Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems

Libo Chen, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Yanhao Wang, *QI-ANXIN Technology Research Institute;* Quanpu Cai and Yunfan Zhan, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Hong Hu, *Pennsylvania State University;* Jiaqi Linghu, *QI-ANXIN Technology Research Institute;* Qinsheng Hou, *QI-ANXIN Technology Research Institute; Shandong University;* Chao Zhang and Haixin Duan, *BNRist & Institute for Network Science and Cyberspace, Tsinghua University; Tsinghua University-QI-ANXIN Group JCNS;* Zhi Xue, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University*

https://www.usenix.org/conference/usenixsecurity21/presentation/chen-libo

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems

Libo Chen *[1], Yanhao Wang*[2], Quanpu Cai[1], Yunfan Zhan[1], Hong Hu[3], Jiaqi Linghu[2], Qinsheng Hou[2,6],
Chao Zhang[4,5], Haixin Duan[4,5], Zhi Xue †[1]

[1]*School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University*
[2]*QI-ANXIN Technology Research Institute* [3]*Pennsylvania State University*
[4]*BNRist & Institute for Network Science and Cyberspace, Tsinghua University*
[5]*Tsinghua University-QI-ANXIN Group JCNS* [6]*Shandong University*
{bob777, zxue}@sjtu.edu.cn, wangyanhao@qianxin.com, honghu@psu.edu, {chaoz, duanhx}@tsinghua.edu.cn

Abstract

IoT devices have brought invaluable convenience to our daily life. However, their pervasiveness also amplifies the impact of security vulnerabilities. Many popular vulnerabilities of embedded systems reside in their vulnerable web services. Unfortunately, existing vulnerability detection methods cannot effectively nor efficiently analyze such web services: they either introduce heavy execution overheads or have many false positives and false negatives.

In this paper, we propose a novel static taint checking solution, SaTC, to effectively detect security vulnerabilities in web services provided by embedded devices. Our key insight is that, string literals on web interfaces are commonly shared between front-end files and back-end binaries to encode user input. We thus extract such common keywords from the front-end, and use them to locate reference points in the back-end, which indicate the input entry. Then, we apply targeted data-flow analysis to accurately detect dangerous uses of the untrusted user input. We implemented a prototype of SaTC and evaluated it on 39 embedded system firmwares from six popular vendors. SaTC discovered 33 unknown bugs, of which 30 are confirmed by CVE/CNVD/PSV. Compared to the state-of-the-art tool KARONTE, SaTC found significantly more bugs on the test set. It shows that, SaTC is effective in discovering bugs in embedded systems.

1 Introduction

IoT (Internet of Things) devices open the door to unprecedented connectivity and bring innovative approaches and services to our daily life. It is believed that 5.8 billion IoT endpoints are in use in 2020 [44]. However, the pervasiveness of IoT devices renders bugs more devastating and leads to a significant security risk. According to the report [32], 57% of IoT devices are vulnerable to medium or high severity attacks, making these devices low-hanging fruit for attackers.

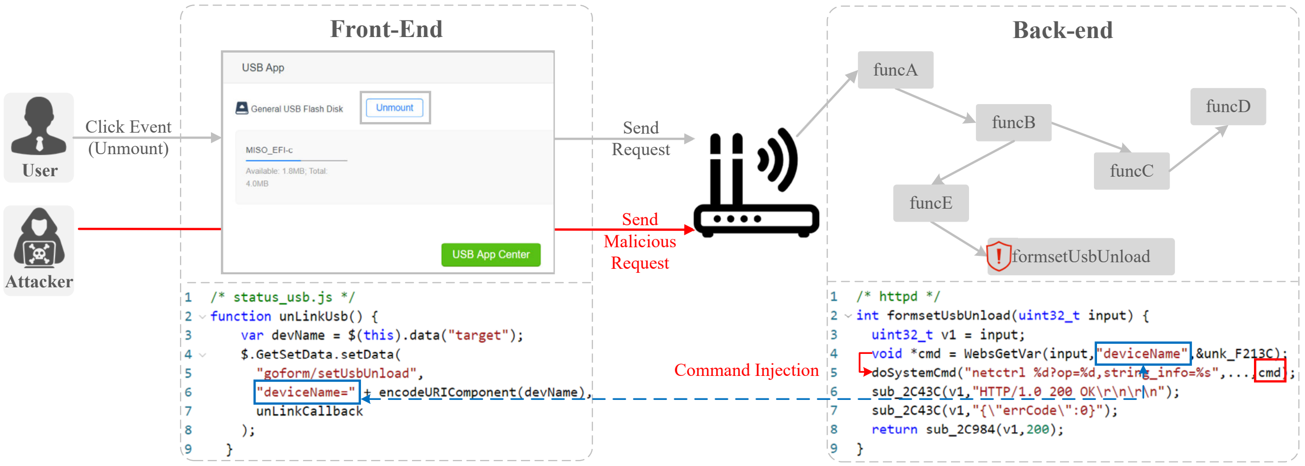
Among all IoT devices, wireless routers and web cameras suffer more attacks than other embedded devices [32, 39, 41–43]. The key reason is that these devices expose web services and network services that usually contain exploitable vulnerabilities. For example, a wireless router usually provides a web-based interface for end-users to configure the system. The underlying firmware contains a web server, various front-end files, and back-end binary programs. The web server accepts HTTP requests from the front-end and summons back-end binaries to handle them. In this scenario, attackers may construct malicious inputs to the front-end in order to compromise corresponding back-end binaries.

Unfortunately, existing methods cannot effectively analyze services in embedded systems to detect vulnerabilities. They are blocked by complicated interactions and implicit data dependencies between the front-end and the back-end. Dynamic solutions, like fuzzing [8, 52] and emulation [7, 23, 50, 53], provide concrete context to run the back-end. However, dynamic executions can only reach a small portion of all possible program states, leading to a lot of false negatives. Static methods, like KARONTE [34], rely on the common inter-process communication (IPC) paradigms between the front-end and the back-end (e.g., environment variables) to locate input-processing code, and perform centralized testing. Unfortunately, these methods may result in many false positives as they ignore the user-input context stored in the front-end files.

We observe that the key point of finding bugs from embedded systems is *to use the front-end of the web service to locate the back-end code that handles the user-supplied data*.

In this paper, we present SaTC (Shared-keyword aware Taint Checking), a novel static analysis approach that tracks the data flow of the user input between front-end and back-end to precisely detect security vulnerabilities. Our insight is that a back-end function handling the user-input usually shares a similar keyword with the corresponding front-end file: in the front-end, the user-input is labeled with a keyword and encoded in the data package; in the back-end, the same or similar keyword is used to extract the user-input from the data package. Therefore, we can use the shared keyword to identify

[*] Co-leading authors.
[†] Corresponding author.



```
/* status_usb.js */
function unLinkUsb() {
    var devName = $(this).data("target");
    $.GetSetData.setData(
      "goform/setUsbUnload",
      "deviceName=" + encodeURIComponent(devName),
      unLinkCallback
    );
  }
```

```
/* httpd */
int formsetUsbUnload(uint32_t input) {
  uint32_t v1 = input;
  void *cmd = WebsGetVar(input,"deviceName",&unk_F213C);
  doSystemCmd("netctrl %d?op=%d,string_info=%s",...,cmd);
  sub_2C43C(v1,"HTTP/1.0 200 OK\r\n\r\n");
  sub_2C43C(v1,"{\"errCode\":0}");
  return sub_2C984(v1,200);
}
```

Figure 1: Motivating example. The left-hand side shows the front-end of the Tenda AC18 router: the USB management interface and the source code of the web page; the right-hand side shows the back-end: the call graph of the message processing process. `deviceName` is used by the code in the front-end and the back-end. An attacker can inject arbitrary command via sending a request with a malicious device name.

the connection between front-end and back-end, and locate the entry of user-input in the back-end. With the user-input entry, we can apply selective data-flow analysis to track the untrusted input and identify its dangerous usage, like using it as a command, which leads to command injection attacks.

To improve the speed of vulnerability detection in embedded systems, we propose three optimizations to traditional taint analysis techniques. First, based on the features of IoT firmware, we develop a coarse-grained taint engine which contains special rules for particular functions to balance the efficiency and accuracy. Second, we speed up the path exploration with the input guidance and the trace merging, which leverages the call graph and sink functions to optimize the searching space. Finally, to handle the infinite paths problem in specific functions (e.g., sanitizer function), we use a prioritization algorithm to efficiently process loops.

We design SaTC with three components: an input-keyword extractor to collect keywords from the front-end files, an input entry recognizer to locate input entry in the back-end binaries, and an input-sensitive taint engine to efficiently detect vulnerabilities. Our prototype is implemented based on Ghidra [31] and KARONTE [34] with around 9800 lines of Python code. It supports parsing multiple types of front-end files, including JavaScript, HTML, and XML files, and could analyze back-end in widely used architectures, such as x86, ARM, and MIPS.

To understand the efficacy of SaTC on detecting vulnerabilities from embedded systems, we apply our tool on 39 firmware samples from six vendors. SaTC successfully discovered 33 unknown vulnerabilities in these latest-version firmware samples, including command injection vulnerabilities and buffer overflow bugs. Among these bugs, 30 of them have been assigned CVE/CNVD/PSV IDs due to their severe security impact. We also compare SaTC with the state-of-the-art bug detection tool, KARONTE. After testing seven firmware samples for two days, SaTC raises 65 alerts which contain 36 true positives, while KARONTE does not detect any true positive. The results show that SaTC is a practical tool to detect bugs in embedded systems.

In summary, we make the following contributions:

- We propose a novel technique that leverages the common keywords between the front-end and back-end of embedded systems to locate the data entry in the back-end binary.
- We design and implement SaTC that utilizes coarse-grained taint analysis and trace merging method to efficiently detect vulnerabilities in embedded systems.
- We evaluate SaTC on 39 real-world firmware samples and discover 33 unknown bugs, including command injection, buffer overflow, and incorrect access control bugs.

To foster future research, we will release the source code of SaTC as well as the experiment data at `https://github.com/NSSL-SJTU/SaTC`.

Roadmap. §2 provides the motivation and background of this work, and gives an overview of our system. §3, §4, §5 and §6 present the design and implementation of our data-relationship recovering technique and the sensitive-data flow analysis. We demonstrate the efficacy of SaTC through experiments and case studies on real-world firmware samples in §7. We discuss the application scenarios of SaTC and its limitation in §8, and compare our system with related work in §9. §10 concludes the paper.

2 Problem and Approach Overview

In this section, we first provide the background of vulnerabilities in embedded systems. Then, we present the overview of our approach and discuss the associated challenges.

