Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

UCRF: Static analyzing firmware to generate under-constrained seed for fuzzing SOHO router

Chuan Qin[a,b], Jiaqian Peng[a,b], Puzhuo Liu[a,b], Yaowen Zheng[c], Kai Cheng[d], Weidong Zhang[a], Limin Sun[a,b,*]

[a] *Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[b] *School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
[c] *Nanyang Technological University, Singapore*
[d] *Sangfor Technologies Inc, Shenzhen, China*

ARTICLE INFO

Article history:
Received 19 August 2022
Revised 12 December 2022
Accepted 21 February 2023
Available online 24 February 2023

Keywords:
Fuzzing
Binary static analysis
Vulnerability detection
IoT

ABSTRACT

SOHO (small office and home office) routers are the key elements of the IoT, providing network services for various smart devices. Recent years have seen increased attacks targeting SOHO routers' web applications. Numerous vulnerabilities are introduced in the process that web servers receive and handle external data directly. Fuzzing is the most popular technique for discovering such vulnerabilities. Previously proposed approaches generate fuzzing seeds in a valid format by analyzing the front-end. Unfortunately, the generated seeds are over-constrained by front-end code legality checks because malicious data can bypass the front-end inspection and be sent directly to the back-end. Moreover, such seeds ignore the semantics of the back-end, which makes the back-end's checking logic hinder the fuzzing's efficiency.

In this paper, we propose a novel approach to fuzzing SOHO routers by generating high-quality test cases via static analysis on the back-end binary. Specifically, we first obtain all communication interfaces in the back-end to avoid missing non-visible front-end interfaces. Then, we extract constraint information of all data fields using data-flow analysis on each interface. Ultimately, efficient and in-depth test cases can be generated only in meaningful test spaces based on constraint information. We implement our approach in a tool named UCRF. To illustrate the effectiveness of UCRF, it is evaluated on 10 real-world firmware from 4 vendors. UCRF found significantly more vulnerabilities of memory corruptions and command injection than the state-of-the-art work SRFuzzer on the five routers we had. Furthermore, UCRF found 41 0-day back-end vulnerabilities in total, 20 of which can be triggered only when the extracted constraints are satisfied.

1. Introduction

Internet-of-Things (IoT) devices are increasingly used in people's daily lives (Alrawi et al., 2019). According to the Global System for Mobile communication Association (GSMA) reports (The Mobile Economy, 2022), the total number of IoT devices worldwide has reached 15.1 billion by 2021 and is expected to grow to 23.3 billion by 2025. Small office and home office (SOHO) routers are extensively used to provide network services for a variety of devices such as smart homes, laptops, and smartphones, so the security of the SOHO router is critical. Unfortunately, common security mechanisms are missing in SOHO routers due to the limited resources in embedded system (Smart Yet Flawed, 2020; Thompson and Zatko, 2018; Yu et al., 2022), and numerous vulnerabilities can be exploited remotely by attackers (Cisco Small, 2022; Dir, 2022). For example, in 2016, the Mirai botnet compromised millions of IoT devices and launched a distributed denial-of-service (DDoS) attack against Dyn, an Internet infrastructure company, causing massive Internet service outages in Europe, and North America (Kolias et al., 2017; Mirai, 2016; Over, 2018; Travel Routers, 2017).

An important reason why routers are vulnerable to attack is the vulnerable web services within them (Cisco Small, 2022; Costin et al., 2016). The SOHO router provides a custom web server for the end-users to configure. A web server generally contains the front-end (HTML and JavaScript Etc.) and the back-end (such as a web server and common gateway interface). User data is entered

* Corresponding author at: Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China.
E-mail addresses: qinchuan@iie.ac.cn (C. Qin), pengjiaqian@iie.ac.cn (J. Peng), liupuzhuo@iie.ac.cn (P. Liu), yaowen.zheng@ntu.edu.sg (Y. Zheng), chengkai@sangfor.com.cn (K. Cheng), zhangweidong@iie.ac.cn (W. Zhang), sunlimin@iie.ac.cn (L. Sun).

https://doi.org/10.1016/j.cose.2023.103157
0167-4048/© 2023 Published by Elsevier Ltd.

at the front-end and sent via Hypertext Transfer Protocol (HTTP) to the back-end for further processing. Malicious user network data can easily bypass front-end inspection (e.g., man-in-the-middle attacks) and be sent to the back-end for handling. Eventually, the vulnerability is exploited to leak information or crash devices.

In recent years, numerous vulnerability discovery approaches for SOHO routers have been proposed, including static analysis (Chen et al., 2021; Cheng et al., 2021; 2018; Redini et al., 2020) and dynamic testing (e.g., fuzzing) (Srivastava et al., 2019; Wang et al., 2013; Zhang et al., 2021; 2019; Zheng et al., 2019a; 2019b). Static analysis has a natural limitation with many false positives (Shoshitaishvili et al., 2016; Vadayath et al., 2022). Significant manual effort is introduced for verifying vulnerability alerts and writing proof-of-concept (PoC) scripts. Fuzzing is a popular technique for vulnerability discovery. Emulation of the firmware provides runtime information of the firmware, including code coverage, which can be used to guide fuzzing. However, emulating the custom firmware requires a lot of manual work due to the peripheral dependencies (Chen et al., 2016; Kim et al., 2020; Zaddach et al., 2014). To avoid the problems in firmware emulation, some research work chose to test the physical router directly. However, it is challenging to generate efficient test cases and perform in-depth testing without the guidance of valid information, which eventually leads to lots of false negatives.

Existing physical SOHO router fuzzing methods generate test cases based on the front-end (Srivastava et al., 2019; Zhang et al., 2019). For example, these methods simulate the behavior of the front-end and generate requests as seeds for fuzzing. However, the front-end strictly verifies the validity of the input data, which makes the generated seeds *over-constrained*. In other words, the legitimacy check of the input data only occurs on the front-end rather than the back-end. Nevertheless, the back-end incorrectly assumes that all external data has been sanitized without checking for legitimacy. Experienced attackers can bypass the front-end inspection and send malicious data to the back-end to directly exploit the vulnerability. In addition, seeds generated based on the front-end lack the semantic information of the back-end code. Hence, it is difficult to satisfy the network data processing logic and trigger deep paths in the back-end. Moreover, crawling communication interfaces in the front-end may miss non-visible interfaces, which only appear on the back-end. In summary, the front-end-based fuzzing approaches limit the efficiency and scope of vulnerability discovery.

Our key observation is that the semantics of the back-end code can assist in generating seeds with the valid format and constraint information for fuzzing without regard to the front-end. Nevertheless, extracting back-end semantic information to guide fuzzing is not a trivial task. We face the following challenges: i) The potential communication interface needs to be correctly identified, which is the primary condition that test cases are not discarded. ii) The back-end code strictly checks the data field composition and content of different communication interfaces, which is a crucial factor affecting the scope of code testing.

Our solution In this paper, we propose an automatic and efficient fuzzing framework UCRF (U̲nder-c̲onstrainted r̲outer f̲uzzer) based on static analysis of router firmware. UCRF identifies all communication interfaces in the back-end according to their characteristics, including the non-visible front-end interfaces. Then, we extract three types of constraint of all data fields using data-flow analysis on each interface. The constraints are validated in the back-end code for data in specific fields, and the data field that satisfies the constraints can pass conditional checks and trigger deeper code protected by checks. Finally, according to the constraint information, UCRF assembles well-structured seeds and mutates them in meaningful test spaces to generate test cases.

We implemented a prototype of our solution UCRF and ran it in 10 popular routers from 4 vendors. UCRF found 41 0-day (i.e., zero-day) vulnerabilities and were assigned 38 CVE IDs, 2 CNVD IDs, and 1 PSV ID. UCRF found significantly more vulnerabilities of memory corruptions and command injection than the state-of-the-art work SRFuzzer on the five routers we had. Furthermore, we evaluated the interface identification and constraint information. The experimental results show that UCRF can identify the back-end communication interface with an accuracy of 96.3%. Regarding constraint information, UCRF can find constraints for 36.4% of the parameter keywords on average (fixed value or numeric range type for conditional computation). Note that 20 vulnerabilities can only be triggered if the constraint is satisfied.

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

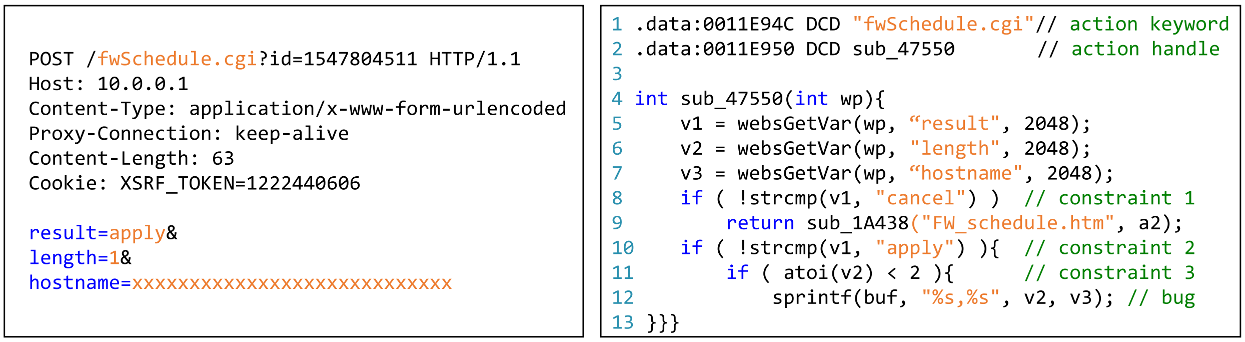
- We propose a novel approach to generating effective test cases for fuzzing routers. We first generate well-structured seeds via static analysis on the back-end binary without analyzing the front-end. Further, three types of constraints on data fields were extracted to guide seed mutation only in meaningful test spaces for efficient test space exploration.
- We implemented the prototype system UCRF, an automated fuzzing framework for SOHO routers to discover back-end vulnerabilities. UCRF identifies all potential communication interfaces at the back-end to avoid missing invisible front-end interfaces and generates interface-specific seeds. Then, UCRF utilizes lightweight data-flow analysis to extract constraints for data fields in the back-end binary with low overhead.
- We evaluated UCRF on 10 SOHO routers from 4 vendors. A total of 41 0-day vulnerabilities were found, 20 of these vulnerabilities can only be triggered if the extracted constraints are satisfied, demonstrating that seeds generated based on back-end semantics are effective for vulnerability discovery.

