Automatic Inference of Taint Sources to Discover Vulnerabilities in SOHO Router Firmware

Kai Cheng, Dongliang Fang, Chuan Qin, Huizhao Wang, Yaowen Zheng, Nan Yu, Limin Sun

Automatic Inference of Taint Sources to Discover Vulnerabilities in SOHO Router Firmware

Kai Cheng[1,2], Dongliang Fang[1,2], Chuan Qin[1,2], Huizhao Wang[1,2], Yaowen Zheng[3], Nan Yu[2], and Limin Sun[2*]

[1] School of Cyber Security, University of Chinese Academy of Sciences, China
[2] Beijing Key Laboratory of IoT Information Security Technology, Institute of Information Engineering, Chinese Academy of Sciences, China
[3] Nanyang Technological University
{chengkai,fangdongliang,qinchuan,wanghuizhao,yunan,sunlimin}@iie.ac.cn, yaowen.zheng@ntu.edu.sg

Abstract. Cyberattacks against SOHO (small office and home office) routers have attracted much attention in recent years. Most of the vulnerabilities exploited by hackers occur in the web servers of router firmware. In vulnerabilities detection, static taint analysis can quickly cover all code without depending on the runtime environment compared to dynamic analysis (e.g., fuzzing). However, existing static analysis techniques suffer from a high false-negative rate due to the lack of resolution of indirect calls, making it challenging to track tainted data from a common source (e.g., *recv*) to a sink. In this work, we propose a new heuristic approach to address the challenge. Instead of resolving the indirect calls, we automatically infer taint sources through identifying functions with key-value features. We can bypass the indirect calls with the inferred taint sources and track the taint to detect vulnerabilities by static taint analysis. We implement a prototype system and evaluate it on 10 popular routers across 5 vendors. The proposed system discovered 245 vulnerabilities, including 41 1-day vulnerabilities and 204 vulnerabilities never exposed before. The experimental results show that our system can find more bugs compared to a state-of-the-art fuzzing tool.

1 Introduction

The *Internet of Things* (IoT) is one of the fastest emerging technologies in the last decade. According to the Statista research, by 2025, there will be more than 21.5 billion active IoT devices installed world-wide [16]. They are usually connected to the Internet through the small office and home office (SOHO) routers. Unfortunately, due to the lack of advanced defense mechanisms and the Internet-facing characteristic, SOHO routers have become hotbeds for remote exploitation [17]. For example, in 2019, Echobot, a new variant of Mirai, hit the Internet [9]. This sophisticated attack exploited up to 71 vulnerabilities in

* Corresponding author

various SOHO routers, leading to serious consequences, such as remote code execution and command injection.

Most SOHO routers implement a customized web server to manage and configure the functionality of the devices. One of the common web servers is the *HTTP* server, which provides network services through the HTTP protocol. These web servers directly receive requests from the network, which are attacker-controlled data, making them susceptible to vulnerabilities. There have been many works for discovering vulnerabilities in SOHO routers in recent years, including static analysis [6, 13] and dynamic analysis [5, 20–22]. Most of the vulnerabilities found in SOHO routers are related to the web server. In this paper, we also aims at discover vulnerabilities in the web server of the SOHO router.

In dynamic analysis, the mainstream approach is to discover vulnerabilities by greybox fuzzing on simulated firmware [21,22] or blackbox fuzzing on physical device [5, 20]. However, both methods currently face the problem of traditional fuzzing itself, such as low code coverage, which can lead to a large number of false negatives. In static analysis, the related works [6,13] use static taint analysis to discover a class of vulnerabilities called the taint-style vulnerability, in which data are passed from an attacker-controlled source to a security-sensitive sink without sanitization [19]. Compared with dynamic analysis, static analysis tests the code without actually executing it on the runtime environment and thus is a more practical and economical option for testing router firmware. For example, static analysis can analyze the firmware of a large number of routers without the need for expensive real-world devices. Moreover, for some types of vulnerability detection, static analysis enables higher code coverage with a lower false negative. In this work, we focus on using static analysis to find taint-style vulnerabilities in SOHO router firmware.

Challenges in discovering taint-style vulnerability. By nature, the effectiveness of finding taint-style vulnerabilities heavily relies on a good data dependency analysis tool; indeed, to find a defect, the tool has to construct a path in which taint is propagated from an attacker-controlled source to a security-sensitive sink. However, in binary analysis, the presence of indirect calls makes it difficult for existing static taint analysis technique to track tainted data from a common source (e.g., *recv*) to a sink. As a result, lots of vulnerabilities were missed. Although existing work used heuristics to bypass indirect calls and found subsets of taint-style vulnerabilities, these heuristics are inefficient. For example, in DTaint [6], the author manually specified some vendor-customized functions (e.g., *find_var* and *websGetVar*) as taint sources. In KARONTE [13], the author used a preset list of network-encoding strings (e.g., "soap" or "HTTP") as the keywords to infer taint sources. However, the former requires manually find special functions by firmware reverse-engineering; the latter's heuristic approach is not comprehensive enough to infer taint source with unknown strings (e.g., "entrys" in Figure 1).

