FIRMKING: Detecting Insecure HTTP Form Handling Functions in IoT Firmware

Shizhi He
Wuhan University

Second Name
Second Institution

Abstract

In recent years, the Internet of Things (IoT) devices have become ubiquitous in people's life. However, a large number of IoT devices are vulnerable to attacks. If a remote attacker compromises an IoT device, he is very likely to gain access to the user network and tamper with the network traffic. IoT firmware architectures are diverse and different from desktop platforms. Traditional analysis and debugging tools are often incompatible with IoT devices. Therefore, IoT firmware security analysis is a troublesome task. Utilizing static or dynamic analysis alone has many limitations. In this paper, we present FIRMKING, a novel framework that combines static function analysis with fuzzing to detect vulnerabilities in HTTP form handling functions (FHF) in IoT firmware. FIRMKING can perform static analysis on the Web server binary, and send the analysis results to fuzzer, improving the efficiency of vulnerability discovery. We evaluated FIRMKING with a real-world dataset of 9,293 firmware images and discovered 17 0-day vulnerabilities, including 7 CVEs, thereby demonstrating its efficiency in detecting vulnerabilities in IoT firmware.

1 Introduction

Over the last few years, the number of IoT devices (eg., routers, IP cameras, network storage) is increasing rapidly. However, a large number of IoT devices are vulnerable to attacks [8, 26]. Researchers have developed techniques and frameworks to discover vulnerabilities in IoT firmware. For example, the RouterSploit [31] framework is an open-source exploitation framework dedicated to IoT devices. A large number of CVEs [12] have been assigned for IoT device vulnerabilities and the exploits are publicly available [13, 23]. Security protection mechanisms (eg., ASLR, canary) are not widely used by IoT devices due to limited resources. Because IoT server binaries usually execute with root privileges, if a remote attacker compromised an IoT device, he is very likely to gain access to the user network and tamper with the network traffic.

Many challenges still exist in IoT security research. For example, IoTFuzzer [7] only supports App-based IoT devices. FIRMADYNE [6] can perform full-system emulation, but IoT binaries often crash due to the lack of hardware environment such as NVRAM. Static approaches such as Firmalice [33] and Karonte [30] require prohibitive time and resources.

Since IoT firmware architectures are diverse and different from desktop platforms, traditional analysis and debugging tools are often incompatible with IoT devices. For example, it is always difficult to debug the programs on IoT systems due to the lack of tools such as GDB [17]. Some vendors encrypt their firmware images or do not share firmware online. Source codes for most IoT binaries are not available. Emulation always crashes due to the lack of hardware dependency. Traditional fuzzing techniques have poor performance in IoT environment. Therefore, static or dynamic analysis on IoT firmware is a troublesome task.

We propose a novel technology that combines static function analysis with fuzzing. Many IoT devices rely on Web interfaces for user interaction and administration, so Web interfaces provide a considerable attack surface. We collected a large-scale dataset of real-world firmware images and extracted Web server binaries from them. In each binary, we searched for the names and parameters of HTTP POST forms and used the information to generate an individual session file for each form. Next, we started fuzzing with these session files one after another and save the results.

In this article, we presented FIRMKING, an automated firmware testing framework that combines static function analysis and fuzzing to detect vulnerabilities in FHFs in Web server binaries. From the user's viewpoint, FIRMKING is able to load the Web server binary, perform static analysis, write the result into session files and efficiently conduct fuzzing to discover IoT vulnerabilities.

We evaluated FIRMKING with a real-world dataset of 9,293 firmware images. FIRMKING can discover 17 0-day vulnerabilities. We reported our findings to vendors and received 7 CVEs. We obtained the following results: (1) The static analysis module can extract information about FHFs inside the Web server binary and save the results to assist the fuzzer. (2) With the information provided by the static analysis module, the fuzzer can find vulnerabilities more quickly. (3) FIRMKING can effectively discover undisclosed vulnerabilities in IoT Web server binaries.

Contribution. In summary, the contributions of our paper are as follows:

- *New Technique.* We develop a novel technology that combines static function analysis with fuzzing to detect vulnerabilities in FHFs in IoT firmware. First, we perform static analysis on the Web server binary, and then the analysis results are sent to a blackbox fuzzer. In this way, we can improve the efficiency of IoT vulnerability discovery.

- *New Framework.* We present FIRMKING, the first framework that combines static function analysis with fuzzing to detect vulnerabilities in FHFs in IoT firmware.

- *New Findings.* We gathered a real-world dataset of 9,293 firmware images across 10 vendors and successfully extracted 2,230 Web server binaries. FIRMKING can discover 17 0-day vulnerabilities. We reported our findings to vendors and received 7 CVEs.

2 Background

2.1 Firmware Security Analysis

An IoT firmware is a program in FLASH or EEPROM chip of the device, which serves as an interface between the upper layer software and underlying hardware. It usually includes a firmware header, boot loader, file system, and provides status monitoring, system control, data collection with limited computing and memory resources. To perform a security analysis on a device, the target firmware is usually required and a debugging environment needs to be established. The firmware images can be downloaded by Web crawlers such as Scrapy [32] or extracted from physical devices.