Roadmap In Section 2, we provide background about the SOHO router and illustrate the limitations of the current fuzzer by an example. Section 3 gives an overview of UCRF and describes the detailed design. In Section 4, we describe the implementation of UCRF, and we evaluate UCRF using real-world devices in Section 5. In Section 6, we discuss the limitations of UCRF and propose future work. We introduce the research work relevant to this paper in Section 7, and finally, conclude the paper in In Section 8.

2. Background and motivation

2.1. SOHO router system

The SOHO router is widely used to provide network services for various devices. In order to properly connect to the Internet and enable network services, the SOHO router usually provides an interactive interface for end users via a web server. The web server typically contains two components: *front-end* and *back-end*, both of which are packed in the firmware. The front-end is the web page used for user interaction, including HTML, Javascript, and other files. Data entered by the user at the front-end is sent to the back-end to handle by HTTP. In the back-end, specific binaries (e.g., httpd) receive and parse user requests from the network and then accomplish pre-defined tasks depending on the request. We refer to this binary as *border binary* (Redini et al., 2020). In this way, data controlled by the attacker is introduced into the firmware. The back-end may be assumed that all received data is trustworthy after the front-end validation. However, the attacker can bypass such validation and send malicious data directly to crash the back-end. Eventually, the misuse of external network data leads to security vulnerability.



```
POST /fwSchedule.cgi?id=1547804511 HTTP/1.1
Host: 10.0.0.1
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: keep-alive
Content-Length: 63
Cookie: XSRF_TOKEN=1222440606

result=apply&
length=1&
hostname=xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

(a) HTTP requests sent from the front-end to the back-end.

```
1  .data:0011E94C DCD "fwSchedule.cgi"// action keyword
2  .data:0011E950 DCD sub_47550       // action handle
3
4  int sub_47550(int wp){
5      v1 = websGetVar(wp, “result", 2048);
6      v2 = websGetVar(wp, "length", 2048);
7      v3 = websGetVar(wp, “hostname", 2048);
8      if ( !strcmp(v1, "cancel") )  // constraint 1
9          return sub_1A438("FW_schedule.htm", a2);
10     if ( !strcmp(v1, "apply") ){  // constraint 2
11         if ( atoi(v2) < 2 ){      // constraint 3
12             sprintf(buf, "%s,%s", v2, v3); // bug
13 }}}
```

(b) Pseudo-code snippets in the back-end web server.

Fig. 1. Motivation example.

To summon specific functions in border binary for data handling, a request sent by the front-end to the back-end includes an action keyword and possibly multiple parameter keywords. The *action keyword* is used to trigger different handle functions that a series of callback functions are registered in the border binary according to the action keyword. The *parameter keyword* is a key used to identify the data field value in a request. This data transmission pattern is also known as the KEY-VALUE model. The front-end and back-end describe the same data field using shared keywords (Chen et al., 2021). The border binary uses the key-value function (Cheng et al., 2021) (also known as input entry (Chen et al., 2021)) to obtain structured data of the KEY-VALUE model, to get the data according to the given key.

2.2. Motivating example

Figure 1 (b) shows an example, this is a buffer overflow vulnerability in Netgear R8500. To illustrate the problem clearly, we simplified the code, and the reality is far more complex. The function *websGetVar* is used to get the corresponding value by key in the network data according to the KEY-VALUE model. In line 12, the bug can be triggered if the length of *hostname* in the network data is longer than the buffer length. In order to trigger this vulnerability, the attacker-controlled network data needs to meet several conditions: (1) At line 8 and line 10, *result* needs to be equal to *apply* and not equal to *cancel*. (2) At line 21, the range of the number *length* should be less than *2*.

Current SOHO router fuzzing methods (Srivastava et al., 2019; Zhang et al., 2019) generate test cases based on the front-end is challenging to satisfy the above constraints. Due to lacking the back-end code semantics, they struggle to bypass such conditions through random mutation (Li et al., 2017; Rawat et al., 2017). Additionally, the front-end strictly checks the legitimacy of input data to prevent malicious data from being sent to the back-end. These tools have difficulty generating the correct values to pass front-end checks or need to introduce much manual work. Moreover, some data validations occur only on the front-end and not on the back-end, which means that seeds generated based on the front-end may be over-constrained, limiting the scope of back-end testing. At last, crawling action keywords in the front-end may miss non-visible interfaces, and it remains challenging to identify the corresponding handle function in the back-end.

We observed that the semantics of the back-end code could assist in generating seeds with valid format and constraint information for fuzzing. To bridge the gaps above, on the one hand, we can identify all interfaces and their corresponding data fields in the back-end for generating valid format seeds. On the other hand, extracting the data validation for a specific field help generate test cases that have the potential to prospect deeper paths and narrow the mutation space. For example, we can use the identified action keyword *fwSchedule.cgi* and corresponding parameter keywords to generate a test case shown in Fig. 1(a). In the test case, the valid range of *result* is *cancel* and *apply*, and the valid range of *length* is less than *2*, which significantly improves the fuzzing efficiency.

3. Detailed design

We pioneer UCRF, which guides SOHO router fuzzing by statically analyzing firmware to assemble well-structured and under-constrained seeds. As shown in Fig. 2, UCRF consists of three core modules: (1) Action handle identification, (2) Constraint collection, and (3) Constraint-based Fuzzing. Firmware is preprocessed to obtain filesystem and border binaries before analysis. The preprocessing module is implemented based on current research work (see Section 4).

- **Action handle identification.** UCRF first identifies all communication interfaces in the back-end, i.e., action handles, with the action keyword used to trigger them, which facilitates the generation of interface-specific seeds for each action handle in subsequent analysis (Section 3.1).
- **Constraint collection.** For each action handle, UCRF extracts constraint information of all data fields using lightweight data-flow analysis. We observe that the firmware mainly includes three types of constraints for data validation, including `strcmp-like`, `number-like`, and `network-like` (Section 3.2).
- **Constraint-based fuzzing.** UCRF assembles validly formatted seeds based on identified action and parameter keywords. Then, test cases, efficient and in-depth, are mutated under the constraints of parameter keywords. In the meantime, UCRF monitors the device's operational status to detect vulnerabilities (Section 3.3).

3.1. Action handle identification

The action handle is a series of functions invoked by different action keywords in the back-end, which handles the request sent by the front-end. Identification of the action handle determines all communication interfaces between the front and back-end interactions. Crawling the front-end can find action keywords (Srivastava et al., 2019; Zhang et al., 2019). However, if action keywords in the back-end do not appear in the front-end, some non-visible interfaces in the back-end may be missed. Listing 1 show an example of non-visible interface in Tenda G1. In line 8, the action keyword *telnet* is used to register an action handle named *TendaTelnet*, and this action handle is used to start a shell that allows the attacker to execute arbitrary commands. Nevertheless, the action keyword

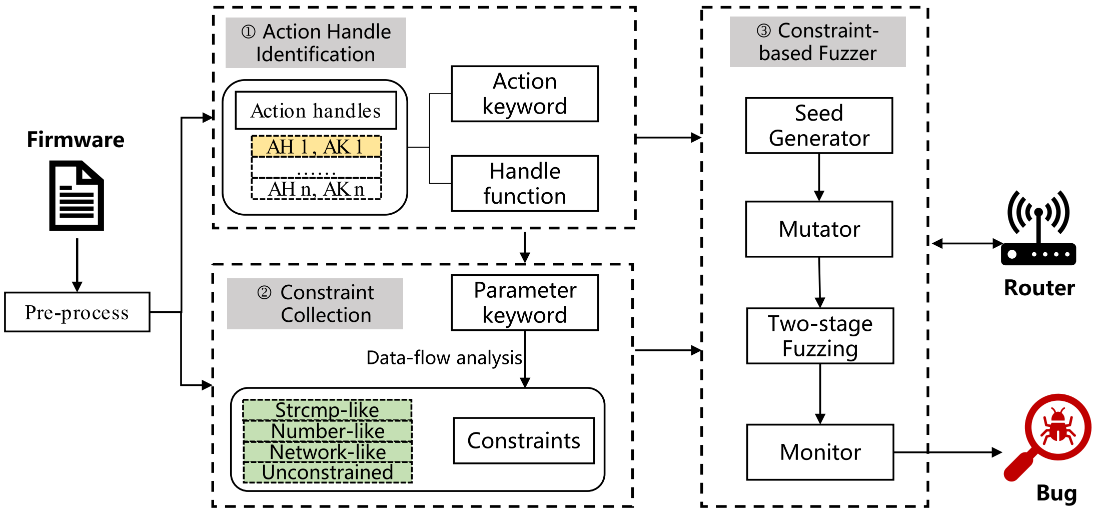


Fig. 2. Overview of UCRF.

```
void TendaTelnet(){
    system("killall -9 telnetd");
    system("telnetd &"); // enabling telnetd
        services
}
void formDefineTendDa(){
    websDefineAction("webInit", formWebInit);
    websDefineAction("setNetwork", formSetNetwork);
    websDefineAction("telnet", TendaTelnet); // non-
        visible interface
}
```

Listing 1. Non-visible interface in Tenda G1.

does not appear in the front-end. In addition, it is still difficult to identify the corresponding action handle in the back-end after crawling the action keyword in the front-end.

We observe that the action handle is usually registered as a series of callback functions in the back-end. There are mainly two methods for registering callback functions through the action keyword.

- **Model 1.** As shown in Listing 1 lines 6 to 8, the back-end uses a unified registration function to register the action handle using the action keyword.
- **Model 2.** As shown in Fig. 1(b) lines 1 to 2, the action keyword and action handle are stored in the data segment consecutively in border binary and be invoked indirectly by different offsets in function call table.

Our insight is that action handles could be identified by analyzing *the data reference of function pointers* in the back-end. A data reference to a function pointer indicates that an action handle was called indirectly. All action handles can be identified in the back-end because the action keyword invokes an action handle with a unique constant string. To further filter false positives, we select the function table that calls the most key-value functions from a list of multiple candidates. The key-value functions are used to obtain network data, such as *websGetVar*.

Algorithm 1 lists the pseudo-code of our approach. We first identify all functions in the border binary as input for the algorithm. Then, we analyze each of the two cases separately, depending on the data reference of each function pointer (lines 4–6). For Model 1, the data reference of the function pointer belongs to the text segment, which means the function pointer is used as a function parameter. For each reference location, we first find all its

Algorithm 1: Action handle identification.