Our Approach. To address these challenges, we propose a new heuristic approach to automatically infer taint sources instead of solving the indirect calls. Specifically, we observed a class of functions that obtain values by indexing keywords from user requests. These values are attacker-controlled. Therefore, these functions can be used as taint source for taint-style vulnerability detection. By inferring these taint sources, our taint analysis does not necessarily start from the common source (e.g., *recv*). Instead, it can start directly with the inferred taint sources. Throughout this paper, we refer to these functions as ***key-value functions***. To identify key-value functions, we first summarize the features of such functions and find key-value functions through data-flow analysis. Second, some key-value functions obtain data locally rather than from the network, so we filter these functions to reduce false positives. Third, a taint source summary is generated and passed to static taint analysis engine for vulnerabilities detection. We implement a prototype system and evaluate it on 10 popular routers across 5 vendors. The proposed system discovered 245 vulnerabilities, including 41 1-day vulnerabilities and 204 vulnerabilities never exposed before.

Contributions. In summary, we make the following contributions in this paper:

- We proposed a new approach for inferring taint source automatically on the firmware of SOHO routers. With the inferred taint source, we achieved static taint analysis on binary to discover vulnerabilities.
- We implemented a system prototype and evaluated it on ten real-world firmware images, showing that our tool can successfully find key-value taint sources, resulting in the discovery of 245 vulnerabilities. Compared to a state-of-the-art fuzzing tool, our prototype can discover more vulnerabilities.

2 Background and Motivation

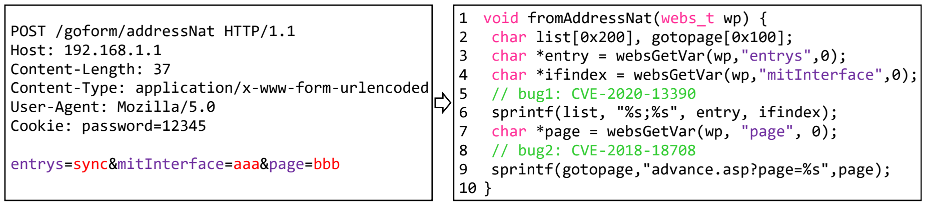
2.1 Typical Architecture of SOHO Router

In addition to providing its routing network services, the SOHO router typically utilizes a built-in web server for administration and configuration. Then, the user can connect to its management interface through a web browser and configure the various functionalities for the router, such as setting the wireless password, IP address, whitelists. Moreover, some router vendors provide mobile APPs to manage the devices [5]. The administration and configuration are usually based on standard protocols, such as HyperText Transfer Protocol (HTTP), and their typical implementation is composed of a frontend, a backend, and a database. The frontend provides an interface to guide the user in configuring the router, the backend processes user requests and parses the requests to perform corresponding functionalities, and the database stores the obtained configuration [20].

2.2 Key-Value Features

A typical user request sent to the SOHO router contains a URL and several different key-value pairs. When a request is received, the backend parses the request to obtains the key and corresponding value. Then the webserver configures the routers according to the obtained value. However, the program often lacks security checks for these values, resulting in many vulnerabilities, e.g., memory corruptions, command injections, cross-site scripting (XSS).

Motivating Example. We demonstrate this in the following example, based on a real-world firmware image. The left half of Figure 1 shows a POST request and

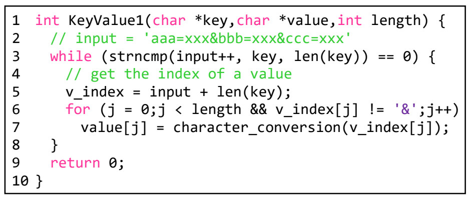


```
POST /goform/addressNat HTTP/1.1
Host: 192.168.1.1
Content-Length: 37
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0
Cookie: password=12345

entrys=sync&mitInterface=aaa&page=bbb
```

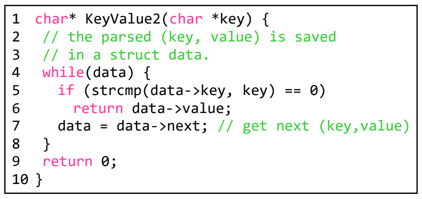
```
1  void fromAddressNat(webs_t wp) {
2    char list[0x200], gotopage[0x100];
3    char *entry = websGetVar(wp,"entrys",0);
4    char *ifindex = websGetVar(wp,"mitInterface",0);
5    // bug1: CVE-2020-13390
6    sprintf(list, "%s;%s", entry, ifindex);
7    char *page = websGetVar(wp, "page", 0);
8    // bug2: CVE-2018-18708
9    sprintf(gotopage,"advance.asp?page=%s",page);
10 }
```

Fig. 1: A motivation example.



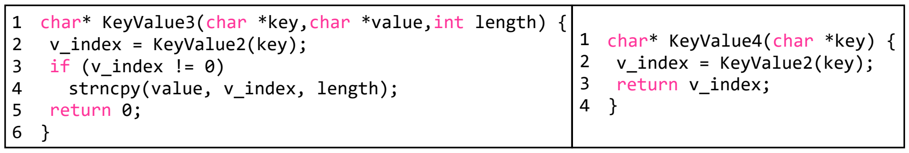
```
int KeyValue1(char *key,char *value,int length) {
  // input = 'aaa=xxx&bbb=xxx&ccc=xxx'
  while (strncmp(input++, key, len(key)) == 0) {
    // get the index of a value
    v_index = input + len(key);
    for (j = 0;j < length && v_index[j] != '&';j++)
      value[j] = character_conversion(v_index[j]);
  }
  return 0;
}
```

(a) key-value model one.



```
char* KeyValue2(char *key) {
  // the parsed (key, value) is saved
  // in a struct data.
  while(data) {
    if (strcmp(data->key, key) == 0)
      return data->value;
    data = data->next; // get next (key,value)
  }
  return 0;
}
```

(b) key-value model two.



```
char* KeyValue3(char *key,char *value,int length) {
  v_index = KeyValue2(key);
  if (v_index != 0)
    strncpy(value, v_index, length);
  return 0;
}
```

```
char* KeyValue4(char *key) {
  v_index = KeyValue2(key);
  return v_index;
}
```

(c) key-value model three.

