Hindawi
Security and Communication Networks
Volume 2022, Article ID 9788219, 14 pages
https://doi.org/10.1155/2022/9788219

Research Article

PDFuzzerGen: Policy-Driven Black-Box Fuzzer Generation for Smart Devices

Yixuan Cheng,[1,2] **Wenqing Fan**,[1,2] **Wei Huang**,[1,2] **Gaoqing Yu**,[1,2] **Yu Han**,[1,2] **Hang Dong**,[3] **and Wen Liu**[1,2]

[1]*State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing, China*
[2]*School of Computer and Cyber Sciences, Communication University of China, Beijing, China*
[3]*China Mobile Communications Co., Ltd., Beijing, China*

Correspondence should be addressed to Wen Liu; lw8206@cuc.edu.cn

Received 16 December 2021; Revised 10 March 2022; Accepted 29 March 2022; Published 27 April 2022

Academic Editor: Tao Zhang

Black-box fuzzing is a testing technique to find both known and unknown vulnerabilities in software. When applying black-box fuzzing to smart devices, the main idea is to take a smart device as a black box and provide random input through a network-based interface, such as a Web interface. Due to the diversity of Web interface implementations and complex data format, a blind mutation of the message makes the message unable to pass the verification of the device component. Therefore, each Web interface needs a unique fuzzer, which precisely defines a message format of the target interface, a state maintenance method, the field positions to be mutated, and a specific input mutation method. At the time of writing, a fuzzer is completely developed by a security engineer. To save human labor, we present PDFuzzerGen, a tool to automatically synthesize complex black-box fuzzers for smart devices. PDFuzzerGen generates multiple fuzzing policies by analyzing raw messages and then synthesizes fuzzers based on policies. PDFuzzerGen requires no human intervention and can be applied to a wide range of smart devices. Furthermore, the generated fuzzers can expose bugs and flaws that rest deep in smart devices. PDFuzzerGen was evaluated to generate fuzzers for 19 different smart devices from 6 vendors. It has found 14 previously unknown vulnerabilities, 5 of which were confirmed and disclosed by the China National Vulnerability Database (CNVD) and 2 of which were confirmed and disclosed by Common Vulnerabilities and Exposures (CVE). The generated fuzzers outperform some manually crafted fuzzers on a few metrics, including the vulnerability detection rate and time cost of a newly developed fuzzer, which demonstrates the effectiveness and efficiency of PDFuzzerGen.

1. Introduction

In recent years, smart devices have become an indispensable part of our life. Such devices include smart routers, cameras, door locks, light bulbs, and so on. By 2020, every person on this planet has four smart devices on average [1]. While these smart devices enrich our lives, unfortunately, they also introduce security risks in the form of vulnerabilities. A notorious example is the Mirai [2] botnet, which exploited a bunch of vulnerabilities on millions of smart devices worldwide (including routers and webcams) and launched a large-scale denial of service attack. In the work-from-home scenarios during COVID-19, Trend Micro has reported that introducing vulnerable smart devices to the household will expose employees to malware and attacks that could slip into a company's network [3]. Therefore, it is crucial to discover the vulnerabilities in smart devices before deploying them in use.

Vulnerabilities in smart devices are usually implementation flaws in the device firmware, which is a software that provides hardware support for upper-level users. In recent years, researchers have developed many novel techniques to discover vulnerabilities in smart devices by analyzing their firmware [4–8]. These approaches, however, present several limitations. The primary one is the difficulty in firmware acquisition because many vendors do not make

their firmware images publicly available [9]. The second one is the difficulty of unpacking firmware images, which may be available in a variety of compression (even with encryption) formats; most of them are undocumented [10]. The third one is the difficulty of binary analysis due to diverse underlying architectures (memory layout, instruction set, and so forth) [11]. This impedes the methods that rely on emulation for certain architectures, and it is still a challenge to analyze the binary file after firmware decompression without the knowledge of the underlying architecture [9, 11].

Due to resource computational constraints, many smart devices provide network-based administration interfaces [12]. The Web management interface is a popular instance of these network-based interfaces (Web interface for short). The Web interface usually packs user data and sends it to the device in a format defined inside the device. After receiving the message, the device parses the message, performs authentication and data format verification, and then passes the message data to a further procedure according to the different functions used [13]. If there is an implementation flaw in message parsing or further procedure, a vulnerability may be exploited [11]. So a smart device that has a Web interface can be treated as a black box. Feeding this black box with malformed messages could trigger potential vulnerabilities in the parsing process. And more importantly, feeding this black box with crafted malicious messages, which have authentication credentials and malicious parameters whose format conforms to different data exchange formats (such as XML, JSON, SOAP, Key-value pairs, etc.), can trigger in-depth vulnerabilities in the further procedure.

This approach is also called black-box fuzzing. State-of-the-art black-box fuzzing tool BooFuzz [14] facilitates smart device fuzzing through the help of an analyst [13]. The analyst writes a set of "fuzzer stubs," a set of functions that define (i) a state maintenance method, (ii) the precise position of the fuzzed parameters in a message, and (iii) specific mutation methods. An analyst must write such a set of stubs for each Web interface of each smart device tested. Interesting fuzzed parameters and how these parameters are mutated are completely determined by the analyst. While this approach mitigates the challenge of fuzzing smart devices, it relies on human knowledge of message semantics and empirical perception of message parameters that may trigger potential implementation flaws.

1.1. Our Approach. In this paper, we propose PDFuzzerGen, a black-box fuzzer generation framework, to automatically generate black-box fuzzers that can discover real-world vulnerabilities. PDFuzzerGen uses a novel technique called policy-driven fuzzer generation. As the name suggests, its main idea is to formulate multiple fuzzing policies by analyzing the raw messages of a Web interface, and a set of "fuzzer stubs" for BooFuzz is generated based on these policies. Finally, these "fuzzer stubs" are used to synthesize fuzzers, and fuzzing inputs are generated by the fuzzers to trigger deep implementation flaws in the firmware of the smart device. PDFuzzerGen automates the manual process of the analyst in creating customized fuzzers for smart devices and specific Web interfaces. Since the Web interface is accessed via a network, this approach is completely independent of the device firmware images, and it can be used to test smart devices that do not publicly release their firmware image.

In our research, we implemented a full-featured prototype of PDFuzzerGen and evaluated it in a real-world environment. To assess its effectiveness, we ran PDFuzzerGen on 19 popular smart devices from 6 vendors. PDFuzzerGen successfully discovered 14 previously unknown vulnerabilities, 2 of which were submitted and confirmed by the Common Vulnerabilities and Exposures (CVE): CVE-2021-31624 and CVE-2021-31627, and 5 of which were submitted and confirmed by the China National Vulnerability Database (CNVD): CNVD-2020-69407, CNVD-2020-67555, CNVD-2021-17400, CNVD-2021-22752, and CNVD-2021-24948. We have reported all these vulnerabilities to CNCERT/CC [15] in pursuit of helping vendors fix them.

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

(i) New technique: we developed a new technique called policy-driven fuzzer generation to automatically generate fuzzers. These policies include state maintenance policies, parameter discovery policies, parameter mutation policies, and monitoring policies.

(ii) New framework: to the best of our knowledge, we present the first firmware-independent black-box fuzzer generation framework, PDFuzzerGen, which is used to find implementation flaws in smart devices. Based on the dynamically created policies, PDFuzzerGen creates BooFuzz fuzzers for different network-based interfaces without human interaction.

(iii) Implementation and findings: we implemented a full-featured prototype of PDFuzzerGen and evaluated it on 19 real-world smart devices. In total, 14 zero-day vulnerabilities were discovered.

1.3. Roadmap. In the remainder of this article, Section 2 provides a review of the background and related work of smart device fuzzing and fuzzer generation. Section 3 presents a detailed design of PDFuzzerGen, and Section 4 covers its implementation details. The evaluation results are summarized in Section 5. Section 6 discusses some limitations of the current design and points out the future work. Finally, Section 7 concludes the paper.

