SRFuzzer: An Automatic Fuzzing Framework for Physical SOHO Router Devices to Discover Multi-Type Vulnerabilities

Yu Zhang[*†‡]
Institute of Information Engineering, CAS, China
zhangyu2@iie.ac.cn

Wei Huo[*†‡§]
Institute of Information Engineering, CAS, China
huowei@iie.ac.cn

Kunpeng Jian[*†‡]
Institute of Information Engineering, CAS, China
jiankunpeng@iie.ac.cn

Ji Shi[*†‡]
Institute of Information Engineering, CAS, China

Haoliang Lu[*†‡]
Institute of Information Engineering, CAS, China

Longquan Liu[*†‡]
Institute of Information Engineering, CAS, China

Chen Wang[*†‡]

Dandan Sun[*†‡]
Institute of Information Engineering, CAS, China

Chao Zhang[¶]
Institute for Network Science and Cyberspace, Tsinghua University
Beijing 100084, China
chaoz@tsinghua.edu.cn

Baoxu Liu[*†‡]
Institute of Information Engineering, CAS, China
liubaoxu@iie.ac.cn

ABSTRACT

SOHO (small office/home office) routers provide services for end devices to connect to the Internet, playing an important role in the cyberspace. Unfortunately, security vulnerabilities pervasively exist in these routers, especially in the web server modules, greatly endangering end users. To discover these vulnerabilities, fuzzing web server modules of SOHO routers is the most popular solution. However, its effectiveness is limited, due to the lack of input specification, lack of routers' internal running states, and lack of testing environment recovery mechanisms. Moreover, fuzzing in general only reports memory corruption vulnerabilities, and fails to discover other vulnerabilities, e.g., web-based vulnerabilities.

In this paper, we propose a solution SRFuzzer to address these issues. It is a fully automated fuzzing framework for testing physical SOHO devices. It continuously and effectively generates test cases by leveraging two input semantic models, i.e., KEY-VALUE data model and CONF-READ communication model, and automatically recovers testing environment with power management. It also coordinates diversified mutation rules with multiple monitoring mechanisms to trigger multi-type vulnerabilities. To the best of our knowledge, it is the first whole-process fully automated fuzzing framework for SOHO routers. We ran SRFuzzer on 10 popular routers across five vendors. In total, it discovered 208 unique exceptional behaviors, 97 of which have been confirmed as 0-day vulnerabilities. The experimental results show that SRFuzzer outperforms state-of-the-art solutions in terms of types and number of vulnerabilities found.

[*]Also with School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China.
[†]Also with Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing, China.
[‡]Also with Beijing Key Laboratory of Network Security and Protection Technology.
[§]Corresponding author.
[¶]Also with Beijing National Research Center for Information Science and Technology.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Software security engineering**.

KEYWORDS

Fuzzing, IoT, automatic vulnerability/bug detection, data inconsistency

ACM Reference Format:
Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. 2019. SRFuzzer: An Automatic Fuzzing Framework for Physical SOHO Router Devices to Discover Multi-Type Vulnerabilities. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359789.3359826

1 INTRODUCTION

More and more end devices, such as laptops, pads, smartphones, smart-home devices, and wearable devices, are used in social life. They are usually connected to the Internet through small office and home office (SOHO) routers. As a result, SOHO routers are in a prominent position that isolates end users from external Internet [15] and processes end users' all traffic. The security of SOHO routers is much more critical than ever.

Unfortunately, the security vulnerabilities pervasively exist in SOHO routers [4], [29]. According to a recent report [6], 83% of popular routers contain vulnerable code. These vulnerabilities are one of the essential exploiting targets by adversaries. In 2018, Cisco Talos found malware *VPNFilter* targeted to Linksys, MikroTik, NETGEAR and TP-Link networking equipments, which are all SOHO routers, and infected at least 500,000 devices in at least 54 countries [5]. Besides, the leading exploit acquisition platform Zerodium [3] has added the requirement for routers in 2018. Therefore, discovering vulnerabilities in SOHO routers becomes significantly important.

A typical architecture of the SOHO router is shown on the right side of Figure 1. As network equipment, the SOHO router provides networking service, e.g., routing, for the end devices connected to it. More importantly, it also leverages web services for administration and configuration, due to its lack of user interface, e.g., keyboard, video, mouse. These web services are provided by some widely used protocols, e.g., HyperText Transfer Protocol (HTTP). We call these protocols as management protocols in this paper.

A typical management protocol is implemented by embedding a web server (backend) into the original device. Usually, web servers in different routers are customized by device vendors and are more vulnerable. Recent works [12], [15], [11] shown that most of SOHO router vulnerabilities identified are be associated with web services, such as command injection vulnerabilities in PHP server-side scripts and memory corruption vulnerabilities in processing mobile application web requests. Therefore, this paper also focus on discovering vulnerabilities of the web server of the SOHO router.

Fuzz testing (i.e., fuzzing) is considered to be a powerful technique to discover vulnerabilities. However, there is little research on fuzzing web server of SOHO router (FWSR), except IoTFuzzer [11], an app-based fuzzing framework. With the help of program logic of mobile APP that could control the device, the approach produces meaningful test cases and triggers device bugs. Although it has partially solved the problems of FWSR, in general, FWSR remains challenging.

Challenge 1: Fuzzing Dedicated System

As an embedded device, the function of the SOHO router is dedicated and cannot be extended easily. Therefore, the internal running state cannot be acquired directly during fuzzing, and its normal running is hard to restore when the device is made to be stuck during testing. These make it difficult for FWSR continuously and effectively. Although fuzzing based on emulation is a promising way to obtain internal executing information that can guide fuzzing, it is limited in emulating various routers in full-system mode. (See Section 6)

Challenge 2: Analyzing Input Semantics

The input of web server is accordant with standard protocols, e.g., HyperText Transfer Protocol (HTTP), however, their internal data that encode the exchanging information are in various format across routers. Without investigating the format of internal data and information exchanging process, FWSR is not effective in terms of code coverage.

Challenge 3: Discovering Multi-type Vulnerabilities

As aforementioned, the web server is almost customized by vendors from the frontend to the backend. Therefore not only web vulnerabilities, such as command injection and cross-site scripting,

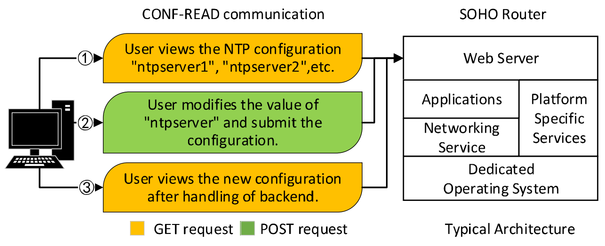

CONF-READ communication
SOHO Router
① User views the NTP configuration "ntpserver1", "ntpserver2",etc.
② User modifies the value of "ntpserver" and submit the configuration.
③ User views the new configuration after handling of backend.
Web Server
Applications
Networking Service
Platform Specific Services
Dedicated Operating System
🟨 GET request 🟩 POST request
Typical Architecture

Figure 1: Typical architecture of SOHO router and the workflow of the NTP configuration

but also memory corruption, should be discovered by FWSR. Unfortunately, different types of vulnerability need to be triggered by different payloads, as well as monitored by different methods. Such requirements make the design of FWSR more difficult.

Our Approach.

In this paper, we propose an automatic fuzzing framework for FWSR based on physical devices to address the above challenges. It drives the fuzzing process continuously by automatic seed generation and automatic power control. To model the input semantics, it leverages two models to constrain test cases, i.e., the KEY-VALUE data model (K-V model for short) to describe the format of internal data of requests, and the CONF-READ communication model (C-R model for short) to describe the temporal sequence of requests. Moreover, our framework coordinates different mutation rules with multiple monitoring mechanisms to effectively trigger four types of vulnerabilities, i.e., vulnerabilities of *memory corruption*, *command injection*, *cross-site scripting (XSS)* and *information disclosure*. To the best of our knowledge, it is the first whole-process fully-automatic framework for FWSR. Furthermore, by a little human effort on collecting web requests and monitoring running state, it could outperform fully-automatic fuzzing.

Firstly, to generate seeds automatically, our framework uses a crawler-based method to collect web request header and request content [20] from the running device automatically. Meanwhile, to restore the devices from the "zombie" state, we add a power control module based on Wi-Fi plug and force the device to reboot when it got stuck. Besides, our framework tests the physical device directly. The design decision is general and practical, as it does not require to emulate firmware that is infeasible for many routers.