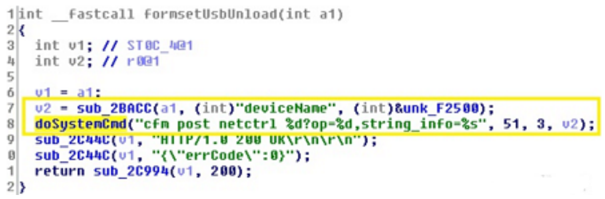
Once the target firmware is acquired, it will be unpacked and extracted by firmware extraction tools such as Binwalk [3], Squashfs-tools [35]. In a few scenarios, the firmware image is encrypted or customized, this makes the task much more difficult. A successful decryption of firmware is needed before extraction. If the target firmware can not be successfully extracted, further firmware-based analysis would be in vain.

After extracting the file system, security analysts will perform static analysis on binary executables such as *httpd*, *upnpd*. IDA [21] and Ghidra [16] are commonly used tools for IoT binary static analysis. Manually static analysis using these tools involves lots of repetitive efforts, like checking insecure function calls and tracing data-flow. On the other hand, IDA scripts and program analysis frameworks automate the process. At the same time, security analysts may perform fuzzing, symbolic execution, dynamic taint analysis and other techniques on important binaries. However, if physical devices are not available, security analysts have to set up an environment to emulate target devices or at least execute the target binaries.

Firmware emulation provides useful interfaces for dynamic analysis when physical devices are not reachable. QEMU [28] is widely used for firmware emulation. There are mainly two levels of emulation: user-level and system-level. User-level emulation is fast and simple, and it only emulates one binary program. System-level emulation fully runs the target operating system including the kernel. It is based on QEMU system mode. However, binary often performs hardware-based operations such as NVRAM. If the physical device is unavailable, the process often crashes in user-level and system-level emulation.

2.2 IoT HTTP POST Data Handling

Many IoT Web server binaries use the HTTP POST method to receive user instructions such as obtaining configuration, modifying configuration, controlling device. Next, the POST parameters are passed to a *form handling function (FHF)*. Figure 1 shows an example of FHF in IDA. In Listing 1, POST parameters *name* and *pass* are retrieved on line 6 and 7.



```
int __fastcall formsetUsbUnload(int a1)
{
  int v1; // ST0C_4@1
  int v2; // r0@1

  v1 = a1;
  v2 = sub_2BACC(a1, (int)"deviceName", (int)&unk_F2500);
  doSystemCmd("cfm post netctrl %d?op=%d,string_info=%s", 51, 3, v2);
  sub_2C44C(v1, "HTTP/1.0 200 OK\r\n\r\n");
  sub_2C44C(v1, "{\"errCode\":0}");
  return sub_2C994(v1, 200);
}
```

Figure 1: An example of FHF in IDA (sub_2BACC is equivalent to websGetVar).

Listing 1: An example of FHF code.

```
 1 static void login_fun(Webs *wp)
 2 {
 3 char *uname = NULL;
 4 char *passwd = NULL;
 5 //get the input value in query stream
 6 uname = websGetVar(wp, "name", NULL);
 7 passwd = websGetVar(wp, "pass", NULL);
 8 printf("username␣=␣%s\n", uname);
 9 printf("password␣=␣%s\n", passwd);
10 websHeader(wp);
11 websWrite(wp, "Name:␣%s,␣Pass:␣%s",
12           uname, passwd);
13 websFooter(wp);
14 websDone(wp);
15 }
```

Popular IoT Web server programs such as GoAhead [18] and Boa [4] implement a URL processor which interprets URLs starting with */goform*, */action*, etc. Each URL is handled by an FHF. If POST parameters are passed without enough security check, a vulnerability such as buffer overflow or command injection will exist. For example, in Tenda AC15 router's *httpd* binary, parameter *deviceName* in FHF *formsetUsbOnload* is retrieved and passed to *doSystemCmd*. This command injection vulnerability (CVE-2020-10987) allows a remote attacker to inject a system command into *deviceName* and execute it with *root* privileges as Listing 2 shows.

Listing 2: HTTP POST data of setusbonload.

```
POST /goform/setUsbUnload HTTP/1.1
Host: 192.168.0.1
User-Agent: Mozilla/6.0 (Windows NT 10.0;
Win64; x64; rv:68.0) Firefox/68.0
Accept: */*
Accept-Language: en-US,en;q=0.6
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urle
ncoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 19
Connection: close
Referer: http://192.168.0.1/samba.html?ra
ndom=0.035894671968671
Cookie: password=40203abe6e81ed98cbc97cdd
6ec4f144flqtgb

deviceName=; reboot
```

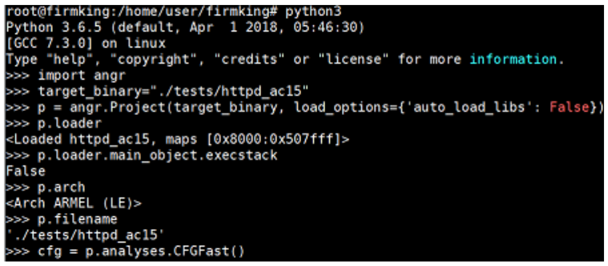
A large amount of IoT Web server binaries share the same procedure of handling user instructions from POST data. Therefore, vulnerabilities in FHFs have a great impact on IoT devices. To make matters worse, a successful attack often results in arbitrary command execution with *root* privileges because IoT server binaries usually run as *root*.