2. Background and Related Work

In this section, we present the fuzzing work in the smart devices scenario and fuzzer generation, a method featured by a higher degree of automated fuzzing in recent years.

2.1. Smart Devices Fuzzing. Fuzzing methods can be classified into white-, grey-, and black-box fuzzing [16]. White-

box fuzzing requires the source code of the target. However, the firmware source code and related documents of smart devices are rarely publicly available, so white-box fuzzing is not appropriate for smart devices. The typical grey-box and black-box fuzzing methods used in smart devices in recent years are summarized in Table 1.

Unlike some methods based on program analysis [20], grey-box fuzzing is usually guided by code coverage, and it is necessary to track the executed code block during the fuzzing execution stage [10]. As a matter of the fact, this type of method needs to distinguish bare-metal devices from simulated devices. For bare-metal devices, this method relies on hardware debugging to achieve code coverage tracking [21], but this function is usually disabled in consumer devices by the manufacturers [6, 7]. For simulated devices, some researchers [5, 22–24] focus on studying how to simulate firmware so that its related services can run expectedly, such as Web services. Some fuzzing methods [6, 7, 25] carry out grey-box fuzzing based on these simulation methods. When the simulation is working, these methods can detect vulnerabilities. Unfortunately, the success rate of simulation and the range of applicable manufacturer's equipment is relatively limited [24], and many simulated devices do not support debugging [26]. At the same time, simulated devices and bare-metal devices may have some functional differences [10, 24]. Vulnerabilities in a simulated device may not be exploitable on a bare-metal device. As a matter of the fact, for those devices that cannot be successfully simulated or that cannot be debugged, the grey-box fuzzing method that relies on simulation and debugging functions is not applicable.

Black-box fuzzing mitigates the issues of grey-box fuzzing at the expense of some efficiency. Black-box fuzzing methods for smart devices can be classified into app- and protocol-based fuzzing. IoTFuzzer [11] is the first method for fuzzing from the mobile app side, which mutates test cases by identifying and reusing app-specific logic. Unlike the method of discovering component vulnerabilities in DEX files [27], IoTFuzzer is still looking for vulnerabilities in smart device firmware. Based on IoTFuzzer, DIANE [10] aims at the challenge that the generated test cases may be restricted by their own code constraints. It transfers the operation of data mutation from the first function entered by the user to the last data encoding function before sending the data to the smart device so that it can directly bypass the previous input data sanitization stage. These methods can effectively discover vulnerabilities in the code that communicates with the app on smart devices. Unfortunately, not all smart devices have corresponding mobile apps. And even if the smart device has a mobile app, this method cannot find vulnerabilities in components that do not communicate with the mobile app.

Protocol-based fuzzing usually starts from the network communication process. Unlike intrusion detection methods that focus on protocol classification, protocol-based fuzzing focuses on flawed protocol components [28]. Using communication messages as a seed, it performs fuzzing on protocol components or functional modules of smart devices through mutation or generation methods. Based on the black-box fuzzer peach, Kamel and Lanet [29] tried to find errors in some modern smart card Web server. Costin et al. [30] extended the range of the research object from smart cards to the entire commercial off-the-shelf (COTS) equipment, such as routers, webcams, and so on. This method relies on the original device firmware through decompression and static analysis of it. After that, it tries to simulate the firmware, and when the simulation is successful, it performs dynamic analysis including black-box fuzzing. Unfortunately, their method completely relies on the success of firmware simulation, which limits its usage. In addition, this method focuses more on the automation of the entire process. When performing black-box fuzzing, the protocol specification is not considered, so there are a lot of test case format errors, which makes the overall fuzzing less efficient. At the same time, their work also pointed out that many of the latest smart devices require authentication to access vulnerable modules.

SIoTFuzzer [19] is another third-party independent work while writing this paper. SIoTFuzzer chooses to start from the Web front-end page and complete stateful message generation through front-end source code review, state analysis, and seed generation. Then, the correct seed message is generated. Finally, the generated seed message is mutated to complete the fuzzing process. This method can effectively generate a raw seed message. Unfortunately, their stateful message generation requires manual pre-analysis of the test target, which limits its large-scale application. In addition, they did not discover unknown vulnerabilities.

2.2. Fuzzer Generation. Although the fuzzing method has proven to be an effective method of vulnerability detection [13, 16, 31], the development of the fuzzer still relies on experienced analysts [32], and the process takes a lot of time [13]. Some modern fuzzing frameworks, such as libFuzzer [33] and Boofuzz [14], facilitate fuzzing through the help of analysts. In the development of fuzzing, researchers need to write a small fuzzer, which is an independent program. For smart device fuzzing, the analyst must write such a fuzzer for each Web interface of each smart device. Analysts can decide for themselves which fields to fuzz, the mutation rules, and whether the authentication process is required. These methods rely on a manual in-depth understanding of different network-based interfaces and their communication processes (for BooFuzz) or internal APIs and their usage (for libFuzzer). And the larger the scale of the application and the more functional interfaces, the greater the manual overhead of writing Fuzzer. Therefore, the cost required to apply this method to many different devices is enormous.

The Google security team first realized the challenge of fuzzing complex interfaces. They tried to use fuzzer generation approach to solve it and developed FUDGE [34] and FuzzGen [32]. The goal of FUDGE and FuzzGen is to automatically generate fuzzers for the open-source libraries of the Linux distributions. FUDGE synthesizes the fuzzer by extracting the API sequence from the existing source code that calls the library function. Based on FUDGE, FuzzGen analyzes the existing source code of calling library functions,

Table 1: Grey- and black-box fuzzing methods in smart devices.

Fuzzer	Fuzzing type	Hardware support	Component	Zero-day detection
IoTFuzzer [11]	Black-box	Bare-metal	APP-related	Yes
FIRM-AFL [7]	Grey-box	Emulation	Web	Yes
FirmFuzz [6]	Grey-box	Emulation	Web	Yes
IoTHunter [17]	Grey-box	Emulation	Protocol	Yes
MultiFuzz [18]	Grey-box	None	Protocol	Yes
SIoTFuzzer [19]	Black-box	Emulation	Web	No
DIANE [10]	Black-box	Bare-metal	APP-related	Yes

infers the API call relationship of the library, and then generates fuzzers, which further improves the degree of automation. IntelliGen [35] addresses the issue of the quality of existing test cases that FuzzGen relies on calling library functions, starting from two aspects: high-value entry function and suitable parameters. It first scans the target program, leverages LLVM to locate the high-value entry function as a potential entry function, and then performs accurate parameter inference to improve performance and compatibility to solve the previous issues.

Inspired by FUDGE and FuzzGen, WINNIE [31] focuses on closed-source Windows programs. Compared with the open-source Linux library, the closed-source and GUI-based Windows software ecosystem has some new challenges. The authors first clarified five challenges of the research object: target discovery, call-sequence recovery, argument recovery, control-flow, and data-flow dependence. Then WINNIE was developed specifically to solve these challenges one by one. Their experimental results show that WINNIE can successfully circumvent some GUI limitations and can effectively find bugs in Windows GUI programs.

These approaches have proved that fuzzer generation is an effective method to improve the efficiency of fuzzing and reduce the labor expense and time consumption of manual development. These approaches are effective for open-source libraries or Windows GUI programs, but unfortunately, many smart devices vendors do not make their firmware images, source code, or documentation publicly available [9–11], and smart devices usually use Web interfaces for communication instead of windows GUI programs. So these fuzzer generation approaches cannot be applied to smart device fuzzer generation.

