IoTCID: A Dynamic Detection Technology for Command Injection Vulnerabilities in IoT Devices

Hao Chen[1], Jinxin Ma[2], Baojiang Cui[3], Junsong Fu[4]
School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, China[1,3,4]
China Information Technology Security Evaluation Center, Beijing, China[2]

Abstract—**The pervasiveness of IoT devices has brought us convenience as well as the risks of security vulnerabilities. However, traditional device vulnerability detection methods cannot efficiently detect command injection vulnerabilities due to heavy execution overheads or false positives and false negatives. Therefore, we propose a novel dynamic detection solution, IoTCID. First, it generates constrained models by parsing the front-end files of the IoT device, and a static binary analysis is performed towards the back-end programs to locate the interface processing function. Then, it utilizes a fuzzing method based on the feedback from Distance Function, which selects high-quality samples through various scheduling strategies. Finally, with the help of the probe code, it compares the parameter of potential risk functions with samples to confirm the command injection vulnerabilities. We implement a prototype of IoTCID and evaluate it on real-world IoT devices from three vendors and confirm six vulnerabilities. It shows that IoTCID are effective in discovering command injection vulnerabilities in IoT devices.**

Keywords—Firmware vulnerability mining; command injection; dynamic detection

I. Introduction

With the development of the Internet and information technologies, IoT devices are extensively used in our life [1], and attacks against IoT devices have been emerged in recent years [2]. The reason is that web interfaces are usually exposed to users to manage the devices, which contain exploitable vulnerabilities.

There are some unique features of the threats caused by vulnerabilities in IoT devices compared to which in PCs or in servers. For example, most existing security mechanism or antivirus products are not available in IoT devices due to the limit of cost and power of IoT devices, which makes it easier to perform further exploits towards certain vulnerabilities [3]. Moreover, one vulnerability may have huge influence on thousands of devices, for the devices from the same vendor usually have similar firmware [4].

Command Injection Vulnerability is one of the most effective and commonest vulnerabilities in IoT devices [14]. Attackers can exploit the target IoT devices through this vulnerability by injecting additional commands into the program. Moreover, the system commands provided by attackers are usually executed with the highest authority in IoT devices.

However, due to the complexity and the specificity of the IoT devices, existing tools cannot effectively detect Command Injection Vulnerability. For example, Dynamic analysis tools, like fuzzing [5, 7, 9], requires valid communication formats to generate fuzzing samples and can only reach a small portion of all the provided interfaces while static analysis tools, like KARONTE [11] and SaTC [12], cannot efficiently generate interaction paradigms between the front-end files and back-end programs, leading to a lot of false positives which requires further manual check.

Therefore, to ensure the safety and reliability of IoT devices, it is urgent to develop security analysis technology towards IoT devices. In this paper, focusing on Command Injection Vulnerability, we propose a novel dynamic detection technology, IoTCID, to effectively detect command injection vulnerabilities in IoT devices.

Inspired by SaTC 12], in order to generate samples in valid communication formats and to cover interfaces as many as we can, IoTCID first performs a logic analysis to the front-end files which interact with the back-end programs to generate constrained models. It utilizes a novel scheduling strategies based on Distance Function to improve the efficiency of command injection vulnerability detection. We design and implement a prototype of IoTCID and evaluate its efficacy through a set of experiments based on real-world IoT devices and confirm six command injection vulnerabilities. It shows that IoTCID is effective in discovering command injection vulnerabilities in IoT devices.

In summary, our major contributions are as follows:

1) We present a dynamic detection technique towards Command Injection Vulnerabilities based on the logic analysis to front-end files and intelligent feedbacks from the back-end programs.

2) We design and implement a constrained model generation technique based on the logic analysis to front-end files, providing a valid format for the generation of fuzzing samples.

3) We design and implement scheduling strategies based on Distance Function feedback to concentrate resources on the fuzzing samples that may cover risk functions in back-end programs.

The rest of the paper is organized as follows. We first summarize related work in recent years in Section II. We then present an overview of IoTCID, and give a detailed description on design and implementation of each component of IoTCID in Section III. We demonstrate the efficacy of IoTCID through a set of experiments and present a vulnerability detection case in Section IV. At last, Section V concludes this paper and show the future work of IoTCID.

II. Related Work

In recent security research on IoT devices, fuzzing is the most discussed technique. The general process of fuzzing is to detect the state of the testing program and guide the generation of fuzzing samples with provided feedbacks.

Chen J et al. propose and implement a generation-based firmware fuzzing method, IoTFuzzer[5], which detects vulnerabilities related to memory in IoT devices by analyzing the corresponding Android application. In view of the shortcomings of IoTFuzzer, DIANE[6] proposed a new method for generating fuzzing samples, which is based on the target fuzzing points in the APP that are located before data conversion and after input validation. However, both IoTFuzzer and DIANE conduct black-box fuzzing directly on real devices, limited to providing guidance and feedbacks based on the testing samples.