2.3 Angr Framework

Angr [2] is Python framework for automated binary analysis which supports multiple architectures. It mainly integrates following techniques:

- Loading the target binary into analysis framework.
- Translating the target binary into intermediate representation (IR).
- Perform a partial or full-program static analysis, such as control-flow graph (CFG) recovery, dependency analysis, program slicing.
- Perform complex dynamic analysis, such as symbolic execution, program state exploration.

Angr uses a module called *CLE* to obtain the loading information of a binary in the virtual address space. The result is called the *loader*. For example, *proj.loader.min_addr* returns the minimum virtual address of the binary, *proj.loader.main_object.execstack* shows whether the stack is executable or not as Figure 2 shows.



```
root@firmking:/home/user/firmking# python3
Python 3.6.5 (default, Apr  1 2018, 05:46:30)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import angr
>>> target_binary="./tests/httpd_ac15"
>>> p = angr.Project(target_binary, load_options={'auto_load_libs': False})
>>> p.loader
<Loaded httpd_ac15, maps [0x8000:0x507fff]>
>>> p.loader.main_object.execstack
False
>>> p.arch
<Arch ARMEL (LE)>
>>> p.filename
'./tests/httpd_ac15'
>>> cfg = p.analyses.CFGFast()
```

Figure 2: Binary analysis with angr.

Angr can generate two types of CFG: a static CFG (CFGFast) and a dynamic CFG (CFGEmulated). The CFG result produces an object called the *FunctionManager*, accessible through *cfg.kb.functions*. Function objects have important properties such as *func.blocks*, *func.get_call_sites()*, *func.string_references()*, etc.

Angr uses *basic blocks* as the unit of program analysis. We use *block = proj.factory.block(address)* to lift a block of code from a given address. Next, *block.pp()* can print a disassembly of that block.

2.4 Fuzzing

Fuzzing is an automated software vulnerability discovering technique. Its idea is to repeatedly execute the target binary with a large amount of malformed data while monitoring the status and behavior of the process. If an abnormal state appears, such as a crash, then security analysts will collect crash samples and examine the error locations. In this way, potential vulnerabilities can be detected. Typically, fuzzing has the following steps: select the test target, determine the input format, generate the test case, execute the test case, monitor the abnormal behavior, judge the vulnerability availability, and generate the utilization code. After getting the target binary and input format, fuzzing utilizes random functions to generate randomized test cases. As long as the randomness of test cases is strong enough and the number of test cases is large enough, some potential vulnerabilities will be discovered with great probability. Many influential vulnerabilities were discovered by fuzzing, such as the OpenSSL Heartbleed vulnerability (CVE-2014-0160), SMB protocol vulnerabilities, Android multimedia file format vulnerabilities, etc. There are different fuzzers according to different targets, such as Wfuzz [37] for Web application, AFL [1] for Linux binary. As for network protocol fuzzers, Sulley [36], Boofuzz [5] and Peach-fuzzer [15] are widely used.

Fuzzing can be categorized into whitebox, greybox and blackbox. A whitebox fuzzer generates input data based on a time-consuming analysis of the binary structure, execution flow and input conditions, and it can avoid generation of useless inputs. A blackbox fuzzer directly generate lots of random input data to test the binary in a short time, and it does not utilize feedback from the binary to guide the generation of inputs. A greybox fuzzer utilizes information to guide the generation of inputs. For example, AFL can instrument the binary either statically or dynamically and collect code coverage information such as new execution paths. The information will be used as seeds to generate new inputs. Greybox fuzzing has greater accuracy than blackbox fuzzing and more efficiency than whitebox fuzzing. However, source code for most IoT binaries is not available. Under this circumstance, dynamic instrumfentation is the only viable choice and the efficiency significantly declines.

3 Design

3.1 Motivation

We aim to develop an automated vulnerability discovering framework for IoT Web server binary. The basic idea of our approach is to perform static analysis on the binary and send the results to a blackbox fuzzer for further vulnerability discovery. Our goal is to eliminate the limitations of pure static or pure fuzzing. Utilizing static analysis alone is difficult to verify the existence of vulnerability. When facing large-scale heterogeneous IoT firmware binaries without source code, the feasibility and efficiency of fuzzing is not ideal. FIRMKING can perform static function analysis and use the results to guide the fuzzer, improving the accuracy and efficiency of vulnerability discovery.