3. Design

In this section, we introduce the detailed design of PDFuzzerGen. Figure 1 shows an overview of the PDFuzzerGen framework. To synthesize customized fuzzers for a smart device, PDFuzzerGen requires the raw messages sent to the device as the initial seed. PDFuzzerGen first scans the raw messages and picks out some candidate functional messages for fuzzing (§3.1). The selected raw messages are passed to the policy generator for policy analysis. The policy generator formulates state maintenance policies, parameter discovery policies, mutation policies, and monitoring policies (§3.2~§3.5). Based on these generated policies, PDFuzzerGen generates a series of fuzzer stubs, which are used to synthesize a set of fuzzers (§3.6). These fuzzers can run independently and generate fuzzing messages to fuzz the target smart device and monitor the response messages to find potential device abnormalities, such as crashes.

3.1. Target Discovery. The first step of fuzzer generation is target discovery. In raw messages, there are many messages requesting static resources, which usually do not trigger deep and different function codes. Therefore, in order to generate fuzzers that can discover in-depth vulnerabilities, the goal of target discovery is to identify functional messages that can trigger deep code. These functional messages are used as seed messages for subsequent policy analysis.

Generally, typical smart device Web interface request messages consist of two types. One is for resource retrievals, such as GET and HEAD messages. The other one is for resource management, such as POST, PUT, DELETE, and so on. Resource management messages usually interact with deep functional components of smart devices and are important for fuzzer generation. Resource retrieval messages are usually used to get some static resources, pages, or device status data, which usually comply with RESTful specifications. However, some of the static resource request messages are rarely related to functional components. In addition, some devices also use resource retrieval messages, such as GET messages for functional interaction and do not comply with RESTful specifications. Accordingly, PDFuzzerGen reserves resource maintenance messages, denoted as *RMM*. At the same time, it filters resource request messages to remove the request for static resources and request messages without query parameters, denoted as RRM_f. According to (1), the set of potential functional messages, I, is

$$I = \mathrm{RMM} \cup \mathrm{RRM}_f. \tag{1}$$

3.2. State Maintenance. Many smart devices require authentication to access vulnerable components. Existing fuzzers usually require manual intervention to authenticate. In other words, an ananalyst manually logs in and infers the authentication method of the target service. After that, authentication credentials, such as cookies or tokens, are filled in a fuzzer to maintain the authenticated session state. The design goal of PDFuzzerGen is to reduce the overhead of manual intervention as much as possible, so it needs to automate the entire process from login to session maintenance. Thus, when formulating a state maintenance policy, several questions need to be answered: (i) does the target

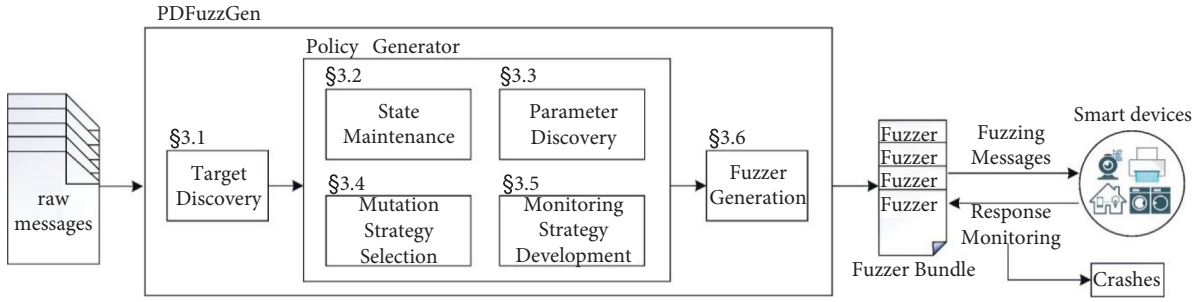

Figure 1: Overview of PDFuzzerGen.

device require authentication? (ii) Is the authentication process replayable? (iii) Can the authenticated state be maintained? We will answer these three questions from four aspects: authentication identification, replayable message identification, unified state maintenance model, and fuzzing execution type.

3.2.1. Authentication Identification. Most smart devices have the function of authentication, but there are still some devices that may not have this function, especially some simulated devices. The design goal of PDFuzzerGen is service-oriented, which means fuzzers can be generated if the Web interface can be accessed normally, regardless of whether the device is bare-metal or simulated. PDFuzzerGen implements authentication identification by identifying the login message in raw messages. PDFuzzerGen recognizes login messages from two aspects. The first aspect is to identify whether the request parameters carry authentication-related information, such as login-related string literals. The second aspect is to determine whether the request message after the login message can be responded to expectedly, including whether the status code and response content are expected.

For status codes, the main judgment is based on the definition of HTTP response status codes in RFC2616 [36]. When the status code is in an error state, the login process usually fails, but when the status code is in a success state, the login process does not necessarily succeed. This is because some Web components set a uniform status code and use a custom response result specification in the response content. Different manufacturers may customize different response result specifications, and it is very difficult to accurately identify these different specifications. However, according to our observations, most Web components have significant differences in the length of the response message for login success and failure. For example, when the login fails, a long error message or a more complex failure page will be returned, which makes the length of the response content in the case of failure significantly larger than that in the case of success. This gave us some inspiration. At the same time, common practice tells us that accessing any existing functional interface will be redirected to the login page or login interface under the condition that the authentication is not passed. Based on this practical experience, by enumerating and traversing the subsequent functional interface messages of the located authentication message, the URLs appearing in the response header and message body are counted and clustered, respectively. If they all have the same login jump request, it indicates that the previous authentication packet authentication failed. Therefore, for the response content, PDFuzzerGen comprehensively uses the above two methods to determine whether the authentication message is successful or not.

3.2.2. Replayable Message Identification. One of the important factors affecting the automated authentication of the fuzzer is whether the login message can be replayed. Based on the idea of differential analysis, when the response result of the replayed message is basically consistent with the response result of the successful login, it indicates that the login message can be replayed. After the above steps, PDFuzzerGen can locate the specific login message and determine whether the login message can complete the login process. After that, the login message is replayed N times for the target Web interface, and after each replay, it is detected whether the login is successful and whether the login status has changed. For example, it can be detected whether the cookie or authentication token has changed. If there is a change, it can be inferred that the login message can be replayed; otherwise, the target authentication is replay resistant.

3.2.3. Unified State Maintenance Model. The authentication categories for the Web interface of smart devices are usually divided into three categories: cookie-based, token-based, and session-based. Some devices also use a mixture of them. For example, both token and cookie are used for authentication.

To enable PDFuzzerGen to handle these complex scenarios at the same time, we propose a unified state maintenance model: *a sliding state maintenance model*. For the session mechanism, the state data is usually recorded on the server side, and there is usually no relevant state maintenance information in the request message. As shown in Figure 2, for the cookie mechanism, the state data is usually in the "Cookie" field in the HTTP request header, and the session state can be maintained if the relevant cookie is carried. For the token mechanism, taking a POST request as an example, the location of the token may exist in the following four locations in the request message: (i) URL, (ii) request header fields, (iii) cookie, and (iv) payload. Therefore, for the created fuzzer to dynamically obtain the

```
POST /some/url/token1/example.http HTTP/1.1
Host: example.com
Token-Param2: token2
Cokie: Token-Param3=token3

{"data_key":"data","Token-Param4":"token4"}
```

Figure 2: Example of Web interface message with authentication credentials.

Input: Login message M_l, Replay time n
Output: Potential authentication field location L_p
(1) Replayed $M_l n$ times. After each login authentication succeeds, select the same group of functional messages M_a that carry state maintenance information.
(2) A pairwise difference is performed on the message in M_a, and its potential k difference fields f_d are located.
(3) Perform sliding traversal on each difference field f_d and filter according to its location and content. When f_d is in the URL or request header fields, the difference field is inferred to be a potential authentication field f_p. When f_d is in the cookie or payload, the parameters in the cookie or payload are parsed, and the values of the same key are compared. If the length of f_d exceeds the threshold k, the field is inferred to be a potential authentication field f_p.
(4) Record the location of all f_p as L_p.

Algorithm 1: Differential Sliding Positioning.

authentication field at runtime, PDFuzzerGen needs to locate the position of the authentication field in each authentication method.