Secondly, our framework can trigger deeper bugs without firmware reverse engineering or instrumentation (required by feedback-guided fuzzing techniques [43]). Instead, the framework mutates the real input with the help of precisely modeling the data format and request sequence, using K-V and C-R model respectively. By K-V model, we are able to distinguish data types for values and assign different mutation rules. By C-R model, we could construct multi-phase communications to trigger bugs under a certain running state.

Thirdly, for different types of vulnerability, we summarize them as data inconsistency vulnerabilities. In order to discover them, we couple mutation with monitoring. More specifically, we have designed six mutation rules and three monitoring mechanisms. When fuzzing a given type of vulnerability, we choose the appropriate mutation rules and monitoring mechanisms.

We have implemented a prototype of our solution SRFuzzer and deployed it in a real-world environment. To evaluate its effectiveness and generality, we ran SRFuzzer on 10 popular routers across 5 vendors. Benefit from the comprehensive new monitoring method, we have got 208 unique exceptional behaviors. Almost half of the exceptional behaviors are confirmed as 101 unique issues that belong to aforementioned four vulnerability types. After responsible disclosing vulnerabilities to the corresponding vendors, we obtained total 97 assigned IDs, i.e., 43 CVE[1] IDs, 52 PSV[2] IDs and 2 CNVD[3] IDs[4].

Contributions. The contributions of the paper are as follows:

- We proposed a fully-automatic fuzzing framework to discover various vulnerabilities for FWSR. We motivated the fuzzing continuously by generating seeds and restoring device automatically.
- We designed KEY-VALUE data model and CONF-READ communication model to reveal the root cause and guide the mutation, and we coordinated six mutation rules with three monitoring mechanisms to improve the effectiveness of vulnerability discovering.
- We implemented a prototype SRFuzzer and evaluated it over 10 real-world SOHO routers. In total, it discovered 101 confirmed issues out of 208 unique exceptional behaviors, 97 of which have been confirmed as 0-day vulnerabilities by vendors. The results showed that SRFuzzer outperformed state-of-the-art solutions in ability of vulnerability discovery.

This paper is organized as follows: Section 2 presents our motivations and insights to overcome the challenges. Section 3 overviews SRFuzzer and describes the detailed design. Section 4 introduces the experiments and evaluation. The shortcomings of our framework and related work are discussed in Section 5 and 6. At last, we concludes in Section 7.

2 MOTIVATION

The design goal of SRFuzzer is to build an automatic fuzzing framework for FWSR and to find as many vulnerabilities as possible. It is straightforward to think building the framework based on firmware emulation. However, such a solution is extremely hard in general because of the diversity in various dedicated component built-in routers. We argue that it would be a general framework to automatically fuzz the physical SOHO router directly, while it is still challenging.

In this section, we first introduce the vulnerabilities we focus on as well as the assumptions of our environment. Then we summarize design challenges in fuzzing real-world routers automatically as well as in depth. We analyze the root cause of the router vulnerabilities based on C-R and K-V models with a motivating example.

[1]Common Vulnerabilities and Exposures.
[2]Vulnerability identifications of assigned by the vendor NETGEAR.
[3]China Nation Vulnerability Database.
[4]All CVE IDs, PSV IDs, and CNVD IDs are listed in appendix.

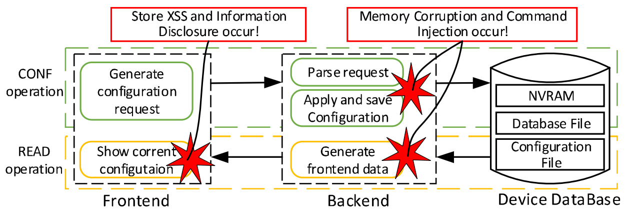

Store XSS and Information Disclosure occur!
Memory Corruption and Command Injection occur!
CONF operation
READ operation
Generate configuration request
Show corrent configutaion
Parse request
Apply and save Configuration
Generate frontend data
NVRAM
Database File
Configuration File
Frontend
Backend
Device DataBase

Figure 2: Typical workflow in term of CONF-READ model

2.1 Scope and Assumptions

Besides networking services, web services are also embodied into SOHO routers for the sake of administration and configuration. They are usually provided by standard protocols, including HyperText Transfer Protocol (HTTP), Simple Object Access Protocol (SOAP), Universal Plug and Play (UPnP) and more. These protocols are implemented by vendors and are used to set Wi-Fi password, to find end devices, etc. We call these protocols as management protocols and focus on HTTP protocol in this paper. Moreover, we suppose SOHO routers use built-in WAN/LAN web services, other than mobile-to-web services [11], as their default settings.

A typical management protocol implementation of a SOHO router consists of three major parts, i.e., a frontend, a backend, and a database. The frontend shows current settings of the router and guides the user for configuration. The backend parses the requests received from the frontend and configures related services. The database stores the current configuration in several places, such as NVRAM, databases and configuration files. Figure 2 describes the workflow of these parts in terms of CONF and READ operations.

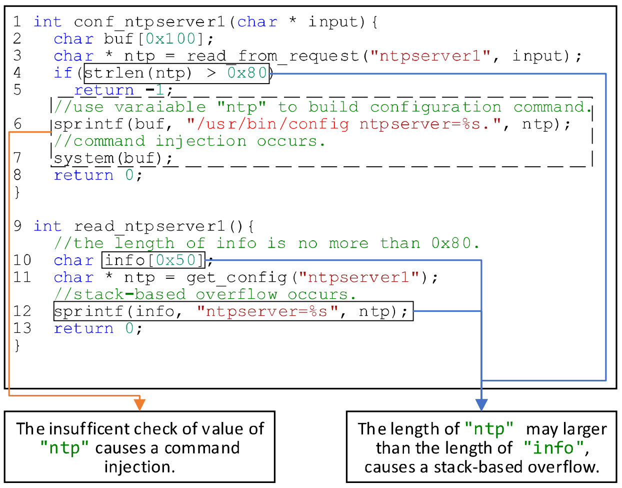
Figure 2 also shows four typical types of vulnerabilities in the workflow. Vulnerabilities of memory corruption and command injection often occur in the backend, while XSS often occurs in the frontend. Information disclosure vulnerability may occur in both frontend and backend. Different types of vulnerability can be triggered by various causes, e.g., memory corruption is usually triggered by improper user's input handling. Information disclosure in this paper refers to the accesses of privileged data without appropriate permission. The other two vulnerabilities are common web vulnerabilities, and most XSS issues are stored XSS in routers.

2.2 Automating the Whole Fuzzing Process

The most important steps for automating the FWSR is input generation and running restoration.

As aforementioned, the web server is implemented differently across device vendors, which means the format of the request to each server is also different. So we choose mutation-based fuzzing since it is hard to adopt general generation-based fuzzing. In order to automatically collect seeds, which is essential for mutation-based fuzzing, we design a crawler to get as many requests as possible. Meanwhile, interactions with web services should be carefully dealt with to keep the server running. These requests are mutated by specific rules, which will be discussed in Section 2.3, and are fed to a physical SOHO router.

Once a specific test case makes the server failure, SRFuzzer should observe the occurrence and drive the fuzzing into a next test. Normally, it is relatively trivial to restart or resume a crashed

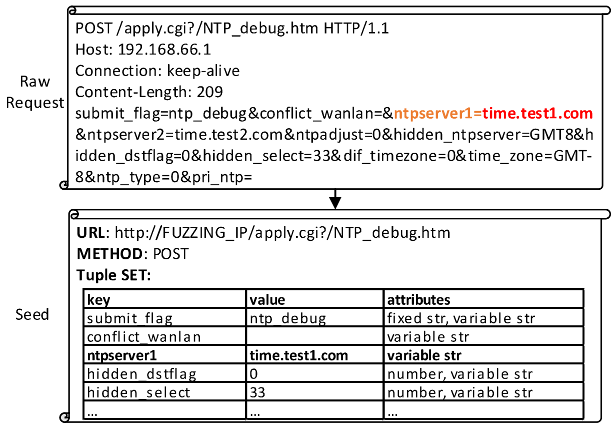


```
1 int conf_ntpserver1(char * input){
2   char buf[0x100];
3   char * ntp = read_from_request("ntpserver1", input);
4   if(strlen(ntp) > 0x80)
5     return -1;
    //use varaiable "ntp" to build configuration command.
6   sprintf(buf, "/usr/bin/config ntpserver=%s.", ntp);
    //command injection occurs.
7   system(buf);
8   return 0;
}

9 int read_ntpserver1(){
    //the length of info is no more than 0x80.
10  char info[0x50];
11  char * ntp = get_config("ntpserver1");
    //stack-based overflow occurs.
12  sprintf(info, "ntpserver=%s", ntp);
13  return 0;
}
```

The insuffcent check of value of "ntp" causes a command injection.
The length of "ntp" may larger than the length of "info", causes a stack-based overflow.