Specifically, our framework focuses on Web server binaries that contain FHFs. In this case, the most prevalent vulnerabilities are stack buffer overflow. Our goal is to detect buffer overflow vulnerabilities within a limited time and resources.

3.2 Overview

Our system includes an extractor, an analyzer and a fuzzer. First, the extractor fetches out the file system. Next, the analyzer loads the Web server binary and detects FHFs. In each FHF, the module finds POST interface, POST parameters and vulnerable functions, then utilize a template file to generate a session file for fuzzer. Finally, the fuzzer uses these session files to perform fuzzing on all the FHFs and alerts the vulnerabilities. We show the overview of FIRMCORN in Figure 3.

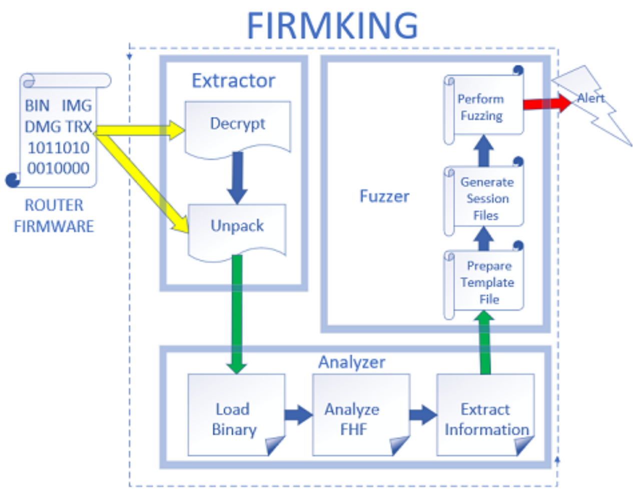


Figure 3: Architectural diagram of FIRMKING.

3.3 Challenges

In the design and implementation of FIRMKING, the following challenges need to be resolved:

- The software and hardware environment of products from different vendors are quite different. It is a hard task to perform large-scale analysis on various firmware images. To solve the problem, we focus on IoT binaries that contain FHFs in them. We implemented a uniform tool to perform static analysis on this kind of binaries. Besides, our blackbox fuzzer can ignore the internal differences between platforms.
- Source codes for most IoT binaries are not available. In this case, whitebox and greybox fuzzing have low feasibility. Therefore, we use a blackbox fuzzer guided by static function analysis.
- A few firmware images are encrypted. We analyzed their encryption algorithm and decrypted them manually.

4 IMPLEMENTATION

4.1 Extractor

Most of the downloaded images have file extensions of *.bin*, *.img*, *.dmg*, *.trx*. We utilized Binwalk [3] and Squashfs-tools [35] to extract the root file system from them. For a few encrypted images, we found out an approach to decrypt them manually.

In most cases, the firmware was not encrypted when it was factory released. An unencrypted middle release of the firmware contains the decryption routine. Subsequent firmware releases are encrypted. In this scenario, we can download all versions of the product on vendor websites, and look for the unencrypted middle release. Once the middle release is found, we extract it and obtain the decryption program to decrypt other firmware releases.

For example, we tried to extract the file system from D-Link DIR-878 v1.12. We collected all releases of DIR-878 (v1.00, v1.01, v1.10, v1.11, v1.12, v1.20) from D-Link support website, and found that v1.10 is the unencrypted middle version. We extracted it and found an interesting binary named */bin/imgdecrypt*. This is the program for firmware decryption. Now we can decrypt other firmware releases with *chroot . ./qemu-mipsel-static ./imgdecrypt target.bin*, and extract the firmware with *binwalk -Me target.bin*. In fact, this program can also decrypt other D-Link DIR products such as DIR-867.

Finally, we should extract all the Web server binaries for analysis. We developed shell command scripts to complete the work automatically.

4.2 Analyzer

Our framework performs static analysis on the target Web server binary. Angr supports MIPS and ARM architectures but sometimes it does not work well on IoT binaries. For example, when we want to get the function reference addresses in MIPS binaries, *list(p.kb.xrefs.get_xrefs_by_dst(func_address))* often returns *null* value. If we want to search certain instructions in a function, it is hard to realize. To solve this kind of problems, we developed our customized auxiliary functions to help automatically analyze the binary, as shown in Table 1.

Table 1: Auxiliary functions.

Function list	Function description
getstringfromaddr	get the string from a specified address inside binary
getintfromaddr	get the integer from a specified address inside binary
getsectionaddr	get the start address of a specified section
getdisasm	get all the instructions of a specified function
getinsfromaddr	get the instruction from a specified address
findaddrfromins	find the address of an instruction in a function
getgp	get the initial value of $gp register in a function (MIPS)
getformfunctionstrings	get POST parameters from a specified FHF
scanexecutable_arm	perform static analysis (ARM)
scanexecutable_mips	perform static analysis (MIPS)