Threat Model. In this paper we aim to detect security vulner-

```
int sub_426B8() {
  Register_Handler("GetSambaCfg",formGetSambaConf);
  Register_Handler("setUsbUnload",formsetUsbUnload);
  Register_Handler("GetUsbCfg",formGetUsbCfg);
}

int formsetUsbUnload(uint32_t input) {
  uint32_t v1 = input;
  void *cmd = WebsGetVar(input,"deviceName",&unk_F213C);
  doSystemCmd("netctrl %d?op=%d,string_info=%s",...,cmd);
  sub_2C43C(v1,"HTTP/1.0 200 OK\r\n\r\n");
  sub_2C43C(v1,"{\"errCode\":0}");
  return sub_2C984(v1,200);
}
```

Listing 1: Back-end code of the motivating example. Function `sub_426B8` registers several handler functions, including function `formsetUsbUnload` which processes USB unload action.

abilities from two types of IoT devices, i.e., wireless routers and web cameras. These devices implement convenient web services and network services to help the system management, configuration and data sharing, like through protocols MQTT (Message Queuing Telemetry Transport) and UPnP (Universal Plug and Play). As these two types of devices are usually the entries to the home network or local network [25], attackers pay a lot of attention to them and like to hack them through the network services [7, 8, 13]. For example, a recent study [45] shows that 75% of IoT attacks in 2018 are directed against routers, while web cameras are second at 15.2%. Meanwhile, most of these devices still have critical flaws [32, 46]. We consider the attackers who have access to a copy of the target device's firmware, but are not physically accessible to the victim device. They can only communicate with the front-end interfaces to affect the values used in the back-end. The back-end is protected with state-of-the-art defense mechanisms on IoT devices, such as Executable Space Protection [26], Address Space Layout Randomization [33], and stack canaries [14]. However, due to the limited resources, no advanced mechanisms (e.g., software defined networking [3], intrusion detection system [1, 49]) are deployed to dynamically recognize these attacks, i.e., command injection and memory corruption attacks.

2.1 Motivating Example

The web services of an IoT device usually consist of two components, the front-end and the back-end. The front-end presents the configurations and functionalities of the device to the end-users, while the back-end parses the requests received from the front-end and executes related services. Figure 1 shows an example where the end-user utilizes the interactive web interface to manage external devices of the Tenda AC18 router. Currently, there is one USB drive called *General USB Flash Disk* mounted to the router and the user decides to remove it. From the front-end web interface, she just needs to click the `Unmount` button. The front-end will automatically synthesize an `unmount` request with the device name attached (line 4 in status_usb.js on the left-hand side), and send the request to the back-end on the right-hand side. The back-end web server will parse the request and invoke function `formsetUsbUnload` to handle the request. Function `formsetUsbUnload` identifies the device name, synthesizes a command string (line 4 in httpd) and executes the command to unmount the specified device (line 5 in httpd).

Unfortunately, the web service contains a typical command injection vulnerability. As function `formsetUsbUnload` generates the unmount command without any sanitization check, an attacker can append extra commands to `deviceName` and thus run arbitrary commands on the router. For example, a malicious `deviceName` `22;telnetd -l /bin/sh -p 3333 &` forces the back-end program to run two commands: 1) `netctrl ... 22` and 2) `telnetd -l /bin/sh -p 3333 &`, where the second command launches a server to accept any future commands. Furthermore, an attacker can directly send the malicious `unmount` command to the back-end via the URL `http://IP:Port/goform/setUsbUnload?deviceName=evalCMD`, showing that the device can be compromised remotely.

Current bug-finding techniques cannot detect these vulnerabilities effectively. Dynamic solutions, like fuzzing [8] and emulation [7, 50, 53], cannot guarantee to cover all program states and may miss many critical bugs. For example, to use the recent work SRFuzzer [52] to identify this bug, we have to insert a USB device into the router and trigger all the normal interactions between the front-end and the back-end, including `Unmount`. However, if we do not have much knowledge of the router and forget to take these manual actions, the dynamic methods will likely miss this highly exploitable bug. Static approaches such as KARONTE [34] focus on back-end binaries and try to analyze all possible paths to find bugs. For example, KARONTE takes common inter-process communication (IPC) paradigms between the web server and the binaries as the starting points for analysis. However, the large number of IPC interfaces bring in a large number of excessive analyses and thus lead to many false positives. We need to identify the real entries of user-inputs in the back-end programs to perform targeted, accurate analysis.

2.2 Observation

Without an oracle to highlight all real entries of user-inputs in the back-end, how can we discover the vulnerability in the motivating example? Our intuition is that *the strings shown in the web interface are commonly used in both front-end files and back-end functions*: in the front-end, the user-input is labeled with a keyword and encoded in the data package; in the back-end, the same or similar keyword is used to extract the user-input from the data package. With these shared keywords, we can connect the front-end and the back-end, and identify the input-processing functions from the latter. Starting from these functions, we can perform the static data-flow analysis and effectively identify exploitable bugs.

Consider our motivating example in Figure 1, the front-end JavaScript file `status_usb.js` contains two strings `goform/setUsbUnload` and `deviceName`. Coincidentally, both of them occur in the back-end binary `httpd`. Listing 1 pro-

Table 1: **Intuition verification. F-Strs** represents the strings selected from front-end and used to encode user-input; **B-allStrs** represents all the printable strings in the back-end; **Intersection** represents the F-Strs strings that are used to retrieve data in the back-end; **Verified** indicates the Intersection strings confirmed to label the same data in the front-end and the back-end; **%** represents the proportion of Verified in Intersection.

Vendor	Device Series	#F-Str	#B-allStrs	#Intersect	Verified	%
Tenda	AC9	101	49,288	86	70	81.4
Tenda	AC15	81	241,314	65	63	96.9
Tenda	AC18	81	119,537	66	57	86.4
Tenda	W20E	161	139,885	89	79	88.8
Netgear	R7000P	114	467,706	59	59	100.0
Netgear	XR300	135	517,254	76	72	94.7
Motorola	M2	133	83,911	31	31	100.0
D-Link	867	85	84,764	53	50	94.3
D-Link	882	100	522,317	86	81	94.1
TOTOLink	A950RG	69	53,931	31	27	87.1
Average	-	106	227,990	64	59	92.4

vides more details of the back-end. `goform/setUsbUnload` is split into two parts, while `setUsbUnload` is used to find the input handler `formsetUsbUnload`. `deviceName` is used by `formsetUsbUnload` to get the device name. With the help of the common keywords `setUsbUnload` and `deviceName`, we can recognize the user-input handler `formsetUsbUnload` (line 3) in the back-end and locate line 9 as the start point of processing the input. Now, we can use the data-flow analysis technique, like taint analysis [30, 35, 40], to track the usage of untrusted input and detect unsafe usage. In this example, we set `cmd` as the taint source and track its usage. At line 10, we find `cmd` is used as the parameter of the security-critical function `doSystemCall` with no constraints. This triggers an alert to signal the potential vulnerability.

To verify that our intuition works on normal IoT devices, we inspect 10 routers from five vendors to check whether the front-end and back-end use common keywords to represent the user input. Specifically, we extracted strings from the back-end and front-end based on the following three principles. 1) We select front-end strings that are used to encode user-input in the network package sent to the back-end. Specifically, the string is some “key” in the network package that has the form of `...&key=value&...`. We manually triggered as many actions as possible in the front-end to cover more request messages. 2) We select back-end strings that are used to retrieve input data from the messages. Based on our knowledge of IoT firmware, we define several functions that are commonly used to obtain input value, such as `websGetVar` in the motivation example. We collect constant-string arguments of these functions as interesting back-end strings. 3) We take an intersection of collect front-end and back-end strings. For each string in the intersection, we mutate the associated data in the front-end to trigger the request message sending to the back-end, and check the value of the associated variables in the back-end. If the back-end variable changes its value accordingly, we confirm that the tested string is a shared keyword to represent user input. We perform the mutation several times to avoid accidentally-changed back-end variables.

Table 1 shows our verification result. On average, 92.4% of the keyword-value pairs captured in the front-end match the ones in the back-end, showing that our intuition works for these common devices. For two devices, all front-end strings match with the back-end ones, where we can completely rely on the shared strings to identify the input data from the back-end. However, for other devices, like Tenda AC9, the matched strings only account for 81.4%, and we have to inspect the other 18.6% to achieve a more accurate analysis.

2.3 Challenges and Our Approaches

Although our method seems straightforward for the motivating example, there are three challenges when we apply it to real-world embedded systems.

C1: Identifying keywords in the front-end. User input is usually labeled with a keyword hidden in the front-end, like `deviceName` in the motivating sample. However, an unpacked firmware contains thousands of strings in the front-end. For example, the firmware of Netgear R7000P listed in Table 1 contains more than 600 front-end files and nearly ten thousand strings. It is challenging to understand the semantics of each string without domain knowledge or real executions.

C2: Locating the input handler in the back-end. The back-end binaries contain many functions, where only a small part of them handle the user input. Meanwhile, they also contain a large number of strings and corresponding reference points. As Table 1 shows, each device contains more than 40,000 strings in the back-end binaries. Therefore, it is challenging to identify the entry point of user input in the back-end. Ideally, the point should be strongly connected with the user input, and its location should be close to the real usage of the input.

C3: Tracking the massive paths of user input to detect vulnerabilities. To detect the vulnerability, we need to track the data flow from the entrance of the input to all sinks, which may contain massive paths. Unfortunately, the state-of-the-art analysis tools [34, 38] introduce high overhead, and cannot handle the elaborate control-flow graph or bypass the user-input sanitization. We need an efficient method for data-flow analysis and path exploration.

In this paper, we design SaTC to address the challenges above to detect common vulnerabilities in embedded systems effectively. Figure 2 provides an overview of our system, which takes as input a firmware sample (i.e., the entire firmware image) and produces various bug reports. As the first step, SaTC unpacks the firmware image using an off-the-shelf firmware unpacker, like `binwalk` [21]. From the unpacked image, it recognizes front-end files and back-end programs based on the file types: HTML, JavaScript, and XML files are usually front-end files, while executable binaries and libraries are back-end files. Then, SaTC analyzes the front-end files and utilizes typical patterns to extract the potential key-

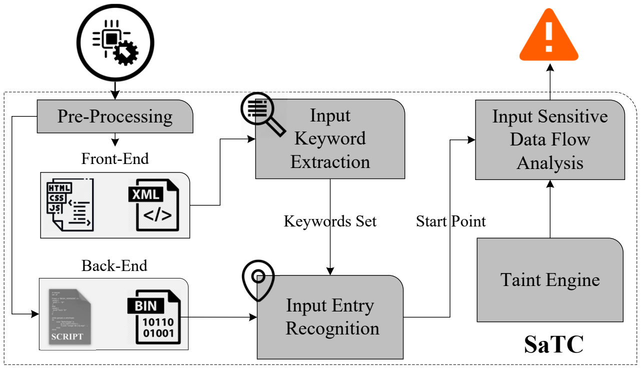


Figure 2: Structure of SaTC. SaTC searches in the firmware front-end to find input keywords and locates their references in the back-end. Starting from reference points, SaTC uses input-sensitive taint analysis to discover vulnerabilities in the back-end.

words of the user input. In Figure 1, `deviceName`, `target` and `goform/setUsbUnload` will be identified as input keywords.