PDFuzzerGen uses the *differential sliding positioning algorithm* for the above state maintenance model to locate the position of the potential authentication field. The specific steps are shown in Algorithm 1:

3.2.4. Fuzzing Execution Type. We divide the fuzzing execution type into two categories: auto-login fuzzing and anonymous fuzzing. This is to enable PDFuzzerGen to handle the above-mentioned multiple scenarios more easily. Auto-login fuzzing refers to fuzzing after logging out. Anonymous fuzzing refers to fuzzing performed at login. When the target service does not require authentication, there is no restriction on authentication. Fuzzing can be performed at any time, so anonymous fuzzing can be applied. Then, when the target service requires authentication and the login message can be replayed, the fuzzer can automatically complete the authentication process before each test case is sent, locate the relevant authentication status field, and fill it into the mutated test case to complete the auto-login fuzzing test. Finally, when the target service requires authentication and the login message cannot be replayed, this situation will be recorded by PDFuzzerGen. In this case, the generated fuzzers will only be able to fuzz the message parsing component because it cannot pass the authentication and reach the deep-level functional code. In practice, the limitations of the last case can be mitigated to some extent, which we discuss in Section 6.

After the above process, PDFuzzerGen constructs a state maintenance policy. The information in the state maintenance policy includes authentication, message replay, authentication state maintenance field location, and fuzzing execution type.

3.3. Parameter Discovery. Through state maintenance, the prerequisites for generating fuzzers have been met. The next thing to do is to enable the generated fuzzer to generate test messages with the correct parameter format and reach deep-level functional components to achieve refined fuzzing. A typical smart device Web interface communication and message processing flow is shown in Figure 3. When an HTTP message is sent to a smart device, the Web server first parses the message, completes the authentication, identifies the structured message parameters, and then calls the parameter analysis component to analyze the structured message parameters. Finally, the extracted message variables are passed to different functional components that specifically handle these parameters. Accordingly, to discover the vulnerabilities of deep-level functional components, the structural integrity and effectiveness of the test messages must be ensured.

For Web interfaces, although in most cases resource request messages, such as GET messages are used to obtain data, resource maintenance messages, such as POST, are used to submit data. However, when each smart device vendor implements related functions, the type of request for submitting data is not fixed. Some smart devices, such as TP-LINK TL-WR940N, use GET messages to complete all data submission operations. The data formats of the request parameters are also different; for example, it can be JSON (such as TP-link TL-IPC43AW equipment), XML (such as D-Link DIR-823G equipment), and other types. Accordingly, PDFuzzerGen needs to deal with various complex parameter scenarios. For the parameter discovery process, considering that in the actual encoding process, before the deep-level parameters reach the target code position, some other parameters usually need to pass the verification. Therefore, we hope that the payload generated by mutation is not only syntactically compliant but also semantically compliant, especially the parameter relationship. Therefore,

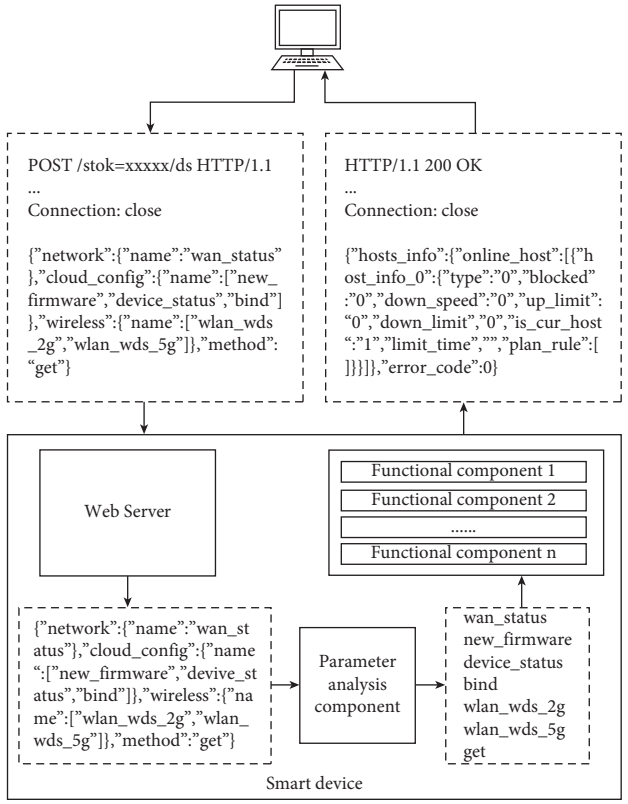

Figure 3: Smart device Web interface communication and message processing flow.

we define two different degrees of parameter relationship to better describe the relationship between parameters: hierarchical relationship and dependency relationship.

The hierarchical relationship is used to describe the parameter relationship at the syntactic level, and the dependency relationship is used to describe the parameter relationship at the semantic level. There are many formats for parameters and usually support nested data structures such as JSON, XML, and so on. In these formats that support nested data structures, we define the hierarchical relationship of parameters as the hierarchical order relationship of each key-value pair from outside to inside. Based on the hierarchical relationship, we define the relationship between a parameter and its parent parameters and ancestor parameters that are shallower than its depth as a dependency relationship. Taking some parameters of the JSON structure in Figure 3 as an example, for the nested key-value pair $''\text{network}'': \{''\text{name}'': ''\text{wan\_status}''\}$, $''\text{network}''$ is the outer key; $\{''\text{name}'': ''\text{wan\_status}''\}$ is the value of the outer layer; $''\text{name}''$ is the key of the inner layer; and $''\text{wan\_status}''$ is the value of the inner layer. The hierarchical relationship of these parameters from shallow to deep is: $''\text{network}''$, $''\text{name}''$, and $''\text{wan\_status}''$. The parameter dependency of $''\text{wan\_status}''$ is that $''\text{wan\_status}''$ depends on $''\text{name}''$ and $''\text{network}''$.

The parameter discovery algorithm of PDFuzzerGen is shown in Algorithm 2:

Input: Potential function message I
Output: Parameter discovery results R_p
(1) For each message in I, identify the message type T and parameter position M of all submitted parameters.
(2) For each position M_i in each type T_i, extract the original complete parameter data structure S.
(3) Determine the type S_i of S. Use the analysis module F_{si} corresponding to S_i to analyze S_i, record all parameter dependency R, and obtain all key-value pair information K: V.
(4) For each K_i: $Vi \in K$: V, mark V_i as the target test parameter, and K_i remains unchanged. At the same time, record the M_i and T_i corresponding to V_i, which are recorded as v_p.
(5) Count all the v_p and record them as v_s.
(6) According to the parameter dependence relationship R, calculate the parameter test value v_t. For each higher level of parameter depth, the potential value increases by one.
(7) Record S_i, R, v_s, and v_t as R_p.

Algorithm 2: Dynamic Parameter Discovery.

After the above process, PDFuzzerGen uses the parameter data structure type, parameter dependency, parameter key-value information, and parameter value information to build the parameter discovery policy.

3.4. Mutation Policy Selection. The mutation is a very important step in a fuzzing process. For some modern fuzzing frameworks, such as libFuzzer and BooFuzz, a mutation operation can be done by an underlying mutation module. In a fuzzer, it is more important to define specific mutation policies. PDFuzzerGen completes mutation policy selection from three origins: parameter value evaluation, policy selection, and policy schedule.

3.4.1. Parameter Value Evaluation. There are many parameter fields in a raw seed message. To perform fuzzing more efficiently, the value of parameters needs to be evaluated. Parameter value evaluation is mainly carried out from three origins: parameter position, semantic feature of parameter value, and parameter type. (1) For a parameter position, in the previous parameter discovery stage, PDFuzzerGen has obtained the dependency and hierarchical relationship of the parameters. According to our practical experiences and observations, the more complex the function parameter, the more complex the data structure, and the deeper the parameter hierarchy, the deeper the function code level that can be triggered. Thus, the default value evaluation basis of PDFuzzerGen is to evaluate according to the depth of the parameter hierarchy. The deeper the hierarchy, the higher its potential value. (2) For the numerical semantic features of parameters, when a parameter value is a human-readable vocabulary, PDFuzzerGen marks the parameter as a potentially high-value parameter. (3) For the parameter type, when the parameter value is a data type

related to the smart device business, such as IP, domain name, and so on, PDFuzzerGen marks the parameter as a potentially high-value parameter.