Zhang Y et al. propose SRFuzzer[7], which mutates the collected network traffic and detects the state of the fuzzing process according to the response-based monitor, routing-based monitor, and signal-based monitor. However, it would be difficult for SRFuzzer to cover the corresponding interface functions without the network traffic in advance.

FirmFuzz[8] runs the target firmware through simulation and collects payloads of different vulnerabilities for fuzzing tests. Zheng Y et al. propose FIRM-AFL[9] to enhance process simulation to fuzz the IoT firmware. However, these methods are all subject to valid inputs.

Although command injection is a common and powerful threat, related detection is less discussed in IoT security research.

Commix[10] is a tool that can automatically detect and exploit command injection vulnerability towards web applications. It sends a data packet attached with a command injection attack vector, and compares the response of the web application with the expected result to determine whether there is a command injection vulnerability. However, Commix needs to collect network traffic in advance, and it makes determine according to the response of the target. When the network delay cannot be guaranteed, there will be a certain false positive, which cannot intuitively reflect the location of the command injection vulnerability.

KARONTE[11], a static analysis framework for embedded firmware, which can detect vulnerabilities caused by its communication by modeling and tracking the interactions between binary programs. However, KARONTE cannot effectively detect command injection vulnerabilities because it does not track the data flow from input entry points to system-like functions. Aiming at the shortcomings of KARONTE, Chen L et al. propose a novel static taint technique, SaTC[12], to effectively detect security vulnerabilities in web services provided by embedded devices. It mainly locates the communication process between front-end files and back-end programs based on the strings used in the front-end web interface, and applies targeted data flow analysis to accurately detect possible vulnerabilities. However, SaTC uses a clustering algorithm to extract the strings interacting between front-end files and back-end programs, and cannot generate an effective input model for the web interface. Besides, it requires additional manual analysis to comfirm the result and eliminate false positives.

In a word, it remains problems in the use of the above detection technologies towards IoT devices. For example, the fuzzing detection technology mainly focuses on memory corruption vulnerabilities, and are subject to the input of valid format while static analysis tools may have low detection efficiency due to excessive analysis. Therefore, aiming at the command injection vulnerability of IoT devices, based on the logic analysis to front-end files and intelligent feedbacks, we propose a dynamic detection model IoTCID, which makes up for the shortcomings of the current command injection detection technology for IoT devices and improves the efficiency and accuracy of command injection vulnerability detection.

III. Methodology

Generally, IoT devices provide user management interfaces, which are mainly composed of front-end files and back-end programs. Front-end files include HTML, Javascript while back-end programs are generally executable binary files. IoTCID is proposed based on the workflow of the front-end files and back-end programs, as Fig. 1 provides the overview of IoTCID. It first generates constrained models by parsing the front-end files of the IoT device, and then performs binary static analysis on the back-end programs to locate the interface processing function. Then, IoTCID selects high-quality fuzzing samples according to various scheduling strategies based on the feedback from Distance Function. The selected samples are given more mutation time slices and priorities, which makes concentration on the interface process functions that may exist command injection vulnerabilities. Finally, IoTCID confirms the command injection vulnerabilities combined with the fuzzing samples and the parameters of risk functions detected by the probe code.

A. Constrained Model Generation

We propose a technology of constrained model generation based on the logic analysis to front-end files. Through syntax analysis, it generates a corresponding abstract syntax tree according to the front-end file, and extracts the variable reference chains of the abstract syntax tree as well as the variables during the interaction process between the front-end files and the back-end programs. The generated models are used as the format of the fuzzing samples, mainly including the URL, the request type, and the request keywords.