Fig. 2: The abstract C codes of three key-value function models.

the right half of Figure 1 shows the procedure of processing the request in the backend. This example corresponds to two known vulnerabilities, CVE-2018-18708 and CVE-2020-13390. The POST data contains three parameter-keys: "entrys", "mitInterface" and "page". While processing these parameter-keys in the backend, their values are read by function `websGetVar` through indexing the key (line 3, 4 and 8) and directly passed to `sprintf` without any sanitization. Therefore, when the length of string *entry* or the length of string *ifindex* is larger than 0x200, a stack-based buffer overflow could be triggered at line 6. Another stack-based buffer overflow could be provoked at line 10 when the length of string *page* is larger than 0x100.

Key-Value Function Model. In Figure 1, function `websGetVar` matchs a key from the data struct *wp* and returns a value. Throughout this paper, we call the function like `websGetVar` as a key-value function. Our goal is to automatically identify key-value functions and discover vulnerabilities by tracking the value through static taint analysis.

Different SOHO router vendors implement key-value functions in different manners. According to our observation, there are two main implementation methods of key-value functions: the value is indexed by the key and (a) saved by a pointer parameter or (b) saved by a return pointer. As shown in Figure 2, the abstract C codes of these two implementation methods correspond to the function `KeyValue1` and `KeyValue2`. For the function `KeyValue1` in Figure 2(a), the data *input* is the raw request posted by user. `KeyValue1` gets the address

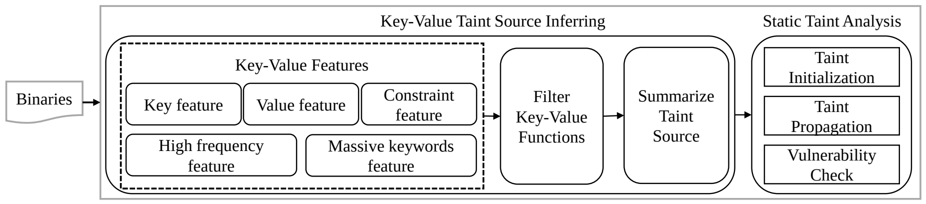


Fig. 3: Overview

v_ index of the corresponding value by finding the keyword *key* in *input* through calling a strcmp-like function (line 3-5). The raw value is then copied to the buffer pointed by the parameter *value* through a loop after examining some special characters (line 6-7). For the function `KeyValue2` in Figure 2(b), the raw request has been split into (key, value) pairs and saved to a structure object *sd*. Then, `KeyValue2` indexes the keyword *key* by calling the strcmp-like function and returns a value *sd->value* (line 3-7).

There are some features in the two implementation of the key-value function. (1) *Key feature*: a parameter to key directly propagates to an argument of the strcmp-like function which is invoked in a loop; (2) *Value feature*: the pointer to value depends on one parameter (e.g., `KeyValue1`) or the return value (e.g., `KeyValue2`), and the data dependency graph contains a loop; (3) *Constraint feature*: key-value function may have a constant parameter that limits the value's length. However, this feature is optional and does not always exists in the implemention.

Besides, the behavior of calling key-value function to process requests has 2 additonal features: (4) *High frequency feature*: the key-value function was called multiple times under different call-sites. (5) *Massive keywords feature*: different call-sites will refer to various constant keywords.

Except for the two implementation methods described above, some functions are wrappers for key-value functions. These functions also have key-value functionality, but they do not implement *key feature* and *value feature* themselves. As shown in Figure 2(c), function `KeyValue3` calls a key-value function `KeyValue2` to get the value *v_ index* and copies it to a memory block pointed by parameter *value* through calling a strcpy-like function. And function `KeyValue4` calls the key-value function `KeyValue2` to get the value *v_ index* and returns it.

To better describe the following design, we refer to the functions like `KeyValue1` as *key-value model one*, refer to the functions like `KeyValue2` as *key-value model two*, and refer to the functions like `KeyValue3` or `KeyValue4` as *key-value model three*.

3 Detailed Design

In this section, we illustrate the detailed design of our system. As shown in Figure 3, the proposed system is comprised of two major components. Its input is a binary with mainstream architectures (e.g., ARM, MIPS) used in embedded systems. To implement architecture-agnostic, it first converts the binary machine code into the intermediate representation (IR) [11]. Therefore, the proposed static binary analysis based on the VEX IR, which is a popular IR widely used in many program analysis tools, including Valgrind [11] and Angr [15]. To infer taint source, it first identifies the key-value features with static analysis. Then, it filters key-value functions by retrieving keywords from the local files extracted from the firmware image. Finally, it summarizes the taint source function and passes the summary information to the static taint analysis engine for vulnerabilities detection. To discover vulnerabilities, it first initializes data from the inferred taint source and tracks the taint through static taint analysis. Then, it detects taint-style vulnerabilities based on the constraints of the tainted data.

