Article

Optimizing IoT Web Fuzzing by Firmware Infomation Mining

Yifei Gao, Xu Zhou *, Wei Xie, Baosheng Wang, Enze Wang and Zhenhua Wang

College of Computer, National University of Defense Technology, Changsha 410073, China;
gaoyf@nudt.edu.cn (Y.G.); xiewei@nudt.edu.cn (W.X.); bswang@nudt.edu.cn (B.W.);
wangenze18@nudt.edu.cn (E.W.); wangzhenhua19@nudt.edu.cn (Z.W.)
* Correspondence: zhouxu@nudt.edu.cn

Abstract: IoT web fuzzing is an effective way to detect security flaws in IoT devices. However, without enough information of the tested targets, IoT web fuzzing is often blind and inefficient. In this paper, we propose to use static analysis to assist IoT web fuzzing. Our insight is that plenty of useful information is hidden in firmwares, which can be mined by static analysis and used to guide the subsequent dynamic analysis—fuzzing. Hence, our approach contains two stages: pre-fuzzing stage and fuzzing stage. In the pre-fuzzing stage, we perform static analysis on the IoT firmwares to exploit helpful information, such as web page paths, interfaces, and shared keywords. These kinds of information are used to construct diverse seeds for covering more web paths and interfaces, and are also used to prioritize seeds according to their importance (related to shared keywords) in the fuzzing stage. Based on this approach, we implement a prototype IoT web fuzzing system—IoTParser. Experiments show that IoTParser increased the vulnerability discovery capability by 44% on average, while increasing the vulnerability discovery efficiency by 48.2% on average compared with state-of-the-art IoT web fuzzer. In addition, IoTParser has found 13 vulnerabilities, including 7 0-day.

Keywords: IoT; firmware; fuzzing; static analysis; vulnerability

Citation: Gao, Y.; Zhou, X.; Xie, W.; Wang, B.; Wang, E.; Wang Z. Optimizing IoT Web Fuzzing by Firmware Infomation Mining. *Appl. Sci.* **2022**, *12*, 6429. https://doi.org/10.3390/app12136429

Academic Editor: Agostino Forestiero

Received: 27 May 2022
Accepted: 17 June 2022
Published: 24 June 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In the era of the IoT, from tiny smart homes to large industrial control networks, a large number of IoT devices are connected to the cloud for services [1]. Due to the limited resources of embedded devices, security issues are rarely considered, and security capabilities are lacking. In addition, the IoT device system architecture is not a general platform and lacks traditional security mechanisms [2]. Once breached, it will cause great harm. Therefore, it is necessary to conduct large-scale vulnerability detection for IoT devices. According to statistics, among the IoT devices, routers, and IP cameras with a large share have a web front-end ratio of 83.6% and 93.2%, respectively. We can see that the web entrance is widely used in IoT devices, and the web entrance integrates most of the functional operations of the device, which is more likely to have vulnerabilities. According to Gartner, more than 70% of vulnerabilities are hosted at the application layer rather than the network layer or system layer [3], so vulnerability detection for IoT web entrance is an effective way to discover vulnerabilities. Among various vulnerability discovery methods, fuzzing [4] is very suitable for large-scale automated vulnerability detection because of its high scalability, high degree of automation, and low false-positive rate.

Current study [5–7] can generally be divided into three stages: (1) Seed generation: In order to generate an initial seed that conforms to semantics, most of the current work uses simulation [8] or physical devices to run the firmware. Then, automatically traversing the front-end web pages through a web crawler [9] while capturing traffic as seeds. (2) Fuzzing: After obtaining the initial seeds, the seeds will be mutated and sent to the device's web entrance for testing. At the same time, vulnerabilities can be discovered using side channels or by monitoring the simulated environment. (3) Schedule: The scheduling part calculates data such as the basic blocks coverage and the number of function calls in the fuzz process, then adjusts the direction of the fuzz in real-time, thereby accelerating vulnerability discovery. However, the existing IoT web fuzzing technology still faces the following two challenges:

Challenge 1: It is not easy for crawlers to cover all front-end pages and interfaces, resulting in incomprehensive test cases and low coverage. The purpose of crawlers is to access as many pages and interfaces as possible, each of which corresponds to a potentially vulnerable function entry on the back end. However, the flexibility of web technology makes the rules of web crawlers unable to cover all possible situations. In addition, crawlers cannot find hidden interfaces that are not displayed on the front-end, pages that are not connected to the crawler entrance, or pages dynamically generated by js scripts, etc. Ultimately, the coverage of crawler is low, and it is impossible to find bugs in uncovered function entries.

Challenge 2: Lack of scheduling mechanism or too much overhead for scheduling. The existing scheduling mechanism [7] first performs dynamic testing on all test cases and counts the corresponding kernel logs. However, the key factor affecting the speed of fuzzing is the low throughput of the simulation environment, coupled with the frequent reading of the kernel log, which puts forward higher requirements on the amount of computation. Although the final result shows that the discovery of vulnerabilities is accelerated, sorting test cases also takes up much time.

These two challenges have limited IoT web fuzz's vulnerability discovery ability and speed. We believe this is because state-of-the-art research does not provide enough information to guide fuzz, resulting in fuzz to be blind and inefficient. Inspired by static analysis, we recognize the need to dig deeper into the firmware for helpful information to guide fuzz. In this paper, we propose IoTParser, aiming at improving vulnerability discovery capabilities and speed of vulnerability discovery. In response to the above two challenges, we divide the fuzzing process into pre-fuzz stage and fuzz stage. In the pre-fuzz stage, to solve the first challenge, we performed a static analysis of the decompressed firmware file system [10] and extracted the interface and page paths. We used this information to assist the front-end crawler in covering more function entries, so that more hidden vulnerabilities could be found and vulnerability discovery capabilities are enhanced. At the same time, by extracting and matching shared keywords at the front- and back-ends of the firmware, we propose a weight calculation method using shared keywords. This approach eliminates the need for dynamic interaction with the web entry and the need to parse the logs frequently during the fuzz process. Furthermore, it only requires a single static analysis of the firmware during the pre-fuzz stage, so scheduling can be completed more quickly and with less overhead, thus solving challenge 2. Finally, in the fuzz stage, using the helpful information provided in the pre-fuzz stage, we can discover vulnerabilities in a larger coverage area and find the vulnerabilities faster.

