limited fuzzing coverage; this method cannot guarantee that all the device's APIs (functionalities) are fully tested because there still remain the APIs not invoked during the automated companion app execution. In this case, security experts need to manually click all the UIs in the app to trigger the remaining APIs, which takes a lot of manual effort and is not suitable for automated testing. Moreover, this method is unable to test APIs that are not called through the app's UI, such as APIs that change internal device settings (see Section VII-C1). Second, if the companion app is obfuscated, this method is impractical because code obfuscation prevents app reverse engineering. Such code obfuscation has become very common in mobile environments due to the surge in mobile app security issues [33].

D. OUR APPROACH

Firmware-based fuzzing is not practical for automated testing. Other fuzzing methods still require a lot of manual work and have limited test coverage due to the lack of information about IoT devices' input message formats. FuzzDocs can

TABLE 1. Summary of related work on IoT device fuzzing. While existing solutions require firmware analysis or a lot of manual work by security experts, such as generating seed messages, FuzzDocs can automatically test the security of IoT devices without such a burden.

	Firmware-based [4]–[7], [23], [24]	*Network-based* [8], [10], [25]–[28]	*Companion-app-based* [9], [11], [16]	*Document-based* (FuzzDocs)
Requirement	Device firmware (Not disclosed)	Seed message or corpus (Not disclosed)	Companion app (Publicly available)	API document (Publicly available)
Manual effort	Required (Firmware de-packing,) emulation setting)	Required (Message format analysis, seed message creation)	Required (Manual de-compilation∗, UI simulation)	N/A
Test coverage	All APIs	Limited	Limited	All APIs

∗Required when the companion app is obfuscated.

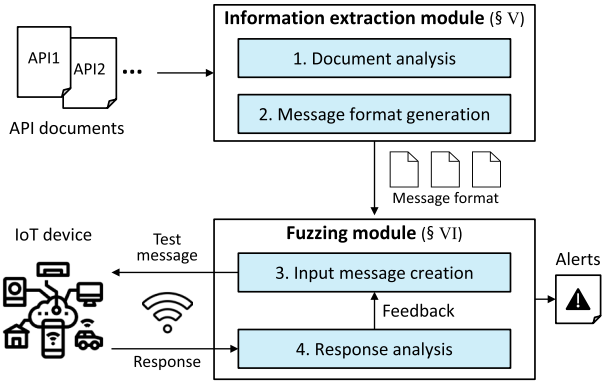


FIGURE 4. Overall workflow of FuzzDocs. The information extraction module takes API documents and creates message formats for security testing. The fuzzing module finds potential vulnerabilities in the IoT device by sending test messages created from the message formats.

tackle these limitations. It leverages the publicly available API documents (see Figure 2) as a testing guideline for functionalities (APIs) of IoT devices. By analyzing API documents, FuzzDocs creates effective input messages for testing all functionalities (APIs) described in API documents and detects potential security holes in target devices in a fully automated manner.

IV. SYSTEM DESIGN

In this section, we provide design requirements for FuzzDocs and its overall workflow.

A. DESIGN REQUIREMENTS

To effectively and efficiently examine the security of IoT devices by leveraging their API documents, the requirements driving our framework can be summarized as follows.

1) R1: AUTOMATING INFORMATION EXTRACTION

As various IoT device vendors have provided their own APIs, the format of the API document may differ depending on each vendor, meaning that there is no unified API document format for IoT devices. For instance, from the separate IoT devices, some API documents use table tags (e.g., `<tr>` or `<td>`), while others use division tags (i.e., `<div>`) with custom styles to describe API specifications. This diversity of format makes extracting API specifications from an API document difficult for a machine. Moreover, it is time-consuming to also extract API specifications from the document manually. To resolve these problems, our framework centers on automating the API specification extraction process.

2) R2: CREATING EFFECTIVE MESSAGE FORMAT

In general, IoT devices reject invalid API request messages at early stages (i.e., message parsing phases) that violate their predefined message format. If we know the detailed message formats for each target device, we can easily create input (request) messages that are *valid enough* not to be easily rejected by the devices but can trigger unexpected internal errors beyond the parsing state. However, manually creating such detailed message formats for target devices is time-consuming. Therefore, our framework needs to automatically generate effective and detailed message formats for each API to create valid enough input messages for testing.

3) R3: MONITORING INTERNAL STATE

It is important to recognize if the target IoT device is entering an abnormal state during security testing. The best way to know the internal states of the device is to directly examine state information through its firmware. In other words, without direct access to the firmware, it is challenging to identify the device's internal state. This lack of state information makes it hard to determine whether the input messages sent by our framework are triggering potential security bugs on the device. Thus, our framework should provide a remote and automated mechanism to recognize the internal states of the target device.

B. DESIGN OVERVIEW

Figure 4 illustrates the key modules of FuzzDocs and its overall workflow. There are two main modules: the information extraction module (top) and the fuzzing module (bottom). The information extraction module takes and parses API documents in various formats to create a layer of abstraction around the target IoT device's API request message formats. (1) First, it extracts API specifications (e.g., URLs, parameters) required for creating valid API request messages from the input API document. When doing so, it uses text-based document parsing to process API documents regardless of their format (**R1**). (2) Next, it utilizes natural language processing (NLP) techniques to analyze the extracted

information and create a message format for each API. Each message format contains crucial information for testing the corresponding API (**R2**). It then passes the message format to the fuzzing module.

The fuzzing module aims to discover potential security bugs inside the target IoT device by transmitting input messages. (3) Based on the message format, the module uses elaborate mutation strategies to automatically create valid-enough input messages that are not easily filtered by the device and can cause security bugs deep inside the device. (4) After sending the input message, this module monitors the response messages and networking behaviors of the device to remotely determine whether the input messages caused the device's security bugs (**R3**). It also offers detailed reporting of bug-inducing messages to aid security experts. The following sections describe each module of FuzzDocs in detail.

V. INFORMATION EXTRACTION MODULE

The key idea of the information extraction module is to reverse engineer the message formats of the target IoT device by analyzing its API document. To do this, the module takes an API document containing information about a specific API. It then produces a message format, a machine-readable description for creating valid request messages for the corresponding API.

A. DOCUMENT ANALYSIS

First, the module takes an API document in an HTML format and parses it to extract API specifications such as a URL, an HTTP method, and parameters. When parsing the API document, this module needs to focus on *text* (i.e., <tag>*text*</tag>) rather than a specific type or structure of HTML tags (e.g., <table>,) because the text is more likely to be related to the API specification and each document uses a different type of tag. However, although we can extract all the text from the document except for the tags, it is challenging to determine whether an individual text is directly related to API specifications without any contextual information because the document is a mixture of a lot of text explaining different content. Fortunately, a typical API document has structured guidelines that inform us which texts are relevant to the API specifications, so the module can leverage this.