```
   Input  : A list of FUNCIONTS, each item represents a function
            address
   Output: A set of tuple AH, each tuple contains keyword and
            func_ea
1  Function Main(FUNCIONTS):
2      func_set ← ∅;
3      for func_ea ∈ FUNCIONTS do
4          data_refs ← DataRefsTo(func_ea);
5          for ref_addr ∈ data_refs do
6              func_set ← GetActionKeywordHandle(ref_addr);
7      function_tables ← FunctionClustering(func_set);
8      AH ← FilterByKeyValueFunc(func_set, func_tables);
9      return AH;
10 Function GetActionKeywordHandle(ref_addr):
11     if ref_addr ∈ text segment then
12         caller ← FindCaller(func_ea);
13         if caller ≠ ∅ then
14             context ← GetCallsiteContext(caller, func_ea);
15             action_key ← GetParams(context);
16             if action_key is constant string then
17                 candidate_func ← (action_key, func_ea);
18     if ref_addr ∈ {data segment ∪ load segment} then
19         action_key ← SearchAdjacentString(ref_addr);
20         if action_key ≠ ∅ then
21             candidate_func ← (action_key, func_ea);
22     return candidate_func;
```

callers and call sites (lines 14–16). Then we find if one of the arguments at the call sites is a constant string representing the action keyword (line 17). Finally, we considered the real registration function to have the highest call frequency and collected the action keyword and action handle corresponding to it (lines 9–10). For Model 2, the data reference belongs to the data segment or load segment. We search for a constant string near the reference address of the function pointer as a candidate for the action keyword (line 24). The approach may find other function call tables that are not used as communication interfaces. We consider that in the correct function call table, some functions will use the key-value function to obtain the network data. We use this as a feature

Table 1
Constraint functions. In this table, we list the constraint type (Constraint type) and its corresponding library function name (Library function).

Constraint type	Library function
Strcmp-like	strcmp, strcasecmp, stricmp, strncasecmp, strncmp
Number-like	atoi, atof, atol,atoll, strtol, strtoll, strtoul, strtod
Network-like	inet_addr, inet_aton, inet_ntoa, inet_ntop, inet_pton, gethostbyname, gethostbyname2

filter to get the real function table and collect the action keyword and the action handle (lines 9–10).

3.2. Constraint collection

As shown in our motivation example (Fig. 1), some parameter keywords are used for data validation in the back-end. Only after checks are passed can the execution trigger deeper code paths. We refer to the validation conditions that the parameter keyword needs to meet as a *constraint*. UCRF first identifies all the parameter keywords in the action handle, then performs data-flow analysis to collect the constraints corresponding to the parameter keyword.

Parameter keyword The parameter keyword labels the corresponding data in the request according to the KEY-VALUE model. Previous work (Cheng et al., 2021) observed that the key-value function in the back-end is explicitly used for parsing structured data such as KEY-VALUE models. The key-value function can be located in the back-end based on the five characteristics of the key-value function. Naturally, the keyword used to obtain data in the key-value function is the parameter keyword. However, in the actual analysis, we find that some keywords are dynamically generated before use, such as using formatting string. To find all parameter keywords, we start backward data-flow analysis with the key-value function until we find a constant string representing the parameter keyword. For some keywords formatted with "%d", we conservatively assume that they correspond to only a single parameter keyword.

Constraint The constraint is a conditional comparison judgment for parameter keywords in the back-end, where the corresponding code path is activated only when the constraint is satisfied. Collecting parameter keywords' constraints helps generate under-constrained seeds to reach deeper code and mutate only in meaningful test spaces. We summarize the cases of the three main types of constraints in the back-end.

- **Strcmp-like.** Fixed string for conditional comparison, obtained by a `strcmp-like` constraint function.
- **Number-like.** Number range for conditional comparison, obtained by a `number-like` constraint function.
- **Network-like.** Determine if a string is a legitimate network address, obtained by a `network-like` constraint function.

It is considering that the embedded system's firmware is implemented in memory-unsafe *low-level language* (Szekeres et al., 2013) such as C/C++. The conditional comparison code heavily uses library functions provided by the C standard library or functions based on library wrappers. We can obtain the constraints by analyzing the semantics of these library functions. We use three types of constraint functions (as shown in Table 1) to collect constraints of the parameter keyword on the data-flow: (1) `strcmp-like` function; (2) `number-like` function; (3) `network-like` function. Moreover, we consider a parameter keyword to be *unconstrained* if it does not access any constraint function in its data-flow, i.e., the parameter keyword is not found to be used for conditional comparison in the analysis. The unconstrained parameter keyword is commonly used to transfer end-user-defined configuration data, which can cause potential vulnerabilities if there is a lack of sanitization in the back-end. Finally, some firmware uses vendor-defined functions semantically equivalent to those in the C standard library, and some functions cannot be located in the stripped binary based on names. UCRF can be easily extended for such firmware as discussed in Section 6.

Code traversal UCRF traces the data-flow starting from the location where the key-value function takes out the parameter keyword's value and collects the constraint function's additional constraints. We analyze the entire action handle function and its callee functions until the end. We use the data-flow analysis method from previous work (Cheng et al., 2021), which is a fine-grained data-flow analysis method based on VEX IR (Nethercote and Seward, 2007). To enable cross-process communication, environment variables or non-volatile random-access memory (NVRAM) are used in the back-end for data transfer. Unique keys are used to read and write the same data. Such communication also exists in the border binary and interrupts the data-flow analysis (For an example, see Section 5.1). We refer to functions that write data based on environment variables or NVRAM as the *setter* and the corresponding functions that read data as the *getter*. UCRF records the keys used to write data to the setter and continues the data-flow analysis from the getter used to read data from the same key, keeping the data-flow analysis going without interruption.

Algorithm 2 demonstrates the code traversal strategy of UCRF. Firstly, we obtain all nodes in the CFG with an in-degree of 0 as

Algorithm 2: Code traversal.