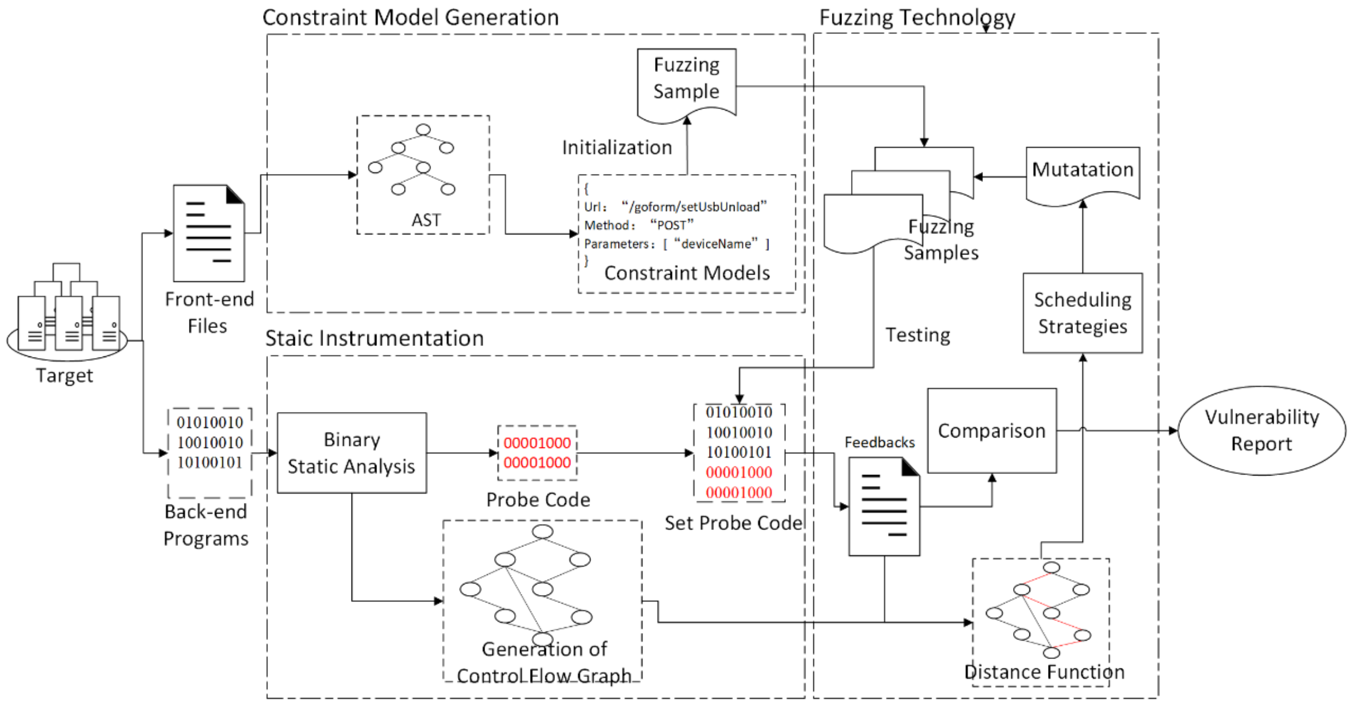


Fig. 1. Structure of IoTCID.

Algorithm 1 shows the analysis process of the function call and variable reference chain of the front-end files, where funcNode is the result of syntax analysis and varName is the parameter of the funcNode. During the analysis, there are two types of function parameters: one of which is directly passed in text value while the other is indirectly passed in variable name. For the direct one, IoTCID uses the regular expression to match and obtain the text value of the parameter. For the indirect one, the text value can be extracted through a recursive method in related function scope.

Algorithm 1: Analysis of Parameters during Interaction

```
Input: funcNode && varNames
Output: Constrained models
1. function GET_VALUE(funcNode, varName)
2.    param ← []
3.    if varName.type == "Literal" then
4.       param.append(ANALYSIS(
             funcNode.expression, varName));
5.       return param
6.    else
7.       funcNode = FIND_PARENT(funcNode);
8.       if funcNode == '' then
9.          return []
10.      end if
11.      GET_VALUE (funcNode, varName);
12.   end if
13. end function
```

B. Static Instrumentation

IoTCID obtains the information that may trigger the command injection vulnerability function in the back-end programs through binary static analysis, and sets the probe code to obtain the performing state of the fuzzing samples. We implement a static analysis technique for back-end binary programs. It obtains information about risk functions such as execve and system which may trigger command injection vulnerabilities. Therefore, we can record the performing paths of fuzzing samples based on binary static instrumentation technology and control flow analysis of the back-end programs.

1) Acquisition of risk function: The purpose of acquiring the potential dangerous function is to locate the interface in the back-end programs, which processes the requests from the front-end files according to the URL extracted in Part A, Section III, and construct its control. flow graph. We further obtain the necessary data by analyzing the header of back-end binary programs, the entry point and relevant segments. Finally, we use Capstone [13] to disassemble the code to obtain the information of the target function and related code blocks as well as building a control flow graph.

2) Generation of probe code: The probe code collects the performing state of fuzzing samples during the execution process and provides feedbacks to the monitor system. Based on the control flow graph of the risk functions, we set the probe code to provide information feedback of the basic blocks, including the address information and the parameters of the risk functions. The trampoline mechanism is used during the generation of probe code to make association between the set point and the risk functions, and provides the necessary environment preparation for the normal execution of the original program.

C. Fuzzing Technology

We apply distance function to the fuzzing technology to select high-quality fuzzing samples according to various scheduling strategies. While IoTCID is sending fuzzing samples to the back-end program for detecting command injected vulnerability, it selects high-quality one and gives them more mutation time and priorities based on the result of measuring the feedbacks of fuzzing samples by distance function, which concentrates resources on the fuzzing samples that may cover risk functions in back-end programs, improving the efficiency of command injection vulnerability detection. Finally, combined with the parameters of risk functions and fuzzing samples, IoTCID makes checks on whether there is a suspicious point of command injection vulnerability. Obviously, the distance function and the scheduling strategies are the cores of the fuzzing technology.

1) Distance function: The weight of basic blocks, the edge vector of basic blocks and the distance of samples are three components of the distance function. The weight of basic blocks and the edge vector of basic blocks are calculated in the process of binary static analysis while the distance of samples calculation is calculated in the fuzzing process. Table I lists the variables and their meanings of the distance function.