We implemented an IoTParser prototype system to evaluate our method, which supports four common vulnerability detections: command injection, XSS, unauthorized access, and overflow [11]. We tested it on five devices from four mainstream vendors and found 13 vulnerabilities, 7 of which are 0-day vulnerabilities. Then, we compared IoTParser with state-of-the-art fuzzing tools, including W13scan [12] and FirmHunter [7], and the final result shows that our tool improves vulnerability discovery capability by 44% and vulnerability discovery speed by 48.2%.

Overall, the contributions of our work are as follows:

1. By extracting interface and page paths from the firmware, web crawlers can have larger coverage and generate more comprehensive test cases;
2. By extracting shared keywords from the firmware, we can statically schedule test cases and accelerate vulnerability discovery;
3. Design and implement the prototype system IoTParser, which is better than state-of-the-art tools through experimental comparison, and found 13 vulnerabilities, 7 of which are 0-day, the source code can be found at https://github.com/jayus0821/IoTParser (accessed on 8 June 2022).

The rest of the paper is structured as follows. In Section 2, we detail the proposed methods to meet the challenges. Section 3 describes the implementation of IoTParser. In Section 4, we evaluate the IoTParser. Section 5 describes the related work. Then, we discuss limitations and some possible improvements in Section 6. Finally, Section 7 concludes the paper.

2. Firmware Information Mining

This section describes how we mine the firmware for helpful information, including interfaces, page paths, and shared keywords. Furthermore, we will describe how we use this information to assist in the fuzzing stage.

2.1. Interfaces and Page Paths Extraction

The page path refers to the web path of all files in the webroot directory. Generally, the web path of the back-end files in the web service is the path relative to the webroot directory. For some test pages or independent pages that have no jump relationship with other pages, the web crawlers cannot find them, and the action forms and interfaces of these pages will be missed. In addition, some IoT devices have imperfect permission restrictions on back-end static files. Directly accessing the page path through the front-end will lead to leakage of sensitive information or leakage of operation interface [13]. For example, in CVE-2021-33251, "model.cgi" of Qihu360-F5C will leak the device's mac address and firmware version due to a lack of permission restrictions. It should be noted that the CGI files of some devices are not located in the webroot directory. We will also match them to obtain the file name and add it to the page paths.

The interface refers to the function entry of a web service. For example, "/ajax_ddnscode.asp" is the configuration interface of "DDNS" in the ASUS RT-N53 router. In addition to the file format, some are in the form of routing, such as "/index/home". the back-end service will resolve the route to the corresponding function entry for processing. IoT web fuzzing needs to find as many interfaces as possible for testing. Each function entry corresponds to a set of processing logic in the back-end, and there may be vulnerabilities.

Due to the flexibility of front-end technology and the limitation of front-end resources, web crawlers cannot obtain all function entries, resulting in low coverage of crawlers, which will miss many potential vulnerabilities. However, all of these can be found in the back-end source code. After decompressing the firmware, we can obtain web-related front-end and back-end files, such as HTML, JS, ASP, CGI, and Httpd. For different file types, we summarize the structures and functions related to interfaces in their grammar rules to extract them in a targeted manner.

For HTML files, HTML files are composed of various front-end components. For the attributes of different components, we extract the values of attributes related to the interface. For example "href" attribute means jumping to a new page or interface, "action" attribute means sending the form content to the interface it points to. PHP, ASP, JSP, and other back-end development languages have their jump functions, such as "header", "redirect", "sendRedirect", etc. [14]; other than that, the more common way is to construct the web page and hand it over to the browser to process. Therefore, in addition to the matching of jump functions, we also use regular matching to extract possible interfaces. The role of XML files in web interfaces is mainly used for XML-based services, such as SOAP. The format is relatively fixed so that the interfaces can be extracted through regular expressions according to the service format. For js files, its content is javascript code, not only in js files, PHP, HTML, and ASP also use some javascript code for dynamic processing. For the javascript code in all files, we also summarize the interface-related technologies in javascript, such as "XMLHttpRequest", "jquery Ajax", "fetch", and other data request technologies [15]. Since there are many public js libraries that provide basic services in js, such as "jQuery", we exclude js files that are referenced by various files for more than a certain number of times based on experience. For some unreadable files, such as CGI files and back-end binary files, we extract potential interface information by regular matching of binary strings contained in them.

As for the page paths, after decompressing the firmware, we first locate the webroot directory by locating the front-end files and then traverse all the files in the webroot directory to obtain the web path relative to the webroot directory. In addition, because some CGI files are not located in the webroot directory, we will search all CGI files in the file system. The processing process is shown in Figure 1.

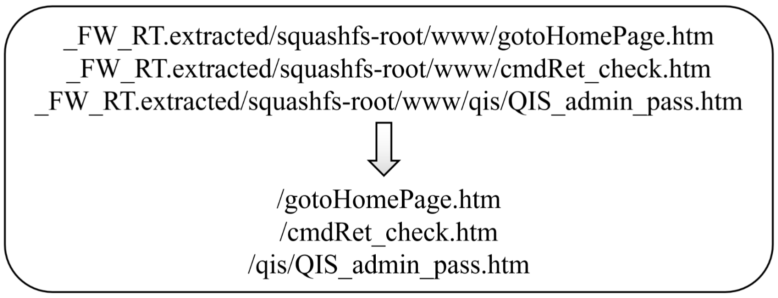
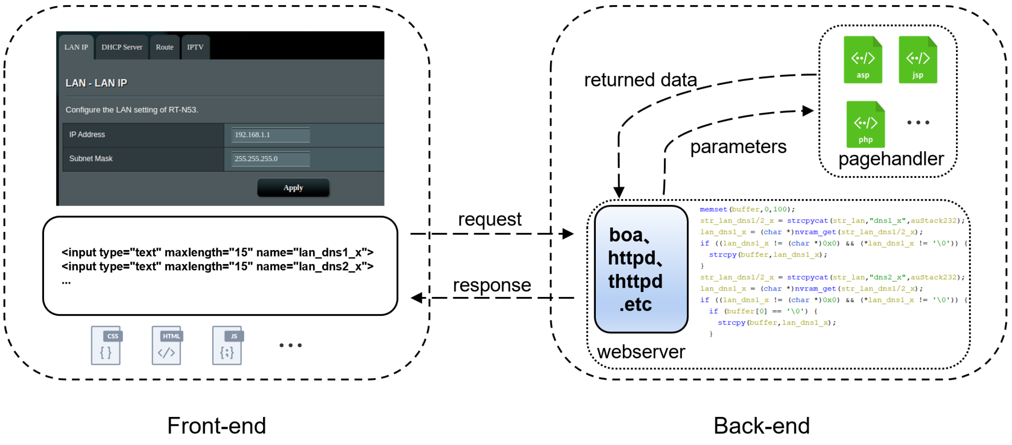