Figure 3: Code Snippet of web server for NTP configuration



```
POST /apply.cgi?/NTP_debug.htm HTTP/1.1
Host: 192.168.66.1
Connection: keep-alive
Content-Length: 209
submit_flag=ntp_debug&conflict_wanlan=&ntpserver1=time.test1.com
&ntpserver2=time.test2.com&ntpadjust=0&hidden_ntpserver=GMT8&h
idden_dstflag=0&hidden_select=33&dif_timezone=0&time_zone=GMT-
8&ntp_type=0&pri_ntp=
```

URL: http://FUZZING_IP/apply.cgi?/NTP_debug.htm
METHOD: POST
Tuple SET:

key	**value**	**attributes**
submit_flag	ntp_debug	fixed str, variable str
conflict_wanlan		variable str
ntpserver1	**time.test1.com**	**variable str**
hidden_dstflag	0	number, variable str
hidden_select	33	number, variable str
...	...	...

Figure 4: Parsing raw request to seeds

process when fuzzing software on a standard computer system. On the contrary, it is not easy to reset or restart a stuck SOHO router without human interference. The router falls into "zombie" state during the fuzzing process mainly for two reasons. Firstly, there might be no response sent back when the process of the web server is crashed by a malformed request. Secondly, self-protection mechanism of some devices will forbid the access to the web server if specific exceptional conditions are met.

In order to restart the web server automatically, we leverage power control equipment to manage the SOHO router. Specifically, we use the smart plug which is widely used nowadays. Each smart plug powers a router and is controlled by SRFuzzer. It is connected to SRFuzzer through Wi-Fi. Once SRFuzzer has monitored a stuck router, it sends the plug restarting command by plug's internal APIs, and consequently restarts the device. Then the web server could be recovered from the crashed state.

2.3 Fuzzing in Depth

Coverage-guided fuzzing techniques, such as AFL [43], have largely enhanced the traditional mutation-based fuzzing. Due to the requirement of instrumentation on code, they are infeasible for real-world dedicated devices. Moreover, the root cause of the vulnerabilities of routers is data inconsistency during requests processing, which makes the effectiveness of FWSR hardly be improved by those techniques. We design a KEY-VALUE data model and a CONF-READ communication model to describe the semantics of requests, i.e., the format of internal data in requests and the relations between different requests, and to guide the design of mutation rules. A motivating example is presented to illustrate these models in a more concrete manner.

Motivating Example. To illustrate the design challenges of mutation rules, we present a configuration process for network time protocol (NTP) by HTTP requests. The left half of Figure 1 shows the general process, and Figure 3 shows the backend handling procedures related to the process.

There are usually three steps to configure NTP option of a router: (i) The administrator of router sends an HTTP GET request to access the configuration web page through a URL, e.g., "http://192.168.0.1/apply.cgi/NTP_debug.htm". He/she also gets the current domain name of `ntpserver1`, which is used by `read_ntpserver1()` shown in Line 9-13 of Figure 3. We call this process READ operation. (ii) The administrator modifies the domain name by sending an HTTP POST request. The new domain name of `ntpserver1` is submitted and stored into the database through the backend procedure `conf_ntpserver1()`. We call this process CONF operation. (iii) he/she checks whether the newly submitted domain name is configured correctly by another READ operation.

KEY-VALUE Data Model. Supposing configuration process is now in CONF operation and the raw request is shown on top half of Figure 4. The backend deals with the request as shown in Line 3, 6, 7 of Figure 3 and a *command injection* issue occurs. Function `conf_ntpserver1()` would match the string "ntpserver1" from the request then get a domain name. The domain name is used as an argument of `/usr/bin/config` command which would be executed by `system()` in turn. However, there would be a command injection if the request contains a string like "ntpserver1=;reboot;", and the shell command "reboot;" would be invoked after executing `/usr/bin/config` command.

The input triggering this vulnerability requires two conditions, i.e., the string "ntpserver1=" keeps unchanged and the format of `config` command keeps valid. Therefore, performing the randomized mutation on raw requests to generate test cases is probably meaningless in this case.

In order to generate meaningful test cases in both READ and CONF operation, we design the KEY-VALUE data model to describe the constraints on a single request. Each request should be composed of key-value pairs (k-v pairs for short), of which the key stands for the variable name like "ntpserver1" and keeps unchanged. The value could be assigned to the variable, so it should be consistent with the type requirement of the key. For example, if a key requires "domain name", the value should be precisely a domain name. Although raw requests might be in other data formats, e.g. json and xml, the model still works well.

We describe the consistency between key and value by labeling attributes on the k-v pair according to the value. We summarize three types of attributes: *number*, *fixed string* and *variable string*. These attributes guide the mutation of the value of a k-v pair. As a result, after parsing k-v pairs from the raw requests and labeling them, the seeds can be generated more effectively.

CONF-READ Communication Model. Another vulnerability shown in Figure 3 is a *stack-based overflow*. This overflow exists in Line 12 of function `read_ntpserver1()`, which does not check the length of `ntp` before using it. To notice that, the value of `ntp` is set in Line 3 of function `conf_ntpserver1()`, and its length constraint is inconsistent with the length of variable `info`. Therefore, the vulnerability could be triggered when reading a malformed domain name from the database, which is set in Line 7, with length between 0x50 and 0x80.

In such a situation, a single request is not enough to trigger the vulnerability, so we design the CONF-READ communication model to form multiple-requests test input. This model consists of two related operations, i.e., CONF operation and READ operation. In CONF operation, the requests of setting or modifying the configurations of devices are constructed, while in READ operation, the requests of getting the corresponding configurations are also constructed.

Lessons learned. In light of the analysis of the configuration workflow in Figure 2 and the motivating example, we conclude the root cause of the vulnerability as data inconsistency. The value inconsistent with key `ntpserver1` causes the *command injection* which is triggered at CONF operation. Meanwhile, the length inconsistent between `conf_ntpserver1()` and `read_ntpserver1()` causes the *memory corruption* which is triggered at READ operation. By coupling the K-V model with the C-R model, SRFuzzer could reveal data inconsistencies in both k-v paired data and temporally related requests. Therefore, it is capable of detecting deep bugs in the web server of the SOHO router.

2.4 Discovering Multi-type Vulnerabilities

Discovering and triggering multi-type vulnerabilities at either CONF or READ operation, is necessary yet difficult in this fuzzing framework. As aforementioned in section 2.3, we can design various of mutation rules according to the K-V model. Especially for the value with "variable string" attribute, we design different mutation rules to trigger exceptional behaviors of overflow, NULL-pointer dereference, command injection and stored XSS respectively. In addition, we establish different types of communications to trigger the vulnerabilities which occur in either CONF or READ operation.

Once a vulnerability is triggered in a device, there will be some exceptional behaviors such as a backend crash, an abnormal response or executing an unexpected command. To monitor the vulnerability in the backend, it is insufficient to use liveness check [32], a most common method for monitoring dedicated devices. The method monitors the exceptional behaviors by only checking the connection state. In fact, the normal connection state (such as status code 200 in HTTP) does not always indicate the normal behavior, e.g., triggering an injected command execution can also return the correct connection state.

We design two general monitoring mechanisms to catch the exceptional behaviors, i.e., a *response-based monitor* and a *proxy-based monitor*. The response-base monitor checks not only connection states but also response contents that might include extra information. The proxy-based monitor receives the network accesses from target router. Inspired by the conclusion of [11], we also design an optional *signal-based monitor*. It could catch more memory corruptions (i.e., silent memory corruptions [32]) at the cost of implanting a compiled executable into the device. By coupling mutation rules with monitoring mechanisms, we can detect multi-type vulnerabilities.

3 DETAILED DESIGN

In this section, we present the detailed design of SRFuzzer. As shown in Figure 5, SRFuzzer consists of five modules to work coordinately. Once connecting to the router, it collects the valid seeds by the *seed generator* module. Then it feeds the seeds into the *mutator* module to generate mutated requests based on various mutation rules. Finally, SRFuzzer triggers and monitors the exceptional behaviors by the collaboration of the *mutator*, the *monitor* and the *power control* module.