As illustrated in Figure 5 (a), the API document is divided into multiple sections (and sub-sections), and a section title indicates the main content of each section. For example, if the section title is ''Path Params'', we can guess that all the following text in that section contains descriptions (e.g., name, value type) about the path parameters. Based on these findings, this module extracts the text in the document and tries to build a section tree that shows the relationship between the text and section titles.

1) BUILDING SECTION TREE

Before building a section tree of an API document, the module conducts the preprocessing as follows to eliminate unnecessary parts and improve parsing efficiency. First, it removes the optional areas (e.g., sidebars and navigation bars) in the API document, which have no meaningful text for the API specifications. Next, it replaces the tags used to emphasize keywords (e.g., , , <italic>) with double quotes when these tags are within the text. For instance, 'The brightness from' is replaced with 'The ''brightness'' from.' Normally, such tags are employed to highlight the keywords (e.g., parameter names), so this tag replacement helps the module extract important keywords for the API.

Algorithm 1 Build a `Section Tree` of an API Document

```
1  BuildSectionTree (T)
       inputs : A set of HTML tags in an API document
                denoted by T; depth-first search function
                DFS; function for adding child node to tree
                addChild; function for adding sibling node
                to tree addSibling
       output: A section tree of the API document
                denoted by ST
2      ST ← {root};
3      l_t ← root;
4      idx ← 1;
5      H = {< h1 >, ..., < h6 >, < heading >};
6      foreach an HTML tag t_i in DFS(T) do
7          if texts ∉ t_i then
8              continue;
9          if t_i ∈ H then
10             if t_i ≤ l_t then
11                 addChild(l_t, t_i, idx);
12             else
13                 addSibling(l_t, t_i, idx);
14             l_t ← t_i;
15         else
16             addSibling(l_t, t_i, idx);
17         idx ← idx + 1;
18     return ST;
```

After the above preprocessing, the module builds the section tree as described in Algorithm 1. It starts (line 6) by traversing all the HTML tags in the API document (T) in a depth-first search (DFS) order to read the tags in the order they appeared in the API document. For each tag, it checks to see if there is text between the opening (<tag>) and closing (</tag>) pairs of the current tag (t_i) (line 7). Note that the module does not consider the text inside a tag's attributes (e.g., *href* attribute). If the current tag has text, it creates a new node for the current tag, which contains a type, the found text, and an insertion index (idx). The insertion position of the new node depends on whether the current tag is a title tag (H). In the case of the title tag, the module compares the priority of the last inserted title tag (l_t) with the current tag (line 10).

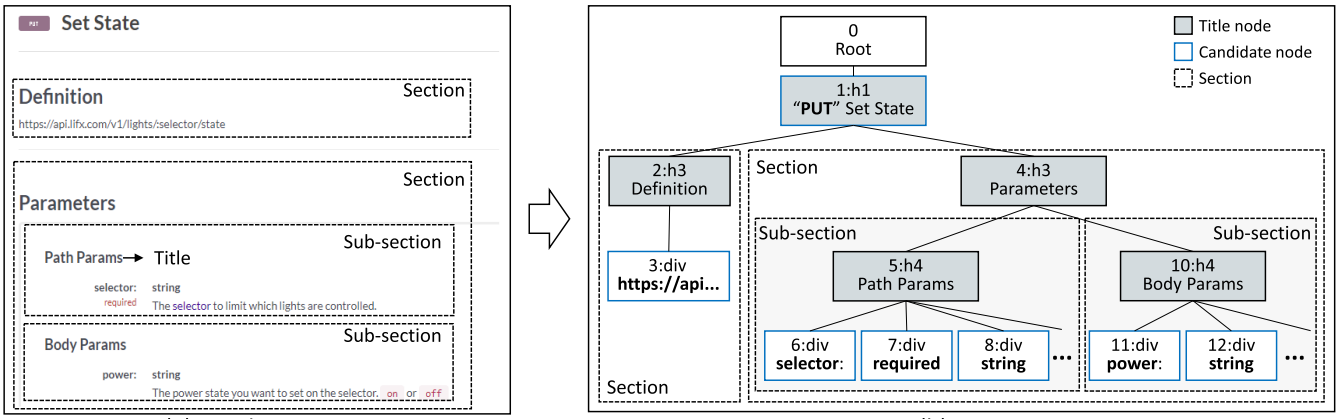


FIGURE 5. Each section in the API document (a) and a partial section tree originated from it (b) and processed by the information extraction module. Regardless of the document format, the section tree captures the hierarchical relationship between the texts in the API document and identifies candidate nodes for each API specification.

TABLE 2. Keywords used as titles of parameter sections in API documentation.

Prefix				*Suffix*
Header	*Path*	*Query*	*Body*	
Header, Authorization	Path, URI,	Query	Body, Request	Parameters, Params Properties, Schema Arguments,

The lower the level of the title, the higher the priority (i.e., h1 > ...> h6). If the current tag has a lower priority than the last title tag (line 11), the new node becomes a child node of the node for the last title tag (i.e., the last title node). Otherwise, it becomes the last title node's sibling node (line 13). The new node becomes the last title node's child node when the current tag is not a title tag (line 16).

Let us explore the example of the partial section tree as shown in Figure 5 (b), which is derived from the API document [34]. The section tree presents the text nodes (i.e., non-title nodes) and the corresponding title nodes as a parent-child relationship. That is, all the nodes containing descriptions of the path parameters are child nodes of the title node whose text is 'Path Params'. It also expresses the hierarchical relationship between sections. The 'Path Params' and 'Body Params' title nodes are child nodes of 'Parameters', which is the main title of the parameter section.

2) IDENTIFYING CANDIDATE NODES

Having created the section tree, the module retrieves the candidate nodes whose texts are related to (1) the URL/HTTP methods or (2) the parameters of the API described in the input API document, and it pushes candidate nodes to a node pool. As the first step, it finds the nodes for the URL and HTTP methods from the section tree. Both the URL and HTTP methods have unique patterns, so we adopt a regular expression-based method [35] to easily discover them. In general, the combination of the URL and HTTP methods is a unique identifier for APIs, meaning that they usually appear at the beginning of the API document. These characteristics are also reflected in the section tree. Thus, if the module finds multiple candidate nodes that include the URL/HTTP method, it selects the node with the lowest index for URL and HTTP methods. It then inserts the selected candidate nodes into the node pool.

Next, the module should find candidate nodes for the parameters, but the parameters have no unique patterns, unlike the URL and HTTP methods. Also, parameters are divided into various types according to their position in an input message (e.g., path or body). These characteristics prevent the module from determining which nodes are associated with which type of parameters. To resolve this problem, the module leverages title nodes as an indicator for locating the candidate nodes for parameters. For this, we build a dictionary for each type of parameter by collecting the keywords used as titles in the parameter sections from about 100 API documents for 10 popular IoT vendors [15], [36], [37], [38], [39], [40], [41], [42], [43], as shown in Table 2.