3.1 Key-Value Taint Source Inferring

This section describes how to infer the taint sources by identifying the key-value features with static analysis.

Identify Key-Value Function. As we mentioned in Section 2.2, a typical key-value function has some apparent features, as follows: *key feature*, *value feature*, *constraint feature*, *high frequency feature*, and *massive keywords feature.*

To find the above five features, our system first uses IDA Pro, the most powerful reverse-engineering tool in the market, to automatically identify functions in the target binary. Then, it selects the candidate functions through *high frequency feature* instead of analyzing all the functions, which can improve the analysis's efficiency. According to our experiments, key-value functions are typically called more than 100 times, which is the threshold for filtering. To identify the *massive keywords feature*, our system analyzes each candidate function's contexts at different call-sites to check whether it contains a pointer argument that points to a constant string. If not found or the number of constant string found does not exceed half the number of times the candidate function is called, our system removes the function from the candidate set. For *constraint feature*, our system also analyzes the function's contexts to summarize whether constant values are assigned to the same parameter at different call-sites. If found, the parameter is marked as a length constraint of the value. In the following, we illustrate how to identify *key feature* and *value feature* through data-flow analysis for the remaining candidate functions.

First, our system adopts the same approach proposed in [13] to identify strcmp-like functions automatically. Second, for a candidate function, our system generates a control flow graph (CFG) and traverses the CFG to find loops. If the CFG contains a loop and calls a strcmp-like function in the loop, our system backward traces the arguments of the strcmp-like function by following the conventional use-def chains. *Key feature* is identified if one of the strcmp-like function arguments depend on one parameter of the candidate function. Third, for *value feature* in model one, the bytes in value are moved through a loop from one buffer to another, where the address of another buffer is a pointer parameter of the candidate function. Our system forward traces all parameters of the candidate function and try to find a parameter, which is a pointer and is used to an address of a byte-store instruction(e.g., the register `R4` in `STRB R0,[R4,R5]` in ARM). If found, our system backward traces the address of the byte-store instruction (e.g., the register `R4` and `R5`) to generate a data dependency graph

(DDG). *Value feature* is found if there is a loop in the DDG. As a result, the candidate function is recognized as the *key-value model one*. Otherwise, our system backward traces the return value of the candidate function to generate a DDG. If there is a loop in the DDG, the candidate function is recognized as the *key-value model two*.

If the candidate function is neither *key-value model one*, nor *key-value model two*, we determine whether it is *key-value model three*. To identify this model, our system analyzes all its callees. If there is a callee that meets the features of *key-value model two*, our system iteratively analyzes its callers and determine whether its caller is *key-value model three* by the following two conditions.

(1) The key-value function and its caller have the same argument *key* that points to a constant string.
(2) For the function like `KeyValue3`, the caller's return value depends on the key-value function's return value. For the function like `KeyValue4`, the value returned from the key-value function is copied to the memory block pointed to by the caller's pointer parameter. The copy is implemented by some library functions (e.g., strcpy, strncpy, memcpy, etc.).

Once founding a new key-value function, our system continues to analyze its callers according to the above two conditions until no other key-value function is found.

Filter Key-Value Functions and Generate Summary of Taint Sources. There are also some functions that show the features, but they are not handling key-value pair from network inputs (e.g., parsing local configuration files). The values read from local files are not controlled by an attacker. Therefore, these key-value functions should not be treated as taint sources. We introduced how to filter them in the following. First, our system collects all constant keywords at its calling contexts for each identified key-value function. Then, our system extracts all the files in the firmware image and matches the content in texts with regular expressions to look up strings like *key=value* or *key:value*. Lastly, if most of the keywords of a key-value function (more than 80% of the number of times the key-value function is called) are in the local file, this key-value function is not considered as a taint source. Besides, our system filters which key-value functions are called no more than 100 times.

For the identified taint sources, our system summarizes their parameters and return values. The summary information includes the parameter or return value that needs to be tainted, and the parameter represents the length constraint of the value and parameter for keywords. This information is passed to the static taint analysis engine for taint-style vulnerabilities detection.

3.2 Static Taint Analysis

The static taint analysis aims at find taint-style vulnerabilities by tracking the tainted data from the taint sources identified above.

Taint Initialization and Propagation. Instead of starting from the entry point (e.g., `main`) of a program, our system starts with the functions that invoke

Table 1: The Taint Propagation Rules in VEX IR

IR Statements	**Description**	**Taint Rules**
$t_i = GET(r_j)$	Assign a value from register r_j to Temp t_i	$r_j \mapsto t_i$
$PUT(r_j) = t_i$	Assign Temp t_i to register r_j	$t_i \mapsto r_j$
$t_i = t_j$	Assign Temp t_j to Temp t_i	$t_j \mapsto t_i$
$t_i = binop(t_n, t_m)$	Assign result of $t_n \Diamond_b t_m$ to Temp t_i $\Diamond_b ::= +, -, *, /, \ll, \gg$ $\Diamond_b ::= <, >, <=, >=, ==, !=$	 $t_n \mapsto t_i, t_m \mapsto t_i$ $t_n \rightsquigarrow t_i, t_m \rightsquigarrow t_i$
$t_i = ITE(t_j, t_n, t_m)$	If t_j is true, $t_i = t_n$, otherwise, $t_i = t_m$	$t_n \mapsto t_i, t_m \mapsto t_i$
$t_i = LDle(t_j)$	Assign value loaded from address t_j to t_i	$t_j \mapsto t_i, *t_j \mapsto t_i$
$STle(t_j) = t_i$	Store t_i to the memory at address t_j	$t_i \mapsto *t_j$