_FW_RT.extracted/squashfs-root/www/gotoHomePage.htm
_FW_RT.extracted/squashfs-root/www/cmdRet_check.htm
_FW_RT.extracted/squashfs-root/www/qis/QIS_admin_pass.htm
/gotoHomePage.htm
/cmdRet_check.htm
/qis/QIS_admin_pass.htm

Figure 1. Process from page path to web path in ASUS RT-N53 router.

2.2. Shared Keywords Extraction

The keywords transmitted by the front-end will be handed over to the back-end for processing, including various customized binary files, such as "httpd" and "nvram". After the back-end processing is completed, the output is handed over to the page handler for rendering and then returned to the front-end page for display. Figure 2 shows the request processing process of the web service in the ASUS RT-N53 router. Click the "Apply" button on the front end, and the two parameters, "lan_dns1_x" and "lan_dns2_x", will be sent together in the form of http packets to the router's back-end web server for processing. Looking at the back-end source code, you can see that the parameter values sent by the front-end are stored in nvram after being received, and then the init process will copy the stored "lan_dns1_x" and "lan_dns2_x" from nvram to the buffer for subsequent configuration. In the init process, the buffer size is 100 bytes, but the length of the "lan_dns1_x" and "lan_dns2_x" parameters is not judged. When the parameter length is greater than 100 bytes, a buffer overflow vulnerability is caused, allowing attackers to gain system privileges and execute arbitrary commands.



Configure the LAN setting of RT-N53.

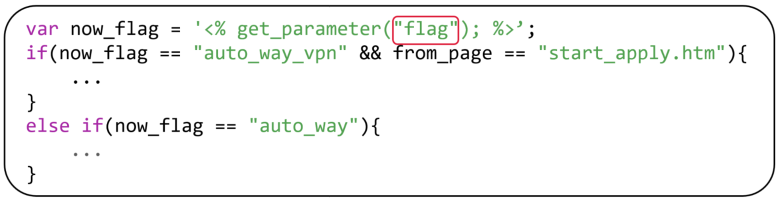
IP Address	192.168.1.1
Subnet Mask	255.255.255.0

```
<input type="text" maxlength="15" name="lan_dns1_x">
<input type="text" maxlength="15" name="lan_dns2_x">
...
```

```
memset(buffer,0,100);
str_lan_dns1/2_x = strcpycat(str_lan,"dns1_x",auStack232);
lan_dns1_x = (char *)nvram_get(str_lan_dns1/2_x);
if ((lan_dns1_x != (char *)0x0) && (*lan_dns1_x != '\0')) {
  strcpy(buffer,lan_dns1_x);
}
str_lan_dns1/2_x = strcpycat(str_lan,"dns2_x",auStack232);
lan_dns1_x = (char *)nvram_get(str_lan_dns1/2_x);
if ((lan_dns1_x != (char *)0x0) && (*lan_dns1_x != '\0')) {
  if (buffer[0] == '\0') {
    strcpy(buffer,lan_dns1_x);
  }
```

Figure 2. The request processing process of the web service in ASUS RT-N53 router.

Shared keywords are keywords that use the same name in the front-end web file and the back-end binary file. With the help of this sharing phenomenon, we can statically parse the firmware web file, extract all possible parameters, and determine whether these keywords are shared. Because shared keywords will be substituted into the back-end binary files, we can assume that shared keywords are more likely to be vulnerable. Among all the parameters transmitted by the front-end, not all parameters will be passed to the back-end for processing. Some of the parameters will be used for logical judgment, as shown in Figure 3, and some of the parameters are meaningless additional values in the form. For example, when the input tag is used as a form submit button, a meaningless "submit" parameter and timestamp, token, etc., will be submitted. A considerable part of the parameters sent by each request will not be passed to the back-end for calculation. Generally, it will not lead to vulnerabilities, except for some logical vulnerabilities, such as unauthorized access.



```
var now_flag = '<% get_parameter("flag"); %>';
if(now_flag == "auto_way_vpn" && from_page == "start_apply.htm"){
    ...
}
else if(now_flag == "auto_way"){
    ...
}
```

Figure 3. The code snippet of the QIS_internet_ip.htm page in RT-N53 router, "flag" is the front-end parameters used for logical judgment.

After the firmware is decompressed, there are many types of web files, and we only extract the keywords related to the parameters. Therefore, the target file types we analyzed include PHP, HTML, ASP, JSP, JS, and XML, which can cover most IoT devices.

For HTML files, we mainly extract forms and variable naming operations, such as "id", "name", etc. In PHP, parameters are mainly received in global variables, such as "$_GET", "$_REQUEST", etc., and keywords are obtained by matching these global variables. Similar to PHP, ASP, and JSP have standard functions to receive parameters. Similarly, we design corresponding extraction rules according to the characteristics of this type of file and use regular expressions to assist in extracting possible keywords. In the js file, we match and extract keywords for assignment behavior and functions related to data transfer.

