Detecting Command Injection Vulnerabilities in Linux-Based Embedded Firmware with LLM-based Taint Analysis of Library Functions

Junjian Ye[a], Xincheng Fei[a], Xavier de Carné de Carnavalet[b], Lianying Zhao[c], Lifa Wu[a,*], Mengyuan Zhang[d]

[a] *Nanjing University of Posts and Telecommunications, Nanjing, China*
[b] *The Hong Kong Polytechnic University, Hong Kong SAR, China*
[c] *Carleton University, Ottawa, Canada*
[d] *Vrije Universiteit Amsterdam, Amsterdam, Netherlands*

Abstract

With the popularization of IoT devices, embedded firmware security has attracted people's attention. Command injection (CI) is one of the most common types of vulnerabilities in Linux-based embedded firmware. It is caused by user input being propagated to functions responsible for command execution without strict sanitization, which can be detected by static taint analysis. Unfortunately, single-binary taint analysis tools cannot find vulnerabilities caused by custom dynamically linked library functions (DLLFs) that are implemented in external library files, while multi-binary analysis tools are time-consuming. In this paper, we present SLFHunter, an approach that leverages Large Language Model (LLM) to analyze sensitive custom DLLFs separately, and imports their information into single-binary taint analysis tools to overcome this challenge. Our approach follows filtering rules to find out sensitive DLLFs that call common sink functions, and analyzes them with LLMs to find sink library functions (SLFs) where input parameters can be passed to executed command strings. Finally, SLFs are marked as new sinks to help existing tools discover CI vulnerabilities caused by them. We implemented SLFHunter as a ChatGPT-based module for EmTaint and evaluated it with a dataset consisting of 100 Linux-based embedded firmware samples from 13 vendors. The results show that our prompts can guide ChatGPT 4.0 to identify SLFs with 95% accuracy after being improved with a trick we dubbed "double-check". SLFHunter can help EmTaint find 42 additional CI vulnerabilities with an average time cost increase of 89 seconds on our dataset, which demonstrates the effectiveness and efficiency of our approach.

Keywords:
Embedded Firmware Security, Command Injection, Static Taint Analysis, Large Language Model

1. Introduction

With the development of emerging technologies such as smart homes and smart cities, more and more Internet of Things (IoT) devices (e.g., routers, IP cameras, and smart TVs) have been deployed. According to the latest available data [1], there are approximately 14.76 billion connected IoT devices. While these devices bring convenience to consumers, they also pose significant security risks because they expose more attack surfaces than traditional devices due to their special structures and requirements [2]. As of 2023, the size of the IoT security market had reached USD 4.94 billion [3].

Embedded firmware is an important part of IoT devices, as it undertakes the implementation of all device features. However, the structure of firmware depends on its embedded operating systems (OS), which leads to differences in the security analysis methods of different OSs. Among them, Linux is the most popular embedded OS due to its open-source nature, customizability and flexibility [4, 5]. These features encourage IoT vendors to develop and add customized programs with various features into the firmware, which brings security risks. Therefore, Linux-based embedded firmware has become a focus of IoT security studies [6, 7, 8, 9].

Command injection (CI) is one of the most common types of vulnerabilities in Linux-based embedded

*Corresponding author
Email addresses: jjy470742953@gmail.com (Junjian Ye), FXC030618@outlook.com (Xincheng Fei), xdecarne@polyu.edu.hk (Xavier de Carné de Carnavalet), lianying.zhao@carleton.ca (Lianying Zhao), wulifa@njupt.edu.cn (Lifa Wu), m.zhang@vu.nl (Mengyuan Zhang)

firmware [10, 11]. The reason is that IoT devices often have several web-based interfaces that accept user input to provide services to users, but the development of back-end binaries of web service (e.g., httpd, lighttpd, cgibin) may not follow best security practices. If user input can reach functions responsible for command execution (e.g., *system*, *popen*) without strict sanitization, adversaries can run arbitrary commands by sending specially crafted packets to the web-based interface.

Static taint analysis is widely used to detect CI vulnerabilities due to its efficiency and independence from real devices or firmware emulators (which still faces many unresolved challenges [12]). It finds CI vulnerabilities by detecting whether the data received by library functions responsible for receiving user input (i.e., taint source functions) can be propagated to library functions responsible for command execution (i.e., taint sink functions) without proper constraints. Several previous efforts [13, 14, 6] have made significant contributions in applying static taint analysis to Linux-based embedded firmware vulnerability detection. Among them, EmTaint [6] demonstrated the strongest performance in detecting CI vulnerabilities.

Challenges. However, the input of EmTaint is a single binary, which means that it cannot analyze dynamically linked library functions (DLLFs) in the target binary because they are implemented in external library files. Therefore, EmTaint can only identify common taint source, propagation and sink functions based on pre-set function names. Unfortunately, in light of our observation, common library functions responsible for command execution may be encapsulated in a library function with another name customized by the firmware vendor. We name these custom library functions that may cause CI vulnerabilities *sink library functions* (SLFs). EmTaint cannot identify SLFs and will miss vulnerabilities caused by them, which is also a common limitation of single-binary taint analysis tools. Multi-binary analysis can solve this problem, but time consumption will significantly increase. For example, on the dataset of EmTaint [6], multi-binary analysis tool Karonte [13] cost 451 hours, while single-binary analysis tool EmTaint only cost 3.5 hours.