The module leverages the dictionary for the parameters as follows. It first selects title nodes containing any combination of suffix and prefix. In the section tree shown in Figure 5 (b), the "Path Params" and "Body Params" are selected. Then the module designates the child nodes (only non-title nodes) of the selected title nodes as the candidate nodes for corresponding parameters (i.e., path parameters and body parameters) and pushes them into the node pool. In Figure 5 (b), the nodes from index 6 to 8 become candidate nodes for path parameters.

An API document typically provides examples of request messages (e.g., CURL [22] commands and JSON objects) along with the descriptions of API specifications. Since such examples have crucial information for the message format, the module should identify them. Identifying examples (i.e., specific code) from the API document is not a new topic, so we can just adopt a regular expression-based approach that is similar to prior work in this field [44]. The module

TABLE 3. Linguistic units used for text expressing the components of a parameter.

Information	*Linguistic unit*	*Example*
Parameter name	Noun	Selector, Power
Value type	Predefined noun or noun phrase	Integer, Array of Integer
Required flag	Predefined noun	Required, Optional
Description	Noun phrase, complete sentence	"The power state you want to set on the selector. 'on' or 'off'"

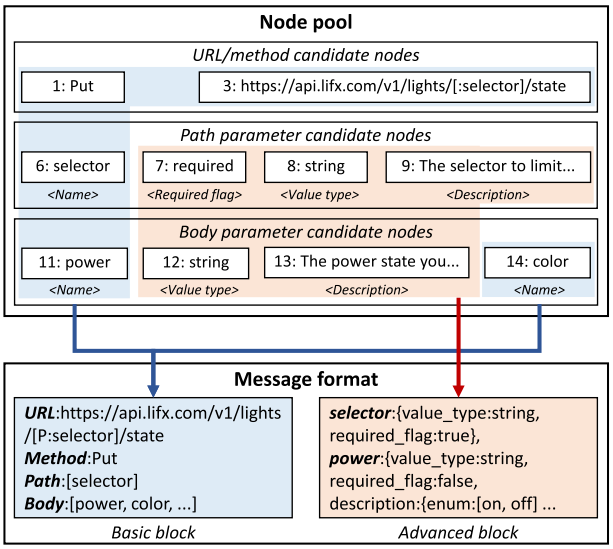


FIGURE 6. Example of the message format generation step. The fuzzing module creates a basic block with the nodes highlighted in blue and an advanced block with red ones.

identifies tree nodes containing examples and assigns them as example candidate nodes.

B. MESSAGE FORMAT GENERATION

While the previous step centers on locating the candidate nodes for a specific API, in this step, the module generates a message format for the API based on the candidate nodes. The lower part of Figure 6 illustrates the message format created from the section tree. The message format comprises a basic block, which contains the requirements for a valid input message, and an advanced block, which elaborates on the message format with more detailed conditions.

1) CREATING BASIC BLOCK

The basic block includes the essential parts of the valid input messages, so the module begins by analyzing the candidate nodes for the URL and HTTP methods from the node pool. Filling the HTTP methods is straightforward because we just adopt the value from the HTTP method candidate nodes (e.g., PUT, DELETE). In contrast, the URL part typically uses a local IP address of the device as a hostname to call the API. However, the hostname are not included in API documents for security reasons, meaning that they are represented as symbolic values (e.g., <IP address>/api/light). For this reason, the module receives the hostname from the administrator and replaces the symbolic values in the URL. In general, all the APIs of one IoT device use the same hostname, so it receives the hostname only once from the administrator during the entire testing process. Note that FuzzDocs performs a ping test to verify whether the hostname is valid and sends alerts if the test fails.

Other symbolic values inside the URL could be the value of a path or query parameter, and these values are typically denoted via the following syntactic symbols [45].

- Path parameter: '{}', '[]', '()', '<>', and ':'.
- Query parameter: '?' and '&'.

By parsing the above syntactic symbols, the module replaces these values with its own symbols ([`P:name`], [`Q:name`]) to determine their values when transmitting input messages. For example, the symbolic value '/api/light/[selector]' will be replaced with '/api/light/[P:selector]' because the selector is surrounded by '[]', which means it is the path parameter.

Next, the module analyzes the parameter candidate nodes to find the names of each parameter used for the input messages calling the API. As shown in the upper part of Figure 6, the parameter candidate nodes offer plenty of information about what the names and value types are, but we have no idea yet which node is related to which parameter names. To address this problem, the module leverages the linguistic units in a node's text. As shown in Table 3, the linguistic units in the text of the node vary depending on what kind of parameter information the text represents. For example, if a node's text is a pre-defined keyword used for data types (e.g., string, integer, or array), it indicates a parameter's value type. Similarly, if a node is a noun and not a pre-defined keyword, the node is the name of a specific parameter. Based on these findings, the module locates nodes containing a parameter's name and feeds the names to the basic block.

2) CREATING ADVANCED BLOCK

The elements in the advanced block are usually the value types, required flags (e.g., optional or mandatory), and a range of allowed values that are used in the parameters. To fill this block, the module revisits the parameter candidate nodes. In the pool, we have marked the nodes with a parameter name, but we should still determine which parameter name the remaining nodes belong to. In general, when writing an API document, the author first gives a parameter's name and then arranges other information about that parameter. Thus, we leverage the index (*idx*) of the node we assigned when building the section tree in the previous step. As shown in Figure 6, the nodes '11: power' and '14: color' are the names of body parameters. The nodes between them (i.e., '12: string' and '13: The power state you…') belong to the node '11: power' because they are not the name and have a lower index than the next parameter name node, '14: color'. In this manner, the module inserts each parameter's information (e.g., value types and descriptions) into the advanced block.

TABLE 4. Examples of target relations triples of the ontology of FuzzDocs and the results. After preprocessing, the relation triples of the descriptions are extracted based on the target (*subject; relation; object*) formats in the ontology.

Description (before preprocessing)	*Target relation triple format example*	*Result relation triple example*
The brightness level from 0.0 to 1.0.	(<*noun*>; is from; <*numbers* with *adposition*>)	(brightness level; is from; 0.0 to 1.0)
Both x and y must be between 0 and 1.	(<*noun*>; is between; <enum* of *numbers*>)	(x; be between; 0 and 1), (y; be between; 0 and 1)
It can take values of 11, 15, 20 or 25.	(<*noun*>; can take; <enum of *nouns* or *noun phrases*>)	(It; can take; values of 11, 15, 20 or 25.)
Action is one of disabled, relay_on, relay_off	(<*noun*>; one of; <enum of *nouns* or *noun phrases*>)	(Action; one of; disabled, relay_on, relay_off)
IP address of the proxy server being used.	(<*noun*>; IP address of; <*noun*>)	(It; is IP address of; proxy server)

*enumeration

While the description of the parameter contains crucial information for creating effective input messages [46], achieving such information is not straightforward because a human-readable description is hard for a machine to understand. For instance, from the description node ''13: The power state you want to set on the selector. 'on' or 'off'' in the node pool, we should identify the fact that the value is one of the predefined strings ('on' or 'off'), as shown in the lower part of Figure 6. To address this task, we adopt natural language processing (NLP) to syntactically and semantically understand the description and precisely extract the conditions and formats of the valid input message.