3.4.2. Policy Selection. Considering the different triggering and detecting methods of different types of vulnerabilities, we divide mutation policies into random-based mutation policies and injection-based mutation policies. The random-based mutation policy is to use the original value of the field as the seed data for random mutation. The injection-based mutation policy refers to a method that adds a command injection payload based on the original value of the resource and then uses the value after adding the load as the seed data for mutation. In addition, in order to facilitate the fuzzer to monitor abnormal conditions outside the device, the typical command injection payload used includes: reverse shell, request external resources, restart the device, and so on. At the same time, for the policy selection of target parameters, PDFuzzerGen prefers to select valuable parameters using an injection-based mutation policy. These parameters include semantic features, numeric types, business-related parameters (such as IP and domain name), and so on.

3.4.3. Policy Scheduling. PDFuzzerGen divides the fuzzing test process into the quick test and full test. By default, PDFuzzerGen performs full test, and all parameters are used for fuzzing. At the same time, PDFuzzerGen provides a quick test option for those users who prefer a result in a short period of time and are less concerned with the number of errors found. When the quick test option is enabled, PDFuzzerGen first performs a quick test on top N parameters based on the parameter value. When no vulnerabilities are found in a quick test, it switches to a full test and performs fuzzing tests on all parameters. It must know one parameter having multiple fuzzing policies after policy selection is reasonable. PDFuzzerGen creates different fuzzers to execute fuzzing tests. In different fuzzers, there are different mutation policies based on the results of the policy selection.

After the above process, PDFuzzerGen regards the parameter value, policy selection result, and policy scheduling result as the mutation policy.

3.5. Monitoring Policy Development. The black-box fuzzing cannot obtain the memory, process, file information, and so on of a smart device, so it needs to be detected from the responses of the smart device. However, simply monitoring service availability may be inaccurate, and there may be under-reporting. Therefore, PDFuzzerGen sets a multigranularity monitoring policy, which can be flexibly configured.

More specifically, we divide monitoring into two categories: active monitoring and passive monitoring. Active monitoring means that the fuzzer actively sends request packets to smart devices to detect the status of the device, including device survival monitoring, service survival monitoring, and function survival monitoring. Device survival monitoring is achieved by monitoring whether the smart device responds to the device survival detection packet. Service survival monitoring is achieved by monitoring whether the Web interface port responds as expected. Function survival monitoring is achieved by monitoring whether a specific function of the Web interface responds expectedly. Passive monitoring refers to determining whether the smart device has an unexpected response or whether it actively sends unexpected data messages to the outside world, such as whether the device has a reverse shell request or an external resource request.

3.6. Fuzzer Generation. After completing the above steps, the key to fuzzer generation, the policy, has been constructed. Next, complete fuzzers need to be synthesized based on these policies. In order to achieve this goal, two steps need to be completed: fuzzer stub generation and fuzzer synthesis.

3.6.1. Fuzzer Stub Generation. An important step in generating fuzzer stubs is to combine these policies reasonably. For example, when the mutation policy is a random mutation policy, the monitoring policy needs to choose active monitoring; otherwise, the status of a device cannot be judged. For a fuzzer stub, once it is generated, its content is fixed. Therefore, in order to enable the generated fuzzer to test more deep components on smart devices, there may be many fuzzer stubs. PDFuzzerGen dynamically matches and combines these policies and then generates multiple fuzzer stubs, which are used in the final synthesis of the fuzzer.

3.6.2. Fuzzer Synthesis. In order to generate fuzzers that can run normally, in addition to multiple fuzzer stubs, some additional basic function codes are required, such as the entry function of the fuzzer, the calling method of each fuzzer stub in the main function, and the parameter information obtained dynamically (e.g., target IP, port, etc.). Like fuzzer stubs, there may be many fuzzers synthesized based on a raw message, and these fuzzers together form a fuzzer bundle. Each fuzzer in the fuzzer bundle can run independently.

4. Implementation

PDFuzzerGen prototype is written in about 4,000 lines of Python code to generate fuzzers. PDFuzzerGen supports fuzzing on bare-metal and simulated smart devices at the same time. We solved the following challenges during implementation.

4.1. Parameter Selection. There are many parameters in an HTTP message, including parameters in a request header and parameters submitted by a client. Fuzzing all these parameters is feasible, but it will reduce the efficiency of fuzzing to a certain extent. According to our experiences, fuzzing parameters in a request header usually triggers errors in a Web server, and fuzzing client-user submitted parameters usually triggers the implementation flaw in the deep component. Therefore, we give a higher priority to parameters submitted by client-user in fast fuzzing. The

parameters in a request header are given less priority and postponed for use in full fuzzing mode.

4.2. State Maintenance and Monitoring Definition. The fuzzers generated by PDFuzzerGen can be run on the BooFuzz framework. For state maintenance, PDFuzzerGen defines the corresponding pre-sent callback function according to the state maintenance policy. The callback function is used to complete continuous state maintenance and position acquisition of authentication parameters. In addition, the offset position of the authentication parameter is synthesized in the main function process to set the value of the configuration of the authentication parameter. For the monitoring definition, PDFuzzerGen defines the corresponding post-send callback function according to the monitoring policy. Each fuzzer has a unique monitoring callback function. When there are multiple monitoring policies for a message, PDFuzzerGen generates multiple different fuzzers to complete the fuzzing.

4.3. Fuzzer Driver. After a fuzzer bundle is generated, although the PDFuzzerGen process has ended, we still hope that the generated fuzzer bundle can be automatically driven. We additionally wrote a fuzzer wrapper to drive a fuzzer bundle to fuzz the target device. The fuzzer wrapper records the logs and test cases during the entire fuzzing process so that the test cases that trigger the crash can be quickly located in the follow-up. At the same time, in order to reduce the time consumption of device service restart after the target service crashes, fuzzer wrapper provides a service recovery interface for the simulated device to automatically restore the target service.

4.4. Useless Fuzzer Filtering. Although PDFuzzerGen uses many techniques to synthesize fuzzers, we cannot guarantee that all synthesized fuzzers are effective, especially for smart devices with a complex authentication process. So we implemented a simple fuzzer filter for PDFuzzerGen. The filter automatically executes a fuzzer in a short time and monitors the response results of the device. If the fuzzer fails to run or there is no difference in different responses (it is common that different authentication results have different responses), the fuzzer is considered invalid and is automatically deleted. An invalid fuzzer is usually caused by a failed login authentication. Failed login authentication makes the target functional component unusable, and all test messages will be discarded by the target smart device. Deleting such invalid fuzzers can significantly reduce the time to trigger implementation flaws and improve the effectiveness of the generated fuzzers.

5. Evaluation

In this section, we evaluated PDFuzzerGen on real-world smart devices to answer the following questions:

(1) Features of PDFuzzerGen. Compared with existing fuzzer generation tools, what are the functional advantages of PDFuzzerGen? (§5.1)

(2) Efficiency of fuzzer generation. Compared with manually developed fuzzers, what are the labor expense and time consumption for PDFuzzerGen to automatically generate fuzzers? (§5.2)

(3) Efficacy of generated fuzzers. How effective are fuzzers generated by PDFuzzerGen and can PDFuzzerGen discover unknown vulnerabilities from real-world devices? (§5.3)

5.1. Evaluation Setup. Since existing fuzzer generation tools do not support smart device fuzzing (e.g., WINNIE [31] and FuzzGen [32]), we mainly compare features with these tools. For the evaluation of fuzzer generation process and generated fuzzers, we mainly compare PDFuzzerGen with BooFuzz [14] and Mutiny [37]. BooFuzz needs a manually written fuzzer before fuzzing; we invited 10 security analysts to complete this part of the test to make a more comprehensive comparison with PDFuzzerGen. To evaluate fuzzers generated by PDFuzzerGen, we used 19 popular real-world devices of different models and from different top key vendors [38]. In order to better monitor the network communication, all devices under test are connected to a local router. We deployed PDFuzzerGen on a Windows 10 desktop PC with Intel Core i7 8-core x 3.70 GHz CPU and 16 GB RAM. The PC is also connected to the router.