```
Input  : Control-flow graph ($CFG$) of the border binary,
         identified key-value function $kv\_func$,
         communication paradigms $setter$ and $getter$
Output: Constraint list $cons\_list$
1  Function DataflowAnalysis($func$):
2      while $dataflow$ is not until the end do
3          if $dataflow$ reach to $setter$ then
4              $pk \leftarrow$ GetParameterKey($kv\_func$);
5              $set\_key \leftarrow$ GetEnvKey($setter$);
6              add ($pk$, $set\_key$) to $nvram\_dict$;
7          if $dataflow$ reach to $constraint\_func$ then
8              $cons \leftarrow$ GetConstraint($constraint\_func$);
9              add ($constraint\_func$, $cons$) to $cons\_list$;
10 $nvram\_dict \leftarrow \emptyset$;
11 $start\_nodes \leftarrow$ GetStartNodes($CFG$);
12 $call\_sequences \leftarrow$ GetAllSuccessors($start\_nodes$);
13 for $call\_chain$ in $call\_sequences$ do
14     for $func$ in $call\_chain$ do
15         if $func$ is $kv\_func$ then
16             $dataflow \leftarrow$ DataflowAnalysis($kv\_func$);
17 for $call\_chain$ in $call\_sequences$ do
18     for $func$ in $call\_chain$ do
19         if $func$ is $getter$ then
20             $get\_key \leftarrow$ GetEnvKey($getter$);
21             if $get\_key$ in $nvram\_dict$ then
22                 $dataflow \leftarrow$ DataflowAnalysis($getter$);
```

a start node for our analysis (line 11). These nodes represent the start point of a call chain, and for a start node, we obtain its subsequent nodes as a call chain (line 12). We determine whether the key-value function is located within each call chain. If so, we initialize the data-flow analysis from where the key-value function starts (line 16). Otherwise, we do not analyze this call chain, considering these paths irrelevant for handling network data. During data-flow analysis, we record the parameter keyword and the unique key used to write the data when the *setter* is accessed (lines 3–6). In the meantime, when the data-flow reaches the constraint function, we record this function and the corresponding constraint conditions as constraints for the corresponding parameter keyword (lines 7–9). At the end of the first data-flow analysis, we determine whether any network data has been written to the setter (lines 17–21). If so, we initialize the data-flow analysis from the location of the *getter* using the same key and continue to collect constraints for the corresponding parameter keywords (line 22).

3.3. Constraint-based fuzzing

UCRF generates well-structured and under-constrained test cases and directly fuzzes the router's back-end. First, UCRF uses the previously identified action keyword and parameter keyword to assemble a valid formatted seed. After that, UCRF mutates the seed only in meaningful test spaces guided by constraint information for efficient fuzzing. Finally, UCRF monitors the status of the remote device to determine if a crash occurred after the test cases are sent.

Seed generation UCRF generates seeds for fuzzing routers based on the results of the back-end analysis. Since UCRF does not perform a complete analysis from the start of the HTTP request received by the back-end binary, we cannot be aware of how the complete HTTP request is processed. In addition, different vendors have different customized implementations of the firmware. For the above reason, UCRF first determines which format (e.g., JSON and SOAP, both following the KEY-VALUE model) is used to transmit data in the HTTP by observing the requests sent by the front-end. Then three types of information are required to generate a seed: URL for request, parameter keyword, and session for authentication. UCRF uses the action keyword as a URL to invoke an action handle in the back-end. In some cases routers use URLs with specific prefixes, whereas UCRF relies on the analyst to specify the URL prefix that should be used. Fortunately, our analysis found that URLs for small embedded devices (such as SOHO routers) often follow the same pattern. Analysts can quickly spot the same pattern in multiple URLs as URL prefixes. For example, the full URL for invoking the interface *telnet* in Listing 1 is "/goform/telnet", with "/goform/" being the prefix for all action keywords. Afterward, UCRF uses the collected parameter keywords for each action handle to generate the interface-specific seed. Finally, after the analyst performs the login operation, UCRF automatically obtains the access credentials and assembles well-structured and authenticated seeds.

Constraint-based mutator To enable constraint-based fuzzing, UCRF determined whether to mutate the value of the parameter keywords based on their constraints. The constraint represents a range of candidate values for a particular parameter keyword. Constraint information can be used to narrow the mutation space and help generate test cases that have the potential to prospect deeper paths. Moreover, for constraints `number-like` and `network-like`, it means the data has been sanitized, and no additional mutation is necessary. Specifically, for each type of constraint, the mutation strategy is as follows:

- **For strcmp-like data.** We select a candidate value from the constraint list corresponding to the parameter keyword for the mutations.
- **For number-like data.** The collected constraint and inverse values are used to assemble the seed.
- **For network-like data.** Random value selection from predefined network-like data to assemble the seed, such as "192.168.0.1/24" and "255.255.254.0".
- **For unconstrained data.** Randomly mutates them using the strategy described subsequently.

UCRF focuses on finding memory-corruption and command injection vulnerabilities, which can be easily extended, as discussed in Section 6. To mutate the values of the parameter keyword, we use the following strategies:

- **String length.** To trigger a memory-corruption vulnerability, such as buffer overflows, we use a random string and modify its length as a mutation.
- **Numerical values.** We use random integers as mutations. In particular, we prioritize specific numbers (e.g., 0 and 1) to bypass possible checks.
- **Crafted command.** To trigger a command injection vulnerability, we randomly select well-constructed payloads from a predefined database as mutation, which can bypass common character restrictions and trigger our proxy-based monitor.

Two-stage fuzzing We observe that the action handles in the back-end are divided into two categories. The *receiver* is used to get data from the front-end and handle it (e.g., POST request), and the *returners* is used to respond to front-end requests and return data (e.g., GET request). To test all action handles as correctly as possible, we send a returner after a receiver and repeat the loop. We use a simple heuristic strategy to distinguish between receivers and returners. Of all the action handles identified, the one that used the key-value function to obtain network data is the receiver, while the remaining action handles are considered to be the returners. We treat requests for web-related files in the firmware as returners as well. As a fuzzing method, UCRF populates the HTTP request method using a mutation (e.g., a predefined GET or POST type) in cases where the HTTP request type cannot be determined.

Monitor UCRF monitors the device's status to detect crashes caused by test cases during the fuzzing process and generates an alert if the monitor detects an abnormal state. We have implemented two monitoring methods to monitor the device's status: (i) Network-based monitor. If the device does not respond or respond differently than before, it is considered that a memory crash has occurred. (ii) Proxy-based monitor, if our local proxy server receives the specially crafted request sent by the test router, it is considered that a command execution has occurred. UCRF will physically restart the router when the monitor finds that the device does not respond after a time and continues the fuzzing process after the router has recovered. Since UCRF tests the physical device directly and does not rely on firmware emulation, we cannot directly obtain feedback on error messages (e.g., segment faults). We use both of the above heuristic monitors to detect bugs. For more discussion of fault observability see Section 6.

4. Implementation

The design of UCRF consists of four steps: firmware preprocessing, action handle identification, constraint collection, and constraint-based fuzzing. We implement UCRF as an automated fuzzing framework.

Firmware pre-processing We downloaded the firmware for all the devices from the vendor's website and then utilized Binwalk (2014) to unpack the firmware image. To determine which of the many binary files is used for network communication, i.e., border binary, we use SaTc (Chen et al., 2021) for discovering border binary as a starting point for subsequent analysis. Furthermore, we utilize IDA Pro (Ida, 2022) to identify functions and build a control flow graph (CFG).

Action handle identification As described in Section 3.1 Model 2, the presence of indirect calls in the back-end binary prevents some functions from being correctly identified. To correctly iden-

Table 2
Dataset.

ID	Vendor	Product	Type	Firmware version	Border binary	Architecture	Service
1	Netgear	R8500	Router	V1.0.2.116, V1.0.2.158	httpd	ARM32(LE)	HTTP
2	Netgear	R7800	Router	V1.0.2.46	net-cgi	ARM32(LE)	HTTP
3	Netgear	WNDR4500v3	Router	V1.0.0.50	net-cgi	MIPS32(BE)	HTTP
4	Tenda	G3	Gateway	V15.11.0.5, V15.11.0.17	httpd	ARM32(LE)	HTTP
5	Tenda	AC9	Router	V15.03.05.19	httpd	ARM32(LE)	HTTP
6	Tenda	AX3	Router	V16.03.12.10	httpd	ARM32(LE)	HTTP
7	D-LINK	DIR-882	Router	FW1.30B06_Hotfix_02	prog.cgi	MIPS32(LE)	HNAP
8	D-LINK	DIR-823-pro	Router	V1.0.2	prog.cgi	MIPS32(LE)	HNAP
9	TOTOLINK	A720R	Router	V4.1.5cu.470_B20200911	cstecgi.cgi	MIPS32(LE)	HTTP
10	TOTOLINK	X5000R	Router	V9.1.0u.6118_B20201102	cstecgi.cgi	MIPS32(LE)	HTTP

tify all functions, we scan the entire data segment using the memory alignment length (4 bytes or 8 bytes, depending on the CPU architecture length) as an offset to determine if the data points to an executable segment address. If so, we treat it as a function and add the function and its successor node to the CFG. Finally, we identify the action handles by analyzing data references as described in Algorithm 2, where we use the API provided by IDA to collect data reference information for functions.

Constraint collection We start the data-flow analysis at a key-value function and collect constraints corresponding to a parameter keyword. During the data-flow analysis, if a pre-defined constraint function (see Table 1) is reached, we record the name of the constraint function and the corresponding comparison condition as a constraint for the corresponding parameter keyword. The VEX IR-based data-flow analysis method and key-value function identification method are built atop previous work (Cheng et al., 2021). Based on the VEX intermediate representation, we can convert the native binary code of multiple architectures into a unified intermediate representation (IR), based on which cross-architectural binary analysis can be achieved. We use PyVex (2022) for intermediate language translation in Python.