2.3. Priority Queue Generation

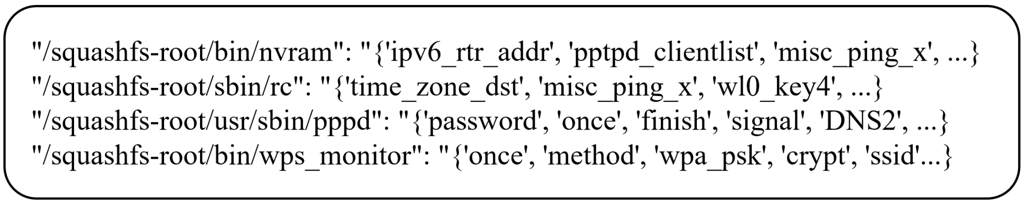
After extracting all the front-end parameter sets, we use the "strings" [16] command to extract the string set contained in the back-end binary file. Then, we traverse the front-end parameters and match the back-end string collection to obtain the shared keyword collection. The calculation process is shown in Algorithm 1.

The result is shown in Figure 4, the key of the dictionary is the binary file, and the corresponding value is the shared keywords contained.

Algorithm 1: Calculate SharedkeySet

```
input  : A collection of parameters ParaSet extracted from the front-end
         Collection of back-end binaries BinfileSet
output : Collection of shared keywords SharedkeySet

for para in ParaSet do
    for binfile in BinfileSet do
        if para in strings(binfile) then
            // Extract strings in binfile through strings command
            put para in SharedkeySet[binfile]
        end
    end
end
```



```
"/squashfs-root/bin/nvram": "{'ipv6_rtr_addr', 'pptpd_clientlist', 'misc_ping_x', ...}
"/squashfs-root/sbin/rc": "{'time_zone_dst', 'misc_ping_x', 'wl0_key4', ...}
"/squashfs-root/usr/sbin/pppd": "{'password', 'once', 'finish', 'signal', 'DNS2', ...}
"/squashfs-root/bin/wps_monitor": "{'once', 'method', 'wpa_psk', 'crypt', 'ssid'...}
```

Figure 4. The set of extracted shared keywords in ASUS RT-N53 router.

After generating the initial seed, our tool will calculate the weight of the parameters carried in the request according to the SharedkeySet. If the parameter exists in the SharedkeySet, it is more likely to be vulnerable. From this point of view, we sort the initial seeds to generate a priority queue. It should be noted that we will reduce the corresponding weight in turn for keywords that appear too many times. This strategy is mainly aimed at some useless keywords that frequently appear, such as "time", "token", etc. The calculation process is shown in Algorithm 2. Our tool can statically generate priority queues, saving much time compared to existing priority queue generation methods because there is no need to interact with the device's web interface. We also do not need to parse the logs during the fuzz process, so the overhead is much lower.

Algorithm 2: Priority queue generation

```
input  : Shared keywords collection SharedkeySet
         The request FuncReq corresponding to each function entry
output : Priority queue SeedQueue for fuzzing seeds

for parameter in FuncReq do
    // Traverse the parameters in the request
    for Binfile in SharedkeySet do
        if parameter in BinkeySet[Binfile] then
            Priority of parameter add // If there is a match, the parameter
                weight is increased
        end
    end
    if Priority of parameter >CriticalValue then
        Priority of parameter sub // If the critical value is exceeded,
            reduce the weight
    end
end
ParaPriorityQueue = sort(parameter) by Priority of parameter // Sort
    parameters by weight
for parameter in ParaPriorityQueue do
    put mutate(FuncReq, parameter) in SeedQueue// Mutation in the order of
        ParaPriorityQueue to generate fuzzing seeds
end
```

3. IoTParser

The architecture of IoTParser is shown in Figure 5, which can be divided into three parts, namely web crawler, static analysis, and fuzzer.

First, we decompress the firmware. Then, through the static analysis, we can obtain the interfaces, page paths, and shared keywords dictionary. After that, we emulate the firmware. The web crawler automatically traverses all the function entries of the web interface through the proxy and uses the request captured as the initial seeds. For the interfaces and page paths obtained in the static analysis of the firmware, if they do not exist in the seed queue, the crawler will crawl and parse the interfaces and pages, then add them to the seed queue to obtain a high-coverage seed queue. Using the shared keyword dictionary, we calculate the weight information of the seed queue, then mutate the seeds with higher weights first, and, finally, generate the seed priority queue. Next, the fuzzer performs fuzzing in the order of the priority queue and finally outputs a vulnerability report.

We use FirmAE [8] with a high simulation success rate to simulate the firmware, and the crawler module uses the popular web crawler tool RAD [17]. At the same time, the interfaces and page paths extracted by the firmware static analysis module can supplement the web crawler to increase the coverage. We transform the fuzzer module based on the popular web fuzzing tool W13scan [12]. Compared with the state-of-the-art tool boofuzz [18], W13scan is more suitable for web protocols, without the need to write templates, and can send more targeted packets.

After the seed priority queue is generated, we mutate the initial request to generate a test queue according to the priority. There are three mutation strategies: (a) delete parameter value; (b) replace parameter value with vulnerability payload; and (c) add vulnerability payload after parameter value. Because web-type vulnerabilities are different from memory-type vulnerabilities, the vulnerability payload usually needs to meet certain grammar rules rather than being randomly generated. Therefore, we use vulnerability payload dictionaries for fuzzing, such as fuzzDict [19].