(1) **Seed Generator.** To generate the initial test cases, i.e., seeds, for mutation, the Seed Generator collects raw requests by the Request Collector submodule and parses them into k-v pairs. Then it labels all k-v pairs with attributes, which could guide mutation later. The Request Collector submodule is composed of two parts, a general crawler as the default setting to collect requests automatically, and an optional passive crawler to collect more requests by interacting with users.
(2) **Mutator.** The Mutator generates mutated requests for the SOHO router through the cooperation of two submodules, i.e., the Mutation Selector and the Pattern Selector. Guided by the K-V model, mutation rules are selected and applied to the value of each k-v pair, according to the attributes of the k-v pair and the vulnerability type being discovered. In the Pattern Selector submodule, the types and the sequence of requests are decided based on the C-R model. The Pattern Selector could generate one request, with the type of either CONF operation or READ operation. While, it could also generate a sequence of requests, e.g., a READ operation after a CONF operation.
(3) **Monitor.** In order to collaborate with the Mutator module tightly and to monitor more exceptional behaviors, the Monitor module consists of two common monitors, a response-based monitor and a proxy-based monitor. The response-based one could usually monitor three types of vulnerability, i.e., memory corruption, XSS and information disclosure. To notice that, for information disclosure vulnerability, it monitors the response of a target URL without enough access permission. The proxy-based monitor is used for command injection and XSS vulnerabilities. In addition, an optional signal-based monitor is also provided to catch deeper memory corruptions. It is developed based on ptrace syscall and could monitor the signal such as SIGSEGV and SIGABRT.

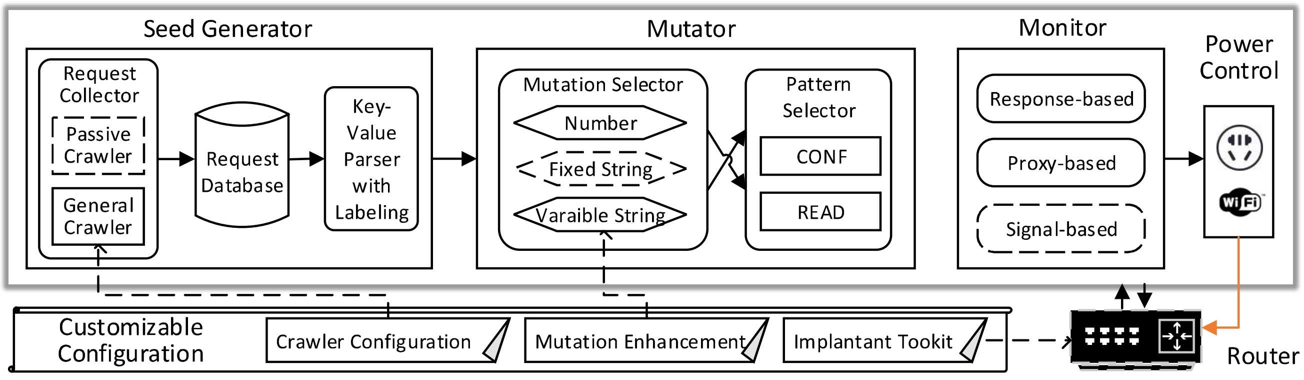


Figure 5: Overview of SRFuzzer

(4) **Power Control.** For the purpose of fuzzing physical router continuously, a Power Control module is introduced. It is supported by a smart plug to control the power of the device. This module is controlled by the Monitor module. If the backend service is stuck into a "zombie" state, i.e., no response, a control command would be sent to the plug and the device would be restarted.
(5) **Configuration.** In order to improve the fuzzing efficiency, we also provide custom configurations for individual modules. All these configurations are optional. We configure the IP address of the default portal for the general crawler. We also provide mutation enhancement techniques for values with variable string attribute to trigger more exceptional behaviors. In order to ease the deployment of the signal-based monitor, we develop an implanting toolkit for the routers. By this toolkit, we can place the signal-based monitor into the device automatically.

3.1 Seed Generation

This module aims at generating the seeds for the following fuzzing. We use the Request Generator to collect the raw requests then store them into the Request Database. Finally, we leverage the Key-Value parser to parse the requests into k-v pairs with attribute labeling.

As we mentioned in Section 2.3, a typical CONF operation is the second step of a web communication, as shown in Figure 4. The Request Collector submodule aims at repeating this step and captures raw requests by two crawlers. By default, SRFuzzer uses a general crawler to collect the requests automatically. However, the collection effects can be improved by the passive crawler.

The general crawler uses the default URL as input, then it fills the web page automatically by parsing the input elements of the web page. In the meanwhile, it identifies all URLs of the page and then fills them recursively like a traditional crawler. It also stores the requests into the request database. In case of interaction during crawling, such as providing specific information, SRFuzzer randomly select data from a predefined database to continue crawling.

The passive crawler is a semi-automatic toolkit which opens a web page and waits for the user input to fill the web page. After submitting the configuration, the passive crawler stores the requests into the request database and prepares for the next web page. Such a crawler is usually used to generate seeds from web pages with user input, such as the login page.

In addition, to collect the raw requests as many as possible and to facilitate the later attribute labeling procedure, both crawlers fill the same web page ten times.

As aforementioned, the K-V model describes the management protocol in a finegrained manner. However, we can dig more information from the k-v pairs for deeper fuzzing. The KEY-VALUE parser analyzes the raw requests and split them into k-v pairs. Meanwhile, it labels the attributes of all k-v pairs according to their values.

There are two features in the value handling procedure of the backend. Firstly, the value is usually handled as a variable string, and the backend parses the crucial information to build related configuration. Secondly, there are always some validity checks, such as to judge whether a value is a number or a fixed string. As a result, if a fixed string is mutated, the check cannot be passed and the code protected by the check becomes unreachable. Therefore, we label a k-v pair with three type attributes, i.e., number, fixed string, and variable string. The attributes of a k-v pair determine the mutating rules applied to the k-v pair. By default, all k-v pairs are labeled with an attribute "variable string". Algorithm 1 shows the attribute labeling process for k-v pairs.

In summary, Seed Generator converts each unique raw request to several seeds. Each seed contains the URL and the set of data tuples, each of which contains a key, a value and attributes. Figure 4 shows the converting process from a raw request to the seeds.

3.2 Mutation

From section 2.3, we know the most crucial factor to trigger the vulnerability is the mutated value. There are two guidances to build mutation rules. Firstly, the root cause of the vulnerabilities is data inconsistency, especially for the variable string. So how to mutate the value of each k-v tuple is more important. Secondly, there are obvious differences between different types of vulnerability, so mutation rules should trigger exceptional behaviors according to the type of vulnerability.

Algorithm 2 shows the mutation algorithm for each seed. We separate the mutation of tuples and URL because it is inefficient to mutate them together. To mutate seeds, we select random number

Algorithm 1 Labeling Attribute Algorithm.

Input: A set of k-v pairs P, each pair p contains $p.key$ and $p.value$;
Output: A set of tuple T, each tuple t contains $t.key$, $t.value$ and $t.attributes$

```
1:  $T \leftarrow \emptyset$
2:  for all $p$ in $P$ do
3:      $t \leftarrow p$
4:      if $t.value$ is a number then
5:          add_attribute($t.attributes$, “number”)
6:      end if
7:      if $t.value$ does not change in various raw requests then
8:          add_attribute($t.attributes$, “fixed string”)
9:      end if
10:     add_attribute($t.attributes$, “variable string”)
11:     $T$.append($t$)
12: end for
        return $T$
```

Algorithm 2 Mutation Algorithm for each Seed.

Input: A single seed S which contains the URL and the set of tuple, T_n; Mutation option *option*;
Output: A mutated seed S_m;
Require: The set of mutation rules for variable strings, R_k;

```
1:  $S_m \leftarrow S$
2:  if option is “TUPLE” then
3:      $T_r \leftarrow$ select_from_set($T_n$, random($n$)).
4:      for all $t$ in $T_r$ do
5:          $attr \leftarrow$ select_from_set($t.attributes$, 1)
6:          if $attr$ is “number” then
7:              $t.value \leftarrow$ mutate_number($t.value$).
8:          end if
9:          if $attr$ is “variable string” then
10:             $rule \leftarrow$ select_from_set($R_k$, 1)
11:             $t.value \leftarrow$ mutate($t.value$, $rule$).
12:         end if          ▷ Do nothing for “fixed string” attribute
13:         $S_m.T_n[t.key] \leftarrow t$
14:     end for
15: else
16:     $S_m.URL \leftarrow$ mutate_URL($URL$)
17: end if
        return $S_m$.
```

of tuples from each seed. According to the attributes of each tuple, we mutate its value with a related mutation rule. The mutation rules for number and fixed string attributes are simple, while for variable string, there are four mutation rules to trigger different types of exception behavior:

(1) **For Overflow:** To trigger the overflow vulnerability, this framework usually duplicates the original value several times. If the key with the empty value, it assigns the key with a random number of payloads selected from a predefined database.