After that, SaTC recognizes the *border* binaries in the back-end, which invokes different handling functions based on the user-input keywords. From these functions, we try to locate the points that retrieve the user input. To find the implicit entry points related to the user-input, we further apply our intuition to multiple back-end programs: user-input may be delivered from one program to another via shared keywords. This helps us track implicit data dependencies among binaries. In Listing 1, the code at line 9 parses the user-input via the input keyword `deviceName`, and thus SaTC treats it as one entry point of user input. Finally, we use our input-sensitive taint analysis to track the usage of the untrusted data. We design several optimizations to make the traditional taint analysis efficient on embedded systems, including coarse-grained taint propagation, input-guided path selection, and the trace merging technique. When SaTC finds the user input is used in any predefined sink, like as a parameter of a system call, it collects the path constraints and judges the reachability. If the sink is reachable while the input has weak constraints, SaTC raises an alert of the potential vulnerability.

3 Input Keyword Extraction

Given an unpacked firmware, SaTC first extracts potential keywords from the front-end files. We classify keywords into two types based on their usage in the back-end: one type is used to label user input, like the `deviceName` in Listing 1 and we call them *parameter keywords*; another type is to label the handler function, such as `setUsbUnload` in Listing 1, and we call them *action keywords*. We identify input keywords and their types based on the common patterns in different front-end files. We also apply different fine-grained rules to two types of keywords to filter out false positives.

In our current design, we consider three categories of front-end resources: HTML files, XML files, and JavaScript files. Since HTML files have a standard format, we use regular expressions to extract the keywords, such as the values of the `id`, `name`, and `action` attributes. The values of the `action` attributes are treated as action keywords. The XML-based services, such as Simple Object Access Protocol (SOAP) and Universal Plug and Play protocol (UPnP), usually have a fixed format in their XML files to label input data. Hence, we only need to do a pre-analysis and then use regular expressions to extract the keywords. The name of the first-level label in the XML body is treated as an action keyword. The format of JavaScript is ever-changing, and thus the regular expression cannot correctly identify the keywords. Hence, we parse a JavaScript file into an abstract syntax tree (AST) and scan every `Literal` node to extract the value from `value` attributes. If the `Literal` node contains the symbol `/`, we take the string as the action keyword. We further search all `CallExpression` nodes to find the ones that use typical application programming interface (API) as their callee, such as `sendSOAPAction`. The API methods or arguments of the matched nodes are also treated as action keywords. With this method, from the code in Figure 1 our extraction module will get `target`, `goform/setUsbUnload`, and `deviceName`.

The strings collected from HTML, XML and AST contain many fake keywords, which not only bring significant burden to string matching in the next step, but also introduce false positives in the bug detection. For example, string `target` is commonly used in the front-end, but does not have the counterpart in the back-end. To filter the invalid keywords, we designed several rules based on our experience. First, we remove strings with special characters, such as `!` and `@`, which will be escaped when the front-end generates the HTTP request. Second, if a string ends with =, we retain the left-hand part and discard the right-hand side. Symbol = is usually facilitated to concatenate parameters and variables, such as `deviceName=` in Figure 1, where only the parameter name will be reused in the back-end. Third, we filter out strings shorter than a threshold (we use 5 in our work) as the parameter keyword and action keyword usually have non-trivial names.

After filtering, the candidate list may still contain many distractors that are not used as input keywords. To reduce the complexity of the subsequent modules, we use two heuristics to identify and exclude them from the keyword set. If a JavaScript file is referenced by a lot of HTML files, we treat it as a common, shared library, like a charting library. As library files usually do not contain input keywords, we will ignore all candidates from such files. If a keyword is referenced by several front-end files, such as `Button` and `Cancel`, it may be a common string rather than an input keyword. We also remove such keywords from the candidate list.

Border Binary Recognition. In the firmware back-end, a *border* binary exports the device functionalities to the front-end, and meanwhile accepts the user input from the front-end [34]. Therefore, the border binary is a good starting point

```
1  SetWebFilterSettings() {//in binary prog.cgi
2    pcVar1=webGetVarString(wp,"/SetWebFilterSettings/
          WebFilterMethod");
3    iVar2=webGetCount(wp,"/SetWebFilterSettings/WebFilterURLs/
          string#");
4    i = 0;
5    if (iVar2<=i) {
6      /* NVRAM operations */
7      nvram_safe_set("url_filter_mode",pcVar1);
8      nvram_safe_set("url_filter_rule",tmpBuf);
9    }
10 }
11 upload_url_filter_rules() {//in binary rc
12   /* NVRAM operations */
13   iVar1=nvram_get_int("url_filter_max_num");
14   __s1=(char *)nvram_safe_get("url_filter_mode");
15   __src=(char *)nvram_safe_get("url_filter_rule");
16 }
```

Listing 2: Pseudocode of NVRAM Operations. Function call at line 2 is the input entry, which uses a superset of the keyword `WebFilterMethod` to retrieve the input.

for our analysis. Based on the input keywords, SaTC can recognize the border binaries in a short time. Specifically, we extract strings from each back-end binary and try to match them with the input candidate keywords. We treat the binary with the maximum matched keywords as the border binary.

4 Input Entry Recognition

After receiving a request from the front-end, the web server invokes the corresponding handling function to parse the input data. The data-extracting point is the target of the subsequent analysis, and we define it as the *input entry*. The input-entry recognition module detects the entry points in the back-end binaries based on the references to front-end keywords.

$$
\begin{aligned}
s(k_i) &= \begin{cases} k_i \\ concat(k_i, str) \end{cases}, && k_i \in keywords, str\,is\,any\,string \\
L: ret &= foo(s_{k_i}, ...), && k_i \in parameter\_keywords \\
P: ret &= bar(s_{k_i}, ..., \&foo), && k_i \in action\_keywords
\end{aligned}
\tag{1}
$$

Keyword Reference Locator. Equation 1 shows our method to locate input entries from the border binaries in the back-end. s_{k_i} represents a string that either exactly equals to one input keyword k_i, or contains a substring that is k_i. The locator detects the location inside the border binaries that references to the string s_{k_i}. As the handling functions usually use the input keywords to extract the target data from the request, SaTC locates function calls L that take the input keywords as parameters, like `foo("devName")`. Consider our motivating example in Figure 1, the input-keyword extractor identifies the string `deviceName` as a parameter keyword, and recognizes `httpd` as the border binary. While searching the keyword references in `httpd`, as Listing 1 shows, our locator finds the function call to `websGetVar` uses `deviceName` as a parameter (line 9). This function call is treated as a keyword reference location, and thus an input entry. In another example in Listing 2, at line 2 the function call to `webGetVarString` uses the concatenation of string `SetWebFilterSettings` and parameter keyword `WebFilterMethod` as its argument. Therefore, this function call is also an input entry.

Among all keyword references, we prioritize the ones inside the action handlers. Specifically, SaTC searches the function calls P that takes the action keywords and function pointers as arguments. As the action keywords are used to retrieve handlers for particular inputs, we treat the routines specified in the function pointer as the action handler. If some reference points L of parameter keywords are inside these handler functions, we will prioritize exploring L before others. In Listing 1, SaTC locates function `formsetUsbUnload` as the action handler, since the function call at line 3 takes the action keywords `setUsbUnload` and `formsetUsbUnload` as the arguments. Therefore, as a reference point, line 9 in `formsetUsbUnload` will be analyzed before other entries.

$$
L_p: ret = foo(p_i, ...),\ p_i \notin keywords,\ \exists L: dist(L_p, L) < MAX \tag{2}
$$

Implicit Entry Finder. During our experiment, we find several real input entries in the back-end do not have corresponding keywords in the front-end. For example, in Listing 3, function `formSetSambaConf` retrieves several elements from the `data` package, and each string should be treated as a valid input entry, like `password`. However, our input-keyword extraction module finds all keywords except `action` and `usbName`. Without `action` in `data`, line 4 will return a `null` pointer, and the condition in line 9 will always be false. Therefore, in the normal execution, the code injection vulnerability in line 10 will never be triggered. SaTC will also miss this vulnerability. However, attackers can directly send arbitrary requests without the help of the front-end. Therefore, they can provide a malicious request that contains both `action` and `usbName`, and launch the code injection attack.

To mitigate this problem, we propose to take similar code patterns around known input entries into consideration for analysis. Equation 2 shows our idea: if we have identified an input entry L, another function call foo around L will be considered as another input entry as long as foo has the similar code pattern as L. We call the missing keyword p_i here as an *implicit keyword*. This method will help SaTC detect some missing entries and thus mitigates false negatives in the bug detection. In Listing 3, both `action` and `usbName` will be treated as implicit keywords. Once SaTC performs data-flow analysis for them, it will identify the code injection bug easily.

Cross-Process Entry Finder. During the data-flow analysis, we find that some data-flow of input could be interrupted at the process boundary. For example, in Listing 2 the input `pcVar1` is saved into the non-volatile random-access memory (NVRAM) in one process `prog.cgi` (line 7), and then is retrieved in another process `rc` from NVRAM (line 14). Fortunately, we can apply our original insight again to connect data-flows across different processes: the data-saving location and the data-retrieving location usually share the same keyword. In Listing 2, both `prog.cgi` and `rc` take `url_filter_mode` to share `pcVar1`, and use `url_filter_rule` to deliver `tmpBuf`.

```
1  int formSetSambaConf(uint32 user_input) {
2    void *data=user_input;
3    void *usbname;
4    action=Extract(data,"action",&unk_F213C);
5    passwd=Extract(data,"password","admin");
6    premit=Extract(data,"premitEn","0");
7    intport=Extract(data,"internetPort","21");
8    usbname=Extract(data,"usbName",&unk_F213C);
9    if (!strcmp(action,"del")) {
10     doSystemCmd("cfm post netctrl %d?op=%d,string_info=%s"
            ,51,3,usbname);
11   }
12 }
```

Listing 3: Pseudocode of implicit keyword sample. Both `action` and `usbName` are missing in the front-end files. SaTC will identify them as implicit keywords and thus can detect the bug inline 10.

Table 2: Optimizations for efficient taint analysis. We embed three techniques to traditional taint analysis techniques to make it efficient and accurate to analyze embedded devices.

Challenge	**Optimization Method**	**Section**
Balance the efficacy and accuracy	Coarse-grained taint	§5.1
Speed up the path exploration	① Sensitive-trace guidance ② Trace merging	§5.2
Handle infinite loop	Path prioritization	§5.3

Algorithm 1 Taint Specifications

```
function TAINT_SPECIFICATION(Ins, Taint_Map)
    if IS_FUNCALL(Ins) then
        func ← GETFUNCADDR(Ins)
        (retv, params) ← GETPARAMS(Ins)
        taint_set ← HAS_TAINT(params)
        if taint_set == NULL then return
        end if
        if HAS_SUMMARY(func) then TAINT_RULE(func, Taint_Map)
        else if IS_NESTFUNC(func) then
            if IS_POINTER(retv) && IS_USED(retv) then
                T(retv)
            else
                T(params)
            end if
        else
            STEPINTO(func, Taint_Map, taint_set)
        end if
    else
        TAINT_RULE(Ins, Taint_Map)
    end if
end function
```

Based on the shared keyword we can connect different binaries or functions that set or use the same user input. Compared with the original input reference point (line 2), the second retrieval of the user input at line 14 is much closer to the real sink function (skipped in the list). Starting taint-track from this point will significantly save the analysis effort.