Our Approach. To overcome this challenge, we propose a new approach: we can first analyze sensitive custom DLLFs separately to identify SLFs, and then import the information into single-binary taint analysis tools. This approach can both discover vulnerabilities caused by SLFs and control cost within a certain range, making up for the shortcomings of single-binary analysis not being able to delve into DLLFs and multi-binary analysis taking too much time.

Identifying SLFs. Traditional static analysis techniques such as reaching definition analysis is a way to analyze sensitive library functions and identify SLFs. However, these techniques require complete data flow information, including information about the called library functions, which is similar to multi-binary analysis and may come at a cost that exceeds expectations. Furthermore, traditional methods are based on the rules governing taint label propagation, but the manually defined rules rely on human expertise and cannot cover all situations. The string matching heuristics in Karonte [13] and the function behavior similarity in FITS [15] can also be employed to identify SLFs, but the former will miss SLFs whose names are not related to common sink functions, and the latter cost an average of 3 hours per sample in their dataset, which is time-consuming. Additionally, similar to predicting function names [16], deep learning techniques can also be applied to identifying SLFs, but they rely on training datasets, which means that the quality of the dataset determines the effectiveness of the model, and it is difficult for the model to identify SLFs that have not been seen before.

In contrast, Large Language Model (LLM) is a good choice for this task, because it can capture the semantics of function names and code efficiently like an experienced security expert without additional human intervention. According to a recent study [17], the limitation of LLMs in vulnerability detection is that they perform poorly in analyzing large and complex code segments, which means that inputting the entire target binary and all library files in the firmware into LLMs is inefficient. Nonetheless, they are effective in understanding potentially vulnerable patterns (including command injection) in a given snippet of code, which just right meets our need for large-scale taint analysis of individual library functions.

In this paper, we propose SLFHunter (Sink Library Function Hunter), an approach that combines traditional techniques with LLM to find potential SLFs from DLLFs in Linux-based embedded firmware. Our approach follows several filtering rules to find out security sensitive library functions first to save the resources and improve the efficiency of LLMs. Then, our prompts are leveraged to guide LLMs to identify sensitive input parameters of these library functions that may cause CI vulnerabilities accurately. Finally, SLFHunter imports the information of these newly discovered SLFs into EmTaint to find more vulnerabilities.

We implemented SLFHunter as a ChatGPT-based module for EmTaint and evaluated it with a dataset consisting of 100 real-world Linux-based embedded firmware samples from 13 well-known IoT vendors. The results show that SLFHunter can help EmTaint find 42 additional CI vulnerabilities with an average time cost increase of 89 seconds. We also checked the results returned by ChatGPT 4.0 and found that our prompts can guide it to analyze the decompiled code of library functions with high accuracy, low false positive rate and zero false negative rate, especially after being improved with a trick we dubbed "double-check". Additionally, we also evaluated the combination of SLFHunter and FITS. We found that FITS can alleviate the limitation of EmTaint and help SLFHunter find more vulnerabilities, but also leads to a significant decrease in efficiency.

Our main contributions are as follows:

- We proposed a novel approach that imports the information of custom dynamically linked library functions into single-binary taint analysis tools to find command injection vulnerabilities caused by sink library functions (SLFs).
- We proposed a LLM-based method to identify SLFs. We designed a set of prompt templates that can guide LLMs to accurately identify the input parameters of a given library function that can cause CI based on the decompiled code of this function.
- We implemented SLFHunter as a ChatGPT-based module for EmTaint and evaluated it with a dataset consisting of 100 real-world Linux-based embedded firmware samples from 13 well-known IoT vendors. The results show that our prompts can guide ChatGPT 4.0 to identify SLFs with 95% accuracy after double-check, and SLFHunter can help EmTaint discover 42 additional CI vulnerabilities at an acceptable cost on our dataset.

2. Motivation

2.1. Limitations of Existing Static Taint Analysis Tools

EmTaint [6] is a state-of-the-art open-source static taint analysis tool that can find more vulnerabilities than other tools, such as Karonte [13] and SaTC [14], on the same dataset in less time. To test its performance in detecting CI vulnerabilities in Linux-based embedded firmware, we collected several firmware images with known vulnerabilities and employed EmTaint to analyze them. The results show that EmTaint cannot handle DLLFs correctly because they are implemented in external library files rather than the binary being analyzed. As shown in Listing 1, the developers of EmTaint manually wrote customized processing scripts for common library functions (e.g., *system*, *recv*, *getenv*, *strcpy*) to alleviate this deficiency, but this approach cannot address the root cause.