$t_n \mapsto t_i$ denotes that the taint propagate from t_n to t_i.
$t_n \rightsquigarrow t_i$ denotes that if t_n is taint, its constraint is t_i.

the taint sources. It marks the argument register or returned register at each call-site that invokes a taint source as taint with a unique id according to its summary information. The unique id can distinguish tainted data that comes from different call-sites and helps us associate the tainted data with various keywords. Then, the tainted data is propagated forward along the def-use chains through the analysis of VEX IR's statements and expressions. In the forward taint analysis, Table 1 shows the taint propagation rules. Specially, when an IR statement's operator is a comparison (e.g., *binop* is *CmpLE*) and the tainted data is one of its operands, our system collects the constraint (e.g., $x < 8$ where x is taint). Moreover, if the comparison operand is not a constant (e.g., $x < y$ where x is taint), our system backward tracks the operand y to find its value within the current function. If operand y is found to be assigned a constant, our system updates the corresponding constraint. Another consideration is the taint propagation rule between the memory accesses with load and store operations. As shown in Table 1, the load operation `LDle` corresponds to two rules $t_j \mapsto t_i$ and $*t_j \mapsto t_i$. The former denotes that a value t_i is read from a tainted address t_j, and the value t_i is marked as taint. In this case, the loaded value t_i may be an integer or a character. The latter denotes that the object to which pointer t_j points is tainted data, and the loaded object t_i is marked as taint. In this case, the tainted data previously stored in memory through the `STle` operation is loaded with the same address. For the taint propagation in memory accesses, our system only tracks the global address, stack address, and simple indirect memory access with the same register base and constant offset (e.g., $STle(r_0 + 0x20)$ and $LDle(r_0 + 0x20)$).

For the interprocedural taint analysis, we apply the method of generating taint summaries to improve the efficiency of taint analysis [12]. When a callee is encountered in the taint analysis, our system ignores the callee if its arguments are not tainted. Otherwise, our system follows the callee and generates a summary for the callee. The taint summary describes the tainted parameters of input and the new parameters or return values that are tainted after analyzing the callee. Therefore, when the same callee is encountered, if the tainted data being tracked is in the taint summary of the callee, our system uses the summary to quickly propagate the taint. Otherwise, it analyzes the callee again and update the taint summary.

For the library functions, our system also adopts the taint summaries to propagate taint. Their taint summaries are generated manually, and we just implement summaries for some common string-related library functions (e.g., strcpy, memcpy, strstr, strcmp, etc.). For example, when meeting the library function `strcpy(*dest, *src)`, the taint propagates from parameter `src` to parameter `dest`. In taint analysis, library functions are not followed if they do not implement summaries.

Vulnerability Check. In our system, we mainly discover the stack-based buffer overflow and command injection vulnerabilities. When the taint reaches a string-copy sink, such as `strcpy(*dest, *src)`, and the copy's memory block pointed to by destination `dest` is a stack address, our system first calculates the maximum size `max_buffer` of the destination buffer. Then, our system checks the constraints of the tainted data. If the constraints are empty, an alert is generated. If the constraints are not empty and contain a symbolic constraint, the sink is security and no alert is generated. such as `len < x`, where the symbol `len` is the length of the string pointed to by the tained pointer and the symbol `x` is a symbolic value. Otherwise, our system solves the constraints to obtain the minimum value of the string length `len`. If the minimum value is larger than the `max_buffer`, an alert is generated. For command injection, if the taint reaches a command-execution sink (e.g., `system` and `popen`) and the constraints of the taint are empty, an alert is generated.

4 Evaluation

4.1 Implementation

We have implemented a prototype on top of VEX IR using python. In particular, we first utilize the IDA Pro to identify functions and generate a control flow graph (CFG) for the target program. Then, we load the target binary and convert assembly code into VEX IR by Angr's API based on the generated CFG. Finally, based on the IR, we implemented the data-flow analysis to infer key-value functions and taint analysis to discover vulnerabilities.

4.2 Experiment Setup

In the experiment, we selected 10 router's firmware images from five different vendors. Table 2 shows the summary information of 10 firmware images. We utilize Binwalk [3] to unpack the firmware images and extract the web-server programs that handle requests. The architecture of these programs includes ARM32 and MIPS32, which are the mainstream architectures used in SOHO routers. All the experiments were conducted on a Ubuntu 18.04.4 LTS OS with a 64-bit 8-core Inter(R) Core(TM) i7-8550 CPU and 24 GB RAM.