First, if a description is an incomplete sentence, the module makes the sentence complete as the preprocessing by leveraging Stanford CoreNLP [47]. Then, the preprocessed description is used to extract detailed information about the target parameter. Here, we need to understand the semantic relationship between the parameter and a word or a phrase meaning the conditions of the valid inputs. Thus, we apply the methods of named entity recognition (NER) and relation extraction (RE) tasks utilizing Stanford OpenIE [48], a popular language analyzer for extracting open-domain relation triples represented as (*subject*; *relation*; *object*).

By analyzing hundreds of API documents, we build an ontology of relations based on PoS (part-of-speech) tags and phrases for specific relations, which represent conditions of valid inputs for given parameters, such as the range of numeral values, enumeration of valid values, and exceptional formats of values (e.g., IP/MAC address, timestamp). Based on the ontology, the module extracts the conditions by semantically analyzing descriptions based on the ontology and adds the conditions to the advanced block. Some examples of ontology maintained by FuzzDocs and extracted relation triples from some API documents are shown in Table 4.

3) COMPLEMENTING MISSING INFORMATION

Lastly, in the case of the example candidate nodes, those nodes offer concrete values about a URL, HTTP method, or some parameters. The module uses such information to supplement missing API specifications that were not identified in the previous steps. For example, when example candidate nodes have CURL commands, the module extracts the correct URL and HTTP method from the commands. Then, it uses the extracted information if it fails to locate URL/HTTP method candidate nodes. In the case of JSON object for the request body, we could obtain body parameters' names and value types. This information is used when the module fails to identify or analyze body parameter candidate nodes.

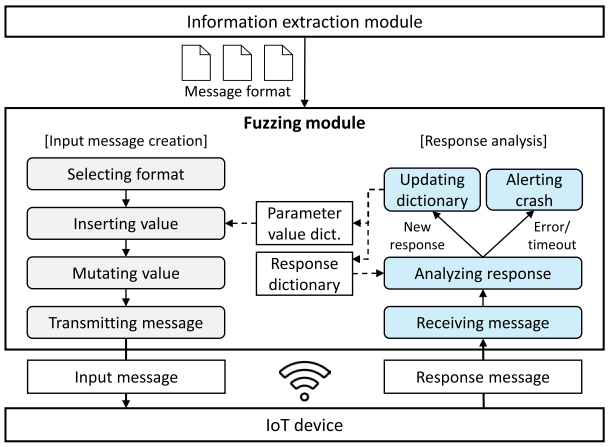


FIGURE 7. Workflow of the fuzzing module. In the input message creation step, the module creates input messages by using the message formats and transmits them to the IoT device. It then analyzes the response messages to detect potential vulnerabilities in the next step.

VI. FUZZING MODULE

We have discussed creating message formats by parsing the API documents of the IoT device. With these message formats, the fuzzing module can examine security vulnerabilities in the target IoT device by conducting the following two steps as shown in Figure 7: (i) input message creation and (ii) response analysis.

A. INPUT MESSAGE CREATION

The left part of Figure 7 shows input message creation. Note that each message format received from the information extraction module is matched one-to-one with an API of the target IoT device. The fuzzing module first selects one of the message formats according to a user-configurable scheduling algorithm (e.g., a round robin). Then, the module uses the message format as a template and fills the input message with the basic block (e.g., URL, parameter names) without specific values.

By default, the input message contains all available parameters. The module randomly discards some parameters that do not have required flags by referencing the advanced block. In addition, to trigger errors in the device's message parsing phase, the module intentionally randomly iterates specific

parameters in the input message. Next, the fuzzing module determines the specific value of each parameter in the input message by leveraging the message format's advanced block. For this, the module maintains a finite dictionary, named a *parameter value dictionary*, that contains the values guided by the advanced block. The module searches the parameter name from the dictionary first, and if there is no value for the parameter, the module creates one initial value and stores it in the dictionary. For instance, in the case of the ''brightness'' parameter shown in Figure 6, the module creates one of the double-type values that are from 0.0 to 1.0 as the advanced block describes, and it writes that value in the message. Also, the parameter dictionary is dynamically updated during the fuzzing test based on the feedback from the target device, which will be discussed in Section VI-B.

In this way, the fuzzing module takes the parameter values from the parameter value dictionary and inserts them into the new input message. After that, the module randomly selects a subset of parameters to mutate instead of mutating all the parameters in the message format. Note that the administrator can specify which parameters are not to be mutated through settings if desired. The module mutates the selected parameters according to their value type and the descriptions specified in the advanced block using the following strategies.

- **String type:** The module changes the contents and length of the string type parameter by adding random strings to trigger errors related to out-of-bound access (e.g., buffer overflows). If the advanced block of the parameter contains the range of its length (N), the module can set the parameter to a string of boundary value (e.g., N+1 or N-1) length or a string of very long length. It also replaces the value with an empty string or a numeric value to trigger misinterpretation and null-pointer de-reference errors.
- **Array type:** Similar to string type parameters, the module modifies contents of an array type parameter by adding or removing arbitrary elements. If the advanced block has a list of allowed elements (i.e., an enum) for the parameter, the module repeatedly inserts elements in that list into the parameter, or else inserts random elements.
- **Numeric type:** The module mutates the value into boundary cases (specifically guided by the advanced block) or extreme values (e.g., INTMAX). It also replaces the value with a random string or random object.
- **Object type:** For the object type parameter (e.g., JSON), the module applies the above strategies to each member value of the parameter in a recursive manner.

Finally, the module assembles the input message for the API based on the concretized format and sends it to the target device over the network.

B. RESPONSE ANALYSIS

After transmitting the created input message, the module listens to the response from the target device to know whether the message causes unexpected errors (e.g., crashes) that can lead to security vulnerabilities. Given that we cannot instrument the device's internal state locally, the fuzzing module automatically analyzes the response message to the corresponding input message remotely and determines whether the input message caused errors. The right of Figure 7 shows the response analysis step. In particular, the module considers an input message to be potentially error-prone if any of the following conditions are met.

- **Timeout:** Before testing an API, the module transmits a normal request message (i.e., without mutation) 10 times with an interval of 1 second, then calculates the average of the response time to set a timeout for that API. Next, when conducting the actual testing for the API, it sends an input message three times because the network environment can affect the timeout. When the timer expires three times, the module gives an alert that a DoS vulnerability has been triggered.
- **Connection Lost:** If the device abruptly closes an active connection, the module considers it a sign that the device has fallen into the error state. The module monitors for cases where the device sends TCP *RST* packets in response to the input message.
- **HTTP Internal Server Error (500):** If the device returns a response message with Internal Server Error (500), it indicates that the device has entered an error state due to the input message.