Constraint-based fuzzing UCRF performs in-depth testing of the router's back-end binary (i.e., the web server) with access credentials. UCRF first launched a browser using Selenium (2022) to allow the analyst to perform a login operation to obtain access credentials. In the meantime, UCRF captures the login request to determine which format is used to transmit the data in HTTP. Currently, UCRF supports the common HTTP messaging formats used in embedded devices, including HTTP, JSON, Simple Object Access Protocol (SOAP), and Home Network Administration Protocol (HNAP), which all follow the KEY-VALUE model. We used the Boofuzz (2014) framework and modified it to support the seed generation, mutator, and monitor. To enable the mutated payload to bypass simple sanitization (e.g., character filtering) in the back-end, we enrich the boofuzz mutator module with a pre-collected attack payload (Payloadsallthethings, 2018). During the fuzzing process, when the router does not respond, the monitor module will physically reboot the router to force reset its state. The monitor module reboots the router by sending control commands to a smart plug to power it down and then up again. We used a Mi Smart Plug (2022) and exposed it to Python by using Python-miio (2022).

5. Evaluation

We evaluate UCRF to answer the following research questions:

- **RQ1**: Can UCRF find real-world vulnerabilities? Are the seeds generated by UCRF helpful for discovering vulnerabilities? (Section 5.1)
- **RQ2**: How many action handles can UCRF accurately identify in the back-end? (Section 5.2)

Table 3
Confirmed 0-day vulnerability. For discovered vulnerabilities, we show the vulnerability types, including buffer overflow (MEM), command injection (CI), and the total number of 0-day vulnerabilities (SUM). For each vulnerability, we classify it as being triggered by only a single keyword (Single Key Vul) or requiring at least two keywords, one of them being constrained (Multi-Key Vul).

Product	MEM	CI	SUM	Single key vul.	Multi-key vul.
R8500	0	5	5	2	3
G3	10	2	12	3	9
AX3	13	0	13	9	4
DIR-882	0	2	2	1	1
DIR-823-pro	0	4	4	4	0
A720R	0	2	2	1	1
X5000R	1	2	3	1	2
Total	24	17	41	21	20

- **RQ3**: How many constraints can UCRF collect? How do the different types of constraints help in finding vulnerabilities? (Section 5.3)

Dataset To evaluate UCRF, we selected some popular devices from different vendors for evaluation, including 10 router devices from 4 vendors (Table 2), these devices are widely used in previous research work (Chen et al., 2021; Redini et al., 2020; Zhang et al., 2019). In the experiments, we test against the physical router since UCRF does not rely on firmware simulation to obtain runtime information to guide fuzzing.

Experiment setup We purchased these router devices (see Fig. 4) and connected them within the subnet. In order to obtain the correct session for fuzzing, we manually set the password for the router and specified the authentication method. At last, we deployed UCRF on a 40-core, 256 GB RAM machine running Ubuntu 20.04.

5.1. Vulnerability identification

We evaluated UCRF using 10 devices with the latest version of the firmware. As shown in Table 3, UCRF found a total of 41 0-day vulnerabilities on 7 devices. We responsibly reported these vulnerabilities to the vendors, and they were confirmed. Of these, 24 vulnerabilities were buffer overflow vulnerabilities, and the other 17 were command injection vulnerabilities. These vulnerabilities can crash devices to enable DDoS attacks or remote command execution. Overall, these vulnerabilities were assigned a total of 38 CVE IDs, 2 CNVD IDs, and 1 PSV ID, all assigned id shown in Table 7.

To explore the effectiveness of seeds generated by UCRF based on constraints, we manually reverse engineering the firmware according to vulnerability alerts and analyze the conditions for vulnerability triggering. The last two columns in Table 3 illustrate that in order to trigger these vulnerabilities, concurrent mutations to one or more parameter keywords are necessary. Of these, 20 vulnerabilities were triggered when two or more parameter keyword

Table 4
Comparison with SRFuzzer and ESRFuzzer. For each device, we report the vulnerabilities number (SUM), including buffer overflow (MEM) and command injection (CI). For ESRFuzzer, we report newly discovered vulnerabilities in D-CONF mode in brackets compared to SRFuzzer.

	UCRF			SRFuzzer			ESRFuzzer		
Product	MEM	CI	SUM	MEM	CI	SUM	MEM	CI	SUM
R8500	14	8	22 †	9	0	9	11(2)	0	11
R7800	16	2	18	10	8	18	10	11(3)	21
WNDR4500v3	13	3	16	13	2	15	13	2	15
G3	20	2	22 ‡	5	0	5	9(4)	0	9
AC9	25	6	31	11	0	11	15(4)	1(1)	16
SUM	88	21	109	48	10	58	58	14	72

† 2 CVE IDs, 2 CNVD IDs and 1 PSV ID were assigned. ‡ 10 CVE IDs were assigned.

values were mutated in one test case, and at least one of the values was based on the constraint information. The other 21 vulnerabilities were triggered when only one value was mutated, and no constraint information was needed. Note that UCRF can help discover vulnerabilities by limiting the mutation space of constrained fields to improve fuzzing efficiency.

Comparison SRFuzzer (Zhang et al., 2019) and ESRFuzzer (Zhang et al., 2021) are state-of-the-art SOHO router fuzzing tools. SRFuzzer (Zhang et al., 2019) generates the test case by simulating the browser's submission behavior and captures the request sent by the front-end as the seed. SRFuzzer repeatedly fills the web page ten times and analyses the requests to label attributes of variables, including fixed string, number, and variable string. SRFuzzer considers fixed strings and numbers as invariants that can be used to pass conditional checks in the back-end. However, the strict input checking of the back-end code makes fuzzing inefficient. UCRF collects constraints in the back-end to obtain more accurate semantic information for reaching deep paths. ESRFuzzer (Zhang et al., 2021) enhances SRFuzzer with D-CONF mode, which leverages NVRAM configuration operations provided by partial routers, and starts fuzzing from the position where data is read from NVRAM key-value pairs in the back-end. A single NVRAM key-value pair represents a parameter keyword and its corresponding value. D-CONF mode can help ESRFuzzer to test the code more deeply under the premise of ignoring part of the back-end data validation.

To compare our approach to SRFuzzer and ESRFuzzer, we contacted the authors but failed to obtain their tool. We selected 5 out of 10 experimental devices from the datasets of SRFuzzer and ESRFuzzer, and performed experiments on the same firmware version. Devices were not selected for evaluation for the following reasons: (1) For Mercury Mer450 and ASUS RT-AC1200, the vendor has taken them off the shelves, and we cannot obtain them. (2) For Netgear Orbi and Netgear Insight, the total price high than 600 USD, which is expensive for us. (3) For TP-Link TL-WVR900G, the device's back-end web server is programmed in Lua, and UCRF is currently only available for memory-unsafe low-level programming languages (e.g., C/C++), with the aim of discovering memory-corruption and command injection vulnerabilities within them. We ran UCRF for 40 h, the same time described in the SRFuzzer and ESRFuzzer papers. Since some of the vulnerability details are not publicly available, we illustrate the vulnerability discovery capabilities of UCRF by comparing the number of discovered vulnerabilities.

As shown in Table 4, UCRF found a total of 109 vulnerabilities, while SRFuzzer found 58 vulnerabilities and ESRFuzzer found 72. Among them, compared with SRFuzzer and ESRFuzzer, UCRF found more vulnerabilities on most devices, except R7800, where UCRF found fewer command execution vulnerabilities. Our further exploration suggests that this may have been due to the internal protection mechanisms of the device causing UCRF to not detect the vulnerability, and we discuss this situation in Section 6. We reported these vulnerabilities to the vendor; fortunately, most of them have been fixed in the newer firmware versions. Among them, the vulnerabilities found in R8500 and G3 can be verified in the latest firmware version and confirmed by the vendor as 0-day vulnerabilities. In total, 12 CVE IDs, 2 CNVD IDs, and 1 PSV ID are assigned. SRFuzzer and ESRFuzzer supports multi-type vulnerability discovery, including cross-site scripting (XSS) and information disclosure. Currently, UCRF is designed to discover memory-unsafe back-end vulnerabilities, including memory-corruption and command injection vulnerabilities, and can be extended as discussed in Section 6.

UCRF found more vulnerabilities benefiting from being aware of the semantics of the back-end code. In contrast, SRFuzzer and ESRFuzzer are insensitive to the semantics of back-end code, allowing fuzzing to remain at a shallow level. ESRFuzzer's D-CONF mode enables independent fuzzing of multiple parameter keywords from deep in the code, but ignoring potential data validation on previous paths can lead to false positives. In addition, in the process of vulnerability reproduction, we found that generating valid requests based on the front-end requires much manual work due to complex front-end interaction logic, and the front-end performs strict checks on the data. While UCRF generates seeds based on back-end information can avoid analyzing the complex front-end.

Case study: CVE-2022-29765 Listing 2 shows the 0-day vulnerability CVE-2022-29765 found by UCRF in Tenda G3, we simplified it to make it clearer. First, UCRF locates the action keyword *setQos* and the corresponding handle function *formQOSSet* by Action handle identification and performs subsequent interface-specific analysis (line 1). In function *formQOSSet*, we can locate three parameter keywords through the key-value function *websGetVar*, including *qosPolicy, qosDefaultRuleEn* and *qosConnecttedNum* (line 4–6). UCRF constructs well-structured seeds using these action and parameter keywords and generates efficient and in-depth test cases by further analysis. Furthermore, UCRF collects field-specific constraints by initializing the data-flow analysis from the location of parameter keywords. For example, we can determine that *qosPolicy* belongs to the `strcmp-like` constraints with valid candidates "user", "auto" and "disable" (lines 15–21). Finally, stack overflow vulnerability at line 37 can be triggered by mutating the unconstrained variable *connectLimit* when *qosPolicy* equals "user" and *qosDefaultRuleEn* equals "true". In addition, UCRF's code traversal algorithm can recover data-flow analysis based on environmental variables or NVRAM reads and writes. For example, the environment variable written in line 10 via "bandwidth.dft.en" is read in line 28. UCRF can collect that the constraint corresponding to parameter keyword *qosDefaultRuleEn* is of the `number-like` and equal to 1. Moreover, UCRF's lightweight data-flow analysis enables quick analysis of deep call chains, free from path explosion.

5.2. Action handle identification

Since no oracle can highlight all communication interfaces in the back-end, we did our best to manually reverse engineering the