SaTC uses the cross-process entry finder (or CPEF) to track the user input across firmware binaries or components. Specifically, it searches various inter-process communication paradigms that use shared strings to label the data, and establishes the data-flow from a set point to a use point. CPEF provides the necessary logic to detect communication paradigms (e.g., NVRAM communication) for sharing data between binaries or functions. It mainly supports two types of inter-process communication paradigms:

- **NVRAM**. NVRAM is a type of RAM that retains data after the host device is power off. It usually keeps the devices' user configurations. The CPEF identifies all `nvram_safe_set` and `nvram_safe_get` functions in order to build cross-process data-flows. In the example of Listing 2, the data dependency between `prog.cgi` and `rc` is built through NVRAM operations.

- **Environment variables**. Processes can share data via environment variables, where the keyword is the variable name. CPEF walks the program path, and collects all function calls that set or get environment variables (e.g., `setenv` or `getenv`). It establishes a data-flow between an environment setter and a getter if they share the same variable name.

5 Input Sensitive Taint Analysis

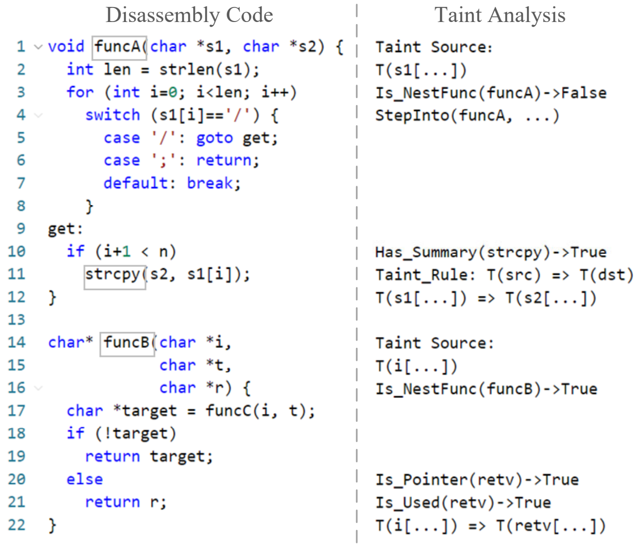
SaTC leverages path exploration and taint analysis technology to track input data to detect dangerous use in the back-end. As Table 2 shows, we design three optimizations based on the unique features of the firmware to balance the efficiency and accuracy, and to speed up the path exploration.

5.1 Coarse-Grained Taint Engine

To perform lightweight data-flow analysis on the user input, we build the taint engine with three principles: (i) the taint source should be related to user input; (ii) it should balance the accuracy and efficiency of the analysis; (iii) it only tracks the data flow from the source to the potential sink.

Taint Source. The taint engine marks taint sources based on the results of the input entry recognition. A taint source can be a variable or a parameter of a target function. As Listing 1 shows, string `deviceName` is used as the parameter of the function `WebsGetVar`, and thus its memory location will be set as the taint source. Since the starting point of SaTC's analysis is a code fragment of a binary, it is usually hard to identify the variable or structure that stores the user-supplied data. However, with our taint source based on input keywords, SaTC could obtain the data flow of the user input data easily.

Taint Specification. SaTC's taint engine propagates taint attributes in the instruction level. We implemented it based on the multi-architecture binary analysis framework angr [38]. The main factor that affects the efficiency and accuracy of taint analysis is the taint specification for function call. To handle the function call appropriately, we first divided the functions into the following categories: summarizable function, general function, and nested function. The summarizable functions are standard library functions related to operations on memory regions, such as `strcpy` and `memcpy`. We can easily summarize the effect of these functions. The general functions contain no function call instructions in its body or only contain branches to summarizable functions (e.g., funcA in Figure 3). The rest



```
            Disassembly Code                    |          Taint Analysis
 1  void funcA(char *s1, char *s2) {            | Taint Source:
 2    int len = strlen(s1);                     | T(s1[...])
 3    for (int i=0; i<len; i++)                 | Is_NestFunc(funcA)->False
 4      switch (s1[i]=='/') {                   | StepInto(funcA, ...)
 5        case '/': goto get;                   |
 6        case ';': return;                     |
 7        default: break;                       |
 8      }                                       |
 9  get:                                        |
10    if (i+1 < n)                              | Has_Summary(strcpy)->True
11      strcpy(s2, s1[i]);                      | Taint_Rule: T(src) => T(dst)
12  }                                           | T(s1[...]) => T(s2[...])
13                                              |
14  char* funcB(char *i,                        | Taint Source:
15              char *t,                        | T(i[...])
16              char *r) {                      | Is_NestFunc(funcB)->True
17    char *target = funcC(i, t);               |
18    if (!target)                              |
19      return target;                          |
20    else                                      | Is_Pointer(retv)->True
21      return r;                               | Is_Used(retv)->True
22  }                                           | T(i[...]) => T(retv[...])
```

Figure 3: Taint specification for different types of functions. The left-hand side shows an example program, while the right-hand side shows the application of taint propagation rules. T(A) indicates the taint tag of A.

of the functions, which contain function call instructions to general functions, are nested functions. In the code of Figure 3, function `strcpy` and `strlen` are treated as summarizable functions. `funcA` does not call non-summarizable functions, and thus it is a general function. `funcB` calls to another general function `funcC` and thus is a nested function.

We designed Algorithm 1 to handle a variety of function calls and instructions. If an instruction `Ins` is not a function call, the taint engine will handle it with the corresponding taint rule (line 19) and update taint map `Taint_Map`. For the data movement instruction, the taint engine will propagate the taint attribute from the source operand to the destination operand. For an instruction `Ins` that calls function `func`, if one actual parameter `param` contains the taint attribute (line 5), the taint engine will track `func`'s effect on the taint map. If `func` is a summarizable function, SaTC treats it as an instruction and applies its taint rule (line 8) that is built based on its semantic. If `func` is a general function, the taint engine will step into its function body and track the data flow from the entry point to the end (line 16). For nested functions, if the taint engine steps into its function body and tracks the data flow for more nested functions, the analysis will be too time-consuming. Hence, we directly propagate the attribute of the argument to its calculation results to balance efficiency and accuracy (line 9∼14). Specifically, if the function returns its results in the return value `retv`, we will label with `retv` with attributes of all arguments; otherwise, we propagate the taint attributes to all pointer arguments.

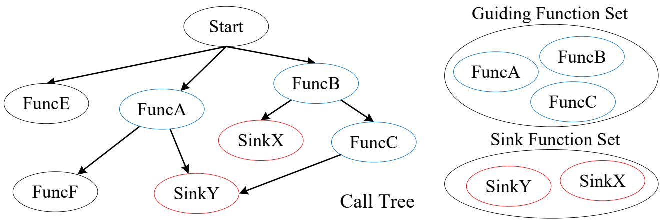


Figure 4: Call tree of an input entry.

5.2 Efficient Path Exploration

SaTC focuses on detecting two classes of vulnerabilities: memory-corruption bugs (e.g., buffer overflows) and command injection. To detect the former class, we first find `memcpy`-like functions within a binary and treat them as sink functions. `memcpy`-like function means a function that is semantically equivalent to `memcpy`, like `strcpy`. Then, if attacker-controlled data unsafely reaches a `memcpy`-like function, like without being sanitized, we raise an alert. For example, for a `memcpy` function, if the attacker-controlled data could affect the value of the source buffer's length, SaTC will raise an alert. To detect the latter class of vulnerabilities, we retrieve the conditions that guard the sink functions (e.g., `system`-like function). Then, we check whether the attacker can construct a proof of concept (PoC) to bypass the constraints. If so, we raise an alert.

Sensitive Trace Guiding. Although previous modules reduce the targets of taint analysis, there could be still a considerable number of input entries that need to be analyzed. To promote the analysis efficiency, SaTC searches the *sink call traces* for each target before exploring any path. A sink call trace represents a function call sequence in the call graph from the input entry to a potential sink function. SaTC searches the sink call traces of a function based on its call tree, which takes the function as the root node. If one function does not contain a call trace, there is no reachable path from this function to the sink point. SaTC will delete all input entries inside this function from the target set. During the exploration, SaTC checks each function call instruction to see whether the target belongs to the call trace. If so, we direct the exploration into the function body.

Call Trace Merging. Starting from one input entry there could be massive call traces, where many call traces share some common paths. To reduce the analysis effort, SaTC merges the call traces with the same input entry as much as possible. To be specific, as Figure 4 shows, we first cluster all traces based on their start points and input keywords. Secondly, we divide all functions in the call traces into two categories: sink functions and guiding functions, and record the types and addresses of the function call instructions. The guiding function represents the dominator of a sink function in a sink call trace. During the exploration, after we encounter a call instruction that jumps to a guiding function, SaTC will step into the function for fine-grained analysis. Otherwise, it

Algorithm 2 Sanitizer Constraints Collection

```
1:  function SANITIZER_CONS_COLLECTION(totalNodes, Max, N, rootNode)
2:      Tree ← ∅
3:      visitedNodes ← ∅
4:      basicNodes ← totalNodes
5:      times ← 0
6:      if !HASSUCCESSOR(rootNode) then return ∅;
7:      end if
8:      while visitedNodes != basicNodes and times < Max do
9:          Tree, visitedNodes ← RANDOM_WALK_SEARCH(rootNode, Tree, visitedNodes, basicNodes)
10:         times++;
11:     end while
12:     for leafNode in Tree.leafNodes do
13:         if num ← GET_FORWARD_NUM(leafNode) > N then
14:             Tree ← REMOVE(basicNodes, leafNode)
15:         end if
16:     end for
17:     Cons ← GET_PATH_CONSTRAINTS(Tree)
18:     return Cons
19: end function
```

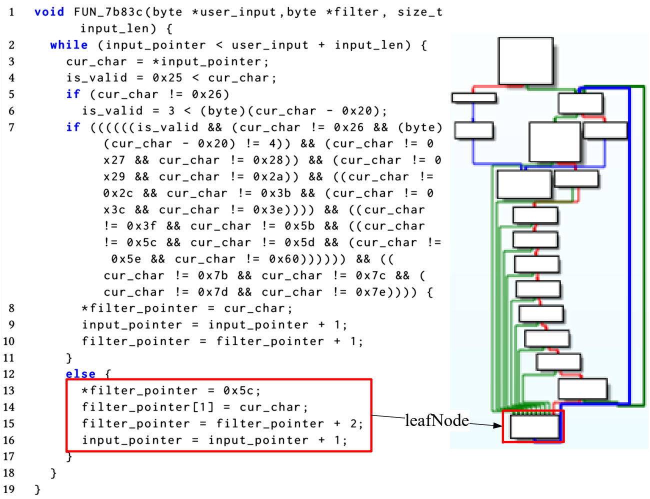
applies the taint specification and strategy (defined in Algorithm 1) to the instruction.

5.3 Path Prioritization Strategy

During our evaluation, we find that some particular functions have a significant impact on the accuracy and efficiency of the path exploration. For example, sanitizer function could result in infinite loops, while parser functions may introduce under-taint problem [34]. To mitigate the negative impact of these particular functions, we identify them and apply special rules. Specifically, if 1) a function contains at least a loop; 2) the number of the function's compare instructions is greater than the threshold; 3) parts of the compare instructions could restrict the content (i.e.,value) of the memory region pointed by the function's arguments, we will treat it as `memcmp`-like function. Based on the amount of information preservation, we can divide these functions into two categories: parsers and sanitizers.