Coverage Feedback Mechanism. Generally, a fuzzer optimizes the input mutation process based on the feedback of executions to find security vulnerabilities more effectively. In the absence of a feedback mechanism, fuzz testing could be blind during input mutation (creation). However, the fuzzing module in FuzzDocs optimizes the input message creation step by using the response message from the target device as feedback, which improves the effectiveness of vulnerability discovery. As shown in Figure 7, the module maintains a response dictionary that stores the contents of the response messages in previous input message transmissions. Thus, whenever the module sends an input message, it stores the contents of the corresponding response message (e.g., status code, keys of a JSON body) in the dictionary. If an input message causes a new response message that does not match the response dictionary, this new response message indicates that a new code block has been executed in the device firmware. Then, the module inserts all the values of parameters in the input message into the parameter value dictionary as well. The inserted values are utilized as starting points for the next input message creation, which could directly explore more code coverage.

VII. EVALUATION

In this section, we present the prototype implementation of FuzzDocs and evaluate it in real-world environments to demonstrate its effectiveness and efficiency.

A. EVALUATION ENVIRONMENT

1) IMPLEMENTATION

FuzzDocs currently includes two main modules: the information extraction module and the fuzzing module. The information extraction module leverages BeautifulSoup4 [49], a representative HTML parsing library to extract HTML tags and texts of the input API documents. For the fuzzing module, we utilize FuzzingBook [50], a Python data fuzzing library, to mutate parameters in test messages according to our mutation strategies. Especially, in the response analysis step, we employ Tcpdump [51], a network traffic monitoring tool, to record the communication traffic between FuzzDocs and the target IoT device. In summary, to support the design features described in Section IV, we implemented FuzzDocs in approximately 3,000 lines of Python code.

2) TEST ENVIRONMENT

IoT devices typically require initial settings such as network connection or user authentication to remotely call their functionalities over the network. Also, some device APIs can be invoked only when the pre-issued authentication token is included in the input messages. Thus, we complete all such necessary initial setups for the target devices using documents and companion apps provided by the device vendors. The prototype of FuzzDocs operates on a commodity Linux server with Intel Xeon 2.1 GHz CPU and 64 GB RAM. We connect the prototype and all the test devices to the same local Wi-Fi router to better capture the network traffic between them.

3) IoT DEVICES SELECTION

IoT devices use various communication channels such as Wi-Fi or Bluetooth, but for convenience, we will only test the devices that communicate via Wi-Fi in this experiment. Also, the API documents of the device to be tested should be publicly available. Thus, as shown in Table 5, we selected five popular IoT devices from different types of home automation services, including a smart bulb and a smart plug. These devices communicate via Wi-Fi, and their API documents are publicly accessible, so anyone can acquire the API documents of these devices from their official websites [15], [36]. At the time of writing the paper, the firmware of all devices is the latest version.

4) BENCHMARK FRAMEWORKS

The goal of FuzzDocs is to test the security of IoT devices in an automated manner without firmware analysis or intervention by security experts, such as analyzing message formats and creating seed messages. To fairly demonstrate FuzzDocs's performance in finding crashes and message format generation, we compared FuzzDocs with the following network-based fuzzers [26], [27] that can run without seed messages (or manual definition of message formats).

- **Doona** [27]: Doona is an extended version of Bruteforce Exploit Detector (BED) designed to find memory-related bugs such as buffer overflows and format string bugs in network protocol implementations. It finds vulnerabilities by mutating general request/response packets without seed messages.
- **Ffuf** [26]: Ffuf is a popular network application fuzzer that supports various protocols, including HTTP and HTTPS. Unlike Doona, Ffuf receives a predefined word list to mutate request messages. We used the word list provided by Ffuf's official website.

We believe this comparison is very reasonable, since both frameworks and FuzzDocs do not require manual effort by security experts. There are many other network-based fuzzing frameworks for IoT devices, such as AFLNET [8] and Boofuzz [12]. However, since they require firmware instruments (AFLNET) or manually created input formats (Boofuzz), it is unfair to employ those tools as benchmark tools for FuzzDocs.

B. EFFECTIVENESS

1) DISCOVERING CRASHES

As IoT devices typically operate on limited hardware resources, sending extremely input messages to the device can make the experimental results worthless. Taking this into account, we limited the maximum number of input messages the three frameworks send to IoT devices to 10 per second during the entire experiment process, and then we inspect each API for an hour. As a result of our experiments, FuzzDocs discovered 35 crashes in a total of nine APIs, including two DoS vulnerabilities (Shelly Plugs and Duo) as summarized in the right part of Table 5.

In particular, FuzzDocs discovered at least one crash on every single device used in the experiment, and in some cases, it found multiple crashes in one API (e.g., Shelly PlugS, shown in Table 5). This means that based on the created message formats,FuzzDocs successfully generated multiple input messages with different combinations of the parameters, which led to causing such distinct crashes in each API. Also, FuzzDocs only spent about 10 minutes on average before finding a crash in each vulnerable API. In contrast, Ffuf and Doona did not find any crashes other than the one found on the Hue bridge. This is because, unlike FuzzDocs, these frameworks create input messages without considering the message formats of IoT devices. Most of the input messages created by them are filtered during the message parsing stage of the IoT devices before triggering bugs deep inside the IoT devices. As a result, FuzzDocs can provide more comprehensive and broad test coverage than existing network-based fuzzers by automatically creating message formats for IoT devices.

2) CASE STUDY

One of the interesting results is a crash that led to a denial of service vulnerability discovered in Shelly PlugS, which is a smart plug device. The smart plug is a key device in home automation services and provides residential energy

```
GET /settings?timezone=seoul&lng=180&mqtt_recon
ect_timeout_min=2&mqtt_pass=[{u'adminadminadmina
dminadmin': u'adminadminadminadminadmin'}]&
coiot_enable=1&mqtt_max_qos=16777215&coiot_peer=
&mqtt_user=admin HTTP/1.1
Host: 192.168.0.102
```

FIGURE 8. **Partial input message created by FuzzDocs that causes the device denial of service (DoS) state. Parameters highlighted in red are mutated and have invalid values.**

monitoring and power control functions, allowing users to remotely monitor and control the power of various electronic products. Although the Shelly PlugS only provides simple on/off functions with monitoring features and timers, it plays a crucial role in the safety of home automation: it controls the power of any appliance that plugs into it. Thus, adversaries can turn on and off all the appliances connected to it and make the user unable to control the appliances.