We define the count of successors of Basic Block B which contain the risk functions as the weight of basic block B, as Equation (1) shows.

$$Weight_B = \begin{cases} \mathrm{Sum}(B, Func)\,, & Func \notin B \\ W_{Max} \,, & Func \in B \end{cases} \tag{1}$$

In Equation (1), $Weight_B$ is the weight of Basic Block B while Sum() is a function that calculates the count of successors of basic block b which contain risk functions. Moreover, if the risk function is in basic block B, $Weight_B$ is recorded as W_{Max}. After calculating the weight of related basic blocks, we infer the edge from Basic Block A to Basic Block B while Basic Block A is the predecessor of Basic Block B.

$$\left|\overrightarrow{Edge_{a,b}}\right| = Weight_b \tag{2}$$

During the process of fuzzing, combined with the probe code we set before, IoTCID obtains the execution path of the fuzzing sample and calculates the function distance according to the control flow graph to evaluate the fuzzing sample with $Score_{test}$.

$$Score_{test} = \frac{\sum\left|\overrightarrow{Edge_{a,b}}\right|}{Count_{test}} \tag{3}$$

2) Scheduling strategy: Therefore, towards one specific interface, we implement our scheduling strategy according to the distance function. It can be inferred that a fuzzing sample with higher $Score_{test}$ is more likely to trigger the risk functions, so that the mutation resources should be concentrated on these high-quality fuzzing samples, improving the efficiency of command injection vulnerability detection.

Moreover, the following situations should be paid more attention on the basis of experience and practical situations. Details are shown in the control flow graph in Fig. 2.

Case 1. Supposed there is a risk function called in the basic block Target. The execution path of one fuzzing sample shows like Start->Basic Block 5->Basic Block 5->…Target-> End, where there is a loop in the path. Since the fuzzing sample can eventually traverse the basic block Target, we should avoid double counting when calculating $Count_{test}$ and $\sum\left|\overrightarrow{Edge_{a,b}}\right|$.

TABLE I. VARIABLES AND MEANINGS IN DISTANCE FUNCTION

Symbol	**Name**	**Meaning**
$Weight_B$	The weight of a basic block	The importance of Basic Block B in the control flow
$\overrightarrow{Edge_{a,b}}$	The edge vector of basic blocks	The edge vector from Basic Block A to Basic Block B while Basic Block A is the predecessor of Baisc Block B.
$Count_{test}$	The count of edges	The count of edges traversed by the sample before triggering the target basic block
$Score_{test}$	The score of a sample	The count of valid edges traversed by the sample

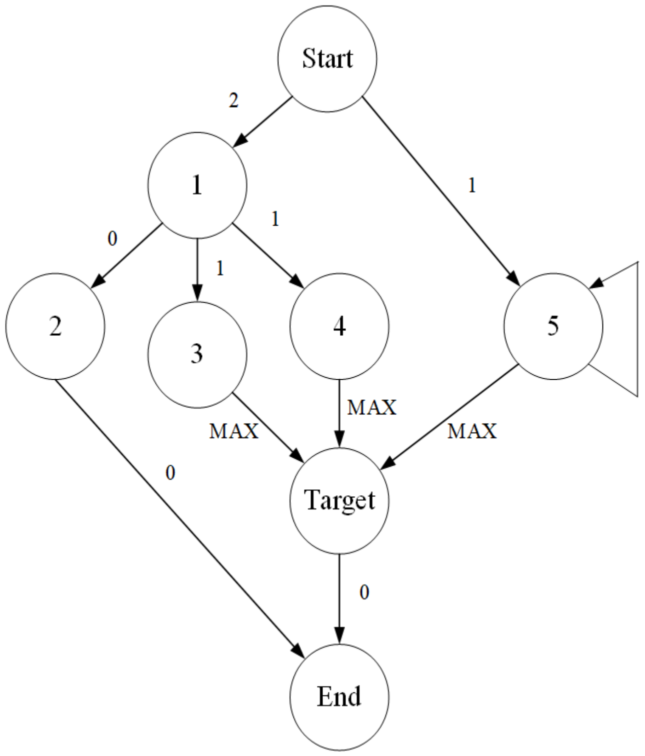


Fig. 2. The Control Flow Graph.

Case 2. Another case is that there are two fuzzing samples respectively travel through basic block 3 and basic block, and both eventually reach the basic block Target. We will find the scores of the two fuzzing samples are the same based on the above theory. However, we find there is a string concatenation functions such as sprintf and strcat, and the fuzzing sample which reaches basic block 4 should be given a higher priority under this circumstance. The reason is that the cause of command injection vulnerabilities generally originate from the back-end program that concatenates the user's input into a string which further directly works as a parameter of the risk function [14].