Table 2: Information of SOHO Routers and Analyzed Programs

ID	Vendor	Product	Version	Architecture	Program	Size (KB)
1	NETGEAR	Orbi RBR20	V2.6.1.36	ARM32	net-cgi	696
2	NETGEAR	WNDR4500v3	V1.0.0.50	MIPS32	net-cgi	694
3	NETGEAR	R8500	V1.0.2.116	ARM32	httpd	1,506
4	NETGEAR	R7800	V1.0.2.46	ARM32	net-cgi	581
5	Tenda	AC9V1.0	V15.03.05.19	ARM32	httpd	960
6	Tenda	AC9V3.0	V15.03.06.42_multi	MIPS32	httpd	2,039
7	Tenda	G3V3.0	V15.11.0.6	ARM32	httpd	1,676
8	Mercury	Mer450	MER1200GV1.0	MIPS32	nginx	759
9	D-Link	DAP-1860	v1.04B05	MIPS32	uhttpd	1,137
10	TP-Link	TL-WR940N	V6_200316	MIPS32	httpd	1,899

Table 3: The Summary of Identified Key-Value Functions

ID	Program	No. All Identified K-V	No. K-V af. Filtering	No. Called	No. Key	Time (sec.)	True Taint Source Functions
1	net-cgi	5	2	1,072	988	34.1	0x39864, 0x19b1c
2	net-cgi	3	2	956	908	26.1	0x40e8dc, 0x42c6c8
3	httpd	5	1	1,307	1,275	38.5	0x190c0
4	net-cgi	3	2	713	692	23.3	0x10678, 0x27bf4
5	httpd	9	1	491	485	27.5	0x2b9d4
6	httpd	7	1	491	485	40.8	0x430468
7	httpd	10	1	613	547	25.7	0x1c634
8	nginx	0	0	0	0	22.4	None
9	uhttpd	2	2	598	585	24.0	0x417574, 0x4a19d0
10	httpd	11	1	1,134	1,103	55.2	0x526710
Total	-	55	13	7,375	7,068	317.6	13

4.3 Key-Value Taint Source Inferring

Table 3 shows the results of key-value functions that our system automatically recognizes. It identified a total of 55 functions with key-value feature in 317.6 seconds. After filtering, it obtained 13 key-value functions, all of which have been verified as the `true taint source` by binary reverse-engineering. The true taint source refers to the constant key and corresponding value obtained by the key-value function are from network requests. The 13 taint source functions were called a total of 7,375 times, and 7,068 constant keys were found at these call-sites. With the exception of firmware MER450 with ID 8, which did not find the key-value function, the other nine firmware images inferred 1 or 2 taint sources. We found that these taint sources can be divided into two categories. One is to get key-value pairs from HTTP protocol-based requests and the other is to get key-value pairs from SOAP protocol-based requests. Firmware images with IDs 1, 2, 4, and 9 contain both types of taint sources, while taint sources in other firmware images fall into the first category.

Table 4: The Summary of the Vulnerabilities that Our System Found

ID	**Product**	**Alerts**		**True Alerts**		**Vulnerabilities**		**Time**
		MEM	**CMD**	**MEM**	**CMD**	**MEM**	**CMD**	**(sec.)**
1	Orbi RBR20	1	2	0	0	0	0	68.7
2	WNDR4500v3	65	5	41	4	31	4	51.8
3	R8500	57	10	37	5	31	5	83.2
4	R7800	68	12	36	2	30	2	65.9
5	AC9V1.0	76	15	67	7	30	5	71.6
6	AC9V3.0	85	11	78	3	35	3	78.5
7	G3V3.0	28	2	25	2	22	2	59.5
8	Mer450	0	0	0	0	0	0	-
9	DAP-1860	52	16	48	16	24	16	42.6
10	TL-WR940N	5	0	5	0	5	0	207.2
Total	-	510		376		245		729.0

Table 5: Known Published Vulnerabilities

Model	**Vulnerabilit ID**
Tenda AC9V1 Tenda AC9V3	CVE-2018-14492, CVE-2018-14559, CVE-2018-16333, CVE-2018-16334, CVE-2018-18706, CVE-2018-18707, CVE-2018-18708, CVE-2018-18709, CVE-2018-18727, CVE-2018-18728, CVE-2018-18729, CVE-2018-18730, CVE-2018-18731, CVE-2018-18732, CVE-2020-10987, CVE-2020-13389, CVE-2020-13390, CVE-2020-13391, CVE-2020-13392, CVE-2020-13393, CVE-2020-13394

4.4 Effectiveness of Vulnerability Detection

In the current prototype, we detect memory corruptions (MEM) and command injections (CMD). All the taint sources came from the key-value functions we inferred. Table 4 summarizes the results. For the 10 firmware images, our system reported 510 alerts in about 12 minutes, among which 376 were true positives (an accuracy of 73.7%). In this paper, we treat the vulnerabilities that meet anyone following the condition as duplicate vulnerability. 1) The tainted data that trigger the vulnerability is propagated from the same source, where the address of the call-site called the taint source is the same. 2) The tainted data from different sources trigger vulnerability at the same sink point. Therefore, we manually verified each true alert by reverse-engineering, removed the duplicate vulnerabilities and found 245 different vulnerabilities, including 208 memory corruptions and 37 command injections.