To test this device, FuzzDocs automatically generated the input message shown in Figure 8 in about 1,800 attempts. The input message targeted the device setting API and mutated the parameters '`mqtt_pass`', '`mqtt_max_qos`', and '`coiot_peer`'. For example, the 'coiot_peer' parameter should have string values, but FuzzDocs mutated it to an empty string. The empty string could cause a null pointer de-reference vulnerability during API processing [52]. As a result, after the input message was delivered to the device, the device could not respond to any input messages until we manually rebooted it, which caused a DoS.

C. EFFICIENCY

1) ACCURACY OF MESSAGE FORMAT GENERATION

One important indicator of evaluating the efficiency of FuzzDocs is automation. FuzzDocs automatically constructs a message format for each API to create an valid-enough input message, so we evaluated the accuracy of message formats. First, we manually compared all the message formats generated by our framework (i.e., 135 APIs) with the original contents of the API documents. Specifically, when comparing them, we examined whether the URL, HTTP method, and parameters created by FuzzDocs were correct.

As shown in Table 6, we confirmed that FuzzDocs extracted the correct URLs and HTTP methods for all the tested APIs. In the case of the parameters, it successfully obtained the correct names, value types, and descriptions for 93% of the APIs, and it only failed to get information for nine APIs. Manual analysis of these failed cases discovered that the API documents of these APIs had no title for the parameter section. We believe the authors of the API documents intentionally omitted the titles for conciseness, as the descriptions of these APIs were short and explicit. Even without the titles, FuzzDocs could create some restricted message formats with a subset of parameters based on the concrete examples in API documents. It created these restricted message formats for seven out of the nine APIs. Therefore, FuzzDocs can successfully generate message formats in various types of documents, except where intentional omission has occurred.

Section trees created by FuzzDocs from the 135 APIs generally had a depth of 4 to 6 and fewer than 100 nodes. The overhead of the message format generation is proportional to the size of the section trees. That is, FuzzDocs can create accurate message formats with negligible overhead.

2) COVERAGE OF API

Next, we evaluated the coverage of the API, which is important for meeting the requirements of an intelligent assessment system. To this end, we compared the API coverage of FuzzDocs with the ones of existing studies [9], [11], which create input messages using companion apps. One especially noticeable point is that these previous studies only tested APIs that could be called through the UI of companion apps and required manual interaction with the UI at the initial stage.

For the comparison, we calculated the number of APIs that could be called from companion apps by first referring to the previous studies. We installed each device's companion app and a traffic monitoring tool on our test mobile phone (Android OS). The monitoring tool started to collect all traffic sent from the mobile phone to the devices as soon as the companion apps launched. We manually simulated all UI inside each companion app according to the vendor-provided manuals, as in the previous studies. After the simulation, we obtained the number of APIs by analyzing the captured traffic. Shelly devices provide additional web-based UI with the companion app, so we conducted the same simulation for the web UI and merged the results.

Figure 9 shows the experimental results. Overall, the companion apps were able to call fewer APIs than listed on the API documents (see Figure 9 (a)) because most of the APIs not available in companion apps are developer-only APIs. Developer-only APIs are used to modify the internal state of devices, so they are not invoked through the companion apps to prevent user error.

In addition to the available APIs, the number of parameters that the companion apps use for each API is also restricted, as shown in Figure 9 (b). For example, when calling the '`/light/`' API of the Shelly Duo, the companion apps cannot use two of the seven parameters listed in its API document, '`effect`' and '`transition`'. This lack of available APIs and their parameters significantly degrades the coverage of the companion-app-based fuzzing. In contrast, FuzzDocs can test any API in the API documents without parameter restrictions.

We did attempt to analyze the Hue companion app's traffic, but we could not obtain correct results because the traffic was encrypted and we were unable to decrypt it. Instead, we manually compared all the UIs in the Hue companion app with the APIs described in the API document. Similar to other devices, we could not modify the device's internal settings through the app's UI. For example, the configuration API [53] of Hue devices can change various internal settings, such as the device's IP address or proxy server, but we could

TABLE 5. Summary of IoT devices under testing and the crashes identified by FuzzDocs and existing network-based fuzzing frameworks (Ffuf [26], Doona [27]). The number in parentheses on the crash field indicates the number of APIs for which a crash was found.

Vendor	*Device model*	*Device type*	*Number of APIs*	*FuzzDocs*		*Ffuf [26]*		*Doona [27]*	
				Crash	*Avg. time*	*Crash*	*Avg. time*	*Crash*	*Avg. time*
Phillips	Hue bridge V3	IoT hub	49	4 (2)	28 mins	1 (1)	3 mins	1 (1)	2 mins
Shelly	PlugS V3	Smart plug	22	12∗ (2)	5 mins	0 (0)	N/A	0 (0)	N/A
Shelly	Button1	Smart button	21	11 (2)	11 mins	0 (0)	N/A	0 (0)	N/A
Shelly	Duo RGB	Smart bulb	23	6∗ (2)	3 mins	0 (0)	N/A	0 (0)	N/A
Shelly	H&T sensor	Smart sensor	19	2 (1)	5 mins	0 (0)	N/A	0 (0)	N/A

∗Including a DoS vulnerability.

TABLE 6. Accuracy of message formats created by FuzzDocs.

Model	*URL*	*HTTP method*	*Parameter*
Hue bridge V3	49 / 49	49 / 49	43 / 49
Shelly PlugS	22 / 22	22 / 22	21 / 22
Shelly Button1	21 / 21	21 / 21	20 / 21
Shelly Duo	23 / 23	23 / 23	22 / 23
Shelly H&T	19 / 19	19 / 19	19 / 19

not change these settings in the companion app. This means that the companion app has restrictions on calling developer-only APIs. FuzzDocs can test APIs and parameters that are not available in companion apps, so it provides a more comprehensive and wider range of testing coverage than existing studies.

VIII. LIMITATIONS AND DISCUSSIONS

Although FuzzDocs can effectively examine the security of IoT devices in various types, it still has room for future work. Here, we discuss the limitations of the current design and suggest ways to improve FuzzDocs.

A. DOCUMENT TYPES

FuzzDocs utilizes vendor-provided API documents for security testing. Currently, our system focuses on processing HTML documents because most IoT vendors publish API documents on the Internet in HTML formats. However, our document analysis method (the section tree) can be easily applied to other document formats because it is designed to leverage the hierarchical relationships between texts inside API documents. In other words, simply by appending the method for extracting text and its metadata from API documents, FuzzDocs could support various document formats (e.g., PDF and DOC). For example, we can extract API specification from API documents in PDF formats by leveraging PDF parsing libraries [54].