We propose a novel algorithm to search FHFs in the binary and obtain the required information from each FHF, as shown in Algorithm 1. Angr loads the binary and generates a static CFG. Next, our framework scans each function and tries to find FHFs. In some devices such as Tenda AC15, the name of FHF starts with *form*. For example, *formexeCommand* handles user commands and *formsetMacFilterCfg* configures MAC address filter rules. However, some binaries stripped function names, so we are not able to identify FHFs by name. In this case, we look for a function that defines all the FHFs in the binary. We name this function *form define table (FDT)*. We studied the characteristics of FDTs and found that FDTs can be distinguished by instruction patterns. We search for instruction patterns and identify FDTs. Next, we look into the FDT and find instruction patterns to locate FHFs and corresponding POST interfaces. We walk into each FHF and search instructions that operates string parameters, as shown in Listing 3. The addresses of two parameters are passed to registers on line 2 and 4. Later, *websGetVar* is called to retrieve these stings from HTTP POST data on line 7. These strings are regarded HTTP POST parameters.

Listing 3: An example of FHF instruction patterns.

```
1 li $v0,0x4e0000
2 addiu $a1,$v0,(param1-0x4e0000)
3 li $v0,0x4e0000
4 addiu $a2,$v0,(param2-0x4e0000)
5 la $v0,websGetVar
6 move $t9,$v0
7 jalr $t9
```

Finally, we generate session files by replacing following fields inside the template session file: *host IP*, *port*, *cookie*, *POST interface*, *POST parameters*.

4.3 Fuzzer

We developed our network protocol fuzzer based on Mutiny [25]. We modified this framework. We do not generate traffic by replaying *PCAPs*. Instead, we directly send data to the target device according to the session file and wait for responses to detect crashes. As shown in Listing 4, a session file contains target port, max failure counts, protocol message definition, etc. In this study, we only send HTTP POST data to the target IoT device. First, we use a text editor to write a template session file. Next, for each FHF, the static analysis module obtains its POST interface and parameters and replace *formname* and *POST parameters* in the template to generate new session files. The *fuzz* keyword indicates the certain parts of the data to be mutated. We use Radamsa [20] to perform mutations. After generating these session files, we run the fuzzer to test the FHFs, one after another, and save the results automatically. Some vendors, such as Cisco, use HTTPS instead of HTTP for Web management interface. Our fuzzer can not generate HTTPS packets. To solve the problem, we use commands like *nvram set https_enable=0* to disable the HTTPS functionality.

Algorithm 1: FIRMKING Static Analysis.

```
Input: IoT Web server binary
Output: scripts for fuzzer
1  Load the binary: p = angr.Project(target_binary);
2  Generate a static CFG: cfg = p.analyses.CFGFast();
3  foreach function in cfg.kb.functions do
4      if function.name start with "form" then
5          Identify function as an FHF;
6          Get the addresses where FHF is called:
             p.kb.xrefs.get_xrefs_by_dst(function.address);
7          Get constant string near the FHF callsite;
8          Identify this string as POST interface name;
9      end
10     if function names are stripped then
11         Search for all FDTs;
12         Locate FHFs and corresponding POST
             interfaces in FDTs;
13     end
14 end
15 foreach FHF do
16     Find instruction patterns that operates strings;
17     Get constant strings near these instructions;
18     Identify strings as POST parameters;
19     foreach function.get_call_sites() do
20         if the function called by FHF is a dangerous
             function then
21             Show alerts;
22         end
23     end
24     Write FHF information to a fuzzer script;
25 end
```

Listing 4: A template session file.

```
max_failure 3
max_timeout 5
server_port 80
send_data 'POST␣/goform/formname
HTTP/1.1\r\n'
'Host:␣192.168.0.1\r\n'
'Cookie:␣password=5gk\r\n\r\n'
'arg0=' fuzz 'aaaaaaaaaaaaaaaaaaaa'
'&arg1=' fuzz 'aaaaaaaaaaaaaaaaaaaa'
'&arg2=' fuzz 'aaaaaaaaaaaaaaaaaaaa'
'&arg3=' fuzz 'aaaaaaaaaaaaaaaaaaaa'
'&arg4=' fuzz 'aaaaaaaaaaaaaaaaaaaa'
'&arg5=' fuzz 'aaaaaaaaaaaaaaaaaaaa'
```

5 Evaluation

In this section, we evaluate FIRMKING for finding vulnerabilities in firmware samples. The purpose of our evalution is to test how effective is our approach in finding real vulnerabilities in firmware. We first evaluate the performance of static FHF analysis. Then, we evaluate the effectiveness of fuzzer in finding vulnerabilities.

5.1 Experiments Setup

We utilized a Web crawler to download firmware images from websites. We wrote an individual parser for each website based on Python Scrapy framework. Sometimes the vendor used dynamically-generated content on the website, such as D-Link. The parser was unable to find the firmware images automatically, so we downloaded them from FTP sites. Some vendors did not provide direct download links or required valid support contracts for downloading firmware images, such as Cisco. In this case, we downloaded them manually.