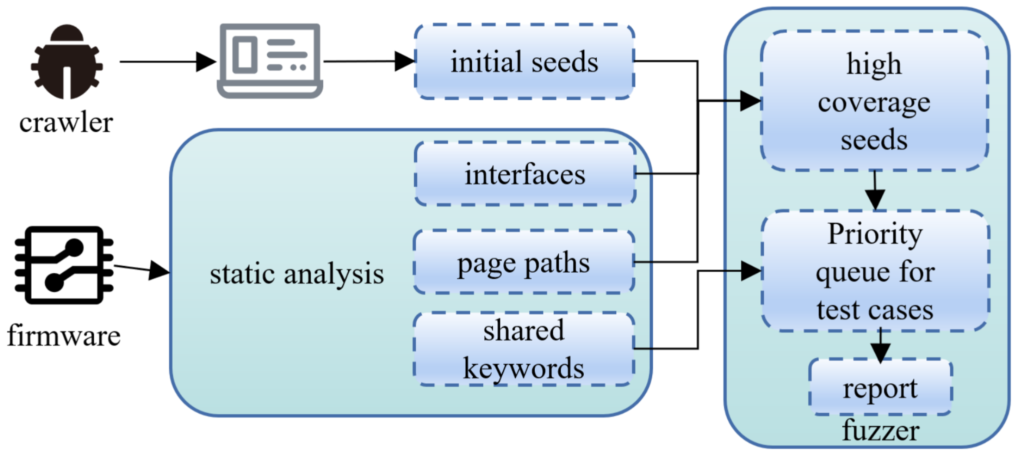


Figure 5. Architecture of IoTParser.

IoTParser supports the detection of four types of vulnerabilities in IoT devices: overflow, command injection, XSS, and unauthorized access. For different vulnerability types, we have designed vulnerability monitoring methods according to the vulnerability characteristics [20].

1. Overflow. By monitoring the running log of the simulation environment, when a memory access error occurs, it can be judged whether an overflow has occurred by performing keyword matching on the log.
2. Command injection. Before the simulation, we inserted a probe into the firmware simulation environment. When command injection occurs, the probe will generate a log file to help us detect command injection vulnerabilities.
3. XSS, matching whether the page response contains a specific string.
4. Unauthorized access. After removing the authentication fields in the request, such as "cookie", "Authorization", etc., if the request without authentication status can still obtain the same return information as before, it can be judged that there is an unauthorized vulnerability.

4. Evaluation

We selected five types of IoT devices from four mainstream manufacturers for testing, including routers and wireless cameras. As shown in Table 1, these five devices can be simulated to access the device web interface for the next step.

Table 1. IoT devices for testing.

Type	Vender	Device	Version
Router	ASUS	NetgeRT-N53	3.0.0.4
	Netgear	WNAP-320	2.0.3
	Netgear	WNDR-3700	1.0.0.10
	D-Link	DIR-825	2.05b
IP Camera	Trendnet	IP110-wn	1.2.2.68

Testing Environment: Experiments are conducted on an 8-core AMD 4900HS 3.0 GHz CPU machine with 8 GB of RAM. The operating system is Ubuntu 18.04 LTS.

4.1. Comparison with Current Work

To illustrate the effectiveness of our tool, we compare IoTParser with W13scan and Firmhunter [7].

Since W13scan has no crawler function, we add the crawler function part to W13scan, marked as W13scan*, to ensure that it has the same initial seed as IoTParser. Firmhunter is the first work to introduce the seed scheduling strategy into the IoT web fuzzing. Compared with the previous work, the speed of vulnerability discovery has been greatly improved, so we chose to compare it.

The final experimental results are shown in Table 2. Since Firmhunter only relies on the web crawler for seed generation, the coverage rate is relatively low, resulting in four vulnerabilities not being discovered. Among the vulnerable seeds that can be covered, the discovery speed of IoTParser is significantly higher than that of Firmhunter, which shows the effectiveness of scheduling based on shared keywords. Since W13scan* cannot monitor the running information of the firmware, it cannot find vulnerability without echo information but can only find XSS and unauthorized access. At the same time, since there is no scheduling method, the tests are performed in sequence, resulting in very low efficiency.

Table 2. Experimental results of IoTParser, Firmhunter, and W13scan*. unassigned-num refers to the serial number of the 0-day vulnerability discovered.

CVE	Vender	Device	Vuln Type	IoTParser	Firmhunter	W13scan*
CVE-2019-20082	asus	RT-N53	BO	21 m 43 s	57 m 8 s	NA
unassigned-0	asus	RT-N53	CI	43 m 40 s	1 h 23 m	NA
CVE-2021-31655	Trendnet	ip110wn	XSS	7 m 37 s	16 m 04 s	17 m 34 s
CVE-2019-11417	Trendnet	ip110wn	BO	8 m 48 s	21 m 24 s	NA
CVE-2018-19241	Trendnet	ip110wn	BO	16 m 26 s	47 m 58 s	NA
unassigned-1	Trendnet	ip110wn	XSS	31 m 40 s	37 m 17 s	57 m 28 s
unassigned-2	Trendnet	ip110wn	XSS	3 m 15 s	9 m 18 s	3 m 7 s
CVE-2016-1555	netgear	wnap320	CI	4 m 9 s	NA	NA
unassigned-3	netgear	wnap320	unauth	4 m 24 s	NA	4 m 11 s
unassigned-4	netgear	wnap320	CI	19 m 35 s	24 m 21 s	NA
unassigned-5	netgear	wndr3700	CI	4 h 35 m	5 h 27 m	NA
unassigned-6	netgear	wndr3700	CI	4 h 16 m	NA	NA
CVE-2019-9126	dlink	dir-825	unauth	6 m 41 s	NA	6 m 53 s

It should be noted that unauthorized access is a logic vulnerability, so we only need to test the initial seed, no mutation is required, so there is no improvement effect. The seed corresponding to the unassigned-2 vulnerability is in the first place in order, and it is still the first after scheduling according to the weight calculation. Therefore, IoTParser and W13scan* behave similarly here, while Firmhunter needs to perform dynamic requests and parse logs which takes a long time.

4.2. Component Validity Testing

In order to explore the effectiveness of each component of IoTParser, we designed in-depth comparative experiments. IoTParser-null refers to removing the static analysis module in IoTParser, and IoTParser-cov refers to adding only the interfaces and page paths in the static analysis module of the firmware. IoTParser is a complete program. We test the 13 vulnerabilities in the previous experiment, and the test results are shown in Table 3. We have two problems to prove:

1. Whether interfaces and page paths extraction can increase coverage, thereby increasing vulnerability discovery capabilities;
2. Is the scheduling strategy based on shared keywords effective?

Table 3. Experimental results of IoTParser-null, IoTParser-cov and IoTParser. Each experimental result is represented by a triple (a, b, c), a: the number of initial seeds; b: the serial number of the request for which the vulnerability was found in the fuzz process; and c: the number of all requests sent by fuzzer.