Vulnerability Verification. For the 245 different vulnerabilities, we verified them using dynamic analysis and static analysis. For dynamic analysis, we acquired three physical devices (NETGEAR R8500, NETGEAR R7800, and Tenda AC9V3) and successfully crafted PoEs for all the 106 vulnerabilities in the three devices, including 96 memory corruptions and 10 command injections. Since we do not have physical devices for the other 7 routers, we did not dynamically validate all vulnerabilities. For the remaining 139 vulnerabilities, we have verified them by binary reverse-engineering. The taint sources of all these vulnerabilities are the key-value function identified by our system. We manually analyzed each of the remaining vulnerabilities and found that the value that key-value function

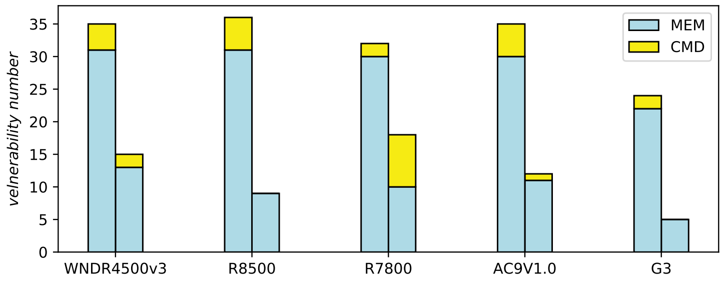


Fig. 4: Comparative Evaluation: our system vs. SRFuzzer

obtained from the request was propagated to the sink without any sanitization. Therefore, we believe that the remaining 139 vulnerabilities are true positives.

Further, to match the known vulnerabilities, we collected vulnerability information for the relevant devices from the Internet (e.g., exploit-db [2] and MITRE CVE [1]). As shown in Table 5, we found 21 CVE IDs with vulnerability details that match the vulnerabilities found by our system. All these IDs belong to Tenda AC9V1 and Tenda AC9V3, and the 21 IDs correspond to 41 different vulnerabilities. The reason is that an ID may correspond to multiple different vulnerabilities in the same device or different devices. For the remaining 204 vulnerabilities, no public disclosure information can be found. We have reported the 204 vulnerabilities to relevant vendors (responsible disclosure).

4.5 Comparison

Lastly, we conducted a comparison experiment with the state-of-the-art tool SRFuzzer [18]. SRFuzzer is the first whole-process fully-automatic framework for fuzzing web server of SOHO router. It obtains key-value pairs by capturing and parsing as many requests as possible and tests the device by mutating the values. In contrast, we search functions that process key-value in binary through static analysis and detect vulnerabilities by tracking values through taint analysis. Note that we only compare the ability to discover memory corruption and command injection.

SRFuzzer is not open-source, and 10 firmware images were analyzed according to the paper [18]. We selected 5 of the 10 firmware images for the comparison experiment. Figure 4 shows the results of the comparison. On the same 5 firmware images, SRFuzzer found 59 vulnerabilities, including 48 memory corruptions and 11 command injections. As a comparison, our system discovered 162 vulnerabilities, including 144 memory corruptions and 18 command injections. For these two types of vulnerabilities, our system detected more vulnerabilities than SRFuzzer in all five firmware images, except for the command injections in R7800.

For the other 5 firmware images, we did not find the corresponding firmware version of NETGEAR Orbi on the Internet, failed to unpack the firmware of NETGEAR Insight through Binwalk [3], and did not identify key-value functions after extracting the target programs from TP-Link TL-WVR900G, Mercury Mer450, and Asus RT-AC1200. In the 5 firmware images, SRFuzzer found a total of 28 command injections and 0 memory corruption, including 1 in firmware of NETGEAR Insight, 2 in firmware of Mercury Mer450, and the remaining 25 in firmware of TP-Link TL-WVR900G. However, all of the 27 command injections in Mer450 and TL-WVR900G are found in `Lua` script files, not binaries. Vulnerability detection in script files is beyond the scope of our system. Overall, even within the same 10 firmware, our system was able to find more bugs than SRFuzzer.

There are two reasons why our system found more vulnerabilities than SRFuzzer on the test set. First, static analysis can achieve higher code coverage than fuzzing. Second, SRFuzzer does fuzzing based on key-values obtained from captured HTTP requests. However, some key-value pairs obtained through static analysis cannot be obtained from these requests. For example, requests that contain key-values are based on the SOAP protocol, which is widely used in router configuration. Another example is some hidden URLs that cannot be triggered by the SOHO router's management interface through a web browser.

5 Discussion

In this section, we discuss the limitations of our system and what improvements could be made in the future. During the experimental evaluation, we found that the backend program in firmware of some routers does not always follow the three patterns described in Section 2.2 to parse the user requests. Therefore, our system cannot automatically infer the taint source and initialize the tainted data, resulting in the system missing some potential vulnerabilities in these firmware.

According to our analysis, there are three main reasons. First, the backend CGI program gets value from the HTTP request by calling function `getenv` with a keyword. For example, in the firmware of NETGEAR R7800, the CGI program *net-cgi* obtains a string value by calling `getenv("HTTP_USER_AGENT")` in function `sub_4DCD8` and propagates the value to `strcpy` without checking the string length. This is a memory corruption vulnerability found by KARONTE [13] and will be missed by our system. Second, the backend program directly reads value from the HTTP request without using any keyword. Third, the backend program invokes the script file (e.g., `Lua` script in Mer450 and TL-WVR900G) to parse and process the HTTP request. For example, in the vulnerability CVE-2017-15614 found by SRFuzzer [18], the value of "outif" keyword is read and propagated to function `sys.fork_exec` in the pptp_client.lua file. In the first case, we can find the corresponding vulnerability by setting `getenv` as the taint source. In the latter two cases, we will analyze more firmware to find other patterns for inferring the taint source and try to analyze scripting languages to discover more potential vulnerabilities.