Our dataset includes 9,293 firmware images with 2,332 models from 10 vendors. After decryption and extraction, we obtained independent IoT file systems from 4,058 images, as shown in Table 2. We failed to extract some file systems and binaries due to unsolvable encryption, unparsable format, non-existent Web components, etc. We utilized shell commands to automatically search and extract Web server binaries from them, and we finally obtained 2,214 binaries. Only a few binaries contain FHFs that apply to FIRMKING.

Table 2: Number of successfully extracted firmware images and Web server binaries.

Vendor	#.bin	#.img	#.dmg	#Other	#Total	#Bins
asus	0	0	0	435	435	383
belkin	36	3	13	0	52	14
centurylinl	10	7	0	0	17	15
cisco	76	34	0	0	110	109
dlink	226	116	41	0	383	126
linksys	52	56	0	0	108	24
mercury	259	0	0	0	259	30
netgear	340	833	3	600	1776	949
tenda	243	0	0	0	243	129
tplink	657	18	0	0	675	435
Total	1899	1067	57	1035	4058	2214

We conducted our experiments on a 2-core Intel(R) Core(TM) i7-7500U 2.70GHz CPU machine with 8GB of RAM 2.5TB hard disk. The operating system is Ubuntu 18.04.1 LTS. Angr version is 9.0.4663, Python version is 3.6.5 and QEMU version is 2.11.1. We purchased 20 devices due to limited research budget.

5.2 Effectiveness

By performing analysis and fuzzing on these Web server binaries, FIRMKING finally identified 598 FHFs and 34 crashes,

Table 3: Results of static FHF analysis and fuzzing.

Vendor	Model	Version	Binary Program	Total Scan Time	#Identified FHFs	Average Time To Crash	#Crashes	#Exploitable Vulnerabilities
tenda	AC15	V15.03.05.18	httpd	3m5s	115	4m48s	12	4
tenda	AC23	V16.03.09.05	httpd	5m47s	100	10m0s	5	3
cisco	RV110W	V1.2.2.5	httpd	3m57s	68	17m0s	2	1
belkin	F9K1111	V1.04.10	webs	4m21s	80	13m20s	3	2
linksys	E1000	V2.1.03.005	httpd	5m58s	99	12m22s	4	4
dlink	DIR-816	V1.11	goahead	6m6s	136	8m30s	8	3
Total	-	-	-	29m4s	598	10m7s	34	17

including 17 (50%) 0-day vulnerabilities across 5 vendors, as shown in Table 3 and Figure 4. These vulnerabilities found by FIRMKING can be exploited by a POST request. We tried to report these vulnerabilities to vendors, but most of them did not reply. Finally, we received 7 CVEs [29]. For example, the details of one CVE are as follows:

- CVE-2021-1167: Stack-based buffer overflow vulnerability in the web-based management interface of Cisco Small Business RV110W. While setting up VPN, attackers can craft a request containing long *vpn_account* to execute arbitrary code as the root user or cause the device to reload, resulting in a denial of service condition.

We noticed that some FHFs contain time-delay operations, such as *system("reboot")*, *sleep(5)*. Our fuzzer regards these cases as crashes. This is the main cause of false positives in our results. To verify whether a crash is an exploitable vulnerability, we looked deep into these FHFs to examine the codes and craft POST requests to reproduce the vulnerability.

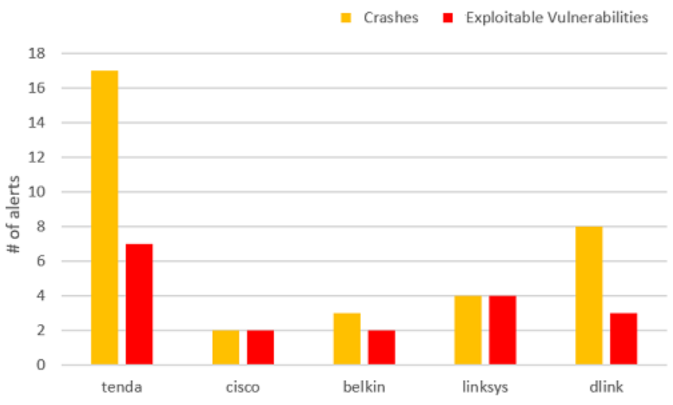


Figure 4: Number of crashes and exploitable vulnerabilities.

5.3 Efficiency

From Table 3 and Figure 6, we can see our static FHFs analysis module can scan most binaries in 6 minutes. Angr CFG generation takes two-thirds of the time. FHF identification and POST parameter extraction takes the remaining one-third of the time. In total, it identified 598 FHFs in 6 binaries taking only about 29 minutes. The fuzzer use scan results to

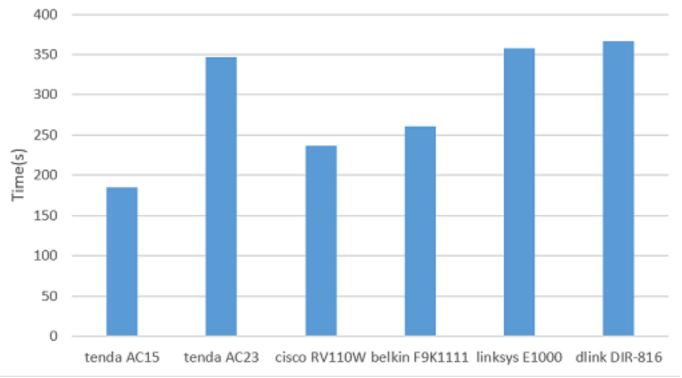


Figure 5: Static analysis time.