```
websDefineAction("setQos", formQOSSet); // action keyword is setQos

void formQOSSet(webs_t wp, ...){ // action handle is formQOSSet
    plcy = websGetVar(wp, "qosPolicy", 2048);
    en = websGetVar(wp, "qosDefaultRuleEn", 2048);
    connectLimit = websGetVar(wp, "qosConnecttedNum", 2048);
    flag = setQosPolicy(plcy);
    if ( flag == 1 ){
        tmp_en = swTobool(en);
        SetValue("bandwidth.dft.en", tmp_en); // write to NVRAM
        SetValue("bhv.dft.connectlimit",connectLimit);
        refresh_default_qos_rule();
    }
}
// The constraints for qosPolicy are "user", "auto" and "disable"
int setQosPolicy(int *policy){
    if ( !strncmp(policy, "user", 5 )) return 1;
    else if ( !strncmp(policy, "auto", 5 )) return 2;
    else if ( !strncmp(policy, "disable", 8 )) return 0;
    else return -1;
}
// The constraints for qosDefaultRuleEn are "true" and "false"
int swTobool(int *input){
    if ( !strncmp(input, "true", 5 )) return 1;
    else if ( !strncmp(input, "false", 6 )) return 0;
}
void refresh_default_qos_rule(){
    GetValue("bandwidth.dft.en", defaultEn); // read from NVRAM
    if ( atoi(defaultEn) == 1 )
        dftQosAdd();
}
void dftQosAdd(){
    dftConnectLimitListAppend();
}
void dftConnectLimitListAppend(){
    GetValue("bhv.dft.connectlimit", connectLimit);
    sprintf(ruleTmplt, "...%s", ..., connectLimit); // stack overflow
}
```

Listing 2. Pseudocode of CVE-2022-29765. Found by UCRF.

firmware to verify the result of the action handle in the back-end. We considered the characteristics of each device's action handles and tried to construct requests to trigger them as a way to identify all the action handles. We also acknowledge that our analysis cannot exclude false negatives, as analyzing real-world firmware is difficult.

Table 5 shows UCRF can find more than 90% action handles in all devices. For six devices, UCRF can discover all the action handles. We found two function call tables in the R8500, and UCRF correctly identified the one used to register the action handle. For the A720R and X5000R, only one of these function call tables exists and is correctly identified by UCRF. For devices R7800 and WNDR4500, there are more than 10 different function call tables in border binary. Our best efforts identified two of these function tables used to register action handles. For the remaining 5 devices that register action handles using Model 2 (see Section 3.1), where multiple similar registration functions exist in border binary. UCRF can precisely identify the unique function used to register action handles. In the case of AC9 and AX3, UCRF identified the function registered by the same registered function as the action handle but generated false positives. We reverse engineered the firmware and found that the false positives came from legacy code. The vendor kept the function entries but removed the corresponding function code, preventing the action handle from being triggered. The nAct column shows the non-visible front-end action handle found by UCRF. Most of them are legacy interfaces after analysis, i.e., the developers only removed them from the front-end but not the back-end. Among them, we found some interfaces that could lead to vulnerabilities, and we left the specific analysis for future work. Finally, during the actual analysis, we found that due to the vendor's custom implementation of the firmware, constructing the correct URL for 8 devices (indicated with †) relied on the analyst specifying a specific prefix before the action keyword, such as "/goform/". Fortunately, URLs in each device contain only one special prefix, and the analysts can identify this prefix based on the same pattern of multiple URLs in no more than 3 min and inform UCRF.

The previous work SaTC (Chen et al., 2021) can identify action handles at the back-end based on information from the front-end. SaTC observed that if a keyword identified in the front-end file (JavaScript/HTML Etc.) and a function pointer are passed into a function as arguments, the keyword is used as the action keyword, and the function pointer is the action handle. Due to the heterogeneity of the front-end, such as the multi-vendor code style and dynamic characteristics of JavaScript code, the method of identifying action keywords in the front-end may produce false positive results. Meanwhile, SaTC cannot identify Model 2 because no registration function is invoked. We show the results of SaTC in Table 5, and the action handle identification method of UCRF is more accurate than SaTC without excessive time spent.

5.3. Constraint

Constraint collection To evaluate the collection of constraints in the back-end by static analysis of UCRF, we ran UCRF on 10 firmware. As shown in Table 6, we can find relevant constraints for 36.41% of the parameter keywords on average, this shows

Table 5
Action Handle Identification. For each device, we report the segment to which the action handle belongs (Seg), the number of all function pointers in the back-end (fPtr), and the total action handle verified by manual (Act). For UCRF and SaTC, We report true positives (TP) and false positives (FP) in their results for action handle identification and report the analysis time in seconds (Time), % represents the proportion of true positive action handle in total action handle. Finally, (nAct) represents non-visible front-end action handles found by UCRF.

				UCRF					SaTC			
Product	Seg.	fPtr	Act	TP	FP	%	Time(s)	nAct	TP	FP	%	Time(s)
R8500	.data	497	177	177	0	100	92.3	14	156	30	88.1	2.7
R7800†	.data	953	230	210	0	91.3	131.4	4	8	48	3.5	9.9
WNDR4500v3	.data	1034	203	183	0	90.1	169.8	2	8	27	3.9	7.0
G3†	.text	474	222	222	0	100	35.0	17	200	81	90.1	15.2
AC9†	.text	427	175	158	10	90.3	35.8	56	105	21	60.0	6.3
AX3†	.text	260	115	110	14	95.7	19.1	8	100	42	87.0	7.5
DIR-882†	.text	165	154	154	0	100	12.3	10	106	542	68.8	31.0
DIR-823-pro†	.text	152	132	132	0	100	19.5	12	103	199	78.0	15.4
A720R†	.data	133	131	131	0	100	18.4	4	1	25	0.8	10.1
X5000R†	.data	139	133	133	0	100	24.7	10	1	14	0.8	10.3
Average	–	–	–	–	–	96.7	55.8	–	–	–	48.1	11.5

† The analyst needs to specify a prefix to construct correct URL.

Table 6
Constraint. For each device, we provide its vendor and product. We show the constrained parameter keyword (cKey), and all parameter keywords found in border binary (Key). % represents the proportion. For constraint information, we show the total number of constraints, and the number of constraints for `strcmp-like` (Str), `number-like` (Num), and `network-like` (Net). Finally, we show the analysis time in seconds (Time).

Vendor	Product	cKey/Key	%	Total	Str	Num	Net	Time(s)
Netgear	R8500	591/1307	45.21	765	474	236	55	115.26
Netgear	R7800	152/838	18.13	254	184	55	15	56.73
Netgear	WNDR4500v3	117/439	26.65	140	69	51	20	42.33
Tenda	G3	287/546	52.56	400	212	184	4	70.18
Tenda	AC9	134/491	27.29	186	106	77	3	74.30
Tenda	AX3	95/352	26.98	148	108	38	2	48.20
D-LINK	DIR-882	163/508	32.08	263	241	22	0	49.23
D-LINK	DIR-823-pro	72/289	24.91	122	111	11	0	44.98
TOTOLINK	A720R	392/548	71.53	415	39	330	46	24.63
TOTOLINK	X5000R	230/593	38.78	254	41	211	2	23.32
Average	–	–	36.41	–	–	–	–	54.92

Table 7
Assigned 0-day vulnerabilities..

Product	Vulnerabity ID
Netgear R8500	CVE-2022-27945, CVE-2022-27946, PSV-2022-0060, CNVD-2022-33446, CNVD-2022-33448
Tenda G3	CVE-2021-45987, CVE-2021-45989, CVE-2021-45991, CVE-2021-45992, CVE-2021-45995 CVE-2021-45996, CVE-2021-45997, CVE-2022-24171, CVE-2022-24169, CVE-2022-24172 CVE-2022-29764, CVE-2022-29765
Tenda AX3	CVE-2022-24146, CVE-2022-24143, CVE-2022-24145, CVE-2022-24142, CVE-2022-24158, CVE-2022-24163, CVE-2022-24159, CVE-2022-24156, CVE-2022-24157, CVE-2022-24152 CVE-2022-24153, CVE-2022-24149, CVE-2022-24155,
D-Link DIR-882	CVE-2021-44881, CVE-2021-45998
D-Link DIR-823-Pro	CVE-2021-46452, CVE-2021-46453, CVE-2021-46455, CVE-2021-46457
TOTOLINK A720R	CVE-2021-44247, CVE-2021-45740
TOTOLINK X5000R	CVE-2021-45733, CVE-2021-45734, CVE-2022-27005

Table 8
Merged Constraint. For each device, we provide its vendor and product. We show the unique constrained parameter keyword (ucKey) and all unique parameter keywords found in border binary (uKey). % represents the proportion. For constraint information, we show the total number of constraints (Total) and the number of constraints for `strcmp-like` (Str), `number-like` (Num), and `network-like` (Net), and the analysis time seems as Table 6.

Vendor	Product	ucKey/uKey	%	Total	Str	Num	Net
Netgear	R8500	311/703	44.23	434	271	128	35
Netgear	R7800	100/400	25	158	110	35	13
Netgear	WNDR4500v3	73/236	30.93	88	45	27	16
Tenda	G3	241/448	53.79	352	200	148	4
Tenda	AC9	97/308	31.49	139	83	55	3
Tenda	AX3	78/233	33.47	116	81	33	2
D-LINK	DIR-882	140/367	38.14	238	220	18	0
D-LINK	DIR-823-pro	41/177	23.16	77	67	10	0
TOTOLINK	A720R	237/350	67.71	273	26	225	22
TOTOLINK	X5000R	103/295	34.91	122	30	92	2
Average	–	–	38.28	–	–	–	-

our observations are proven. These constraints can help seed narrow down the mutation space and reach deeper codes protected by checks. Specifically, UCRF found 71.53% of the constrained parameter keyword for TOTOLINK A720R. We manually checked the firmware and determined that the constraints were valid, and the back-end extensively used function *atoi* to convert the network data to numbers. For Netgear R7800, UCRF found only 18.13% of the constrained parameter keyword. Our manual analysis shows constraints not present in the border binary but in other edge binary. We discuss the imperfections of UCRF in Section 6. The average time spent for firmware analysis was 54.92 s, indicating that the statical analysis of UCRF is lightweight and practical.

UCRF generates specific seeds for each interface, and parameter keywords of the same name in different action handles may have different roles and constraints. The constraints after de-duplication are shown in Table 8. For parameter keywords with the same name, the constraints are merged. The results are similar compared to the unmerged ones.

Constraint effectiveness To verify the usefulness of the different constraint types of UCRF for vulnerability discovery, we used individual constraint types to evaluate the number of crashes found by UCRF over 40 h. We used the following fuzzing schemes as benchmarks.

- **UCRF.** Enable all three constraint types.
- **UCRF-str.** Use only `strcmp-like` constraints.
- **UCRF-num** Use only `number-like` constraints.
- **UCRF-net.** Use only `network-like` constraints.
- **boofuzz.** We configure Boofuzz (2014) with the correct action keyword and parameter keyword. It is worth noting that UCRF is based on the boofuzz implementation, and with the correct

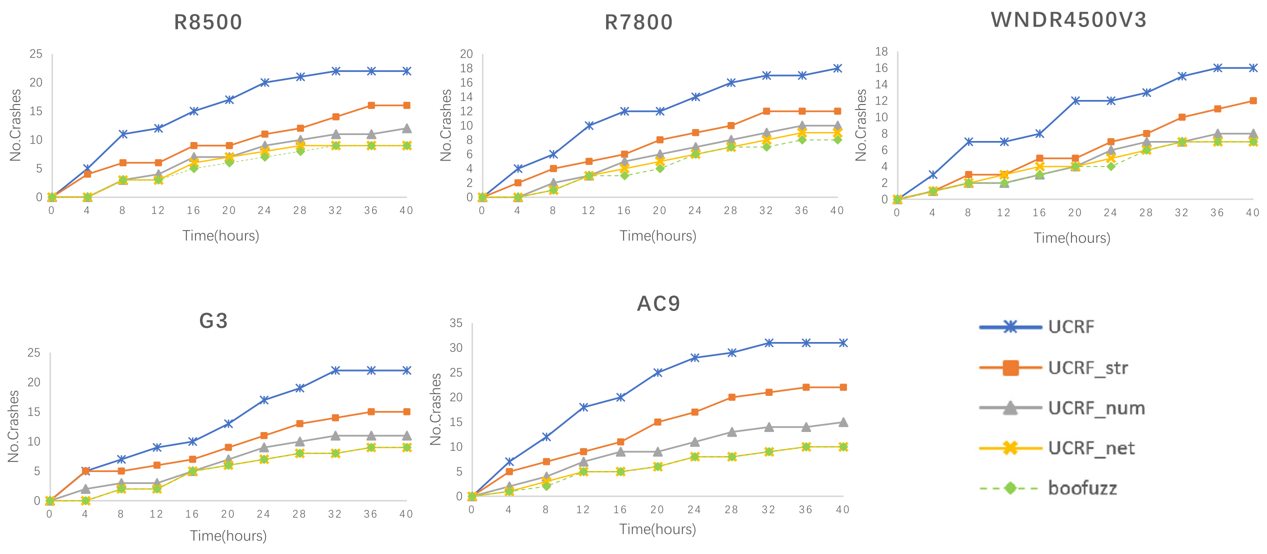


Fig. 3. The number of crashes discovered over time.

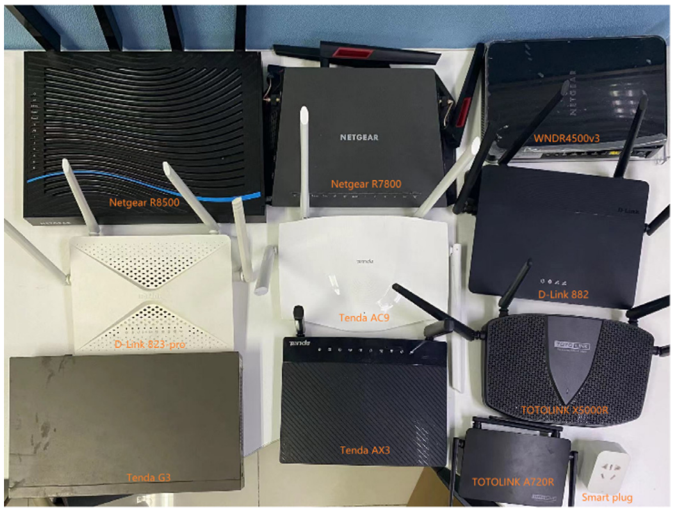


Fig. 4. Experimental devices.

configuration, we can regard boofuzz as UCRF without using constraint information.

We experimented with five devices where more vulnerabilities had previously been found, making the results more straightforward. In addition, we skipped subsequent mutations to the parameter keyword currently being mutated after a crash was detected. We manually recorded the PoCs that triggered the crashes and performed a reverse engineering analysis to ensure we got the number of unique crashes.

Figure 3 shows the efficiency of five fuzz schemes for triggering crashes. UCRF outperforms the other four schemes and, as we mentioned previously, the three constraint types in the back-end function together for the sake of accessing deeper paths. Of the other four strategies, UCRF-str has found the most vulnerabilities in a limited time. Considering the reverse engineering analysis of the firmware, we believe that it is because `strcmp-like` constraints are frequently used for earlier conditional comparisons. The `number-like` constraint used by UCRF-num can help bypass the conditional checks for number classes. However, the mutation space for `strcmp-like` data is much larger than for `number-like`. UCRF-num has difficulty generating eligible string class constraints, leaving the test stuck at a superficial level. For UCRF-net, the efficiency is similar to boofuzz, as such constraints are usually located in deep paths and are difficult to reach without information on conditional comparisons. In addition, the efficiency of the three types of schemes using only single constraint information is proportional to the number of constraints extracted (see Table 6), which indicated that the constraint information helped narrow the mutation space.