Table 1: Example of Mutation Rules for Variable Strings

Section	Mutation Rule	Example of Mutated Value
ntpserver1	Overflow	time.test1.comtime.test1.com... (repeat 20 times)
	NULL-pointer dereference	(empty value)
	Command Injection	time.test1.com";wget http://PROXY_SERVER/ntpserver1;
	XSS	time.test1.com";<script> alert('xss_ntpserver1')</script>
URL	http://DEVICE_IP/apply.cgi?/NTP_debug.htm/../../etc/passwd	

(2) **For NULL-pointer dereference:** For the key with the non-empty value, this framework provides the empty value to trigger the potential NULL-pointer dereference vulnerability.

(3) **For Command Injection:** To cooperate with the proxy-based monitor, SRFuzzer constructs the value with the malformed payloads which are based on built-in tools such as `ping` or `wget`. If a command injection is triggered, these payloads will connect to the outside proxy-based monitor which contains a proxy server. If this monitor catches the requests sent from the router, it collects the detailed information about the exceptional behavior. It can also help to locate the vulnerability efficiently for the follow-up analysis.

(4) **For Stored XSS:** There are two rules to construct the payloads for XSS. The most common payload contains the malformed JavaScript code to eject a message box. If the response-based monitor catches the message box that contains the string prefix with “xss_”, it records the exception and locates the vulnerability tuple. The other payload constructed for the proxy-based monitor is similar to the rule for command injection, e.g., “<script>(new Image()).src= "http://PROXY_SERVER/MUTATION_KEY_INFO/" </script>”.

For all mutation rules of variable string, we also use special characters, e.g., “;”, “$”, whitespace characters, and all kinds of quotation marks, to trigger more exceptional behaviors. These special characters can help to bypass the validity checks in the backend. For example, the `inet_addr(const char *cp)` function only extracts the part of a string before the first whitespace to check whether it is a valid IP address. Therefore, a value with the form “IP+Whitespace+Additional String” is wrongly considered as a valid IP address. For the 3rd and 4th mutation rules, we encode the key into the mutation value to assist in locating the exactly k-v pair.

For the URL mutation, this framework generates URLs that contain the special paths or sensitive file paths such as “/etc/passwd” or “/etc/shadow” beyond the permission when fuzzing. If a malformed URL can be accessed with a normal response, the response-based monitor will report it as an exceptional behavior.

Table 1 shows how these rules are applied to the motivating example illustrated in Figure 4. The original values are “http://DEVICE_IP/apply.cgi?/NTP_debug.htm” and “time.test1.com”.

3.3 Triggering and Monitoring the Exceptional Behavior

The common fuzzing method of the device is to monitor the response status after sending a mutation packet. There is always one communication in this procedure. However, it is only useful for vulnerabilities that occur in CONF operation of the C-R model. We also

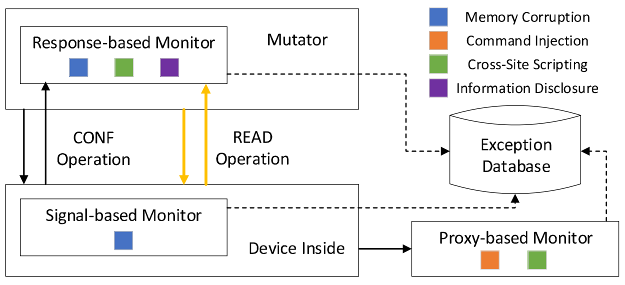


Figure 6: Workflow of Triggering and Monitoring

need to trigger the vulnerabilities that occur in READ operation. To overcome this limitation, we trigger a READ operation after a CONF operation immediately. We also separate the request phases and monitoring methods. Moreover, we rollback CONF operation with the original value after each communication cycle.

As we mentioned in Section 2.4, because "liveness check" is limited to the valid information from the response, it can hardly monitor various types of vulnerability nor catch deeper exceptional behaviors such as memory corruptions that occur in the subprocess. To make up for its limitation, we expand the liveness check into a response-based monitor. Moreover, we design the proxy-based monitor and the signal-based monitor to improve the monitoring performance in depth. Figure 6 shows three typical monitoring mechanisms with their monitoring scopes.

(1) **Response-based monitor:** Besides *liveness check*, analyzing the response content of the communication can monitor XSS issues that triggered in the frontend. We craft the payload for each key to mutate during CONF operation and analyze the response content during the separated READ operation. Then we can locate the crucial keys that trigger the exceptional behaviors easily. Moreover, information disclosure can also be monitored by judging the response status.
(2) **Proxy-based monitor:** For command injection and XSS issues, the Mutator module crafts malformed payloads to request the server of router outside. We build a monitoring server in the local network where the router can approach. So we can detect command injection and XSS issues when the monitoring server is accessed by the router. By cooperating with the crafted payload, the proxy-based monitor can efficiently locate the vulnerable URL and the crucial k-v pairs that trigger the exceptional behaviors.
(3) **Signal-based monitor:** For the memory corruptions, the most common signals are SIGSEGV and SIGABRT. These two signals always occur when a process even though a subprocess crashes. So monitoring these signals can catch more true positive exceptional behaviors with less false positives. For Linux-based routers, we can develop the monitor based on `ptrace` syscall.
Although the signal-based monitor can monitor the memory corruption more widely and accurately than the response-based one, it requires permission to implant a binary into the router, which is not always the case. So only the devices that satisfy the permission requirement can take advantage of this monitor. There are three ways to acquire this permission, i.e., through debug shell, exploiting known vulnerability such as command injection, and connecting the built-in serial port. We also develop an implant toolkit to put an executable into the device automatically.

Because three monitors work individually, we have to synchronize requests and exceptional behaviors for them. We record a timestamp for each request and a timestamp for each exceptional behavior, to make sure the request related to an exceptional behavior.

4 EXPERIMENT AND EVALUATION

4.1 Implementation

We have implemented the automatic fuzzing framework with around 12,000 Python lines of code and 500 C lines of code.

For the Seed Generator module, we implemented a general crawler and a passive crawler both with Selenium [35]. For the Monitor module, we implemented the three types of monitors independently. Specifically, the signal-based monitor is implemented in C with `strace` [39] and is implanted into devices by the help of device feature, known vulnerabilities or the serial port if possible. It is cross-compiled with Buildroot [8] as a statically linked binary. Now, it supports multiple architectures, including x86, x86-64, ARM32 (LE), ARM32 (BE), MIPS32 (BE) and MIPS32 (LE).

In order to restart the device when it hangs, SRFuzzer firstly monitors the device status by trying to establish the TCP connections to the target device repeatedly. Then it restarts the device by the Power Control module if the TCP connection could not be established normally several times. We implemented this module based on the Mi Smart Plug [2] with the help of python-miio [34] protocol.

4.2 Experiment Setting

In the experiment, we selected 10 devices from five different vendors to test. Table 2 shows their information. Last Column of the table shows the methods, to acquire permission to implant signal-based monitor, supported by each device.

Except signal-based monitor, all modules of SRFuzzer were deployed on Ubuntu 16.04. It connected the router by cable and a Mi Smart plug via a stable wireless connection. SRFuzzer fuzzed each device continuously for 40 hours. During the fuzzing process, it would restart the device if the Power Control module does not receive any response for more than six minutes.

4.3 Overall Experiment

In total, we monitored 208 unique exceptional behaviors. With the help of extra information that added by the Mutator module, we could confirm 101 unique issues and manually complete the PoCs[5]. Their details are shown in the last column of Table 3.

After reported to the related vendors under the responsible disclosure policy, 97 of 101 unique issues have been confirmed by their vendors, and another four issues are under assessing process. These

[5]For memory corruption, the PoC can cause the backend crash or hijack its control flow. For command injection, the PoC can execute a shell command such as "reboot". For XSS, the PoC can eject a message box in the browser with the content "Hello, XSS.". For information disclosure, the PoC can disclose some sensitive information.