B. API IMPLEMENTATION

As HTTP-based APIs are the most common API implementation in modern IoT devices [19], FuzzDocs centers on IoT devices that use HTTP-based APIs. This design choice was decided based on the ease of implementation and evaluation. Note that other API implementations, such as MQTT [55] and

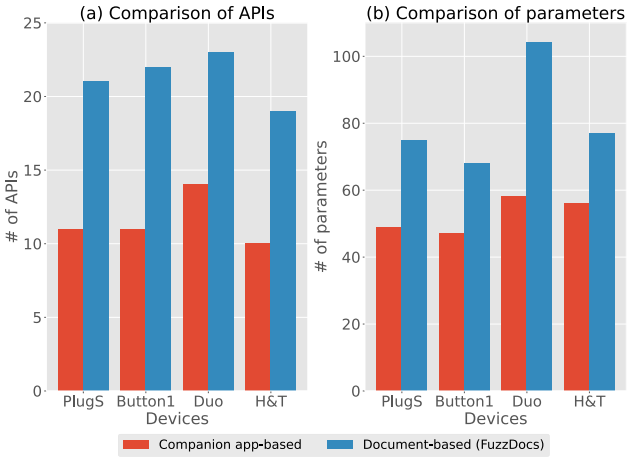


FIGURE 9. Comparison results of the number of APIs and parameters available in companion apps and API documents. FuzzDocs can provide more comprehensive test coverage than the app-based solution in terms of the number of APIs and parameters.

COAP [56], are all similar in that they operate by exchanging request/response messages with various parameters, except for the underlying network protocols and structures for transmitting messages. This means that FuzzDocs can be easily extended to other API implementations by modifying the method for assembling messages.

C. NESTED JSON PARAMETERS

For now, FuzzDocs cannot extract message parameters with nested JSON objects from descriptions in API documents. However, this limitation does not mean that FuzzDocs cannot use nested JSON-type parameters for security testing at all. If API documents contain concrete examples for the nested JSON parameters, FuzzDocs can extract the parameters from the examples and mutate them for testing. We plan to resolve this limitation in our future work.

IX. CONCLUSION

In this work, we have proposed FuzzDocs, the first document-based black-box IoT testing framework. To achieve more automatic and effective security testing for IoT devices, section-tree-based document parsing enables FuzzDocs to extract API specifications from human-readable API documents and creates valid-enough input messages based

on them. Implementing the prototype of FuzzDocs and evaluating it in real-world IoT environments showed that FuzzDocs outperformed existing frameworks in testing coverage and uncovered 35 potential vulnerabilities in five IoT devices, including two DoS vulnerabilities.

ACKNOWLEDGMENT

(Myoungsung You and Yeonkeun Kim contributed equally to this work.)

REFERENCES

[1] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 636–654.

[2] E. Ronen and A. Shamir, "Extended functionality attacks on IoT devices: The case of smart lights," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroSP)*, Mar. 2016, pp. 3–12.

[3] R. Graham, "Mirai and IoT botnet analysis," in *Proc. RSA Conf.*, 2017, pp. 1–63.

[4] J. Kim, J. Yu, H. Kim, F. Rustamov, and J. Yun, "FIRM-COV: High-coverage greybox fuzzing for IoT firmware via optimized process emulation," *IEEE Access*, vol. 9, pp. 101627–101642, 2021.

[5] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for Linux-based embedded firmware," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2016, p. 1.

[6] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, and K. R. Butler, "FIRMWIRE: Transparent dynamic analysis for cellular baseband firmware," in *Proc. NDSS*, 2022, pp. 1–19.

[7] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *Proc. USENIX Secur. Symp. (USENIX Security)*, 2019, pp. 1099–1114.

[8] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *Proc. IEEE 13th Int. Conf. Softw. Test., Validation Verification (ICST)*, Oct. 2020, pp. 460–465.

[9] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[10] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 337–350.

[11] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 484–500.

[12] *Boofuzz: Network Protocol Fuzzing for Humans*. Accessed: Sep. 22, 2022. [Online]. Available: https://boofuzz.readthedocs.io/en/stable/

[13] *Bed—A Network Protocol Fuzzer*. Accessed: Sep. 22, 2022. [Online]. Available: https://www.kali.org/tools/bed/

[14] Royal Philips Electronics. *Philips Hue*. Accessed: Sep. 22, 2022. [Online]. Available: http://www2.meethue.com/en-us

[15] *Shelly Cloud*. [Online]. Accessed: Sep. 22, 2022. Available: https://shelly.cloud/

[16] X. Wang, Y. Sun, S. Nanda, and X. Wang, "Looking from the mirror: Evaluating IoT device security through mobile companion apps," in *Proc. USENIX Secur. Symp. (USENIX Security)*, 2019, pp. 1151–1167.

[17] H. Garg and M. Dave, "Securing IoT devices and securely connecting the dots using REST API and middleware," in *Proc. 4th Int. Conf. Internet Things, Smart Innov. Usages (IoT-SIU)*, Apr. 2019, pp. 1–6.

[18] J.-Y. Yu and Y.-G. Kim, "Analysis of IoT platform security: A survey," in *Proc. Int. Conf. Platform Technol. Service (PlatCon)*, Jan. 2019, pp. 1–5.

[19] P. P. Ray, "A survey of IoT cloud platforms," *Future Comput. Informat. J.*, vol. 1, nos. 1–2, pp. 35–46, Dec. 2016.

[20] *Shelly Plugs: The WiFi Smart Plug That Fits Everywhere*. Accessed: Sep. 22, 2022. [Online]. Available: https://shelly.cloud/products/shelly-plug-s-smart-home-automation-device/

[21] *Shelly Cloud API*. Accessed: Sep. 22, 2022. [Online]. Available: https://shelly-api-docs.shelly.cloud/gen1/

[22] *Curl: Command Line Tool and Library for Transferring Data With URLs*. Accessed: Sep. 22, 2022. [Online]. Available: https://curl.se/

[23] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "FirmFuzz: Automated IoT firmware introspection and analysis," in *Proc. 2nd Int. ACM Workshop Secur. Privacy Internet-of-Things*, 2019, pp. 15–21.

[24] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2020, pp. 733–745.

[25] C. Song, B. Yu, X. Zhou, and Q. Yang, "SPFuzz: A hierarchical scheduling framework for stateful network protocol fuzzing," *IEEE Access*, vol. 7, pp. 18490–18499, 2019.

[26] *FFuF—Fuzz Faster U Fool*. Accessed: Sep. 22, 2022. [Online]. Available: https://github.com/ffuf/ffuf

[27] *Doona—Network Fuzzing Tool*. Accessed: Sep. 22, 2022. [Online]. Available: https://github.com/wireghoul/doona

[28] Z. Shu and G. Yan, "IoTInfer: Automated blackbox fuzz testing of IoT network protocols guided by finite state machine inference," *IEEE Internet Things J.*, early access, Jun. 13, 2022, doi: 10.1109/JIOT.2022.3182589.

[29] *American Fuzzy Lop*. Accessed: Sep. 22, 2022. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[30] S. Vasile, D. Oswald, and T. Chothia, "Breaking all the things—A systematic survey of firmware extraction techniques for IoT devices," in *Proc. Int. Conf. Smart Card Res. Adv. Appl.* Cannes, France: Springer, 2018, pp. 171–185.