Parsers. A parser function usually contains a loop, such as `funcA` in Figure 3. If the variable `s1` is unconstrained, there will always be a path from the default statement to the head of the `for` loop. Among these paths, only those passing through the first case statement (line 5) would propagate the taint outside the function. In other words, an analysis missing these paths would mistakenly establish that the user input cannot affect variable `s2` and later execution paths. SaTC uses the same solution with KARONTE to handle this problem, which valorizes those paths within a function that potentially propagate the taint also outside the function.

Sanitizers. A sanitizer function either cleans malicious data or warns about the potential threat. Consider the sample in Figure 5, to filter the specific strings, such as `?`, Netgear inserts a sanitizer function `FUN_7b83c` before the `system`-like function. It contains a complex check on the `user_input` (line 7). The `while` loop and comparing operations result in many paths. However, to get the complete constraints on the `user_input`, we are only interested in the longest path.



```
1   void FUN_7b83c(byte *user_input,byte *filter, size_t
          input_len) {
2     while (input_pointer < user_input + input_len) {
3       cur_char = *input_pointer;
4       is_valid = 0x25 < cur_char;
5       if (cur_char != 0x26)
6         is_valid = 3 < (byte)(cur_char - 0x20);
7       if ((((((is_valid && (cur_char != 0x26 && (byte)
              (cur_char - 0x20) != 4)) && (cur_char != 0
              x27 && cur_char != 0x28)) && (cur_char != 0
              x29 && cur_char != 0x2a)) && ((cur_char !=
              0x2c && cur_char != 0x3b && (cur_char != 0
              x3c && cur_char != 0x3e)))) && ((cur_char
              != 0x3f && cur_char != 0x5b && ((cur_char
              != 0x5c && cur_char != 0x5d && (cur_char !=
               0x5e && cur_char != 0x60)))))) && ((
              cur_char != 0x7b && cur_char != 0x7c && (
              cur_char != 0x7d && cur_char != 0x7e)))) {
8         *filter_pointer = cur_char;
9         input_pointer = input_pointer + 1;
10        filter_pointer = filter_pointer + 1;
11      }
12      else {
13        *filter_pointer = 0x5c;
14        filter_pointer[1] = cur_char;
15        filter_pointer = filter_pointer + 2;
16        input_pointer = input_pointer + 1;
17      }
18    }
19  }
```

Figure 5: Pseudocode of sanitizer function. This function tries to remove invalid characters from the user input before the program invokes a `system`-like function.

We use Algorithm 2 to explore the longest path and get the constraints. The `rootNode` is the start basic block of the function. Firstly, it uses `Random_Walk_Search` function to explore the reachable paths and basic blocks `visitedNodes` in limited total times `Max` (line 8). `Random_Walk_Search` randomly chooses a successor from the `rootNode` and recursively calls itself until encountering a `leafNode` and records the `leafNode` (line 9). The `leafNode` represents the basic block that has no successor or the source basic block of the back edge of a loop. Secondly, it scans all `leafNodes` and removes a `leafNode` if the number of its in-degree is more than the threshold value `N` (line 13). Finally, it re-explores the function and outputs the constraints of the longest path (line 17). As Figure 5 shows, the else branch of the complex check is a `leafNode`.

6 Implementation

We implemented the prototype system with around 9800 lines of Python code. The input keyword extraction module is implemented based on standard XML processing library and JavaScript parsing library Js2Py [16]. The input entry recognition module is implemented based on Ghidra library and extended KARONTE's CPF which covers shared tainted variable with NVRAM [10]. The taint engine of the input sensitive analysis is built on top of angr [38], a multi-architecture binary analysis framework. The path selection part is implemented based on Ghidra library [31]. To make SaTC more available for MIPS architecture, we fix the binary loader of angr and the register misuse problem of KARONTE. Now, the prototype system supports multiple architectures, including x86, ARM and MIPS.

Table 3: Dataset of device samples. We selected 39 device samples from six vendors, including 37 routers and two cameras on two architectures. SaTC found 33 previously unknown bugs, and 30 have been confirmed by developers. SizeP and SizeUP represent the average size before and after unpacking, respectively.

Vendor	**Type**	**Series**	**#**	**SizeP**	**SizeUP**	**Arch**	**Bugs**
Netgear	Router	R/XR/WNR	19	38M	192M	ARM32 (LE)	5
Tenda	Router	AC/G/W	9	12M	105M	ARM32 (LE)	10
TOTOLink	Router	A/T	2	5M	60M	ARM32 (LE)	3
D-Link	Router	DIR/DSR	5	8M	123M	MIPS32 (LE)	12
Motorola	Router	C1/M2	2	12M	64M	MIPS32 (LE)	3
Axis	Camera	P/Q	2	60M	700M	ARM32 (LE)	0
Total	**2**	**14**	**39**	**135M**	**1,244M**	**2**	**33**

7 Evaluation

We evaluate SaTC on real-world embedded systems to answer the following research questions:

- **Q1**: Can SaTC find real-world vulnerabilities? How effective is it compared to the state-of-the-art tool? (§7.1)
- **Q2**: Can SaTC accurately detect the input keywords? (§7.2)
- **Q3**: How efficient and accurate is our taint analysis? (§7.3)

Dataset. To evaluate our approach, we selected six major IoT vendors that have provided their device firmware online, specifically, Netgear, D-Link, Tenda, TOTOLink, Motorola and Axis. As shown in Table 3, we eventually collected 39 firmware samples from 14 series, including 37 routers and 2 cameras. Among the samples, 32 adopt the ARM32 architecture, while another seven use the MIPS32 architecture. On average, each firmware is 26 megabytes and totally SaTC has processed 1,024 megabytes.

Existing Tool. We compared our tool with KARONTE [34], the state-of-the-art static bug-hunter for embedded systems. It monitors the interactions between multiple binaries in the firmware back-end, and utilizes taint analysis to track dataflow between binaries to detect vulnerabilities.

Bug Confirmation. Each alert produced by SaTC contains the call trace from the start point to the sink function, and the corresponding input keywords. We distinguish true positives from false positives according to whether the path is reachable in the back-end. If we can manually generate the proof-of-crash (PoC) based on the alert and verify it on the physical device, we consider the true positive as a real bug.

7.1 Real-world Vulnerabilities

As shown in Table 4, SaTC detected 33 previously unknown bugs, and at the time of paper writing, 30 of them have been confirmed by their developers. 25 bugs are command injection vulnerabilities; two of them are buffer overflow bugs; the other six belong to incorrect access control which could result in privacy disclosure. As we define more sinks related to `system`-like functions, our tool found more command injection vulnerabilities than other types. 30 bugs have assigned CVE/CNVD/PSV numbers due to their severe security consequence, while developers are still actively inspecting another one. The last column shows the services where SaTC detects vulnerabilities. Other than the common HTTP protocol, SaTC also supports other services such as UPnP and HNAP. These results show that SaTC can effectively find common vulnerabilities in various network services of embedded systems.

Table 4: Vulnerabilities discovered by SaTC. For the bug type, BoF means buffer overflow; CI represents command injection; IAC indicates incorrect access control. **Ksrc** represents the type of the front-end file where the vulnerability-related keyword is found. **Service** represents the service where the vulnerability occurs.

Vendor	**Device Series**	**Type**	**Bug IDs**	**Ksrc**	**Service**
Netgear	R7000/R7000P	BoF	PSV-2020-0267	HTML	HTTP
			CVE-2020-28373	XML	UPnP
	R6400v2	CI	CNVD-2020-15102	HTML+	HTTP
			CNVD-2020-28091	HTML+	HTTP
	XR300	CI	PSV-2020-0277	HTML	HTTP
Tenda	W20E	CI	CNVD-2019-22866	JS	HTTP
			CNVD-2019-22867	JS	HTTP
			CNVD-2019-22869	HTML	HTTP
		IAC	1 unassigned	JS	HTTP
	G1/G3	CI	CNVD-2020-46058	JS	HTTP
			CNVD-2020-46059	JS	HTTP
	AC15/AC18	CI	CNVD-2020-29725	JS	HTTP
			CNVD-2020-40766	JS	HTTP
			CNVD-2020-40767	JS	HTTP
			CNVD-2020-40768	JS	HTTP
TOTOLink	T10	CI	CNVD-2020-28089	JS	HTTP
	A950RG	CI	CNVD-2020-28090	JS	HTTP
			1 unassigned	JS	HTTP
D-Link	DIR 823G	IAC	CVE-2019-7388	JS	HTTP
			CVE-2019-7389	JS	HTTP
			CVE-2019-7390	JS	HTTP
			CVE-2019-8392	JS	HTTP
	DIR 878	CI	CVE-2019-8312	XML	HNAP
			CVE-2019-8314	XML	HNAP
			CVE-2019-8316	XML	HNAP
			CVE-2019-8317	XML	HNAP
			CVE-2019-8318	XML	HNAP
			CVE-2019-8316	XML	HNAP
	DIR 878 882	IAC	1 unassigned	JS	HTTP
		CI	CNVD-2020-23845	XML	HNAP
Motorola	C1 M2	CI	CVE-2019-9117	JS	HTTP
			CVE-2019-9118	JS	HTTP
			CVE-2019-9119	JS	HTTP
Total		3	**33**	3	3

Comparison with KARONTE. We compared SaTC with the state-of-the-art static analysis tool KARONTE on discovering vulnerabilities.

We use the dataset[1] and experiment result released by KARONTE, which includes four major IoT vendors (i.e., Netgear, TP-link, D-Link, and Tenda) and totally 49 firmware samples. Table 5 shows our evaluation results. SaTC raised 2,084 alerts and out of them, 683 are true positives; KARONTE produced 74 alerts, among which 46 were true positives. The result shows that SaTC can find more true positives than KARONTE. On the design level, SaTC takes a similar method as KARONTE, which both rely on common strings to connect different components of the IoT devices: KARONTE uses common strings between multiple back-end binaries to connect data flow, while SaTC identifies common

[1] https://github.com/ucsb-seclab/karonte#dataset

Table 5: Compared with KARONTE on its dataset. For each vendor we report the device series, the number of firmware samples, the average analysis time (hour), the total number of alerts (#Alert) and the total number of true positives (#TP).

Vendor	Device Series	#Samples	KARONTE			SaTC		
			#Alerts	#TP	Time	#Alerts	#TP	Time
Netgear	R/XR/WNR	17	36	23	17:13 h	1,901	537	16:47 h
D-Link	DIR/DWR/DCS	9	24	15	14:09 h	32	22	1:57 h
TP-Link	TD/WA/WR/TX/KC	16	2	2	1:30 h	7	2	4:13 h
Tenda	AC/WH/FH	7	12	6	1:01 h	144	122	12:19 h
Total	-	49	74	46	33:57 h	2,084	683	35:16 h

Table 6: Compared with KARONTE. We list the analysis time (min), the number of alerts (#Alert) and true positives (#TP).