5.2. Features of PDFuzzerGen. In this section, we evaluate PDFuzzerGen and existing fuzzer generation tools. As shown in Table 2, as far as we know, PDFuzzerGen is the first fuzzer generation framework for smart devices. PDFuzzerGen automatically generates black-box fuzzers that use BooFuzz as a fuzzer engine, which has a wider range of use than other fuzzer generation tools.

Next, we compare PDFuzzerGen with state-of-the-art open-source network fuzzers BooFuzz [14] and Mutiny [37], as shown in Table 3. We compare them all in each stage of writing fuzzers for smart devices. Mutiny only implements the target discovery function and provides an interface for monitoring policy selection. BooFuzz provides related interfaces for mutation policy selection and monitoring policy selection, but it needs to be implemented manually by researchers. PDFuzzerGen implements all the functions of each stage and can automate the entire process.

5.3. Efficiency of Fuzzer Generation. To evaluate the differences in time expense between PDFuzzerGen and a manually developed fuzzer, we invited 10 security analysts to complete the task of manually coding fuzzers. We first generated about 100 MB and more than 400,000 raw messages for 19 smart devices. We divide the experiment into 2 stages. In the first stage, about 200 fuzz test seed messages need to be filtered from more than 400,000 raw messages. In the second stage, 10 security analysts and PDFuzzerGen write fuzzers based on the filtered seed messages. The time spent by each of the 10 security analysts is accumulated as the total time to jointly complete the second stage of the test.

Table 2: Comparison of fuzzer generation tools.

Fuzzer	Platform	Component	Fuzzer engine	Fuzz type
PDFuzzerGen	Smart device	Smart device component	BooFuzz	Black-box
FUDGE [34]	PC	C/C++ library	libFuzzer	White-box
FuzzGen [32]	PC	C/C++ library	libFuzzer	White-box
WINNIE [31]	PC	Windows application	WinAFL	Grey-box
IntelliGen [35]	PC	C/C++ library	libFuzzer	White-box

Table 3: Features comparison with BooFuzz and Mutiny.

Fuzzer	Target discovery	State maintenance	Parameter discovery	Mutation policy selection	Monitoring policy development	Fuzzer generation
PDFuzzerGen	✓	✓	✓	✓	✓	✓
BooFuzz [14]	✗	✗	✗	Manual	Manual	✗
Mutiny [37]	✓	✗	✗	✗	Manual	✗

To be fair, 10 security analysts and PDFuzzerGen used the same raw messages to conduct 5 rounds of testing and calculate the average time.

The experimental results are shown in Table 4. The average time for a security analyst to code a fuzzer is 54.9 minutes, and PDFuzzerGen reduces this time cost by nearly three orders of magnitude. Experimental results show that PDFuzzerGen can significantly improve the development efficiency of fuzzers.

Table 4: Fuzzer generation time consumption comparison.

Time consumption	The first stage (s)	The second stage (s)
PDFuzzerGen	253	758
Artificial	13,572	659,241

5.4. Efficacy of Generated Fuzzers. In this section, we evaluate the efficacy of fuzzers generated by PDFuzzerGen. We selected 19 devices from 6 vendors. These devices are the best-selling products offered by mainstream manufacturers. We compare PDFuzzerGen with default configured open-source network protocol fuzzers: BooFuzz and Mutiny. Note that we believe that these two fuzzers are handled fairly. Firstly, they are designed for general requests such as HTTP. Secondly, we intentionally selected the same raw data message for them to ensure the consistency of their fuzzing seeds. Finally, because they cannot complete the automatic state maintenance, we help them complete the state maintenance function manually to ensure the validity of the test data. In order to compare the efficacy of fuzzers generated by PDFuzzerGen, we reconfigured BooFuzz and Mutiny to run them for 24 hours. When a crash occurs, we automatically reset the device to resume testing.

From Table 5, we can see that PDFuzzerGen generally surpasses popular fuzzers in terms of the number of detected vulnerabilities and the time expense. It identified 11 memory corruption vulnerabilities and 3 command injection vulnerabilities. It only took 5.7 hours, with 46,913 fuzzing messages in total. PDFuzzerGen uses parameter discovery policies to identify different data structures and ensure the validity of the message. Therefore, Buffer Overflow 8 and Buffer Overflow 9 are discovered. Through the selection of mutation policies, Command Injection 1 and Command Injection 2 were discovered. While meeting the above two policies at the same time, Command Injection 3 was discovered. Neither BooFuzz nor Mutiny can handle the above situation to discover none of these vulnerabilities. Figure 4 shows the number of vulnerabilities discovered by PDFuzzerGen, Mutiny, and BooFuzz over time. PDFuzzerGen found more vulnerabilities than Mutiny and BooFuzz and found vulnerabilities faster. In some cases, the time for PDFuzzerGen to find vulnerabilities is very short, even within 2 minutes, such as Buffer Overflow 1 and Buffer Overflow 9. This is because PDFuzzerGen evaluated the value of the parameters in the message and selected the most valuable parameters for priority fuzzing. Therefore, vulnerabilities were discovered within a short period of time. However, PDFuzzerGen must spend more time testing in some cases. In addition, when testing Buffer Overflow 5, Buffer Overflow 6, Buffer Overflow 7, and Uncategorized Crash 1, BooFuzz had a better time performance than PDFuzzerGen did. This is because when the number of parameters goes higher and the structure of parameters are becoming more complex, PDFuzzerGen has more test cases to run. Even in these cases, the time cost difference is reasonably small.

In these real-world devices, we manually judged the triggered crashes and exceptions one by one to determine whether they were previously unknown vulnerabilities. Finally, PDFuzzerGen found 14 zero-day vulnerabilities, 2 of which were assigned CVEs and 5 were assigned CNVDs, as shown in Table 6. The results show that PDFuzzerGen can be applied to real smart devices from different vendors and can discover real-world unknown vulnerabilities.

6. Discussion and Future Work

Our prototype demonstrates the automated generation and policy-driven capabilities of black-box fuzzers for smart devices. Since this is the first step towards automation, we want to highlight some opportunities for improvement.

Table 5: Fuzzing time statistics.

Vulnerability	Device	PDFuzzerGen	Mutiny	BooFuzz
Buffer Overflow 1	TP-Link TL-WR940N	16 s	1,060 s	190 s
Buffer Overflow 2	Tenda AC9	143 s	1,479 s	1,473 s
Buffer Overflow 3	TP-Link TL-WR810N	209 s	N/A	635 s
Buffer Overflow 4	Tenda AC18	316 s	216 s	400 s
Buffer Overflow 5	TP-Link TL-WR841N	1,424 s	2,087 s	1,221 s
Buffer Overflow 6	TP-Link TL-WR841N	1,979 s	854 s	1,213 s
Buffer Overflow 7	Tenda AC9	1,610 s	N/A	1,179 s
Buffer Overflow 8	D-Link DIR-806	3,199 s	N/A	N/A
Buffer Overflow 9	TP-Link TL-WR940N	78 s	N/A	N/A
Uncategorized Crash 1	TP-Link TL-WR941N	1,774 s	N/A	931 s
Uncategorized Crash 1	TP-Link TL-WR810N	1,587 s	1,342 s	N/A
Command Injection 1	Tenda AC9	3,603 s	N/A	N/A
Command Injection 2	Tenda AC9	1,310 s	N/A	N/A
Command Injection 3	D-Link DIR-823G	3,269 s	N/A	N/A