Table 2: Information of Routers under Fuzzing

ID	Vendor	Product	Firmware Version	Architecture	Signal-based Monitor
1	NETGEAR	Orbi	V15.03.05.19 (6318)_CN	ARM32 (LE)	D, S
2	NETGEAR	Insight[**]	WAC505-510_firmware_V5.0.5.4	ARM32 (LE)	Not Support
3	NETGEAR	WNDR-4500v3	WNDR4500v3-V1.0.0.50	MIPS32 (BE)	D, S
4	NETGEAR	R8500	R8500-v1.0.2.100, R8500-V1.0.2.116	ARM32 (LE)	D, S
5	NETGEAR	R7800	R7800-V1.0.2.44, R7800-V1.0.2.46	ARM32 (LE)	D, S
6	TP-Link	TL-WVR900G	V3.0_170306	MIPS32 (BE)	Not Support
7	Mercury	Mer450	MER1200GV1.0	MIPS32 (BE)	Not Support
8	Tenda	G3	V15.11.0.5	ARM32 (LE)	E
9	Tenda	AC9	V15.03.05.19	ARM32 (LE)	E
10	Asus	RT-AC1200	RT-AC1200-3.0.0.4.380.9880	MIPS32 (LE)	D

[*]Stand for the three methods to acquire the permission to implant Signal-based Monitor: **D** for Device Feature, **E** for Existed Vulnerability and **S** for Serial Port.
[**]**Insight** is short for "Insight Managed Smart Cloud Wireless Access Point".

Table 3: Confirmed Issues and their Types

Product	SEED	CONF		READ				SUM
		MEM	CMD	MEM	CMD	XSS	INFO	
Orbi	216	0	0	0	0	1	1[*]	2
Insight	108	0	1	0	0	0	0	1
WNDR-4500v3	188	6	2	7	0	0	1[*]	16
R8500	208	9	0	0	0	3	1[*]	13
R7800	232	0	8	10	0	5	1[*]	24
TL-WVR900G	176	0	24	0	1	0	0	25
Mer450	91	0	2	0	0	0	0	2
G3	98	5	0	0	0	0	0	5
AC9	111	11	0	0	1	0	0	12
RT-AC1200	168	0	0	0	0	0	1	1
SUM	1586	31	37	17	2	9	5	101

[*] These four Information Disclosure issues are being assessed after submitting to the vendor.

97 vulnerabilities are assigned official IDs, including 43 CVE IDs[6], 52 PSV IDs[7] and 2 CNVD IDs.

Confirmed Issues. For each device, the number of collected seeds for SRFuzzer is given in column 2 and the number of issues for each aforementioned types (i.e., MEM for memory corruptions, CMD for command injections, XSS for cross-site scripting and INFO for information disclosure) grouped by two triggering phases (i.e., CONF and READ) is given in columns 3-8. Among the 101 confirmed issues, 48 issues are memory corruptions (47.52%) while the other includes 39 command injection issues (38.61%), 9 XSS issues (8.91%) and 5 information disclosure issue (4.95%). There are 67.33% of issues triggered in CONF operation and 32.67% in READ operation. All of the five devices that are confirmed more than ten issues were discovered multiple types of vulnerabilities, except for TL-WVR900G. After analyzing its implementation, we have found that its backend is implemented by Lua language, which could avoid memory corruption.

Furthermore, we evaluated the impact of 97 officially confirmed vulnerabilities from their CVSS version 3 scores [1] and their effects, which are shown in the Figure 7. We use the official four rankings based on CVSS scores to show the severity of the vulnerabilities. Specifically, more than a quarter of issues (25/97) are at high level. We also counted the number of issues for four categories according their effects, i.e., escalation of privilege (EoP), scripts execution, denial of service (DoS) and information disclosure. The majority of issues fall into EoP category, since we can craft their PoCs to hijack the control flow to execute a command of the low-level system, and hence we escalate the web-management privilege to the root privilege. Almost all PoCs can compromise the target device with one single message.

Effect of Monitors. The distribution of different confirmed issues caught by different monitors are shown in Table 4. We can observe that most of the confirmed issues (77.23%) are caught by the response-based monitor and the proxy-based monitor, showing the effectiveness of these device-independent monitors. We also find additional 23 issues are caught by the signal-based monitor, showing its ability to discover deep memory corruption vulnerabilities. To notice that, signal-based monitor can catch much more issues

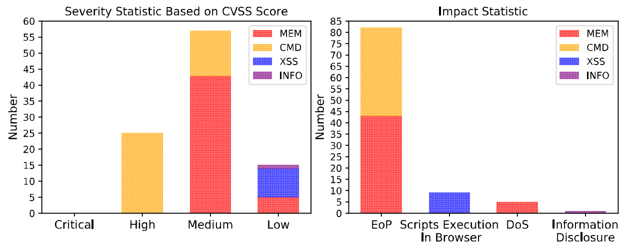


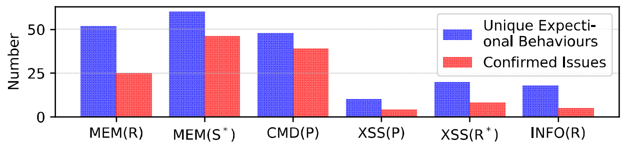
Figure 7: Impact of 97 Vulnerabilities

Table 4: Effectiveness of Monitors for 101 Confirmed Issues

Product	MEM		CMD	XSS		INFO
	R	S	P	P	R	R
Orbi	0	0	0	1	0	1
Insight	0	N/A	1	0	0	0
WNDR-4500v3	3	10	2	0	0	1
R8500	7	2	0	1	2	1
R7800	2	8	8	2	3	1
TL-WVR900G	0	N/A	25	0	0	0
Mer450	0	N/A	2	0	0	0
G3	4	1	0	0	0	0
AC9	9	2	1	0	0	0
RT-AC1200	0	0	0	0	0	1
SUM	25	23	39	4	5	5

R represents response-based monitor. **S** represents signal-based monitor. **P** represents proxy-based monitor. **N/A** represents this monitoring method is not supported by the device.

[6]We haven't disclosed all of the CVE IDs on the *oss-security mailing list* after being assigned, so not all of them can be found through the Internet now.
[7]The vendor haven't list all vulnerabilities on its security advisory yet, which causes several of the PSV IDs cannot be found through the vendor security advisory now.



* Compared with Table 4, there are 23 more confirmed issues for signal-based monitor and three more confirmed issues of XSS for response-based monitor. They are all duplicate issues that could be caught by other monitors.

Figure 8: Monitor Accuracy for 101 Confirmed Issues

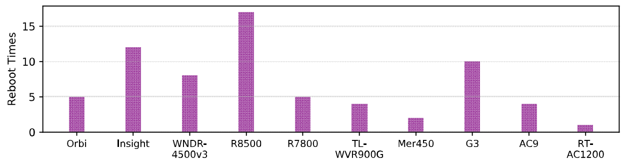


Figure 9: Device Reboot Times

for WNDR-4500v3 and R7800. It is caused by their special implementation. In their backends, they create subprocesses to handle the requests and always respond with "configuration failure" when subprocesses are crashed. In such cases, the signal-based monitor other than the reponse-based monitor can deal with them.

Figure 8 shows the number of unique exceptional behaviors reported by SRFuzzer, compared with issues we confirmed. In this experiment, we run each monitor independently. So some confirmed issues were caught by different monitors, as explained in Figure 8. On average, the accuracy of our monitors is 48.56%. The accuracy for all types of vulnerabilities is higher than 40.00%, except for information disclosure, which is very promising.

Reboot. The times of reboot for each device during the fuzzing is given in Figure 9. During 40 hours in our testing, each device rebooted 6.8 times on average. It is interesting that R8500 rebooted 3.6x as many as R7800 did. This is because R8500 handles all requests in only one process, while R7800 handles the requests by creating subprocesses. Moreover, the devices with Openwrt-based [18] operating system (e.g., TL-WVR900G) are more stable than others.

4.4 Comparison

We compared SRFuzzer with three popular open-source fuzzers in terms of discovering vulnerabilities. For memory corruption vulnerabilities, we chose *boofuzz* [27], a fork and successor of famous protocol fuzzer *Sulley* [21], as the comparative tool. For command injection vulnerabilities, we chose *Commix* [37] instead of *boofuzz*, as *Commix* are better with more mutation rules and monitoring methods. For XSS vulnerabilities, we selected *wfuzz* [28], a popular web fuzzing tool supporting XSS detection. SRFuzzer did not compare with others in information disclosure vulnerabilities, as most of the results (4 out of 5) are being assessed. We will perform the comparison in the future work.