Compare		AC15	AC18	W20E	878	R6400	R7000	XR300
SaTC	#Alert	10	10	4	22	4	5	10
	#TP	4	4	2	16	4	2	4
	Time	573	576	55	502	1,823	694	1,377
KARONTE	# Alert	17	17	0	0	0	0	0
	# TP	0	0	0	0	0	0	0
	Time	222	210	153	214	2,880	2,880	2,880

identifiers between the front-end and the back-end to locate entries of user input. However, the difference in the final result is significant. As SaTC can analyze front-end files to reveal input entries in the back-end, we can significantly find more analysis points for taint analysis and thus improve the bug-detection capacity. In contrast, common strings identified by KARONTE in back-end binaries cannot guarantee to be related to the user input. Therefore, KARONTE suffers from inefficient analysis and misses a lot of bugs.

We further selected seven new firmware samples from three vendors, specifically, Tenda, D-Link, and Netgear, to confirm the advantage of SaTC on more IoT devices. We ran KARONTE and SaTC until it completes the analysis or time out (2 days), and set SaTC to only detect command injection vulnerability. The result is given in Table 6. SaTC successfully found 36 true positives, while KARONTE could not find any true positive in any samples. For AC15 and AC18, KARONTE provided 17 paths to the sink addresses. We manually verified that all the warnings are false positives. For W20E, KARONTE found no potential vulnerability. KARONTE did not find any border binary in D-Link DIR 878 and thus could not raise any alert. For R6400, R7000 and XR300, KARONTE could not finish analysis within 48 hours. We found it hangs while analyzing a basic block and thus found no vulnerability. These results indicate that SaTC outperforms KARONTE on discovering vulnerabilities in embedded systems.

We further manually checked the alerts that are only found by our tool (in Table 5) and identified the underlying reasons that KARONTE missed them. Firstly, SaTC and KARONTE adopt different features to identify border binaries. Specifically, KARONTE uses the features of instructions and functions to identify border binaries, while SaTC considers the string extracted from the front-end files instead. These different heuristics cause KARONTE and SaTC to select different border binaries. For example, in Tenda AC series, SaTC selects httpd as the border binary while KARONTE selects app_data_center. SaTC raises 144 alerts, which are missed by KARONTE. Secondly, SaTC and KARONTE identify different entry points. KARONTE focuses on the shared data between binaries, while SaTC focuses on the entry points of the user input. For example, SaTC finds the keyword `ed_url` in the border binary httpd of Netgear R6400 that labels the user input, which cannot be found by KARONTE and results in a false negative. As another example, for the keyword `http_user` in Netgear R7900, both KARONTE and SaTC could find the same buffer-overflow bug (i.e., the traces in the reports of SaTC and KARONTE are the same). However, SaTC could find one more buffer-overflow bug because KARONTE misses an entry point related to the string. Finally, KARONTE cannot detect any command injection vulnerabilities as it does not track the data flow from the input entry points to the `system`-like functions. For example, KARONTE misses 12 command injection alerts in Netgear R7300.

In terms of the analysis time, KARONTE and SaTC have their own pros and cons. The analysis time of SaTC depends on the protocols the device uses and the number of sensitive input entry points it extracted. For example, SaTC found more than 31,000 back-end entry points in 17 Netgear samples and found only 779 entry points in nine D-Link samples, and therefore, the average analysis time for the Netgear samples is 14 hours longer than the D-Link samples (shown in Table 5). In contrast, the time spent of KARONTE depends on the number of data keys found in the border binaries, which are used to label the IPC (inter-process communication) paradigms. For example, SaTC found 10,228 sensitive entry points in 7 Tenda samples, but KARONTE found less than 100 data keys. Hence, KARONTE is faster than SaTC on Tenda samples.

Case Study: Command Injection. Listing 4 shows a command injection vulnerability in D-Link DIR 878, detected by SaTC. The front-end HTML file `Network.html` contains an input keyword `SetNetworkSettings/IPAddress` (line 4). Our input entry module detects a reference of the keyword in the border binary `prog.cgi` (line 10). The cross-process entry finder recovers the data dependency between `prog.cgi` and `rc` base on the shared string `SysLogRemote_IPAddress` and finds the entrance of the code fragment that uses the input data in function `FUN_44fa0c` (line 17). The input-sensitive taint analysis module finds a call trace to the sink function `twsystem` at line 28 and raises an alert based on the path exploration result and path constraints (line 25).

Case Study: Incorrect Access Control. We discover incorrect access control vulnerability of a device based on the action keywords identified by SaTC. First, we send requests with action keywords to trigger the corresponding handler functions in the back-end of device. Then, we check the responses and verify whether an API of the device is correctly

```
 1 /* Keywords: SetNetworkSettings/IPAddress
 2    front-end: /cpio-root/.../www/web/Network.html */
 3 function SetResult_3rd(e){ ...
 4    e.Set("SetNetworkSettings/IPAddress",
 5          document.getElementById("lanIP").value)...
 6 }
 7 /* Keywords Reference Point: FUN_43a08c
 8    back-end: /cpio-root/bin/prog.cgi */
 9 void FUN_43a08c(uint32 p) {
10    s=webGetVarString(p,"/SetNetworkSettings/IPAddress");
11    ModifySyslogServerIpNetAddr(s,s_00,&lac,&l9c);
12 }
13 void ModifySyslogServerIpNetAddr (uint32 param_1, ...) {
14    snprintf(acStack72,0x10,"%s",param_1);
15    iVar2 = ModifyIpNetAddr(&local_58,0x10,acStack72);
16    if (iVar2 == 0)
17       nvram_safe_set("SysLogRemote_IPAddress",&local_58);
18 }
19 /* Sink point : /cpio-root/bin/rc */
20 void FUN_44fa0c(void) {
21    /* Located by Cross-Process Entry Finder */
22    pcVar1 = nvram_safe_get("SysLogRemote_IPAddress");
23    iVar2 = strcmp(__s1,"1");
24    if (iVar2 == 0) {
25       if (*pcVar1 != '\0') {
26          memset(acStack112,0,100);
27          sprintf(acStack112,"syslogd -L -R %s",pcVar1);
28          twsystem(acStack112,1);
29 }}}
```

Listing 4: Pseudocode of CVE-2019-8312, a command injection vulnerability detected by SaTC at line 28.

restricted for access. In our data set, we found six incorrect access control vulnerabilities that could result in privacy disclosure. For example, in CVE-2019-7388, D-Link 823G incorrectly restricts access to a resource from an unauthorized actor. An attacker only needs to call an HNAP API `GetClientInfo` remotely and could get the information of all clients in the wireless local network (WLAN), such as IP address, MAC address and device name.

7.2 Accuracy of Keyword Extraction

The `Ksrc` column of Table 4 shows the type of front-end file where the vulnerability-related keyword is found. 20 out of 33 bugs are related to input keywords found in JavaScript files; eight are related to keywords in XML files; four of them rely on the keywords in HTML files. Among 33 bugs, only two are related to the keywords in the `form` component of HTML files (labeled as HTML+ in Table 4). For the bug in XR300, we use the implicit finder to identify the entry that is closer to another normally located entry. The result means all three types of front-end files used in the input keyword extraction (§3) are necessary to locate the input entries. Table 8 shows the number of input keywords selected by each step of SaTC during the evaluation. Only 10% of all strings from front-end files are finally used as input keywords.

False Positive of Parameter Keywords. To understand the false positives of the input keyword extraction (§3), we extend the input entry locators in §4 to find all data-retrieval functions. These functions are commonly used to obtain input data from the request package with the parameter keywords, such as the function `WebsGetVar`. We treat parameter keywords used by these functions as true positives. For other parameter keywords, we apply manual analysis: if they are used to label some user-input from some related requests, we treat them as true positives; otherwise, they are false positives.

Table 7: Categories of the false positives of the keywords. In this table, we list the type (Type) and the sample case (Sample).

Type	Sample
Value of *id* label (HTML)	`id="adv_connect_time"`
Constant string (JavaScript)	`if (typeof(event.pageX) == "undefined")`
Function's parameter (JavaScript)	`R.module("macFilter", view, module)`

As shown in the vPar/tPar column of Table 8, SaTC collects sustainable true positives in parameter keywords, especially for TOTOLink (80%), Tenda samples (69%) and Netgear samples (32%). For devices from D-Link and Motorola, the true positive rate is relatively lower, meaning that SaTC collects more false positive keywords. We manually analyzed these false positives and show the common reasons in Table 7. Most of the false positives are constant strings, function's parameters and values of the *id* labels. We plan to investigate these false positives and will add corresponding methods in the input keyword extraction module to filter them out.

False Positives of Action Keywords. We take a method similar to the above one to check the false positives of action keywords. The key difference is that we only use `register`-like functions to search the true positives, such as `Register_Handler` in Listing 1. The vAct/tAct column of Table 8 shows the result of our verification. For most devices from Tenda, TOTOLink and Motorola, SaTC can achieve higher than 70% true positive rate. For the other devices, the true positive rate is lower, and even reaches zero for two Netgear routers. We manually checked these results and found a common reason that renders SaTC to have a significant false positive rate under our verification method: the real action keywords are not used to register or call handler functions; For example, Netgear R7000 router stored function pointers of all handler functions inside a function call table and merely uses the action keywords to get the index of the associated function in the table. In this way, even if SaTC successfully identified real action keywords, our verification method relying on `register`-like functions cannot confirm their correctness. We plan to identify such code patterns for particular devices, and define specific rules to handle them properly.

False Negative. To understand the false negatives of our bug detection results, we conservatively treat all strings in a border binary as the taint sources and launch data-flow tracking for each of them. Our goal here is to check whether we can effectively find vulnerabilities starting from back-end strings that have no appearance in the front-end. Since this experiment relies on tedious human effort to verify each alert (true positive or not), we randomly select seven devices to conduct the false negative verification. We keep the taint engine running for each device until all strings have been tested, which takes 5 to 113 hours. For all 408 reported alerts, we manually check whether they are true positives or not.

Table 8: Input keywords collected, filtered and used during our evaluation. For each device, we provide the number of front-end files (Input). For input keyword extraction, the table shows the number of unfiltered keywords (str), filtered keywords (fKey) and the analysis time. For border binary recognition, we show the number of all strings in the back-end binaries (strAll), the border binary name (borderBin), and the keywords matched in border binary (borderKey). In the verification part, (vPar) and (tPar) represent numbers of verified and total parameter keywords, while (vAct) and (tAct) represent numbers of verified and total action keywords; % represents the proportion. Other than `httpd`, Netgear samples contain border binaries for other services, such as `upnpd`.