6 Related Work

Fuzzing on Embedded Devices. Many related works utilize fuzzing to discover vulnerabilities on physical device or on emulated firmware. Wang et al. [18] proposed a fuzzing framework RPFuzzer for detecting vulnerabilities in specific router protocols. To tackle the challenge of lacking in access to firmware images of some embedded devices, IoTFuzzer [5] detects memory corruption by performing black-box fuzzing on the real-world devices with the rich communication information in the companion mobile apps. Zhang et al. [20] proposed a fully-automatic fuzzing framework called SRFuzzer to discover variation vulnerabilities in web servers of SOHO routers. It utilized the key-value feature in the request and the configuration-read model in the communication to guide the mutation for fuzzing. However, the fuzzing based on physical devices is limited by the high expensive cost and is not suitable for large-scale devices analysis. Muench et al. [10] achieved effective memory corruption detection on embedded devices through partial and full firmware emulation. Zheng et al. [22] proposed a technique called "augmented process emulation" to enhance throughput and guarantee correct emulation of the given program by switching between user-mode emulation and full system emulation. To improve the efficiency of fuzzing, they also adopted static analysis to help generate useful inputs for fuzzing [21]. However, due to different peripherals and frequent I/O interactions, current emulation techniques do not guarantee successful emulation of various firmware.

Static Vulnerability Discovery. Most of the vulnerability discovery based on static analysis foucus on traditional PC programs [4,12] and source code [7,8,19]. Few techniques exist to detect vulnerabilities on embedded devices. Firmalice [14] was specifically developed to detect the authentication bypass vulnerability in binary firmware. DTaint [6] detected taint-style vulnerabilities in firmware through building intra- and inter-procedural data flow in a bottom-up manner with pointer alias analysis. KARONTE [13] first proposed to discover multi-binary vulnerabilities through modelling interaction between multiple binaries in the firmware. However, DTaint relies on the symbols of the function to manually specify specific taint source to detect associated vulnerabilities; KARONTE relies on network-encoding keywords to infer the taint source. For example, keyword strings associated with "HTTP". In contrast, our system is able to automatically infer taint source functions and obtain large amounts of tainted data to reduce false negatives.

7 Conclusion

In this work, we propose a new heuristic approach to discover vulnerabilities in the firmware images of SOHO routers whithout addressing indirect calls. Specially, we automatically infer taint sources through identifying functions with key-value features. With the inferred taint sources, we track the taint to detect vulnerabilities by static taint analysis. We implement a prototype system and evaluate it on 10 popular routers across 5 vendors. The proposed system discovered 245 vulnerabilities, including 41 1-day vulnerabilities and 204 vulnerabilities never exposed before. The experimental results show that our system can find more bugs compared to a state-of-the-art fuzzing tool.

8 Acknowledgement

This work was supported by the National Key R&D Program of China (Grant No.Y950201104), and Key Program of National Natural Science Foundation of China (Grant No.U1766215).

References

1. "Common vulnerabilities and exposures," https://cve.mitre.org/.
2. "Exploit database of the website," https://www.exploit-db.com/.
3. "Firmware analysis tool," https://github.com/ReFirmLabs/binwalk.
4. G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86—a platform for analyzing x86 executables," in *CC*, 2005.
5. J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *NDSS*, 2018.
6. K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: detecting the taint-style vulnerability in embedded device firmware," in *DSN*, 2018.
7. N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *USENIX Security*, 2018.
8. D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "{FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *USENIX Security*, 2013.
9. M. Z. Eli Kreminchuker, "Echobot malware now up to 71 exploits, targeting scada," https://www.f5.com/labs/articles/threat-intelligence/echobot-malware-now-up-to-71-exploits–targeting-scada, 2019.
10. M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." in *NDSS*, 2018.
11. N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.
12. S. Rawat, L. Mounier, and M.-L. Potet, "Static taint-analysis on binary executables," http://web.cs.iastate.edu/ weile/cs513x/5.TaintAnalysis2.pdf, 2011.
13. N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *SP*, 2020.
14. Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, 2015.
15. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *SP*, 2016.
16. Statista, "Internet of things (iot)," https://www.statista.com/topics/2637/internet-of-things/, 2020.
17. TrendMicro, "Smart yet flawed: Iot device vulnerabilities explained," https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/smart-yet-flawed-iot-device-vulnerabilities-explained, 2020.
18. Z. Wang, Y. Zhang, and Q. Liu, "Rpfuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing." *KSII TIIS*, 2013.

19. F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *SP*, 2015.
20. Y. Zhang, W. Huo, K. Jian, J. Shi, H. Lu, L. Liu, C. Wang, D. Sun, C. Zhang, and B. Liu, "Srfuzzer: An automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities," in *ACSAC*, 2019.
21. Y. Zheng, Z. Song, Y. Sun, K. Cheng, H. Zhu, and L. Sun, "An efficient greybox fuzzing scheme for linux-based iot programs through binary static analysis," in *IPCCC*, 2019.
22. Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *USENIX Security*, 2019.