improve the efficiency of vulnerability discovery. For example, FIRMKING generated 115 session files after scanning Tenda AC15 binary. The fuzzer will use one session file at a time. For most binaries, we set the time interval between packets to 2 seconds, and each FHF interface is fuzzed 15 times. Therefore, it costs 30 seconds to test each FHF interface. Figure **??** shows average crash time per device. Facing the appropriate target binary, FIRMKING can detect a crash every 607s on average. This demonstrates the efficiency of our framework.

6 Discussion

Our framework has some limitations.

- Currently, our framework only supports the testing of

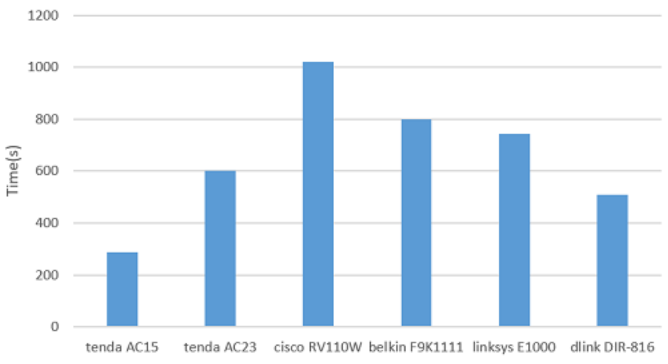


Figure 6: Average crash time.

binaries with FHF in them. We believe the main idea of our approach can be applied to other binaries.

- Our fuzzer requires the binaries to be properly executed, but only a few Web server binaries meet the requirement.
- Fuzzing requires the target device to reboot frequently in order to renew a clean state for the next test case. This usually takes several minutes.

In the future, We plan to continue vulnerability discovery and extend our study to more firmware images.

7 Related Work

7.1 IoT Static Analysis

Angr, an automated binary analysis framework, was proposed by Shoshitaishvili et al. [34]. Angr has limitations on MIPS and ARM binary analysis, so we extended our static analysis module based on it. Costin et al. [10] developed an automated framework to collect and unpack 32,000 firmware images into 1.7 million individual files. They found 38 previously unknown vulnerabilities, but they only performed simple static analysis. Shoshitaishvili et al. [33] proposed the Firmalice framework to detect authentication bypass vulnerabilities in IoT devices. Firmalice mainly performs static analysis, symbolic execution, and authentication bypass check on the target firmware image to detect backdoors. However, Firmalice was only evaluated on 3 firmware samples. It requires manual annotation and does not apply to large-scale analysis. Pewny et al. [27] presented a system to derive bug signatures for known bugs in binaries of different CPU architectures. Their approach is to compare binary code similarity via semantic hashes and identify previously-unknown bugs that are similar to known bugs. However, their approach cannot verify whether the code part is actually vulnerable and the algorithm degrades performance quite significantly. Redini et al. [30] presented Karonte framework to find vulnerabilities in IoT firmware. They focused on vulnerabilities in multi-binary interactions. Karonte does not utilize dynamic analysis and takes prohibitive resources to verify the large-scale scan results.

7.2 IoT Dynamic Analysis

Several studies aimed to improve emulation success rate, [6, 14, 22] tried to perform a comprehensive analysis of failure cases, then create an emulation environment as similar as possible to real device. FirmAE can successfully emulate 892 (79.36%) of 1,124 firmware images, including Web servers, boosting the emulation rate of FIRMADYNE by 487%. P^{2}IM introduced a new approach for microcontrollers (MCU) firmware fuzzing. By abstracting diverse peripherals and handling firmware I/O on the fly based on automatically generated models, P^{2}IM successfully executed 79% of the sample images (eg., drone, robot, PLC).

Some recent works focused on IoT fuzzing. [24] demonstrated the challenges in fuzzing IoT device and proposed possible solutions. Chen et al. [7] proposed IoTFuzzer, an automated framework to detect vulnerabilities in IoT devices without firmware. Their approach is to utilize the information carried by IoT mobile apps and app program logics. However, IoTFuzzer only supports App-based IoT devices. Corteggiani et al. Yun et al. [38] implemented QSYM, a fast concolic execution engine to support hybrid fuzzing. However, QSYM can only test programs that run on x86_64 architecture and does not support IoT platforms. [19, 40] introduced novel IoT fuzzing technologies such as augmented process emulation and optimized virtual execution. However, their frameworks only support binaries that can be properly emulated and take a considerable time to crash the target binary.