Vendor	**Series**	**Input**	**Keyword Extraction**			**Border Binary Recognition**				**Verification**			
			str	fKey	time(s)	strAll	borderBin	borderKey	time(s)	vPar/tPar	%	vAct/tAct	%
Tenda	AC15	119	7,771	995	254	241,314	httpd	447	51	223/319	69.91	101/128	78.91
Tenda	AC18	119	7,663	984	145	119,537	httpd	447	57	222/319	69.59	101/128	78.91
Tenda	W20E	134	10,581	1,744	102	139,885	httpd	834	102	423/589	71.82	222/245	90.61
Tenda	G1	147	14,241	137	1,952	123,960	httpd	636	75	422/586	72.01	5/56	8.39
Tenda	G3	147	14,241	137	1,952	123,960	httpd	636	75	422/586	72.01	5/56	8.39
Netgear	XR300	864	18,889	4,232	683	517,254	httpd	1,226	1,280	330/1,014	32.54	11/211	5.21
Netgear	R6400	489	5,692	1,729	32	478,005	httpd	887	449	288/706	40.79	10/180	5.56
Netgear	R7000	610	9,421	2,304	167	330,087	httpd	1,132	452	456/920	49.57	0/211	0.00
Netgear	R7000P	607	8,670	2,257	67	467,706	httpd	1,121	579	455/919	49.51	0/201	0.00
D-Link	878	251	26,389	3,415	492	139,948	prog.cgi	735	170	223/735	45.44	140/520	26.92
D-Link	882	252	25,608	3,025	1,149	522,317	prog.cgi	878	670	256/416	61.54	91/461	19.74
D-Link	823G	110	10,200	2,544	370	48,005	goahead	255	78	27/167	16.17	24/87	27.59
TOTOLink	T10	59	6,217	869	231	51,898	system.so	64	24	35/41	85.37	20/23	86.96
TOTOLink	A950RG	73	7,520	1,267	303	53,931	system.so	180	31	53/66	80.30	35/114	30.70
Motorola	C1	105	12,347	2,133	315	90,652	prog.cgi	370	89	44/147	29.93	175/223	78.48
Motorola	M2	103	10,982	1,863	303	83,911	prog.cgi	333	93	38/137	27.74	143/196	72.96

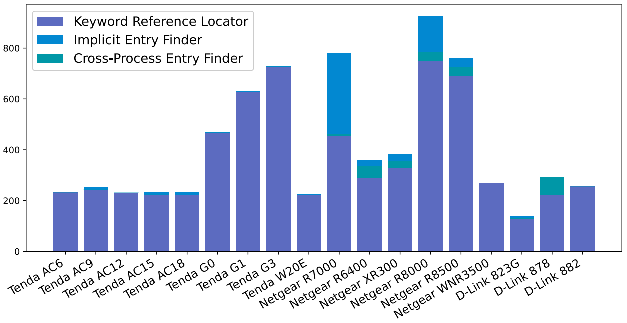


Figure 6: Number of the keywords detected by keyword reference locator, implicit entry finder and cross-process entry finder.

According to our analysis, all alerts related to strings absent in the front-end are confirmed to be false positives, except for two cases in Tenda AC18, which are related to strings `cmdinput` and `data`. The string `cmdinput` does not appear in the front-end, and `data` exists in `main.js` but is filtered by the input keyword extraction module as many front-end files use it (see §3). The result shows that compared to testing all back-end strings in a tedious way, SaTC just introduces very few false negatives (2 out of 408).

Source of True Input Keywords. Figure 6 lists the number of keywords in each sample detected by keyword reference locator, implicit entry finder and cross-process entry finder. We can see that most of the input keywords are collected by the keyword reference locator, especially for Tenda devices. Netgear samples contain relatively more keywords located by implicit entry finder, while D-Link samples contain more keywords related to shared data between different binaries.

Table 9: Performance of Trace Merging. We list the number of the sensitive trace (#Sensitive), the number of the path after trace merging (#Merging) and the ratio of the merged traces (%).

Vendor	**Series**	**Command Injection**			**Memory Corruption**			**SUM**
		#Sensitive	#Merged	%	#Sensitive	#Merged	%	%
Tenda	AC18	207	113	54.59	38917	1,634	4.20	4.47
Tenda	AC15	202	110	54.46	38923	1,638	4.21	4.47
Tenda	W20E	93	48	51.61	955,123	1,287	0.13	0.14
Tenda	G1	45	36	80.00	794,104	1,082	0.14	0.14
Tenda	G3	45	36	80.00	794,104	1,082	0.14	0.14
Netgear	WNR3500	69	22	31.88	1,635	164	10.03	10.92
Netgear	XR300	14,728	718	4.88	24,079	1,363	5.66	5.36
Netgear	R6400	31,605	605	1.91	41,120	1,109	2.70	2.36
Netgear	R7000P	62,840	858	1.37	143,455	2,192	1.53	1.48
Netgear	R8000	19,588	718	3.67	38,929	1,616	4.15	3.99
Netgear	R8500	23,537	528	2.24	35,740	893	2.50	2.40
D-Link	878	17,153	246	1.43	64,075	1,545	2.41	2.20
D-Link	823G	6,811	121	1.78	257,410	313	0.12	0.12
TOTOLink	T10	62	21	33.87	1	1	100.00	34.92
TOTOLink	AR950	95	28	29.47	18	16	88.89	38.93

7.3 Efficacy of Taint Analysis

We further inspect the taint analysis process to understand the benefits of our three optimizations proposed in §5.

Trace Merging. SaTC reduces the number of paths to be explored by merging the call traces with the same input entry (§5.2). Table 9 shows the number of explored paths before and after trace merging. The results confirm that the trace merging strategy is useful: for Netgear, D-Link and Tenda devices that have many sensitive traces to sink functions, SaTC merges more than 89% of redundant paths; for other devices, this technique also merges more than 61% of their start points.

Path prioritization. SaTC found five parser and sanitizer functions in Netgear samples. Three of them are used to encode the character entities. Two of them are used to resolve the

```
1  void formDelVpnUsers(...)
2  {
3    // reference point
4    taint = websGetVar(wp, "vpnUserIndex", byte_E945C);
5    strncpy(sUserIndexCopy2, taint, 0x3Fu);
6    getVpnServerType(sServerType);
7    for (pIndex = (unsigned int8 *)strtok_r((char *)
          sUserIndexCopy2, "\t", (char **)&pSavePtr); pIndex;
          pIndex = (unsigned int8 *)strtok_r(0, "\t", (char **)&
          pSavePtr)) {
8      v6 = atoi((const char*)pIndex);//over-tainting -> v6
9      get_item_in_list("vpn.ser.pptpuser", "&", v6 + 1, 1,
           sUserId);//over-tainting -> sUserId
10     doSystemCmd("cfm post netctrl %s?op=%d,index=%s", (const
           char *)sServerType, 10, (const char *)sUserId);
11   }
12 }
```

Listing 5: Pseudocode of false positive sample.

```
1 int sub_42383C(...) {
2   char* body;
3   char log[0x1388];
4   /*sub_432D28 extracts message body from the request.*/
5   sub_432D28(body);
6   memset(log, 0, 0x1388);
7   snprintf(log, 0x1387, "echo '%s' >/var/hnaplog", body);
8   system(log);
9 }
```

Listing 6: Pseudocode of CVE-2019-7298. The device uses the Home Network Administration Protocol (HNAP) to provide service for users to configure and manage it. However, while handling the POST request of HNAP, the function sub_42383C does not check and filter the message body and writes it into a log file directly via executing *echo* command (line 8). A malicious message body will result in command injection vulnerability.

input string, escape the characters and generate the internal variables.

Taint Engine. For all firmware samples in Table 9, SaTC raised 101 alerts, 46 of them are true positives. We manually analyzed twenty false positives in the alerts. As Listing 5 shows, some over-taint problems occur because of missing abstracts for the common functions, such as `atoi`. The taint state of the character variable `pIndex` is passed to an integer variable (line 8), which is used as an index to extract data from a list and store the data into a string `sUserId` (line 9). SaTC raises the alert because `v6` finally affects the `doSystemCmd` function (line 10). In fact, the attacker cannot control the string through interface keywords `vpnUserIndex`.

8 Discussion

In this section, we discuss the ability and limitation of SaTC, and explore the improvement direction in the future.

Circle of Competence. Our evaluation shows that shared keywords between different components of IoT devices can effectively bridge points inside complicated data-flows. This short path saves a lot of analysis effort and thus improves the efficacy of bug finding. In fact, we can extend SaTC to detect bugs in other systems, as long as they use shared keywords to deliver data. For example, environment variables are widely used in malware applications as a stealthy way to share information. In this case, we can use the same variable names to find the connection between different malware processes and to help detect critical operations in malware.

Implicit Data Dependency. During our evaluation we find several cases that the input entry in the back-end programs does not have a corresponding keyword in the front-end. Our implicit entry locator (§4) helps SaTC mitigate this issue to detect more implicit input entries. However, there are cases that even the implicit entry locator fails to build connections, where SaTC will miss potential bugs associated with these entries. For example, in an old vulnerability CVE-2019-7298, the back-end program directly reads data from the HTTP message without using any keyword (more details in Listing 6), and thus SaTC will miss this bug. In another example, the firmware of D-Link 823G uses function `apmib_set` and `apmib_get` to share data between different functions, without using any keyword. SaTC will miss the associated vulnerability CVE-2019-7297. We will analyze more cases and attack surfaces and try to find hidden patterns to build the relationship between front-end and back-end, so as to enhance the ability to discover vulnerabilities.

Efficiency v.s. Completeness. Analysis efficiency and bug completeness are two key factors of any bug detection mechanisms. Compared to previous work like KARONTE, SaTC trades the completeness of bug finding for the analysis efficiency. On the one hand, the method will help us detect the vulnerabilities related to the front-end in a more timely manner. According to our evaluation in §7.2, it requires five times more effort to test all potential data entries of the back-end in a brute-force way. On the other hand, our tool may result in false negatives if the back-end entries do not have common strings associated with the front-end, or the entries cannot be detected via the heuristic methods. Fortunately, the empirical evaluation shows that our method introduces very few false negatives for seven devices from two vendors. Therefore, SaTC achieves an empirically reasonable balance between the analysis completeness and the bug-finding efficiency.

Encryption and Obfuscation. As the majority of the security threats in IoT devices exist at the application layer and the network layer, parts of the IoT device manufacturers adopt code encryption or obfuscation to protect intellectual property from reverse engineering attacks [2, 6, 15, 20, 47]. These obfuscation techniques will limit the capabilities of SaTC. For example, the string encryption technique could hinder SaTC from building the relationship between the front-end and the back-end. We leave the solutions for dealing with these obfuscation techniques to future work.

To measure the applicability of SaTC regarding the concern of encryption and obfuscation, we conducted an empirical evaluation. Specifically, we collected 186 widely used home Wi-Fi routers from seven leading vendors and inspected them to find encryption and obfuscations. We found that only four out of 186 devices are protected with encryption and all of them are D-Link routers. After we manually decoded these four samples, SaTC can handle them just as other devices. We can use off-the-shelf unpacking tools (e.g., binwalk) to unpack all devices except one failure due to the unsupported filesystem (Tenda AC11). Only one device, specifically NetGear R6400 v2, uses obfuscation to protect parts of the front-end JavaScript code, and SaTC failed to extract any keywords from these JavaScript files. However, the HTML files are not obfuscated, where SaTC still can extract many useful keywords and successfully found two command injection vulnerabilities. Overall, encryption and obfuscation techniques have not been widely used in real-world IoT devices, and SaTC is still able to discover vulnerabilities for a large number of firmware samples. We plan to use existing deobfuscation approaches [4, 19, 29] to make SaTC more applicable.

9 Related Work

Instead of listing all related work, we focus our discussion on the most related ones: dynamic and static methods of vulnerability discovery for firmware, and taint tracking.