6. Discussion

Limitation of the scope UCRF is designed to test vulnerable web servers in the SOHO router efficiently. Technically, UCRF can work against web servers of any embedded device with a client-server mode. We plan to extend UCRF to support other embedded devices such as cameras, programmable logic controllers (PLC), etc. Our evaluation of 10 routers from 4 vendors demonstrates the scalability of UCRF. Currently, UCRF focuses only on analyzing Linux-based firmware, relying on symbol tables for constraint collection. Although most router firmware fits the above criteria, UCRF fails to work for some *blob* or stripped firmware. Constrained functions can be discovered using function semantic analysis (Redini et al., 2020) or binary similarity matching (Yu et al., 2020), and integrating UCRF with these methods can be a feasible solution.

Since different vendors have different customized implementations of firmware, we reserve the possibility of having a different data processing model on additional devices. For the same reason, in some cases, the action keywords we collect from the back-end are missing a specific prefix (e.g., "/goform/") compared to the actual URL used. Analysts can quickly determine the URL prefixes that should be used in UCRF based on the same pattern of multiple URLs. Although the full URL can be obtained directly by analyzing the front-end, such an approach can introduce false positives due to the diversity of front-end code styles (e.g., SaTC Chen et al., 2021, which we have shown in Table 5). Alternatively, manual work needs to be introduced to handle complex front-end interaction logic (e.g., SRFuzzer Zhang et al., 2019). In addition, parameters may be encoded (e.g., by MD5 or base64) in the request of some devices, in which case it is necessary for the analyst to analyze the request and determine the type of encoding, and then specify which way to encode the mutation value at the end of the mutation module.

Constraints' granularity UCRF performs lightweight data-flow analysis to collect constraints instead of symbolic execution (e.g., angr Shoshitaishvili et al., 2016). Because in our practical analysis that for data transferred using the KEY-VALUE model, the back-end performed coarse-grained comparisons of one or more keys (e.g., character comparisons using *strcmp*) rather than byte-grained precise validation. This allows us to ignore complex and dynamic constraints, such as comparisons of memory and function references. Our experiments demonstrate that UCRF can effectively collect constraints for many parameter keywords (see Section 5.3). On the other hand, symbolic execution faces the path explosion problem when encountering deep call stacks. In contrast, data flow analysis enables us to traverse deeper paths and collect as many constraints as possible.

Vulnerability types UCRF is currently targeted at discovering two types of back-end vulnerabilities, memory-corruption, and command injection since the collected constraints are effective for finding these two typical types of taint-style vulnerabilities. Previous work SRFuzzer (Zhang et al., 2019) additionally supports the discovery of cross-site scripting (XSS) and information disclosure vulnerabilities. To detect XSS, we can modify UCRF's mutator module to discover such vulnerabilities, and our proxy-based monitor can support detecting such vulnerabilities. For the detection of information disclosure, determining the occurrence of the vulnerability usually requires manual validation, and we will support such vulnerabilities in the future through further analysis of the returned messages. UCRF found a few non-visible front-end interfaces, some of which may be vulnerable, and we leave the specific analysis for future work. Previous work KARONTE (Redini et al., 2020) observed that multi-binary interactions could lead to vulnerabilities, where some data-flow is forwarded by border binary to other binaries for processing. UCRF can find such vulnerabilities by fuzzing border binary directly, but currently, we do not consider constraints of data-flow in other binaries and leave it for future work.

Fault observability UCRF observes network behavior to determine the occurrence of memory crash vulnerabilities. However, as discussed by Muench et al. (2018), not all errors can be observed on embedded devices. Due to the lack of protection mechanisms common in desktop systems (Smart Yet Flawed, 2020; Thompson and Zatko, 2018; Yu et al., 2022), embedded devices cannot provide error messages (e.g., segment faults) and feedback when an application crashes. In fundamental analysis, we found the error handling code of the web server or self-protection mechanisms of the embedded device (e.g., watchdog) often eliminates exceptions by restarting the application or operating system, allowing us the opportunity to determine the occurrence of the crash by observing network behavior. In this paper, we performed reverse engineering according to the crashes detected by UCRF and proved that UCRF did not have false positives. However, false negatives may occur in some cases by observing network behavior only. For example, SRFuzzer detected 8 command execution vulnerabilities through the instrumentation of physical devices, ESRFuzzer in D-CONF mode found 3 through the same mechanism, however, UCRF found only 2 by observing network behavior. Using QEMU (Bellard, 2005) to emulate heterogeneous firmware and catch exceptions, or instrumentation of physical device (Zhang et al., 2019), are both possible approaches to detect *silent memory corruptions*. We plan to use these methods to enhance UCRF in the future.

7. Related work

Fuzzing embedded systems Fuzzing is a popular technique for vulnerability discovery. Several previous research efforts have focused on feedback-driven fuzzing methods (American Fuzzy Lop, 2014; Lee et al., 2021; Li et al., 2017; Rawat et al., 2017; Stephens et al., 2016; Wang et al., 2017) that obtain runtime information, such as code coverage, to guide the seed generation, and giving higher weight to seeds that can trigger deeper paths. The firmware cannot be natively executed for embedded devices without emulation support. FIRMADYNE (Chen et al., 2016), and FirmAE (Kim et al., 2020) are designed to improve the simulation success rate and perform large-scale dynamic analysis. Zheng et al. (2019b) and FirmFuzz (Srivastava et al., 2019) integrate a fuzzing framework for vulnerability discovery based on firmware emulation. In order to improve the throughput of firmware emulation, Firm-AFL (Zheng et al., 2019a) proposes augmented process emulation by running the firmware program in a user-mode emulator, and redirecting system calls to the system-mode emulator when the program runs to require special hardware dependencies. EQUAFL (Zheng et al., 2022) uses enhanced user-mode emulation to execute firmware programs, thus avoiding the significant overhead of system-mode emulation. In order to successfully execute the program in user-mode emulation, EQUAFL first runs the program using full-system emulation to collect valid information and then migrates the needed environment for user-mode emulation. μAFL (Li et al., 2022) uses the debug interface on the embedded device to collect runtime information from the device, enabling feedback-driven fuzzing based on code coverage. Firmware emulation can often provide fuzzer with runtime information about the program, on which advanced feedback-based fuzzing methods can be employed. However, most of these methods use random strings or HTTP messages as input and do not consider the characteristics of network communication in embedded devices.

To avoid problems in firmware emulation, some studies test against physical devices. SRFuzzer (Zhang et al., 2019) is the state-of-the-art tool for fuzzing the SOHO router, which simulates the browser's submission behavior and captures the request sent to the back-end as the seed. SRFuzzer analyses the requests to find the invariant, including fixed string and number, which are used to pass the conditional checks in the back-end. However, the strict input checking of the back-end code makes fuzzing inefficient. Note that UCRF collects a list of all potential constraints from the back-end rather than just invariants, and UCRF generates seeds based on the back-end to avoid over-constraints from the seeds generated at the front-end. ESRFuzzer (Zhang et al., 2021) extends SRFuzzer with D-CONF mode, which uses the NVRAM configuration operations provided by the partial router (e.g., "backup and restore configuration" features), and tests the back-end with NVRAM-related key-value pairs. In D-CONF mode, ESRFuzzer can fuzz a single key-value pair directly by mutating the value. D-CONF mode can help ESRFuzzer to test the code more deeply under the premise of ignoring part of the back-end data validation. However, not all key-value pairs are relevant to NVRAM operations, and ignoring NVRAM-independent key-value pairs hinders the discovery of back-end vulnerabilities. SRFuzzer and ESRFuzzer implement a signal-based monitor to obtain more accurate in-process information through common signals (e.g., SIGSEGV and SIGABRT) when having permission to deploy monitors inside real devices, and thus to catch silent memory crashes. RPFuzzer (Wang et al., 2013) target to fuzz router protocol such as SNMP. Snipuzz (Feng et al., 2021) uses a mutation strategy based on message snippets to test against IoT devices, inferring the message snippets used for mutation based on the response, thus narrowing the search space for probing information. IoTInfer (Shu and Yan, 2022) leverages a heuristic based on FSM (Finite State Machine) inference to guide black-box fuzzy testing of IoT network protocols, including Bluetooth and Telnet protocols. IoTFuzzer (Chen et al., 2018) and DIANE (Redini et al., 2021) leverage the companion app of IoT devices to generate well-structured test cases. However, such test cases ignore potential constraints in the back-end. Furthermore, UCRF collects constraint information in the back-end to help test cases reach deeper paths.

Binary static analysis Binary static analysis techniques are another method for embedded system vulnerability discovery (Davidson et al., 2013; Redini et al., 2017; Shoshitaishvili et al., 2015). Cheng et al. (2021) proposed a method for automatic inference of taint sources in SOHO router firmware, which summarizes the features of functions used to parse structured data of the KEY-VALUE model, and uses data-flow analysis to identify such functions as taint sources. UCRF further performs data-flow analysis on this foundation to collect constraints in handling network data, then used to guide fuzzing. SaTC (Chen et al., 2021) observed that variable names are commonly shared between front-end files and back-end functions, using the keywords found in the front-end can locate action handle and input entry in the back-end. However, inaccurate front-end keywords can result in false positives and false negatives for action handle identification. KARONTE (Redini et al., 2020) observed that data-flow in multi-binary interactions could lead to vulnerabilities, which first built the binary dependency graph and used symbolic execution for taint propagation to discover vulnerabilities. DTaint (Cheng et al., 2018) is the first work to detect the taint-style vulnerability in firmware binaries, which detect vulnerabilities by generating context-sensitive and interprocedural data-flow. UCRF can do verification against web server-related vulnerabilities in embedded devices, and we believe that UCRF can be used as a complement to static analysis tools.

8. Conclusion