There are several recent works that rely on other dynamic techniques. Zaddach et al. [39] implemented the Avatar framework to support dynamic security analysis of embedded firmware images. Avatar executes binary instructions in the emulator, and forwards I/O accesses from the emulator to physical devices. However, due to the frequent interaction between software and hardware, the performance of Avatar is significantly lower than that of physical devices. Muench et al. An automated framework [11] achieved scalable and automated analysis of embedded Web interfaces. Their work also has limitations on emulating the firmware images and executing Web server binaries. They mainly utilized dynamic analysis techniques to discover XSS, CSRF, command injection, and file inclusion vulnerabilities, but our work mainly focuses on the buffer overflow vulnerabilities. [9] introduced Inception, a security testing framework of embedded firmware. Inception proposed novel techniques for symbolic execution but sometimes the testing is not scalable because of state explosion and interrupt inconsistencies.

8 Conclusion

Using static or dynamic analysis alone has limitations, so we combined static and dynamic techniques and presented FIRMKING framework to discover vulnerabilities in IoT firmware. FIRMKING utilizes static analysis to extract the names and parameters of HTTP POST forms inside the Web server binary. Next, for each form, the static analysis module uses information to generate an individual script. Finally, the fuzzer module starts to fuzz with all the scripts one after another and save the results. We evaluated FIRMKING on a real-world dataset of 9,293 firmware images. FIRMKING can discover 17 0-day vulnerabilities, including 7 CVEs. We demonstrated the efficiency and effectiveness of FIRMKING vulnerability discovery.

References

[1] AFL. American fuzzy lop. https://github.com/google/AFL.

[2] Angr. A powerful and user-friendly binary analysis platform. https://github.com/angr.

[3] Binwalk. A fast, easy to use tool for analyzing, reverse engineering, and extracting firmware images. https://github.com/ReFirmLabs/binwalk.

[4] Boa. Embedded web server. http://www.boa.org/.

[5] Boofuzz. A network protocol fuzzing framework, the successor to sulley. https://github.com/jtpereyda/boofuzz.

[6] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[7] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[8] L. Constantin. Hackers found 47 new vulnerabilities in 23 iot devices at defcon. https://www.csoonline.com/article/3119765/security/.

[9] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 309–326. USENIX Association, 2018.

[10] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 95–110. USENIX Association, 2014.

[11] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 437–448. ACM, 2016.

[12] CVE. Common vulnerabilities and exposures. https://cve.mitre.org/.

[13] Exploit Database. Exploits for penetration testers, researchers, and ethical hackers. https://www.exploit-db.com/.

[14] Bo Feng, Alejandro Mera, and Long Lu. P2IM: scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1237–1254. USENIX Association, 2020.

[15] Peach Fuzzer. An advanced and extensible fuzzing framework. https://www.peach.tech/.

[16] Ghidra. A full-featured, high-end software reverse engineering framework created and maintained by the national security agency research directorate. https://github.com/NationalSecurityAgency/ghidra.

[17] GNU. Gdb debugger. http://www.sourceware.org/gdb/.

[18] GoAhead. The world's most popular, tiny embedded web server. https://www.embedthis.com/goahead/.

[19] Z. Gui, H. Shu, F. Kang, and X. Xiong. Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution. volume 8, pages 29826–29841, 2020.

[20] Aki Helin. A test case generator for fuzzing. https://gitlab.com/akihe/radamsa.

[21] IDA. State-of-art disassembler and debugger hosted on windows, linux, mac os x. https://www.hex-rays.com/.

[22] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. FirmAE: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual Computer Security Applications Conference (ACSAC)*, Online, December 2020.

[23] Metasploit. The world's most used penetration testing framework. https://www.metasploit.com/.

[24] M. Muench, Jan Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[25] Mutiny. A network fuzzer that operates by replaying pcaps through a mutational fuzzer. https://github.com/Cisco-Talos/mutiny-fuzzer.

[26] 360 netlab. Is hajime botnet dead. https://blog.netlab.360.com/hajime-status-report-en/.

[27] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 709–724. IEEE Computer Society, 2015.

[28] QEMU. A generic and open source machine emulator and virtualizer. https://www.qemu.org/.

[29] CVEs received. CVE-2021-1167, CVE-2020-26737, CVE-2020-26738, CVE-2021-27713, CVE-2020-28181, CVE-2020-28182, CVE-2020-28725.

[30] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1544–1561. IEEE, 2020.

[31] RouterSploit. Exploitation framework for embedded devices. https://github.com/threat9/routersploit.

[32] Scrapy. A fast, powerful and extensible web crawling framework in python. https://scrapy.org/.

[33] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[34] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016.

[35] Squashfs-tools. Tools to create(mksquashfs) and extract(unsquashfs) squashfs filesystems. https://github.com/plougher/squashfs-tools.

[36] Sulley. A fuzzing framework consisting of multiple extensible components. https://github.com/OpenRCE/sulley.

[37] Wfuzz. A famous web fuzzer. https://github.com/xmendez/wfuzz.

[38] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 745–761. USENIX Association, 2018.

[39] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[40] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1099–1114. USENIX Association, 2019.