Dynamic Analysis. Many works [22, 48, 54, 55] use fuzzing technology to detect vulnerability in IoT devices. SRFuzzer [52] is an automated fuzzing framework for testing physical SOHO (small office/home office) routers, which needs to capture a large number of web requests from the running devices firstly and then could model the user-input semantics to generate test cases. FIRMADYNE [7] is a state-of-the-art firmware emulation framework, which designed for automated dynamic analysis for a large-scale embedded firmware. Although FIRMADYNE is promising, its emulation rate of network reachability and web service availability is considerably low. FIRMAE [23] uses several heuristics to address the problems and increases the emulation success rate. However, it can only handle observed cases and may not apply to new devices and new configurations. IoTFuzzer [8] tries to find memory corruption vulnerabilities in IoT devices via their official apps, therefore it is firmware-free. However, it's trapped in the coverage of code and attack surface, which is a common challenge for dynamic fuzzing analysis. FIRM-AFL [53] is a greybox fuzzer for IoT devices via emulating the target firmware. However, it's hard for researchers to achieve a faithful emulation with various kinds of CPU architectures.

Static Analysis. Static analysis-based techniques are very common in the field of IoT vulnerability detection. KARONTE [34] leverages static analysis techniques to perform multi-binary taint analysis. However, the researchers only focus on back-end binaries and ignore the user-input context stored in the front-end files, which will cause a large number of false negatives. Firmalice [37] provides a framework for detecting authentication bypass vulnerabilities in binary firmware based on symbolic execution and program slicing. However, it suffers from overwhelming the constraint solver. FIE [17] utilizes the symbolic execution to analyze open-source MSP430 firmware programs. However, complete analyses are intractable for some firmware and various sources of imprecision in the analysis may lead to false positives or false negatives.

Taint Tracking. Several prior works [5, 24, 34] use taint analysis to discover the vulnerability in IoT devices. DTaint [9] focuses on the data generated by `recv` and other similar functions, but it ignores the semantic of the front-end files. CryptoREX [51] only identifies the crypto misuse problem of IoT devices. Some researchers [12, 27, 28] focus on enhancing the availability of taint analysis. TaintInduce [11] tries to increases the accuracy of individual propagation rules via learning platform-specific taint propagation rules from pairs of instructions. Greyone [18] proposes a fuzzing-driven taint inference solution FTI, which is utilized to get more taint attributes as well as the precise relationship between input offsets and branches. Neutaint [36] uses neural program embeddings to track information flow, and utilizes symbolic execution to generate training data with high quality to improve the flow coverage. However, accumulated errors and large overhead are still big challenges for Dynamic Taint Track.

10 Conclusions

We propose SaTC, a novel approach to detect security vulnerabilities in embedded systems. Based on the insight that variable names are commonly shared between front-end files and back-end functions, SaTC precisely identifies the input entry in the back-end binaries. Then, it utilizes our taint engine customized for embedded systems to efficiently detect dangerous use of untrusted input. SaTC has successfully discovered 33 zero-day software bugs from 39 firmware samples, and 30 of them have been assigned CVE/CNVD/PSV IDs. Our evaluation result shows that SaTC outperforms the state-of-the-art tool on discovering bugs in firmware samples.

Acknowledgement

We would like to thank our shepherd, Dr. Kevin Butler, and the anonymous reviewers of this work for their helpful feedback. We thank Yue Liu, Yuwei Liu, Minghang Shen, Huikai Xu, and Chutong Liu for their valuable feedback on earlier drafts of this paper. This research is supported, in part, by National Natural Science Foundation of China under Grant No. U1836113, 61772308, 61972224 and U1736209, Beijing Nova Program of Science and Technology under grant Z191100001119131, BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009, National Key Research and Development Program under Grant 2019QY0703, and Science and Technology Commission of Shanghai Municipality Research Program under Grant 20511102002. All opinions expressed in this paper are solely those of the authors.

References

[1] Eirini Anthi, Lowri Williams, Małgorzata Słowińska, George Theodorakopoulos, and Pete Burnap. A supervised intrusion detection system for smart home iot devices. *IEEE Internet of Things Journal*, 6(5):9042–9053, 2019.

[2] Kimia Zamiri Azar, Farnoud Farahmand, Hadi Mardani Kamali, Shervin Roshanisefat, Houman Homayoun, William Diehl, Kris Gaj, and Avesta Sasan. COMA: Communication and Obfuscation Management Architecture. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 181–195, 2019.

[3] Suman Sankar Bhunia and Mohan Gurusamy. Dynamic attack detection and mitigation in IoT using SDN. In *2017 27th International telecommunication networks and applications conference (ITNAC)*, pages 1–6. IEEE, 2017.

[4] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 643–659, Vancouver, BC, 2017. USENIX Association.

[5] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. Sensitive Information Tracking in Commodity IoT. In *Proceedings of the 27th USENIX Security Symposium*, pages 1687–1704, 2018.

[6] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Embedded software security through key-based control flow obfuscation. In *International Conference on Security Aspects in Information Technology*, pages 30–44. Springer, 2011.

[7] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. FIRMADYNE: Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 23th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2016.

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IOTFUZZER: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2018.

[9] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. DTaint: Detecting the Taint-style Vulnerability in Embedded Device Firmware. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 430–441. IEEE, 2018.

[10] Per Christensson. NVRAM Definition. https://techterms.com/definition/nvram, 2010.

[11] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. One Engine to Serve'em All: Inferring Taint Rules without Architectural Semantics. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2019.

[12] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[13] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 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*, pages 437–448, 2016.

[14] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[15] Benjamin Cyr, Jubayer Mahmod, and Ujjwal Guin. Low-cost and secure firmware obfuscation method for protecting electronic systems from cloning. *IEEE Internet of Things Journal*, 6(2):3700–3711, 2019.

[16] Piotr Dabkowski. Js2Py: JavaScript to Python Translator. https://github.com/PiotrDabkowski/Js2Py, 2020.

[17] Drew Davidson, Benjamin Moench, Somesh Jha, and Thomas Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium*. USENIX Association, 2013.

[18] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data Flow Sensitive Fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, Boston, MA, 2020.

[19] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 694–707, 2020.

[20] Ujjwal Guin, Swarup Bhunia, Domenic Forte, and Mark M Tehranipoor. Sma: A system-level mutual authentication for protecting electronic hardware and firmware. *IEEE Transactions on Dependable and Secure Computing*, 14(3):265–278, 2016.

[21] Craig Heffner. Binwalk - Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk, 2014.

[22] Yikun Jiang, Wei Xie, and Yong Tang. Detecting Authentication-bypass Flaws in a Large Scale of IoT Embedded Web Servers. In *Proceedings of the 8th International Conference on Communication and Network Security*, pages 56–63, 2018.

[23] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual Computer Security Applications Conference*, ACSAC '20, page 733–745, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Amit Mandal, Pietro Ferrara, Yuliy Khlyebnikov, Agostino Cortesi, and Fausto Spoto. Cross-Program Taint Analysis for IoT Systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, page 1944–1952, New York, NY, USA, 2020. Association for Computing Machinery.

[25] Vincentius Martin, Qiang Cao, and Theophilus Benson. Fending off IoT-hunting attacks at home networks. In *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*, pages 67–72, 2017.

[26] Microsoft. Data Execution Prevention (DEP), 2006. http://support.microsoft.com/kb/875352/EN-US/.

[27] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled Offline Symbolic Taint Analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 308–319. IEEE, 2016.

[28] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined Symbolic Taint Analysis. In *Proceedings of the 24th USENIX Security Symposium*, pages 65–80, 2015.

[29] Omid Mirzaei, José María de Fuentes, J Tapiador, and Lorena Gonzalez-Manzano. Androdet: An adaptive android obfuscation detector. *Future Generation Computer Systems*, 90:240–261, 2019.

[30] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, pages 3–4, 2005.

[31] NSA. Ghidra. https://github.com/NationalSecurityAgency/ghidra, 2019.

[32] Palo Alto Networks. 2020 Unit 42 IoT Threat Report. https://iotbusinessnews.com/download/white-papers/UNIT42-IoT-Threat-Report.pdf, 2020.

[33] PaX Team. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[34] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[35] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In *Proceedings of the 31st IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[36] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 364–380, Los Alamitos, CA, USA, 2020. IEEE Computer Society.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *the 22nd Annual Network and Distributed System Security*, San Diego, California, USA, February 2015.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

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

[40] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[41] Web-Release. Multiple Netgear Routers are Vulnerable to Arbitrary Command Injection. https://www.kb.cert.org/vuls/id/582384/, 2016.

[42] Web-Release. Securing IoT Devices: How Safe is Your Wi-Fi Router? https://www.theamericanconsumer.org/wp-content/uploads/2018/09/FINAL-Wi-Fi-Router-Vulnerabilities.pdf, 2018.

[43] Web-Release. Targeted Attacks Now Moving into the IoT and Router Space. https://us.norton.com/internetsecurity-emerging-threats-targeted-attacks-moving-into-iot-router.html, 2018.

[44] Web-Release. Gartner Says 5.8 Billion Enterprise And Automotive IoT Endpoints Will Be In Use In 2020. https://web-release.com/gartner-says-5-8-billion-enterprise-and-automotive-iot-endpoints-will-be-in-use-in-2020, 2019.

[45] Web-Release. ISTR 2019: Internet of Things Cyber Attacks Grow More Diverse. https://symantec-enterprise-blogs.security.com/blogs/expert-perspectives/istr-2019-internet-things-cyber-attacks-grow-more-diverse, 2019.

[46] Web-Release. Home Router Security Report 2020. https://www.fkie.fraunhofer.de/content/dam/fkie/de/documents/HomeRouter/HomeRouterSecurity_2020_Bericht.pdf, 2020.

[47] Dixing Xu, Mengyao Zheng, Linshan Jiang, Chaojie Gu, Rui Tan, and Peng Cheng. Lightweight and unobtrusive data obfuscation at iot edge for remote inference. *IEEE Internet of Things Journal*, 7(10):9540–9551, 2020.

[48] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. Poster: Fuzzing IoT Firmware via Multi-stage Message Generation. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, pages 2525–2527, 2019.

[49] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2015.

[50] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *the 21st Annual Network and Distributed System Security Symposium*, volume 14, pages 1–16, San Diego, California, USA, 2014.

[51] Li Zhang, Jiongyi Chen, Wenrui Diao, Shanqing Guo, Jian Weng, and Kehuan Zhang. CryptoREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, pages 151–164, 2019.

[52] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. 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*, pages 544–556, 2019.

[53] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *Proceedings of the 28th USENIX Security Symposium*, pages 1099–1114, Santa Clara, CA, 2019. USENIX Association.

[54] Yaowen Zheng, Zhanwei Song, Yuyan Sun, Kai Cheng, Hongsong Zhu, and Limin Sun. An Efficient Greybox Fuzzing Scheme for Linux-based IoT Programs Through Binary Static Analysis. In *Proceedings of the 38th International Performance Computing and Communications Conference*, pages 1–8. IEEE, 2019.

[55] Lipeng Zhu, Xiaotong Fu, Yao Yao, Yuqing Zhang, and He Wang. FIoT: Detecting the Memory Corruption in Lightweight IoT Device Firmware. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering*, pages 248–255. IEEE, 2019.