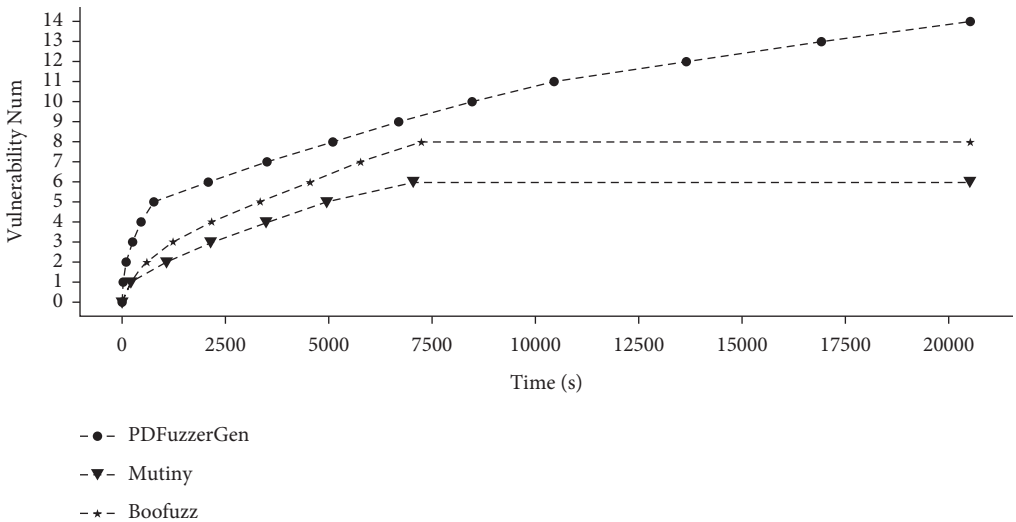

Figure 4: The number of vulnerabilities discovered over time.

Table 6: Assigned CVEs and CNVDs for the vulnerabilities found by PDFuzzerGen.

Vulnerability	Severity	Vulnerability type
CVE-2021-31624	High	Buffer Overflow
CVE-2021-31627	High	Buffer Overflow
CNVD-2020-69407	High	Command Injection
CNVD-2020-67555	Medium	Buffer Overflow
CNVD-2021-17400	Medium	Buffer Overflow
CNVD-2021-22752	Medium	Buffer Overflow
CNVD-2021-24948	Medium	Buffer Overflow

6.1. Automatic Generation of Raw Seed Messages. The initial input of PDFuzzerGen is raw messages. Although PDFuzzerGen automates the process from raw messages to generated fuzzers, the automatic generation of working raw messages itself is still a challenge. A promising solution is to start with the front-end page of each device and analyze the dependencies between the front-end parameters and input rules to generate effective raw seed messages. This will be our next step to achieve a higher degree of automation.

6.2. Message Dependency. In the current implementation of PDFuzzerGen, we only focus on those messages that can complete the corresponding functions with a single request. But, in fact, there are still a few functions that require multiple request messages to collaborate to work out. In this case, splitting multiple requests and creating fuzzers separately may prevent the generated fuzzers from working properly. In order to work around this issue, it is necessary to manually participate in these complex interaction processes and make corresponding modifications to the fuzzer. One possible automatic method is to track and model the entire conversation flow during the initial analysis of the message and then handle it on the established interactive process model. Automating this process to reduce human involvement is on our roadmap.

6.3. Message Replay. Although login messages can be replayed on most of the smart devices we tested, there are still some devices where login messages cannot be replayed, which limits the discovery of bugs in components that require login authentication. In practice, this situation can be

alleviated with the help of appropriate manual configuration. First, PDFuzzerGen is able to flag that messages are in this case. Next, for the raw message, in this case, the analyst can manually complete the login process and fill in the corresponding authentication credential into the corresponding raw message authentication field. Finally, the analyst can use PDFuzzerGen to regenerate the fuzzer for the message carrying the authentication credentials. This situation is transformed into anonymous fuzzing. Because most anti-replay mechanisms limit the target to programmable robots rather than normal users, researchers only need to take a few extra minutes to complete the modification of the original message during the whole process. And, for a single device, the authentication credential can be reused within a certain period of time, so multiple raw messages of the same device can be modified by simple replacement. Compared to the complicated encoding process, this time cost is acceptable. Of all the 19 smart devices we selected, login messages were not replayable for 3 devices. In addition to fuzzing the parsing module for these 3 devices, we also performed the above steps to alleviate the limitation caused by the inability to replay the login message. After completing mitigations, all generated fuzzers can fuzz modules that require authentication. Further automation of the login process is one of the directions for future improvements.

6.4. Parameter Relationship. In the process of parameter discovery, when different parameter data exchange formats are found, all hierarchical relationships of each parameter can be identified by using the corresponding parsing tool. The accuracy of this process depends on two aspects: on the one hand, it depends on the implementation accuracy of the firmware components for the definition of various data format standards (e.g., ECMA-404 [39] defines the JSON data exchange format). On the other hand, it depends on the accurate implementation of the standard by each parsing tool. For example, the json library[40] of Python implements parsing of the json data format. Therefore, if the firmware component correctly implements the data format standard and the data format parsing tool can correctly parse the standard data structure, the accuracy of the hierarchical relationship and the dependency relationship can be effectively guaranteed. Since the dependency relationship is defined above the hierarchical relationship, it can be directly derived from the hierarchical relationship. Therefore, when the accuracy of the hierarchical relationship is guaranteed, the accuracy of the dependency relationship can be guaranteed.

6.5. Valuable Parameters. According to our experience, although the types and values of parameters may vary, those parameters with specific semantic meaning are more important for the functions involved. For example, for a smart device management function that uses the ping command to test network connectivity, among its various parameters, the parameter of type IP address may be directly used for splicing and executing the ping command. In this scenario, command injection vulnerabilities may occur. Typical vulnerabilities include: CVE-2021-41653, CVE-2021-38470, and so on. Among the seven vulnerabilities we found, two involve parameters that meet the valuable parameters we defined, namely CVE-2021-31624 and CNVD-2020-69407. Prioritizing testing with valuable parameters reduced the time to discover the above two vulnerabilities by approximately $1x$ and $5x$. Among them, CNVD-2020-69407 has higher parameter complexity, with a total of nine tested parameters. This shows that selecting valuable parameters for priority fuzzing can effectively reduce the time of vulnerability discovery in some scenarios with a large number of parameters. The determination of valuable parameter categories mainly relies on knowledge entries to the knowledge base. At present, the way of adding specific knowledge items in the knowledge base is mainly the accumulation of artificial experience. Automated knowledge entry augmentation would be a promising approach to knowledge base construction. This will be one of the future work directions.

6.6. False Positives. The fuzzer generated by PDFuzzerGen judges whether an exception is triggered by monitoring the return result of the callback function. Consequently, inaccurate fuzzers may produce invalid crashes and exceptions, which may not happen on a normal use device. There are two possible reasons for this. On one hand, it may be due to the limited performance of the bare-metal device. When the message sending of fuzzing is too fast, some devices may experience short-time freezes or wait for the request to be processed. When the waiting time exceeds the threshold of the fuzzer, an exception will be thrown, leading to a false positive. On the other hand, it may be because the behavior of some simulated devices is not the same as the bare-metal device. Some crashes triggered on simulated devices may not be triggered on bare-metal devices. The first case can be mitigated by adjusting the timeout waiting time of the monitoring policy in PDFuzzerGen, but this will result in a certain degree of fuzzing efficiency reduction. But what is interesting is that timeouts caused by device performance problems are often accompanied by potential denial of service vulnerabilities, which are common in smart devices. So the short time manual verification here is worthwhile. The second case can be solved by testing on the bare-metal device and the simulated device at the same time, but this will undoubtedly increase the labor expense and time consumption. But, fortunately, the number of such cases is relatively small, so this verification will only generate less time overhead.