Considering the above situations, we propose Algorithm 2 to evaluate the quality of fuzzing samples where sampleInfo is the fuzzing samples and funcGraph is the control flow graph of function. It first traverses the basic blocks of the interfaces in the back-end program in deep first search (DFS), gathering necessary information, and then calculates $Score_{test}$ of fuzzing samples based on their feedbacks.

Algorithm 2: Accessment of Fuzzing Samples

```
Input: sampleInfo && funcGraph
Output: Scores of Fuzzing Samples
1. function ACCESSMENT(sampleInfo, funcGraph)
2.     infoGraph ← DFSTraverse(funcGraph)
3.     totalEdges ← makeEdges(
         sampleInfo.executionBlocks)
4.     edgeCount ← 0
5.     edgeTotal ← 0
6.     for edge in totalEdges do
7.         if infoGraph.edge.|$\overrightarrow{\text{Edge}}$| != 0 and
             sampleInfo.edge.flag != 1 then
8.             edgeTotal += infoGraph.edge.|$\overrightarrow{\text{Edge}}$|;
9.             edgeCount += 1;
10.            sampleInfo.edge.flag = 1;
11.        end if
12.    end for
13.    if edgeTotal >= MAX and
           IMPORTANT_BLOCKS(
               sampleInfo.executionBlocks) then
14.        SET_PRIORITY(sampleInfo)
15.    end if
16.    Score ← CALCULATE(
           sampleInfo, edgeTotal, edgeCount)
17.    return Score
18. end function
```

Finally, according to $Score_{test}$ of fuzzing samples, we select high-quality fuzzing samples, given higher priority and more mutation time. We further select the samples which have reached the basic block Target and confirm whether there is a command injection vulnerability by comparing its data to the parameter of risk function.

IV. Results and Discussion

The prototype system of IoTCID consists of three subsystems including the constrained model generation subsystem, the binary static analysis subsystem, and the fuzzing subsystem. The constrained model generation subsystem implemented based on the standard HTML parsing library BeautifulSoap [15] and the standard Javascript parsing library Esprima [16] uses Algorithm 1 to extract the variables during the interaction between front-end files and back-end programs and generate the constrained models of fuzzing samples. The binary static analysis subsystem implemented based on Capstone [13] first obtains the disassembly code of back-end programs, and then establishes the control flow graph as well as sets the probe code using the trampoline mechanism. On the basis of the above two subsystems, the fuzzing subsystem initializes the fuzzing samples by LibFuzzer [17], and performs fuzzing test according to our scheduling strategies.

A. Preparation

We evaluate IoTCID on real-world IoT devices from three vendors, including six routers on two architectures, which are commonly used in our daily life. The target firmware can be obtained from the official website or extracted from the device based on binwalk [18].

In this paper, we design two experiments to prove the efficiency of IoTCID, one of which is the assessment of front-end files analysis while the other is the assessment of fuzzing test. Besides, we compared our tool with SaTC, the state-of-the-art static bug-hunter for IoT devices, which locates the strings between front-end files and back-end programs based on the interaction and applies data flow analysis to detect vulnerabilities. We perform our experiments on Ubuntu 18.04LTS 64-bit operating system, with Intel Core i5-6300HQ @ 2.30GHz and 16.0 GB RAM.

B. Result and Discussion

Table II lists the result of the instrumentation information of target IoT devices, where T_1 represents the average response time of IoT devices under normal working state while T_2 represents the average response time after the static instrumentation. We get the final result after performing multiple tests to reduce the impact of fluctuations caused by the test environment. It shows that the setting of the probe codes only increases the response time by about 25%, which is an acceptable expense for the next fuzzing test.

Table III lists the analysis result of front-end files towards our target IoT devices. Among them, tURL means the total number of URL interfaces that have data interaction between the front-end files and the back-end programs. eURL means the total number of URL interfaces extracted from the front-end file by the tools. gMod means the total number of the generated constrained models. g% means the accuracy rate of the generated constrained models.

According to Table III, it can be seen according to vURL. We define TP rate (True Positive rate) and FP rate (False Positive rate) for further explanation.The TP rate means the ratio of the correct results to the actual total, which can be inferred by vURL/tURL while the FP rate means the ratio of the incorrect results to the actual total, which can be inferred by (eURL-vURL)/tURL.

As shown in the left side of Fig. 3, it can be found that, except for the X12 series, the TP rates of IoTCID and SaTC are achieving an appreciable rate, which means that both IoTCID and SaTC have correctly extracted most of the URL interfaces provided by the target IoT devices and IoTCID does a better job.

However, the FP rates are various as shown in the right side of Fig. 3. Besides a few identification errors, the main reason of the difference is that IoTCID extracts URLs in the front-end files by analyzing the calling procedures which have data interaction with the back-end programs, while SaTC extracts URLs through regular expressions and clustering algorithms directly, which causes a higher FP rate. For instance, certain URL interfaces that only provide the status of devices should not be presumed to be risks and will not by extracted by IoTCID.