CVE	Vender	Device	Vuln	IoTParser-null	IoTParser-cov	IoTParser	Speed Improve
CVE-2019-20082	asus	RT-N53	BO	(114, NA)	(233, 5581, 10,875)	(233, 1576, 10,875)	71%
unassigned-0	asus	RT-N53	CI	(114, 6756, 12,580)	(233, 9136, 24,650)	(233, 3639, 24,650)	60%
CVE-2021-31655	Trendnet	ip110wn	XSS	(15, 61, 120)	(117, 211, 4590)	(117, 121, 4590)	42%
CVE-2019-11417	Trendnet	ip110wn	BO	(15, NA)	(117, 601, 2295)	(117, 91, 2295)	84%
CVE-2018-19241	Trendnet	ip110wn	BO	(15, NA)	(117, 1921, 2295)	(117, 211, 2295)	89%
unassigned-1	Trendnet	ip110wn	XSS	(15, NA)	(117, 2191, 4590)	(117, 1231, 4590)	43%
unassigned-2	Trendnet	ip110wn	XSS	(15, NA)	(117, 1, 4590)	(117, 1, 4590)	0%
CVE-2016-1555	netgear	wnap320	CI	(41, NA)	(76, 1123, 2686)	(76, 1, 2686)	100%
unassigned-3	netgear	wnap320	unauth	(41, NA)	(76, 4, 76)	(76, 4, 76)	0%
unassigned-4	netgear	wnap320	CI	(41, 953, 1938)	(76, 1021, 2686)	(76, 783, 2686)	23%
unassigned-5	netgear	wndr3700	CI	(181, 6495, 28,050)	(312, 14,629, 43,656)	(312, 16,525, 43,656)	−12%
unassigned-6	netgear	wndr3700	CI	(181, NA)	(312, 23,869, 43,656)	(312, 15,403, 43,656)	35%
CVE-2019-9126	dlink	dir-825	unauth	(69, NA)	(186, 33, 186)	(186, 33, 186)	0%

Q1: Due to the limitation of crawler capabilities, IoTParser-null cannot cover many vulnerabilities. It can be seen that Trendnet IP-110wn if only relying on the web crawler, only 15 initial seeds are generated. By analyzing the front-end page information of the device, we found that all menus on the front-end page of IP-110wn implement the jump function by binding js events on the "img" tag, which is very different from the traditional front-end menu technology. Although the flexibility of the front-end technology supports this implementation method, the crawler cannot face this complex situation, resulting in deficient seed coverage. After adding the interfaces and page paths, the limitation of the menu bar is bypassed by directly parsing different function pages. At the same time, for some hidden page information and interfaces that cannot be found from the front end, such as CVE-2016-1555, unassigned-3, unassigned-6, and CVE-2019-9126, IoTParser can further compensate for the crawler coverage by extracting firmware information to enhance vulnerability discovery capabilities.

Q2: From the experimental results, it can be seen that after adding the scheduling policy, the sequence number of the requests that discovered the vulnerability decreased by 48.6% on average, indicating that the speed of finding the vulnerability was accelerated by 48.6% on average. It should be noted that logic vulnerabilities, such as unauthorized access, are not considered here. However, for unassigned-5, since the request with this vulnerability is very high in the order of generation, it is faster than after scheduling. However, our scheduling strategy is to schedule all requests, and this situation does not affect the overall efficiency improvement.

The final experimental results show that with the help of the mining and analysis of firmware information, IoTParser's vulnerability discovery ability is improved by 44% on average, and the vulnerability discovery efficiency is improved by 48.2% on average.

4.3. Case Study

CVE-2016-1555 is a command injection vulnerability in the WNAP-320 router. The exploit entry is located in "boardDataNA.php" and "boardDataWW.php", but these two files are independent and are not referenced in other files. Therefore, it is impossible to capture this interface directly through the crawler at the front end. However, by extracting the page paths in the firmware, the crawler can successfully cover this function entry.

Unassigned-3 is unauthorized access in WNAP-320. When an existing user is logged in, "recreate.php" will be called to prompt the user whether to cut off another access state when logging in again elsewhere. However, due to improper permission restrictions, the cookie information of all users will be leaked. Since the web crawler cannot meet the conditions, it cannot obtain this interface. Similar to unassigned-6, this vulnerability is a command injection in the WNDR-3700 router. Only if certain conditions are met front-end will dynamically render the vulnerable page, so the crawler cannot obtain this interface. These problems can be solved by extracting the firmware page paths.

CVE-2019-9126 is unauthorized access in the DIR-825 router. This interface leaks various configuration information about the router. However, it does not exist in a web-related file but is received and processed in the back-end binary file "httpd". Therefore, IoTParser can successfully discover this vulnerability by extracting interfaces from firmware.

5. Related Work

Vulnerability detection for firmware can be divided into two research trends: static analysis and dynamic analysis. Our work focuses on dynamic analysis.

5.1. Static Analysis

In IoT scenarios, static analysis can be performed on all files in the firmware file system but generally on back-end binaries. Vulnerabilities are found through reverse engineering or source code analysis, for example.

Thomas [21] et al. proposed a static analysis method for hard-coded vulnerabilities. The method first extracts comparison functions in the program and then uses control flow analysis techniques to evaluate and rank these comparison functions. For the generic taint class vulnerability type, Cheng [22] et al. proposed a method for detecting firmware taint class vulnerabilities in IoT devices by analyzing functions in the program and extracting variable descriptions, data types, and other information for modeling. This work found eight known vulnerabilities and 13 0-day vulnerabilities and outperformed angr [23] in terms of time overhead. Text-based similarity comparison is an emerging approach in static analysis. Costin [24] et al. conducted the first public, large-scale static analysis of firmware security, confirming that certain vulnerabilities infected 140,000 devices in cyberspace through similar file associations. This work is the first to propose the discovery of homologous vulnerabilities utilizing firmware module correlation. Chen et al. proposed SaTC [25], a static analysis tool for IoT devices. The core principle is that they found such a phenomenon of sharing between front-end and back-end keywords. By extracting keywords from the front-end and using them as the entry for taint analysis [26] on the back-end binary, SaTC accelerates vulnerability discovery and improves accuracy.