6.7. More Protocols. In the current implementation of PDFuzzerGen, we only focus on the Web interface on smart devices. However, there are no special technical obstacles to migrating the policy-driven fuzzer generation method and PDFuzzerGen to other smart device services, such as FTP, SMB, UPnP services, and so on. An inspiring example is that we simply migrated PDFuzzerGen to the SMB service and then tested a smart device with SMB service enabled by default, which finally found a memory corruption vulnerability and was assigned a CNVD: CNVD-2021-30168.

7. Conclusion

We came up with a new technique called policy-driven fuzzer generation. Supported by this technique, PDFuzzerGen, a black-box fuzzer generation framework was proposed to support an automatic fuzzing of deep-level functional components on smart devices through a Web interface. Instead of directly mutating a raw message, PDFuzzerGen analyzes a message, formulates a variety of driving policies, and synthesizes fuzzers to reduce labor expense and development time consumption. We tested PDFuzzerGen on 19 real-world devices. PDFuzzerGen found more vulnerabilities than manually crafted black-box fuzzers in less time. Overall, PDFuzzerGen discovers 14 previously unknown vulnerabilities with two CVEs and five CNVDs.

Data Availability

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

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported, in part, by the Open project of the State Key Laboratory of Computer Architecture, Neural Network Enhanced Symbolic Execution Algorithm Research (CARCH201910) and the Fundamental Research Funds for the Central Universities under grant nos. 3132018XNG1814 and 3132018XNG1815.

References

[1] K. Lant, *By 2020, there will be 4 devices for every human on earth*, Futurism, Italy, 2017.

[2] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[3] T. Micro, "Smart yet Flawed: IoT Device Vulnerabilities Explained," Security News, Technical Report, 2020.

[4] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proceedings of the NDSS*, vol. 1, San Diego, California, February 2015.

[5] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proceedings of the NDSS*, vol. 1, pp. 1–8, San Diego, California, February 2016.

[6] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "Firmfuzz: Automated Iot Firmware Introspection and analysis," in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-Of-Things*, pp. 15–21, London, United Kingdom, November 2019.

[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 *Proceedings of the 28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1099–1114, SANTA CLARA, CA, USA, August 2019.

[8] N. Redini, A. Machiry, R. Wang et al., "Karonte: detecting insecure multi-binary interactions in embedded firmware," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1544–1561, IEEE, San Francisco, CA, USA, May 2020.

[9] D. Wang, X. Zhang, T. Chen, and J. Li, "Discovering Vulnerabilities in COTS IoT Devices through Blackbox Fuzzing Web Management interface," *Security and Communication Networks*, vol. 2019, Article ID 5076324, 19 pages, 2019.

[10] N. Redini, A. Continella, D. Das et al., "DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy 2021*, San Francisco, CA, USA, May 2021.

[11] J. Chen, W. Diao, Q. Zhao et al., "IoTFuzzer: Discovering Memory Corruptions in IoT through App-Based Fuzzing," in *Proceedings of the NDSS*, San Diego, California, February 2018.

[12] 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.

[13] X. Feng, R. Sun, X. Zhu et al., "Snipuzz: Black-Box Fuzzing of IoT Firmware via Message Snippet Inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, November 2021.

[14] P. J. boofuzz, "Network Protocol Fuzzing for humans," 2017.

[15] "National computer network emergency response technical team/coordination center of china," 2021, https://www.cert.org.cn/publish/english/index.html.

[16] M. Eceiza, J. L. Flores, and M. Iturbe, "Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded System," *IEEE Internet of Things Journal*, vol. 8, no. 13, 2021.

[17] P. Khandait, N. Hubballi, and B. Mazumdar, "IoTHunter: IoT network traffic classification using device specific keywords," *IET Networks*, vol. 10, no. 2, pp. 59–75, 2021.

[18] Y. Zeng, M. Lin, S. Guo et al., "MultiFuzz: a coverage-based multiparty-protocol fuzzer for IoT publish/subscribe protocols," *Sensors*, vol. 20, no. 18, p. 5194, 2020.

[19] H. Zhang, K. Lu, X. Zhou, Q. Yin, P. Wang, and T. Yue, "SIoTFuzzer: fuzzing web interface in IoT firmware via stateful message generation," *Applied Sciences*, vol. 11, no. 7, p. 3120, 2021.

[20] H. Lu, C. Jin, X. Helu et al., "Research on intelligent detection of command level stack pollution for binary program analysis," *Mobile Networks and Applications*, vol. 26, no. 4, pp. 1723–1732, 2021.

[21] M. Smith, M. Helmi, and J. Miller, "Comparison of approaches to use existing architectural features in embedded processors to achieve hardware-assisted test insertion," in *Proceedings of the Work-in-Progress Session*, Brussels, Belgium, January 2010.

[22] 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 NDSS*, vol. 23, pp. 1–16, San Diego, California, February 2014.

[23] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A Multi-Target Orchestration platform," in *Proceedings of*

the Workshop on Binary Analysis Research (Colocated with NDSS Symposium, p. 18, San Diego, California, February 2018.

[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 *Proceedings of the Annual Computer Security Applications Conference*, pp. 733–745, Austin, USA, December 2020.

[25] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: fuzzing iot firmware via multi-stage message generation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2525–2527, London, United Kingdom, November 2019.

[26] M. Yu, J. Zhuge, M. Cao, Z. Shi, and L. Jiang, "A survey of security vulnerability analysis, discovery, detection, and mitigation on IoT devices," *Future Internet*, vol. 12, no. 2, p. 27, 2020.

[27] H. Lu, C. Jin, X. Helu, N. Guizani, and Z. Tian, "AutoD: intelligent blockchain application unpacking based on JNI layer deception call," *IEEE Network*, vol. 35, no. 2, pp. 215–221, 2020.

[28] N. Hu, Z. Tian, H. Lu, X. Du, and M. Guizani, "A multiple-kernel clustering based intrusion detection scheme for 5G and IoT networks," *International Journal of Machine Learning and Cybernetics*, vol. 12, no. 11, pp. 3129–3144, 2021.

[29] N. Kamel and J. L. Lanet, "Analysis of HTTP protocol implementation in smart card embedded web server," *International Journal of Information and Network Security*, vol. 2, no. 5, p. 417, 2013.

[30] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the 23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 95–110, San Diego, California, August 2014.

[31] J. Jung, S. Tong, H. Hu, J. Lim, J. Jin, and T. Kim, "WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS), Virtual*, San Diego, California, April 2021.

[32] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "{FuzzGen}: Automatic Fuzzer Generation," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, pp. 2271–2287, Boston, MA, USA, August 2020.

[33] K. Serebryany, "Libfuzzer -a library for coverage-guided fuzz testing," 2022, https://llvm.org/docs/Reference.html.

[34] D. Babić, S. Bucur, Y. Chen et al., "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 975–985, Tallinn, Estonia, August 2019.

[35] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, "IntelliGen: automatic driver synthesis for fuzz testing," in *Proceedings of the 2021 IEEE/ACM 43rd international conference on software engineering: software engineering in practice (ICSE-SEIP)*, pp. 318–327, IEEE, Madrid, ES, May 2021.

[36] R. Fielding, J. Gettys, J. Mogul et al., "RFC2616: Hypertext Transfer Protocol--HTTP/1.1," *Network Working Group*, 1999, https://dl.acm.org/doi/pdf/10.17487/RFC2616.

[37] "Mutiny fuzzing framework," 2021, https://github.com/Cisco-Talos/mutiny-fuzzer.

[38] "Wireless Router Market Share 2021 – Global Trends, Industry Analysis, Opportunities, Development History and Key Players and Forecast 2027," 2021, https://www.wicz.com/story/45198505/wireless-router-market-share-2021-ndash-global-trends-industry-analysis-opportunities-development-history-and-key-players-and-forecast-2027.

[39] "ECMA-404 the JSON Data Interchange Syntax," 2017, https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

[40] "JSON encoder and decoder," 2022, https://docs.python.org/3/library/json.html.