TABLE II. RESULT OF STATIC INSTRUMENTATION

Vendor	Device Series	T1(ms)	T2(ms)
Tenda	AC9	3.87	4.66
	AX12	3.47	4.28
D-Link	D605L	3.94	4.64
	D816	3.81	4.68
L-Blink	X12	4.32	5.53
	X22	4.38	5.49

TABLE III. ANALYSIS RESULT OF FRONT-END FILES

Vendor	Series	tURL	IoTCID				SaTC	
			eURL	vURL	gMod	g%	eURL	vURL
Tenda	AC9	106	94	88	100	87.00	123	86
	AX12	98	102	72	108	87.04	130	71
D-Link	D605L	64	58	53	60	95.00	60	52
	D816	66	57	47	60	91.67	50	40
L-Blink	X12	114	117	101	127	74.80	25	16
	X22	114	117	101	127	74.80	25	16

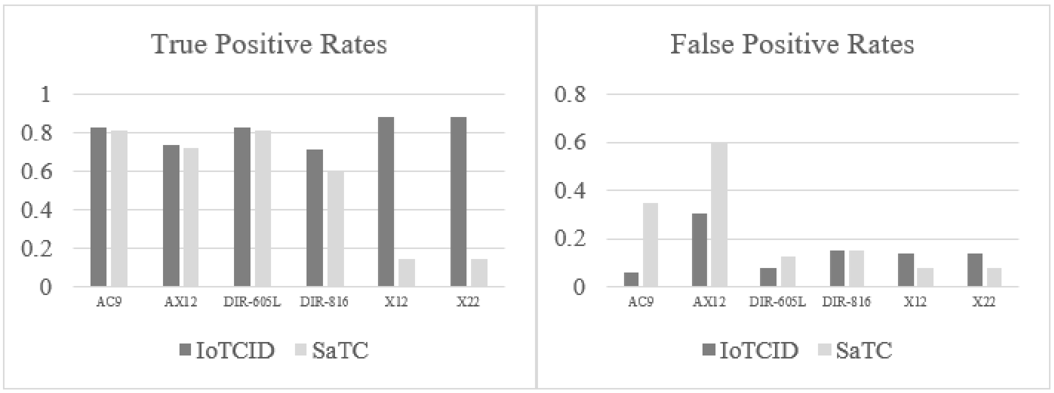


Fig. 3. TP Rates and FP Rates Comparison.

Moreover, IoTCID generates constrained models related to the extracted URLs based on the abstract syntax tree, and has a high accuracy rate in g%. In this experiment, it shows that IoTCID has a better performance in the analysis of the interactions between the front-end files and the back-end programs providing a foundation for the following command injection vulnerability detection.

As Table IV shows, the IoTCID completes all the command injection vulnerability detections towards the target IoT devices and confirms six command injection vulnerabilities, while SaTC only completes a few of them. The reason is that the implementation of SaTC is developed based on angr [19], which is limited in supporting MIPS architecture programs, and prone to cause crash during data flow analysis while IoTCID is designed based on the generation of the constrained models, and confirms whether there is a command injection vulnerability by comparing the fuzzing samples to the feedbacks of the probe code, representing a high degree of support for the detection of multi-architecture programs. Moreover, SaTC only raises alerts after completing the detection and requires further manual analysis to confirm whether the alert is reliable and the command injection vulnerability is controllable, existing certain false positives and costing additional manual analysis time. However, the detection result of IoTCID is based on the feedbacks of the executing program sent by the probe code, so it requires no more manual analysis and the accuracy of the results is guaranteed.

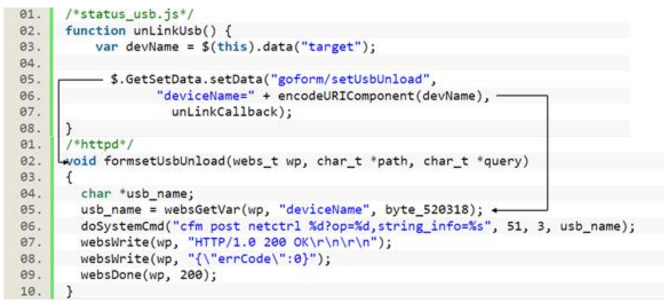
From the above experiments, compared with the existing command injection vulnerability detection tool SaTC towards IoT devices, IoTCID can effectively extract the constrained models based on the analysis of the interaction between the front-end files and the back-end programs, and improve the accuracy and efficiency of detecting the command injection vulnerabilities through various scheduling strategies.

C. Case Analysis

Taking CVE-2018-14558 as an example, when a user manages an external device, the related front-end file will generate a request (Line 5 in status_usb.js, up side of Fig. 4) combined with the device name and send it to the back-end programs for parsing. The back-end program processes the request through the function "formsetUsbUnload", and generates a string containing the device name as the parameter of the system call. (Line 5 and 6 in httpd, low side of Fig. 4). Because the function "formsetUsbUnload" does not verify the validity of the parameter "deviceName", there exists a typical command injection vulnerability in the function "doSystemCmd", which can be exploited by attackers to execute arbitrary commands.