Compared to dynamic analysis, static analysis usually has a higher coverage rate. However, the static analysis relies on manual rules of thumb and has not yet found an appropriate balance between accuracy and efficiency, which makes its implementation in real-life scenarios unsatisfactory. Taking advantage of the high coverage of static analysis, we extract as many function entries as possible through static analysis in the pre-fuzz stage, and then hand them over to the fuzz stage for vulnerability detection. This combination makes our tool much more capable of vulnerability discovery.

5.2. Dynamic Analysis

Dynamic analysis (e.g., fuzz) has attracted much attention from researchers in the past few years, and related systems have been deployed and used on a large scale in the industry. However, the special characteristics of IoT devices, such as limited hardware resources and complex system architecture, make traditional fuzz not directly applicable to vulnerability discovery in IoT devices.

Recent dynamic analyses for IoT devices can be classified according to the interfaces tested. Previously we described the prevalence of the web interface and the advantages of using it for fuzz. More and more state-of-the-art is choosing this interface for vulnerability detection. Firmadyne [20] accesses web pages, collects SNMP information, and uses existing attack scripts to test the emulated firmware. FirmAE [8] is based on Firmadyne, which mainly solves the problem of the low success rate of emulation. They proposed an arbitration mechanism, which increases the simulation success rate from about 20% to about 80%. FirmFuzz [5] generates test cases through the web crawler and then mutates them for testing. As a result, it achieves better vulnerability detection results than previous dynamic analysis research. Building on FirmFuzz, FirmHunter proposes a stateful queue method and a two-level scheduling mechanism to discover vulnerabilities with stateful dependencies and speed up vulnerability discovery. WMIFuzzer [24] adopts mandatory interface automation technology and performs input and click operations on all elements to increase coverage as much as possible. It aims to extend the coverage of the crawler to enhance vulnerability discovery, and experiments show that WMIFuzzer can find more vulnerabilities than mainstream fuzz tools. IoTScope [13] constructs probing requests through static analysis to test physical devices, and narrows down the scope of identification by filtering out irrelevant requests and interfaces through differential analysis. It pinpoints hidden interfaces by attaching various device-setting parameters in the probing requests and matching keywords of sensitive information.

From the initial simple analysis of web pages to the continuous optimization of crawlers for more function entries, dynamic analysis has been confined to the limited resources of web entry. None of these studies could discover hidden interfaces that are not displayed on the web entry or some function pages that are dynamically generated by js scripts, etc. IoTScope can obtain more function entries through static analysis of the firmware, but instead of using fuzz techniques to find vulnerabilities, they only looked at the issue of permissions for the interface by constructing requests. Our work extracts interfaces, page paths, and shared keywords from firmware to aid fuzz. Compared to existing work, we have enhanced the capability and speed of vulnerability discovery. Furthermore, because the firmware is only statically extracted once before fuzz testing, it is faster and has less overhead.

In addition to the web interface, some research is based on other interfaces. To reduce the difficulty and effort of reversing executable files when analyzing vulnerabilities, Chen [27] et al. designed IoTFuzzer to detect memory vulnerabilities in IoT devices by analyzing the device's Android APP. It can mutate the source data that constitutes the message and trigger memory vulnerabilities by mutating the length and type of the data. Matheus [28] et al. proposed a new type of vulnerability, SweynTooth, which affects devices running the low-energy Bluetooth protocol. Their experiments found that more than 480 devices were using the vulnerable SDK. Research on these interfaces is not suitable for large-scale vulnerability discovery due to limited interface resources or the small number of devices that contain this interface.

6. Discussion

From the experimental results, we can see that IoTParser's vulnerability discovery capability and vulnerability discovery speed are significantly better than state-of-the-art FirmHunter, with an average improvement of 44% and 48.2%, respectively. However, there were still some ways for future improvements.

Maintaining authentication status: IoT devices are different from traditional web applications. Many operations need to interact with the underlying layer, which can easily lead to state loss during the fuzz process. If the authentication status is lost, the fuzz will not be able to access the function entries that require authentication, resulting in undiscovered vulnerabilities. Therefore, we will introduce monitoring of the authentication status in future work to ensure the effective operation of fuzz.

Decompression of encrypted firmware: Many IoT vendors have recently packaged their firmwares with encryption to prevent information leakage. These firmwares cannot be decompressed by conventional tools [10], so IoTParser cannot perform vulnerability discovery on such devices. We will work on encrypted firmware in the future to enhance the scope of IoTParser.

Detecting more types of vulnerabilities: With the development of IoT, web portals are becoming more powerful, and the proportion of traditional web vulnerabilities in IoT devices is increasing year on year. Detection for these vulnerabilities must continue to be refined in the following research.

Support for more protocol types: In IoT devices, there are many customized protocol types, such as UPnP protocols, which send data in a different format to traditional data request methods, and for which future work could be further refined to enhance the discovery of vulnerabilities.

Inspired by research into existing static and dynamic analysis techniques, we extracted interface and page paths through firmware information mining to extend the coverage of the crawler, and also extracted hidden shared keyword information from the firmware to speed up vulnerability discovery and reduce overhead, resulting in good experiments. In addition to the three types of information we extracted, there are many other hidden correlations in the firmware waiting to be discovered, and depending on the use case, different information can be analyzed and extracted to aid vulnerability discovery, all of which can be further investigated in the future.

7. Conclusions

We propose IoTParser, the first web fuzzing tool using firmware information mining to aid fuzzing. IoTParser extracts interfaces, page paths, and shared keywords from firmware. Experimental comparisons show that IoTParser's vulnerability discovery capabilities and speed have improved by 44% and 48.2%, respectively, compared to the state-of-the-art tools. In addition, IoTparser has found 13 vulnerabilities, seven of which are 0-day. IoTParser can provide an idea for future IoT vulnerability detection research: Static analysis (firmware information mining) can be used to guide dynamic analysis to achieve better results. However, our research is only on firmware that can be decompressed, and the supported protocols and vulnerability types are not comprehensive enough. These limit the scope of IoTParser and will be the target of our future work.