In the comparison, seven devices among four vendors were selected. On these devices, we ran all of the tools for 40 hours without interruption, which was the same as SRFuzzer. We fed them with the raw requests, which were also same as SRFuzzer. To satisfy the special input requirement of boofuzz, we converted each seed with k-v pairs into the data representation of boofuzz[8]. As shown in Figure 10, SRFuzzer outperformed those three comparative fuzzers in all types of vulnerabilities. Specifically, it has found more memory corruption issues than boofuzz by 53.57% and more command injection issues than Commix by 25.81%. Meanwhile, SRFuzzer found one more XSS issue than wfuzz. We analyze the results in details.

Memory Corruption. *Boofuzz* cannot find any vulnerabilities with its default data representation. We encodes our seeds, which consists of k-v pairs, in boofuzz's data representation. In such a way, boofuzz can mutate request content field effectively, and trigger vulnerabilities. However, due to its lack of multiple monitor methods, boofuzz could miss issues that occur in READ operation and in a subprocess.

Command Injection. The true positives of *Commix* are less than those of SRFuzzer for two reasons. Firstly, among its many monitor methods, "time-related" injection monitoring technique is suitable for the devices. However, the technique relies on response time, which makes Commix miss some short-time exceptions. Secondly, Commix only monitors exceptions during CONF operation, which make it ignore the issues that occur in READ operation, such as issues in AC9 and TL-WVR900G.

Cross-site Scripting (XSS). SRFuzzer found one more XSS issue than *wfuzz*. It is in R8500. It is missed by wfuzz because the input generated cannot bypass the backend validity check without the guidance of K-V model. Therefore, the crafted value cannot be stored successfully through a CONF operation and the value returned to the frontend did not trigger a XSS issue.

5 DISCUSSION

In this section, we discuss the limitation of the current fuzzing framework and explore the improvement direction in the future.

Limitation of the Scope. The IoT device whose management protocols satisfy the C-R model and the K-V model can take advantage of SRFuzzer. We will extend our work to apply more types of device such as camera, switch, and printer, as well as other widely used management protocols such as SOAP.

Vulnerability of severity. Although SRFuzzer can fuzz devices without authentication, it does not find any pre-authenticated issues in our experiment. So we will examine the attack surfaces more thoroughly, to find hidden interface, so as to enhance the ability to discover authentication bypass vulnerabilities.

Research on data inconsistency. Based on the C-R model and the K-V model, we notice that there are several data inconsistencies between different backend procedures for a specific k-v pair. In this paper, we focus on the automatic fuzzing process and leave the analysis of the semantic relevance systematically for future investigations. The in-depth research can also help vendors to consolidate their security design.

Monitoring. From the monitoring perspective, SRFuzzer takes advantage of the two aspects: the various monitors for different types of vulnerability and the various monitoring mechanisms for the same type of vulnerability. We will try to improve the generality of the intrusive monitor, e.g., proposing a framework to repack the firmware or support direct flash writing.

[8]An example of the conversion to the data representation of boofuzz is listed in the Appendix

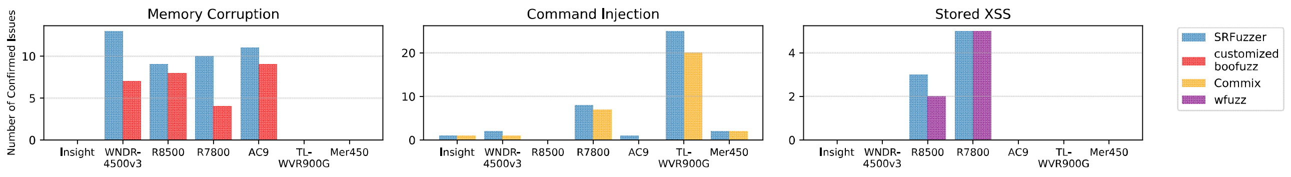


Figure 10: Comparison with other Fuzzers

6 RELATED WORK

There are many researches on detecting vulnerabilities of IoT devices, which might also be applied to SOHO routers. A. Costin et al. [13] performed a large scale analysis on the firmware images while not finding any issues at runtime. H. Bojinov et al. [7] audited several types of embedded management interface and B. Gourdin et al. [24] proposed *WebDroid* to build secure embedded web interfaces. Similar works [25], [17] for the SOHO devices are also proposed. Another type of large scale vulnerability detection is to scan for venerable devices in the internet. A. Cui [14] presented a quantitative lower bound on the number of the vulnerable embedded devices on a global scale. They found over 540,000 embedded devices are configured with factory default root passwords.

Fuzzing is an effective method to automaticlly discover vulnerabilities. Feedback-driven fuzzers [9], [22], [23], [31], [43], [33] used the runtime code coverage to guide the following inputs generation. You [41], Jain [26] worked on the automatic input type inference of input bytes. Fuzzing process may stick at particular branches with complex conditions. To solve this problem, researchers combined the fuzzing with symbolic execution [10], [30], [38]. Though they are promising, they are not suitable for the routers because of lacking internal runtime information.

Emulating devices is a potential solution to get the runtime information. A. Costin et al. [12] used qemu to emulate the web interface of certain linux-based devices. The method cannot work for web interfaces that contain special hardware related operations, like Wi-Fi configuration. FIRMADYNE [15] emulated Linux-based COTS firmware by supporting the emulation of NVRAM of devices. But it was limited only for ARM-based devices in our practice.

To overcome the firmware acquisition and emulation problems, fuzzing on physical devices was proposed. Z. Wang et al. [40] developed RPFuzzer to fuzz the router protocol like SNMP. IoTFuzzer [11] was an app-based fuzzing framework which aimed at finding memory corruptions in physical IoT devices without firmware images. It took advantage of the collaboration of fuzzing and taint analysis. It only focused on mobile-to-web interface and detected memory corruptions with only liveness check, which was not enough for finding web server vulnerabilities. Moreover, M. Muench [32] analyzed the challenges of fuzzing embedded devices and presented six heuristics to detect memory corruptions.

Program analysis techniques are also used for IoT devices to discover vulnerability. Q. Feng et al. [19] adopted a graph-based method to search for vulnerabilities in firmware images. They converted control flow graphs to numeric feature vectors, and used several hashing techniques to achieve real-time search. Y. Shoshitaishvili et al. [36] used static symbolic execute and program slicing to find backdoor. Besides, dynamic symbolic execution were also used. FIE [16] was a symbolic execution framework to find bugs for MSP430 firmware. Avatar [42] was a framework to coordinate emulator and device when analyzing the firmware. SRFuzzer is a complement to these techniques.

7 CONCLUSION

We have presented SRFuzzer to identify multi-type vulnerabilities of SOHO routers in a fully-automatic mode without device emulation. To find vulnerability comprehensively and deeply, we designed the KEY-VALUE data model and the CONF-READ communication model to reveal the data inconsistency and to guide the fuzzing. By the collaboration of six mutation rules and three monitoring mechanisms, we have tested 10 physical routers and been assigned 43 CVE IDs, 52 PSV IDs, and 2 CNVD IDs after the responsible disclosure.

8 ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their comments on an earlier version of this paper and our shepherd, Dr Sangho Lee, for helping us improve the presentation of the final version. This work is supported in part by Chinese National Natural Science Foundation (61602470, 61702508, 61802394, U1836209, 61772308, 61972224, U1736209), National Key Research and Development Program of China (2016QY071405), Strategic Priority Research Program of the CAS (XDC02040100, XDC02030200, XDC02020200), Program No.2017-JCJQ-ZD-043-01, and BNRist Network and Software Security Research Program (BNR2019TD01004, BNR2019RC01-009).

REFERENCES

[1] 2015. Common Vulnerability Scoring System (CVSS). https://nvd.nist.gov/vuln-metrics/cvss.
[2] 2015. Mi Smart Plug. https://www.mi.com/us/mj-socket/.
[3] 2015. Zerodium. https://zerodium.com/program.html.
[4] 2016. Multiple Netgear routers are vulnerable to arbitrary command injection. https://www.kb.cert.org/vuls/id/582384/.
[5] 2018. New VPNFilter malware targets at least 500K networking devices worldwide. https://blog.talosintelligence.com/2018/05/VPNFilter.html.
[6] 2018. Securing IoT Devices: How Safe Is Your Wi-Fi Router? https://www.theamericanconsumer.org/wp-content/uploads/2018/09/FINAL-Wi-Fi-Router-Vulnerabilities.pdf.
[7] Hristo Bojinov, Elie Bursztein, Eric Lovett, and Dan Boneh. 2009. Embedded management interfaces: Emerging massive insecurity. *BlackHat USA* (2009).
[8] buildroot. 2001. Buildroot - Making Embedded Linux Easy. https://buildroot.org/.
[9] CENSUS. 2016. Choronzon - An evolutionary knowledge-based fuzzer. https://github.com/CENSUS/choronzon.
[10] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (S&P)*.
[11] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IOTFUZZER: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Network and Distributed System Security Symposium (NDSS)*.