TABLE IV. RESULT OF VULNERABILITY DETECTION

Vendor	Series	IDs	Time（min）	
			IoTCID	SaTC
Tenda	AC9	CVE-2018-14558	4:40h	-
	AC*	CNNVD-202109-1174	4:26h	3:52h
D-Link	D605L	CVE-2018-20057	3:55h	-
	D816	CVE-2021-39510 CVE-2018-17066	3:31h	-
L-BLink	X*	CNNVD-202011-1320	3:10h	3:40h

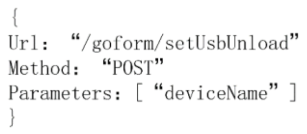


```
/*status_usb.js*/
function unLinkUsb() {
    var devName = $(this).data("target");

        $.GetSetData.setData("goform/setUsbUnload",
             "deviceName=" + encodeURIComponent(devName),
               unLinkCallback);
}
/*httpd*/
void formsetUsbUnload(webs_t wp, char_t *path, char_t *query)
{
  char *usb_name;
  usb_name = websGetVar(wp, "deviceName", byte_520318);
  doSystemCmd("cfm post netctrl %d?op=%d,string_info=%s", 51, 3, usb_name);
  websWrite(wp, "HTTP/1.0 200 OK\r\n\r\n");
  websWrite(wp, "{\"errCode\":0}");
  websDone(wp, 200);
}
```

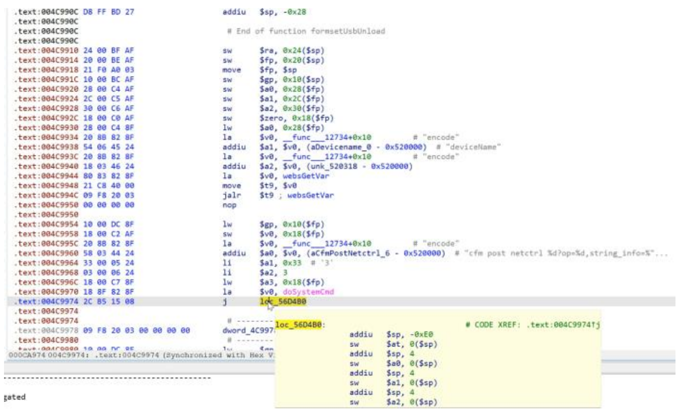
Fig. 4. Interaction between the Front-end Files and the Back-end Programs.

First, IoTCID generates a constrained model by analyzing the interactions between the front-end files and the back-end program, as shown in Fig. 5.



```
{
Url: "/goform/setUsbUnload"
Method: "POST"
Parameters: [ "deviceName" ]
}
```

Fig. 5. A Constrained Model.



```
.text:004C990C D8 FF BD 27             addiu   $sp, -0x28
.text:004C990C
.text:004C990C                          # End of function formsetUsbUnload
.text:004C990C
.text:004C9910 24 00 BF AF             sw      $ra, 0x24($sp)
.text:004C9914 20 00 BE AF             sw      $fp, 0x20($sp)
.text:004C9918 21 F0 A0 03             move    $fp, $sp
.text:004C991C 10 00 BC AF             sw      $gp, 0x10($sp)
.text:004C9920 28 00 C4 AF             sw      $a0, 0x28($fp)
.text:004C9924 2C 00 C5 AF             sw      $a1, 0x2C($fp)
.text:004C9928 30 00 C6 AF             sw      $a2, 0x30($fp)
.text:004C992C 18 00 C0 AF             sw      $zero, 0x18($fp)
.text:004C9930 28 00 C4 8F             lw      $a0, 0x28($fp)
.text:004C9934 20 8B 82 8F             la      $v0, __func___12734+0x10        # "encode"
.text:004C9938 54 06 45 24             addiu   $a1, $v0, (aDevicename_0 - 0x520000)  # "deviceName"
.text:004C993C 20 8B 82 8F             la      $v0, __func___12734+0x10        # "encode"
.text:004C9940 18 03 46 24             addiu   $a2, $v0, (unk_520318 - 0x520000)
.text:004C9944 80 83 82 8F             la      $v0, websGetVar
.text:004C9948 21 C8 40 00             move    $t9, $v0
.text:004C994C 09 F8 20 03             jalr    $t9 ; websGetVar
.text:004C9950 00 00 00 00             nop
.text:004C9950
.text:004C9954 10 00 DC 8F             lw      $gp, 0x10($fp)
.text:004C9958 18 00 C2 AF             sw      $v0, 0x18($fp)
.text:004C995C 20 8B 82 8F             la      $v0, __func___12734+0x10        # "encode"
.text:004C9960 58 03 44 24             addiu   $a0, $v0, (aCfmPostNetctrl_6 - 0x520000)  # "cfm post netctrl %d?op=%d,string_info=%"...
.text:004C9964 33 00 05 24             li      $a1, 0x33  # '3'
.text:004C9968 03 00 06 24             li      $a2, 3
.text:004C996C 18 00 C7 8F             lw      $a3, 0x18($fp)
.text:004C9970 18 8F 82 8F             la      $v0, doSystemCmd
.text:004C9974 2C B5 15 08             j       loc_56D4B0
.text:004C9974
.text:004C9974                          # --------
.text:004C9978 09 F8 20 03 00 00 00 00 dword_4C997
.text:004C9980                          # --------
000CA974 004C9974: .text:004C9974 (Synchronized with Hex V

loc_56D4B0:                                         # CODE XREF: .text:004C9974↑j
                addiu   $sp, -0xE0
                sw      $at, 0($sp)
                addiu   $sp, 4
                sw      $a0, 0($sp)
                addiu   $sp, 4
                sw      $a1, 0($sp)
                addiu   $sp, 4
                sw      $a2, 0($sp)
```