[31] T. N. Phu, K. H. Dang, D. N. Quoc, N. T. Dai, and N. N. Binh, "A novel framework to classify malware in MIPS architecture-based IoT devices," *Secur. Commun. Netw.*, vol. 2019, Dec. 2019, Art. no. 4073940.

[32] C. Palmiero, G. Di Guglielmo, L. Lavagno, and L. P. Carloni, "Design and implementation of a dynamic information flow tracking architecture to secure a RISC-V core for IoT applications," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2018, pp. 1–7.

[33] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products," in *Proc. 40th Int. Conf. Softw. Eng.*, May 2018, pp. 421–431.

[34] *The LIFX Switch Range Just Got a Whole Lot Smarter*. Accessed: Sep. 22, 2022. [Online]. Available: https://api.developer.lifx.com/

[35] P. A. Ly, C. Pedrinaci, and J. Domingue, "Automated information extraction from web APIs documentation," in *Proc. Int. Conf. Web Inf. Syst. Eng.* Berlin, Germany: Springer, 2012, pp. 497–511.

[36] *Philips Hue*. Accessed: Sep. 22, 2022. [Online]. Available: http://www2.meethue.com/en-us

[37] *Google Cloud IoT Rest API*. Accessed: Sep. 22, 2022. [Online]. Available: https://cloud.google.com/iot/docs/reference/cloudiot/rest

[38] *Google Home Rest API*. Accessed: Sep. 22, 2022. [Online]. Available: https://rithvikvibhu.github.io/GHLocalApi/#section/Google-Home-Local-API

[39] *KAA IoT Platform Rest API*. Accessed: Sep. 22, 2022. [Online]. Available: https://docs.kaaiot.io/KAA/docs/v1.3.0/Features/Device-management/EPR/REST-API/

[40] *Microsoft IoT Hub Rest API*. Accessed: Sep. 22, 2022. [Online]. Available: https://docs.microsoft.com/en-us/rest/api/iothub/

[41] *Rest API for Oracle Internet of Things Cloud Service*. Accessed: Sep. 22, 2022. [Online]. Available: https://docs.oracle.com/en/cloud/paas/iot-cloud/iotrq/rest-endpoints.html

[42] *Losant IoT Platform Rest API*. Accessed: Sep. 22, 2022. [Online]. Available: https://docs.losant.com/rest-api/data/

[43] *Smartthings API (1.0-Preview)*. Accessed: Sep. 22, 2022. [Online]. Available: https://developer-preview.smartthings.com/docs/api/public

[44] S. Gupta, G. Kaiser, D. Neistadt, and P. Grimm, "DOM-based content extraction of HTML documents," in *Proc. 12th Int. Conf. World Wide Web*, 2003, pp. 207–214.

[45] A. Rodriguez, "Restful web services: The basics," IBM Developer Works, Armonk, NY, USA, Tech. Rep. 33, 2008.

[46] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2009, pp. 307–318.

[47] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proc. 52nd Annu. Meeting Assoc. Comput. Linguistics, Syst. Demonstrations*, 2014, pp. 55–60.

[48] G. Angeli, M. J. J. Premkumar, and C. D. Manning, "Leveraging linguistic structure for open domain information extraction," in *Proc. 53rd Annu. Meeting Assoc. Comput. Linguistics 7th Int. Joint Conf. Natural Lang. Process.*, vol. 1, 2015, pp. 344–354.

[49] *Beautiful Soup Documentation*. Accessed: Sep. 22, 2022. [Online]. Available: https://beautiful-soup-4.readthedocs.io/en/latest/
[50] *Fuzzing: Breaking Things With Random Inputs*. Accessed: Sep. 22, 2022. [Online]. Available: https://www.fuzzingbook.org/html/Fuzzer.html/
[51] *TCPDUMP and LIBPCAP*. Accessed: Sep. 22, 2022. [Online]. Available: https://www.tcpdump.org/
[52] D. Romano, M. Di Penta, and G. Antoniol, "An approach for search based testing of null pointer exceptions," in *Proc. 4th IEEE Int. Conf. Softw. Test., Verification Validation*, Mar. 2011, pp. 160–169.
[53] *Hue Configuration API*. Accessed: Sep. 22, 2022. [Online]. Available: https://developers.meethue.com/develop/hue-api/7-configuration-api/
[54] *PDF Parser and Analyzer for Python*. Accessed: Sep. 22, 2022. [Online]. Available: https://pypi.org/project/pdfminer/
[55] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S—A publish/subscribe protocol for wireless sensor networks," in *Proc. IEEE Int. Conf. Commun. Syst. Softw. Middleware Workshops (COMSWARE)*, Jan. 2008, pp. 791–798.
[56] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An application protocol for billions of tiny internet nodes," *IEEE Internet Comput.*, vol. 16, no. 2, pp. 62–67, Mar. 2012.

MYOUNGSUNG YOU received the B.S. degree in computer science from Chungbuk National University, South Korea, and the M.S. degree in information security from KAIST, where he is currently pursuing the Ph.D. degree with the School of Electrical Engineering. His research interests include programmable network data planes, cloud security, and distributed systems.

YEONKEUN KIM received the B.S. degree in computer science engineering from the Ulsan National Institute of Science and Technology (UNIST), South Korea, and the M.S. degree in information security from KAIST, where he is currently pursuing the Ph.D. degree with the Graduate School of Information Security, KAIST. His research interests include network security issues of the IoT and embedding systems.

JAEHAN KIM received the B.S. and M.S. degrees from the School of Electrical Engineering, KAIST, where he is currently pursuing the Ph.D. degree. His research interests include cyber threat intelligence, natural language processing, and data mining.

MINJAE SEO received the B.S. degree in computer engineering from Mississippi State University, and the M.S. degree from the Graduate School of Information Security, KAIST. He is currently a Researcher at the National Security Research Institute, Daejeon, South Korea. His current research interests include software-defined networking security, network fingerprinting, and deep learning–based network systems.

SOOEL SON (Member, IEEE) received the Ph.D. degree from the Department of Computer Science, The University of Texas at Austin. He is an Associate Professor at the School of Computing, KAIST. He is working on various topics regarding web security and privacy.

SEUNGWON SHIN (Member, IEEE) received the B.S. and M.S. degrees from KAIST, both in electrical and computer engineering, and the Ph.D. degree in computer engineering from the Electrical and Computer Engineering Department, Texas A&M University. He is an Associate Professor at the School of Electrical Engineering, KAIST. His research interests include software-defined networking security, dark web analysis, and cyber threat intelligence.

SEUNGSOO LEE received the B.S. degree in computer science from Soongsil University, South Korea, and the M.S. degree in information security from KAIST, and the Ph.D. degree in information security from KAIST, in 2020. He is an Assistant Professor at the Department of Computer Science and Engineering, Incheon National University. His research interests include secure and robust SDN controllers and protecting SDN environments from threats.