[12] A. Costin, J. Zaddach, and A. Francillon. 2016. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *ACM Asia Conference on Computer and Communications Security (ASIACCS).*

[13] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. 2014. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium.*

[14] Ang Cui and Salvatore J Stolfo. 2010. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Annual Computer Security Applications Conference (ACSAC).*

[15] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Network and Distributed System Security Symposium (NDSS).*

[16] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security Symposium.*

[17] Independent Security Evaluators. 2017. SOHO Network Equipment (Technical Report). https://www.securityevaluators.com/wp-content/uploads/2017/07/soho_techreport.pdf.

[18] Florian Fainelli. 2008. The OpenWrt embedded development framework. In *Free and Open Source Software Developers European Meeting (FOSDEM).*

[19] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *ACM Conference on Computer and Communications Security (CCS).*

[20] Roy Fielding and Julian Reschke. 2014. RFC 7231-Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. *Internet Engineering Task Force (IETF)* (2014).

[21] Fitblip. 2012. Sulley - a pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley.

[22] Google. 2015. Honggfuzz. https://github.com/google/honggfuzz.

[23] Google. 2015. syzkaller - linux syscall fuzzer. https://github.com/google/syzkaller.

[24] Baptiste Gourdin, Chinmay Soman, Hristo Bojinov, and Elie Bursztein. 2011. Toward Secure Embedded Web Interfaces. In *USENIX Security Symposium.*

[25] HP-Fortify-ShadowLabs. 2014. Report: Internet of Things Research Study. https://www8.hp.com/us/en/hp-news/press-release.html?id=1744676.

[26] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: Using Input Type Inference To Improve Fuzzing. In *Annual Computer Security Applications Conference (ACSAC).*

[27] jtpereyda. 2012. A fork and successor of the Sulley Fuzzing Framework. https://github.com/jtpereyda/boofuzz.

[28] jtpereyda. 2014. Wfuzz - The Web Fuzzer. https://github.com/xmendez/wfuzz.

[29] Swati Khandelwal. 2018. Thousands of MikroTik Routers Hacked to Eavesdrop On Network Traffic. https://thehackernews.com/2018/09/mikrotik-router-hacking.html.

[30] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. 2017. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *USENIX Annual Technical Conference (USENIX ATC).*

[31] LLVM. 2015. libFuzzer - a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html.

[32] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Network and Distributed System Security Symposium (NDSS).*

[33] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium (NDSS).*

[34] rytilahti. 2018. python-miio:Python library & console tool for controlling Xiaomi smart appliances. https://github.com/rytilahti/python-miio.

[35] Selenium. [n. d.].

[36] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Network and Distributed System Security Symposium (NDSS).*

[37] Anastasios Stasinopoulos, Christoforos Ntantogian, and Christos Xenakis. 2015. Commix: Detecting and exploiting command injection flaws. *BlackHat EU* (2015).

[38] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Network and Distributed System Security Symposium (NDSS).*

[39] strace. 2000. strace - linux syscall tracer. https://strace.io/.

[40] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. 2013. RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing. *KSII Transactions on Internet and Information Systems (TIIS)* (2013).

[41] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *IEEE Symposium on Security and Privacy (S&P).*

[42] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Network and Distributed System Security Symposium (NDSS).*

[43] Michal Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

9 APPENDICES

A DATA REPRESENTATION SAMPLE

Listing 1 shows a data representation sample for boofuzz, this sample is related to the requests from Figure 4.

```
if s_block_start("post blob"):
  s_static("submit_flag=")
  s_string("ntp_debug")
  s_static("&conflict_wanlan=")
  s_string("")
  s_static("&ntpserver1=")
  s_string("time.test1.com")
  s_static("&ntpserver2=")
  s_string("time.test2.com")
  s_static("&ntpadjust=")
  s_string("0")
  s_static("&hidden_ntpserver=")
  s_string("GMT8")
  s_static("&hidden_dstflag=")
  s_string("0")
  s_static("&hidden_select=")
  s_string("33")
  s_static("&dif_timezone=")
  s_string("0")
  s_static("&time_zone=")
  s_string("GMT-8")
  s_static("&ntp_type=")
  s_string("0")
  s_static("&pri_ntp=")
  s_string("")
s_block_end()
```

Listing 1: A Data Representation Sample for boofuzz

B SOAP REQUEST SAMPLE

Listing 2 show a SOAP request sample for NTP configuration, the KEY-VALUE pairs are emphasized with red and green.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.
xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header>
    <SessionID>ABCDEFGHIJKLMNOPQRST</SessionID>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
    <M1:SetNTP xmlns:M1="urn:ROUTER:service:DevConfig:
    1">
        <Option>Preferred</Option>
        <NTPServer1>time.test1.com</NTPServer1>
        <NTPServer2>time.test2.com</NTPServer2>
        <TimeZone>GMT-8</TimeZone>
    </M1:SetNTP>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 2: A SOAP Request Sample

C ASSIGNED VULNERABILITIES

Table 5 shows the assigned CVE, PSV and CNVD ids during the fuzzing. The exact numbers are hidden for the anonymity and they will be replaced with the original value if this paper is accepted.

Table 5: All assigned vulnerability IDs

Model	Vulnerabity ID
NETGEAR Orbi	PSV-2017-3093
NETGEAR Insight	PSV-2018-0610
NETGEAR WNDR-4500v3	PSV-2017-3169, PSV-2017-3167, PSV-2017-3170, PSV-2017-3168, PSV-2017-3154, PSV-2017-3158, PSV-2017-3159, PSV-2017-3152, PSV-2017-3165, PSV-2017-3166, PSV-2017-3157, PSV-2017-3156, PSV-2017-3160, PSV-2017-3155, PSV-2017-3153
NETGEAR R8500	PSV-2017-3065, PSV-2017-2460, PSV-2017-2427, PSV-2017-2428, PSV-2017-2254, PSV-2017-2226, PSV-2017-2229, PSV-2017-2228, PSV-2017-2227, PSV-2018-0244, PSV-2018-0243, PSV-2018-0242
NETGEAR R7800	PSV-2018-0116, PSV-2018-0115, PSV-2018-0148, PSV-2018-0144, PSV-2018-0141, PSV-2018-0142, PSV-2018-0139, PSV-2018-0132, PSV-2018-0173, PSV-2018-0140, PSV-2018-0136, PSV-2018-0138, PSV-2018-0145, PSV-2018-0171, PSV-2018-0146, PSV-2018-0147, PSV-2018-0137, PSV-2018-0135, PSV-2018-0133, PSV-2018-0172, PSV-2018-0174, PSV-2018-0159, PSV-2018-0158
TP-Link TL-WVR900G	CVE-2017-15613, CVE-2017-15614, CVE-2017-15615, CVE-2017-15616, CVE-2017-15617, CVE-2017-15618, CVE-2017-15619, CVE-2017-15620, CVE-2017-15621, CVE-2017-15622, CVE-2017-15623, CVE-2017-15624, CVE-2017-15625, CVE-2017-15626, CVE-2017-15627, CVE-2017-15628, CVE-2017-15629, CVE-2017-15630, CVE-2017-15631, CVE-2017-15632, CVE-2017-15633, CVE-2017-15634, CVE-2017-15635, CVE-2017-15636, CVE-2017-15637
Mercury Mer450	CVE-2018-12488, CVE-2018-12489
Tenda G3	CVE-2018-12057, CVE-2018-12058, CVE-2018-12059, CVE-2018-12060, CVE-2018-12061
Tenda AC9	CVE-2018-8742, CVE-2018-8743, CVE-2018-8744, CVE-2018-8745, CVE-2018-8746, CVE-2018-8747, CVE-2018-8748, CVE-2018-8749, CVE-2018-8750, CVE-2018-8751, CNVD-2019-00015, CNVD-2019-00016
Asus RT-AC1200	CVE-2017-16901