In this paper, we propose UCRF focusing on SOHO router fuzzing, which generates high-quality test cases via static analysis on the back-end binary without firmware emulation. To generate the well-structured seed, UCRF identifies all communication interfaces in the back-end to avoid missing non-visible front-end interfaces. Then, UCRF utilizes static data-flow analysis to extract constraint information for each action handle, and generates efficient and in-depth test cases only in meaningful test spaces based on constraint information. We implemented a prototype system of UCRF, and evaluated it on 10 real routers from 4 popular vendors, a total of 38 CVE IDs, 2 CNVD IDs, and 1 PSV ID were assigned.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments to help us improve this paper. This work is financially supported by the National Key Research and Development Program of China, under Grant 2020YFB805405, and National Natural Science Foundation of China under Grant nos. 61702504, U1766215.

Appendix A

A1. Experimental devices

Figure 4 shows the experimental devices we used.

A2. Assigned vulnerabilities

Table 7 shows all assigned vulnerability IDs.

A3. Merged constraint

Table 8 shows merged constraint.

References

Alrawi, O., Lever, C., Antonakakis, M., Monrose, F., 2019. SOK: security evaluation of home-based IoT deployments. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1362–1380.
American fuzzy lop, 2014. http://lcamtuf.coredump.cx/afl/. Accessed October 18, 2022.
Bellard, F., 2005. QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track, vol. 41. Califor-nia, USA, pp. 10–5555.
Binwalk, 2014. https://github.com/ReFirmLabs/binwalk. Accessed October 18, 2022.
boofuzz, 2014. https://github.com/jtpereyda/boofuzz. Accessed October 18, 2022.
Chen, D.D., Woo, M., Brumley, D., Egele, M., 2016. Towards automated dynamic analysis for linux-based embedded firmware. In: NDSS, vol. 1, p. 1.
Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W.C., Sun, M., Yang, R., Zhang, K., 2018. Iotfuzzer: discovering memory corruptions in IoTthrough ap-p-based fuzzing. NDSS.
Chen, L., Wang, Y., Cai, Q., Zhan, Y., Hu, H., Linghu, J., Hou, Q., Zhang, C., Duan, H., Xue, Z., 2021. Sharing more and checking less: leveraging common input keywords to detect bugs in embedded systems. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 303–319.
Cheng, K., Fang, D., Qin, C., Wang, H., Zheng, Y., Yu, N., Sun, L., 2021. Automatic inference of taint sources to discover vulnerabilities in soho router firmware. In: IFIP International Conference on ICT Systems Security and Privacy Protection. Springer, pp. 83–99.
Cheng, K., Li, Q., Wang, L., Chen, Q., Zheng, Y., Sun, L., Liang, Z., 2018. Dtaint: detecting the taint-style vulnerability in embedded device firmware. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 430–441.
Cisco small business rv110w, rv130, rv130w, and rv215w routers remote command execution and denial of service vulnerabilities. 2022. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-sb-rv-rce-overflow-ygHByAK. Accessed October 18, 2022.
Costin, A., Zarras, A., Francillon, A., 2016. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 437–448.
Davidson, D., Moench, B., Ristenpart, T., Jha, S., 2013. Fie on firmware: finding vulnerabilities in embedded systems using symbolic execution. In: 22nd USENIX Security Symposium (USENIX Security 13), pp. 463–478.
Dir-882 & dir-882-us multiple vulnerabilities. 2022. https://supportannouncement.us.dlink.com/announcement/publication.aspx?name=SAP10287. Accessed October 18, 2022.
Feng, X., Sun, R., Zhu, X., Xue, M., Wen, S., Liu, D., Nepal, S., Xiang, Y., 2021. Snipuzz: black-box fuzzing of IoT firmware via message snippet inference. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 337–350.
Ida pro, 2022. https://hex-rays.com/ida-pro/. Accessed October 18, 2022.
Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., Kim, Y., 2020. Firmae: towards large-scale emulation of IoT firmware for dynamic analysis. In: Annual Computer Security Applications Conference, pp. 733–745.
Kolias, C., Kambourakis, G., Stavrou, A., Voas, J., 2017. DDoS in the IoT: mirai and other botnets. Computer 50 (7), 80–84.
Lee, G., Shim, W., Lee, B., 2021. Constraint-guided directed greybox fuzzing. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 3559–3576.
Li, W., Shi, J., Li, F., Lin, J., Wang, W., Guan, L., 2022. μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware. ICSE.
Li, Y., Chen, B., Chandramohan, M., Lin, S.-W., Liu, Y., Tiu, A., 2017. Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 627–637.
Mi smart plug, 2022. https://www.mi.com/us/mj-socket/. Accessed October 18, 2022.
Mirai IoT botnet blamed for 'smashing liberia off the internet', 2016. https://www.theregister.com/2016/11/04/liberia_ddos/. Accessed October 18, 2022.
Muench, M., Stijohann, J., Kargl, F., Francillon, A., Balzarotti, D., 2018. What you corrupt is not what you crash: challenges in fuzzing embedded devices. NDSS.
Nethercote, N., Seward, J., 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan Not. 42 (6), 89–100.
Over 65,000 home routers are proxying bad traffic for botnets, apts, 2018. https://www.bleepingcomputer.com/news/security/over-65-000-home-routers-are-proxying-bad-traffic-for-botnets-apts/. Accessed October 18, 2022.
Payloadsallthethings, 2018. https://github.com/swisskyrepo/PayloadsAllTheThings. Accessed October 18, 2022.
python-miio, 2022. https://github.com/rytilahti/python-miio. Accessed October 18, 2022.
Pyvex, 2022. https://github.com/angr/pyvex. Accessed October 18, 2022.
Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H., 2017. Vuzzer: application-aware evolutionary fuzzing. In: NDSS, vol. 17, pp. 1–14.
Redini, N., Continella, A., Das, D., De Pasquale, G., Spahn, N., Machiry, A., Bianchi, A., Kruegel, C., Vigna, G., 2021. Diane: identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices. In: 2021 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 484–500.

Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2017. Bootstomp: on the security of bootloaders in mobile devices. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 781–798.
Redini, N., Machiry, A., Wang, R., Spensky, C., Continella, A., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2020. Karonte: detecting insecure multi-binary interactions in embedded firmware. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1544–1561.
Selenium, 2022. https://www.selenium.dev/. Accessed October 18, 2022.
Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G., 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: NDSS, vol. 1, p. 1.
Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al., 2016. SOK: (state of) the art of war: offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 138–157.
Shu, Z., Yan, G., 2022. Iotinfer: automated blackbox fuzz testing of IoTnetwork protocols guided by finite state machine inference. IEEE Internet Things J. doi:10.1109/JIOT.2022.3182589.
Smart yet flawed: IoT device vulnerabilities explained, 2020. https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/smartyet-flawed-iot-device-vulnerabilities-explained. Accessed October 18, 2022.
Srivastava, P., Peng, H., Li, J., Okhravi, H., Shrobe, H., Payer, M., 2019. Firmfuzz: automated IoT firmware introspection and analysis. In: Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, pp. 15–21.
Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: augmenting fuzzing through selective symbolic execution. In: NDSS, vol. 16, pp. 1–16.
Szekeres, L., Payer, M., Wei, T., Song, D., 2013. SOK: eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy. IEEE, pp. 48–62.
The mobile economy, 2022. 2022. https://www.gsma.com/mobileeconomy/. Accessed October 18, 2022.
Thompson, P., Zatko, S., 2018. Build safety of software in 28 popular home routers. Cyber-ITL (Dec 2018).
Travel routers NAS devices among easily hacked IoTdevices, 2017. https://threatpost.com/travel-routers-nas-devices-among-easily-hacked-iot-devices/124877/. Accessed October 18, 2022.
Vadayath, J., Eckert, M., Zeng, K., Weideman, N., Menon, G.P., Fratantonio, Y., Balzarotti, D., Doupé, A., Bao, T., Wang, R., et al., 2022. Arbiter: bridging the static and dynamic divide in vulnerability discovery on binary programs. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 413–430.
Wang, J., Chen, B., Wei, L., Liu, Y., 2017. Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 579–594.
Wang, Z., Zhang, Y., Liu, Q., 2013. Rpfuzzer: a framework for discovering router protocols vulnerabilities based on fuzzing. KSII Tran. Internet Inf. Syst. (TIIS) 7 (8), 1989–2009.
Yu, R., Del Nin, F., Zhang, Y., Huang, S., Kaliyar, P., Zakto, S., Conti, M., Portokalidis, G., Xu, J., 2022. Building embedded systems like it's 1996. Ndss.
Yu, Z., Zheng, W., Wang, J., Tang, Q., Nie, S., Wu, S., 2020. Codecmr: cross-modal retrieval for function-level binary source code matching. Adv. Neural Inf. Process. Syst. 33, 3872–3883.
Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al., 2014. Avatar: a framework to support dynamic security analysis of embedded systems' firmwares. In: NDSS, vol. 14, pp. 1–16.
Zhang, Y., Huo, W., Jian, K., Shi, J., Liu, L., Zou, Y., Zhang, C., Liu, B., 2021. Esrfuzzer: an enhanced fuzzing framework for physical soho router devices to discover multi-type vulnerabilities. Cybersecurity 4 (1), 1–22.
Zhang, Y., Huo, W., Jian, K., Shi, J., Lu, H., Liu, L., Wang, C., Sun, D., Zhang, C., Liu, B., 2019. Srfuzzer: an automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities. In: Proceedings of the 35th Annual Computer Security Applications Conference, pp. 544–556.
Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L., 2019. Firm-AFL: high-throughput greybox fuzzing of IoTfirmware via augmented process emulation. In: 28th USENIX Security Symposium (USENIX Security 19), pp. 1099–1114.
Zheng, Y., Li, Y., Zhang, C., Zhu, H., Liu, Y., Sun, L., 2022. Efficient greybox fuzzing of applications in linux-based IoT devices via enhanced user-mode emulation. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 417–428.
Zheng, Y., Song, Z., Sun, Y., Cheng, K., Zhu, H., Sun, L., 2019. An efficient greybox fuzzing scheme for linux-based IoT programs through binary static analysis. In: 2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC). IEEE, pp. 1–8.

Chuan Qin received the B.E. degree from Central South University, China, in 2019. He is currently pursuing the Ph.D. degree with the Institute of Information Engineering, Chinese Academy of Sciences, China. He focuses on the security of Internet of Things and industrial control systems. His research interests include IoT security and program analysis.

Jiaqian Peng received the B.E. degree from Hefei University of Technology, China, in 2020. He is currently pursuing the Master degree with the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include IoT security and vulnerability detection.

Puzhuo Liu received the B.E. degree from Jilin University, China, in 2018. He is currently pursuing the Ph.D. degree with the Institute of Information Engineering, Chinese Academy of Sciences, China. He focuses on the security of Internet of Things and industrial control systems. His research interests include binary program analysis, fuzzing, vulnerability and risk analysis.

Yaowen Zheng received the Ph.D. from University of Chinese Academy of Sciences, China, in 2020. He is a research fellow at Nanyang Technological University. His research interests are related to system security. In particular, his research mainly focuses on vulnerability analysis techniques such as fuzzing, dynamic emulation for embedded system. He has also been in the University of California, Riverside as a visiting scholar for one year.

Kai Cheng received the Ph.D. from University of Chinese Academy of Sciences, China, in 2021. He is now a senior engineer at Sangfor Technologies Inc. His research interests include IoT security and vulnerability detection based on static analysis.

Weidong Zhang received the Ph.D. from University of Chinese Academy of Sciences, China, in 2021. He is now a Research Assistant in Beijing Key Laboratory of IoT information security technology, Institute of Information Engineering, CAS, China. His research interests include software security and and vulnerability detection.

Limin Sun received the Ph.D. degree from the National University of Defense Technology. He is currently a Professor with the Institute of Information Engineering, Chinese Academy of Sciences. He is also the Secretary General of the Select Committee of CWSN and the Director of the Beijing Key Laboratory of IoT Information Security Technology. His main research interests include the Internet of Things security and industrial control system security. He is an Editor of the Journal of Computer Science and Journal of Computer Applications.