Author Contributions: Conceptualization, Y.G. and X.Z.; methodology, Y.G.; software, Y.G.; validation, Y.G., X.Z.; data curation, Y.G.; writing—original draft preparation, Y.G.; writing—review and editing, Z.W., W.X., B.W., E.W. and Z.W.; All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Natural Science Foundation of China (61902412), and the Research Project of National University of Defense Technology (ZK20-17).

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

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

Abbreviations

The following abbreviations are used in this manuscript:

IoT	Internet of things
XSS	Cross-site Scripting
CVE	Common vulnerabilities and exposures
PHP	Hypertext Preprocessor
CGI	Common Gateway Interface

References

1. Internet of Things. Available online: https://en.wikipedia.org/wiki/Internet_of_things (accessed on 9 June 2022).
2. Alaba, F.A.; Othman, M.; Hashem, I.A.T.; Alotaibi, F. Internet of Things security: A survey. *J. Netw. Comput. Appl.* **2017**, *88*, 10–28. [CrossRef]
3. IoT Connected Devices Worldwide 2030. Available online: http://timmurphy.org/2009/07/22/line-spacing-in-latex-documents/ (accessed on 15 August 2021).
4. Chen, C.; Cui, B.; Ma, J.; Wu, R.; Guo, J.; Liu, W. A systematic review of fuzzing techniques. *Comput. Secur.* **2018**, *75*, 118–137. [CrossRef]
5. Srivastava, P.; Peng, H.; Li, J.; Okhravi, H.; Shrobe, H.; Payer, M. Firmfuzz: Automated iot firmware introspection and analysis. In Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, London, UK, 15 November 2019; pp. 15–21.
6. Wang, D.; Zhang, X.; Chen, T.; Li, J. Discovering vulnerabilities in COTS IoT devices through blackbox fuzzing web management interface. *Secur. Commun. Netw.* **2019**, *2019*, 5076324. [CrossRef]
7. Yin, Q.; Zhou, X.; Zhang, H. FirmHunter: State-Aware and Introspection-Driven Grey-Box Fuzzing towards IoT Firmware. *Appl. Sci.* **2021**, *11*, 9094. [CrossRef]
8. Kim, M.; Kim, D.; Kim, E.; Kim, S.; Jang, Y.; Kim, Y. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In Proceedings of the Annual Computer Security Applications Conference, Austin, TX, USA, 7–11 December 2020; pp. 733–745.
9. Web Crawler. Available online: https://en.wikipedia.org/wiki/Web_crawler (accessed on 12 April 2022).
10. binwalk. Available online: https://github.com/ReFirmLabs/binwalk (accessed on 26 July 2014).
11. Attacks. Available online: https://owasp.org/www-community/attacks/ (accessed on 25 April 2020).
12. w13scan. Available online: https://github.com/w-digital-scanner/w13scan (accessed on 20 Augest 2019).
13. Xie, W.; Chen, J.; Wang, Z.; Feng, C.; Wang, E.; Gao, Y.; Wang, B.; Lu, K. Game of Hide-and-Seek: Exposing Hidden Interfaces in Embedded Web Applications of IoT Devices. In Proceedings of the ACM Web Conference 2022, Association for Computing Machinery, Lyon, France, 25–29 April 2022; pp. 524–532.
14. PHP-Header. Available online: https://www.php.net/manual/en/function.header.php (accessed on 10 June 2010).
15. Request JavaScript API. Available online: https://www.javascripture.com/Request (accessed on 9 Feburary 2022).
16. Strings—Linux Man Page. Available online: https://linux.die.net/man/1/strings (accessed on 11 Feburary 2009).
17. Chaitin/Rad. Available online: https://github.com/chaitin/rad (accessed on 30 Feburary 2021).
18. Jtpereyda/Boofuzz. Available online: https://github.com/jtpereyda/boofuzz (accessed on 4 December 2015).
19. fuzzDicts. Available online: https://github.com/TheKingOfDuck/fuzzDicts (accessed on 26 May 2019).
20. Chen, D.D.; Woo, M.; Brumley, D.; Egele, M. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. *NDSS* **2016**, *1*, 1.
21. Thomas, S.L.; Chothia, T.; Garcia, F.D. Stringer: Measuring the importance of static data comparisons to detect backdoors and undocumented functionality. In *European Symposium on Research in Computer Security*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 513–531.
22. Cheng, K.; Li, Q.; Wang, L.; Chen, Q.; Zheng, Y.; Sun, L.; Liang, Z. DTaint: Detecting the taint-style vulnerability in embedded device firmware. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg, 25–28 June 2018; pp. 430–441.
23. Wang, F.; Shoshitaishvili, Y. Angr—The Next Generation of Binary Analysis. In Proceedings of the 2017 IEEE Cybersecurity Development (SecDev), Cambridge, MA, USA, 24–26 September 2017; pp. 8–9. [CrossRef]
24. Costin, A.; Zaddach, J.; Francillon, A.; Balzarotti, D. A Large-Scale Analysis of the Security of Embedded Firmwares. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 95–110.
25. Chen, L.; Wang, Y.; Cai, Q.; Zhan, Y.; Hu, H.; Linghu, J.; Hou, Q.; Zhang, C.; Duan, H.; Xue, Z. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 11–13 August 2021; pp. 303–319.
26. Alashjee, A.M.; Duraibi, S.; Song, J. Dynamic Taint Analysis Tools: A Review. *Int. J. Comput. Sci. Secur. (IJCSS)* **2019**, *13*, 231–244.
27. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Lin, Z.; Wang, X.; Lau, W.C.; Sun, M.; Yang, R.; Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 18–21 February 2018.
28. Garbelini, M.E.; Wang, C.; Chattopadhyay, S.; Sumei, S.; Kurniawan, E. SweynTooth: Unleashing Mayhem over Bluetooth Low Energy. In Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20), Philadelphia, PA, USA, 15–17 July 2020; pp. 911–925.