Fig. 6. Probe Code Set.

Then, IoTCID sets up the probe code that records the execution path of the fuzzing samples in the located interface functions and provides with feedbacks by performing binary static analysis on the back-end program, as shown in Fig. 6.

Finally, IoTCID confirms whether there is a command injection vulnerability in the back-end program by comparing the fuzzing samples and the parameters of the risk function, which are provided by the probe code we set before.

V. Conclusion

In this paper, we propose and implement a state-of-the-art dynamic detection tool towards command injection vulnerabilities in IoT devices, IoTCID, which generates constrained models based on the logic analysis to front-end files, and selects high-quality fuzzing samples by various scheduling strategies based on the Distance Function. IoTCID has successfully detected seven command injection vulnerabilities in six real-world IoT devices, two of which are previously unknown vulnerabilities and assigned IDs by CNNVD-202109-1174、CNNVD-202011-1320 after being confirmed by CNNVD.

However, there still remains shortcomings in our tool, such as the limit of the constrained model generation when facing with complex variable references in front-end files and the limit of the throughput of IoTCID for the experiments are currently performing on the devices. Therefore, our future work is as follows:

1) Optimization is needed to improve the capability of the constrained model generation in complex variable references in the front-end file.

2) Optimization is needed to improve the throughput of IoTCID by building a simulation framework environment.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. The research was supported by the National Natural Science Foundation of China (Grant No. 61872386, 62001055).

References

[1] State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globall. https://iot-analytics.com/number-connected-iot-devices/.

[2] Common IoT Attacks that Compromise Security. https://socradar.io/common-iot-attacks-that-compromise-security/.

[3] Governments Must Promote Network-Level IoT Security at Scale. https://www.paloaltonetworks.com/blog/2021/12/network-level-iot-security/.

[4] Tackle IoT application security threats and vulnerabilities. https://www.techtarget.com/iotagenda/tip/Tackle-IoT-application-security-threats-and-vulnerabilities.

[5] Chen J, Diao W, Zhao Q, et al. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing[C]//NDSS. 2018.

[6] Redini N, Continella A, Das D, et al. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices[C]//2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021: 484-500.

[7] Zhang Y, Huo W, Jian K, et al. SRFuzzer: an automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities[C]//Proceedings of the 35th Annual Computer Security Applications Conference. 2019: 544-556.

[8] Srivastava P, Peng H, Li J, et al. Firmfuzz: Automated iot firmware introspection and analysis[C]//Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things. 2019: 15-21.

[9] Zheng Y, Davanian A, Yin H, et al. FIRM-AFL:High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation[C]//28th USENIX Security Symposium (USENIX Security 19). 2019: 1099-1114.

[10] Stasinopoulos A, Ntantogian C, Xenakis C. Commix: Automating evaluation and exploitation of command injection vulnerabilities in web applications[J]. International Journal of Information Security, 2019, 18(1): 49-72.

[11] Redini N, Machiry A, Wang R, et al. Karonte: Detecting insecure multi-binary interactions in embedded firmware[C]//2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020: 1544-1561.

[12] Chen L, Wang Y, Cai Q, et al. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems[C]//30th USENIX Security Symposium (USENIX Security 21). 2021: 303-319.

[13] Quynh N A. Capstone: Next-gen disassembly framework[J]. Black Hat USA, 2014, 5(2): 3-8.

[14] Command Injection. https://owasp.org/www-community/attacks/Command_Injection.

[15] Richardson L. Beautiful soup documentation[J]. Dosegljivo:https://www.crummy. com/software/BeautifulSoup/bs4/doc/. [Dostopano: 7. 7. 2018], 2007.

[16] Hidayat A. Esprima: Ecmascript parsing infrastructure for multipurpose analysis[J]. 2017.

[17] libFuzzer. https://llvm.org/docs/LibFuzzer.html.

[18] C. Heffner, "Binwalk - firmware analysis tool designed to assist in the analysis, extraction, and reverse engineering of firmware images," https://github.com/ReFirmLabs/binwalk, 201 Shoshitaishvili Y, Wang R, Salls C, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis[C]//2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016: 138-157.