As an example, a CI vulnerability in adm.cgi in Wavlink WN575A3 router firmware cannot be found by EmTaint. The back-end binary adm.cgi calls *fgets* to obtain user input from received request packets and passes them into *set_sys_init*, one of the functions responsible for parsing the contents of user input. As shown in Listing 2, *set_sys_init* calls a custom DLLF *web_get* to extract values of parameters from user input "*a1*" (line 4). Then, it calls another library function *do_system* to run a command string containing the value of "*username*", one of extracted parameters, without any sanitization (line 7 and 10). Adversaries can run arbitrary commands by adding commands into the value of "*username*" and sending the constructed request packet to adm.cgi.

Thanks to manually written scripts, EmTaint can mark common library function *fgets* as a taint source and propagate the tainted data to *set_sys_init*. However, it does not mark custom DLLF *do_system* as a taint sink function because this function is a typical SLF. As shown in Listing 3, *do_system* is implemented in external library file libwebutil.so and calls *system*, a common sink function that can be identified by EmTaint, to run command strings. EmTaint only analyzes adm.cgi and does not know that the input parameters of *do_system* can be propagated to *system*, which leads to EmTaint not finding any vulnerabilities in *set_sys_init*. In other words, just a simple encapsulation causes EmTaint to miss one (or even more) vulnerability.

SaTC [14], another state-of-the-art static taint analysis tool that can detect CI vulnerabilities, is also based on manually defined sink functions and cannot find vulnerabilities caused by SLFs. Therefore, we decide to improve EmTaint by taking custom DLLFs that may cause CI vulnerabilities into consideration.

2.2. Taint Analysis Capability of LLMs

Comparison with Existing Methods. To perform taint analysis of a single library function, traditional static analysis techniques such as reaching definition analysis are good choices. However, in practice, we found that the cost of these techniques may exceed expectations. The main reason is that the analyzed library functions may call other library functions. For traditional techniques, it is necessary to follow up on these called library functions in depth to figure out whether they will

```python
import dataflow
from dataflow.data_collector import weaks_command_exec

class system(dataflow.SimProcedure):
    def run(self, command):
        if self.flow_dir == 'F' and self.purpose == 0:
            for trace_expr in self.block.forward_exprs:
                trace_sims = trace_expr.expr.sims
                trace_ast = trace_expr.expr.ast
                flag = trace_expr.expr.flag

                if trace_ast.op == 'BVS' and flag & 0x100 and command in trace_sims:
                    self.block.is_tainted = 2
                    weaks_command_exec[self.block.addr].append(trace_expr)
                    print("Good, find taint in system! %s %s 0x%x %d" % (self.block, trace_expr, trace_expr.expr.
                          taint_source, len(trace_expr.inter_funcs)))
        return 1

    def infer_type(self, command):
        self.label_variable_type(command, 'ptr')
        self.label_return_type('N')
```

Listing 1: Customized processing script of EmTaint for *system*.

```c
int __fastcall set_sys_init(char *a1)
{
  v3 = (const char *)nvram_bufget(0, "Login");
  v2 = (const char *)web_get("username", a1, 0);
  v5 = strdup(v2);
  ...
  do_system("sed -e 's/^%s:/%s:/' /etc/passwd > /etc/newpw", v3, v5);
  do_system("cp /etc/newpw /etc/passwd");
  do_system("rm -f /etc/newpw");
  do_system("chpasswd.sh %s %s", v5, v9);
  ...
}
```

Listing 2: Decompiled code of *set_sys_init* in adm.cgi (simplified).

```c
int do_system(char *a1, ...)
{
  va_start(va, a1);
  vsprintf(byte_21768, a1, (va_list *)va);
  sprintf(byte_21768, "%s 1>%s 2>&1", byte_21768, "/dev/console");
  return system(byte_21768);
}
```

Listing 3: Decompiled code of *do_system* in libwebutil.so.

propagate the tainted data. The inter-procedural analysis needs to analyze multiple binaries and makes the analysis process more and more complex and require much time. Furthermore, traditional methods are based on the rules governing taint label propagation, but the manually defined rules rely on human expertise and cannot cover all situations.

Several previously proposed methods can also be employed to identify SLFs. Karonte [13] cannot find CI vulnerabilities, but they also considered library functions that are implemented in external library files. They proposed applying string matching heuristics on the name of these functions to detect whether they are similar to *memcmp*, *memcpy* or *strlen*. This approach is also suitable for *system*-like functions but not rigorous. For example, *sqlite_Stat_hook* in ASUS TUF-AX3000 V2 is an SLF, but it cannot be identified by this method because its name is not similar to any common sink functions. FITS [15] compares the behavior features of custom functions with those of anchor functions to identify intermediate taint sources (ITSs). We can also identify SLFs based on function behavior features, but FITS cost an average of 3 hours on each sample in their dataset, which means that extracting features and calculating their similarity are time-consuming.

In addition, similar to predicting function names [16], deep learning techniques can also be applied to identifying SLFs. Deep learning techniques can automatically learn features of function behaviors to identify the patterns of SLFs. However, the main limitation is that they rely on training datasets, which means that we need a large dataset that contains both normal and taint sink

```
int __fastcall wlcsm_wl_sta_assoc_auth_status(const char *a1, int a2, _BYTE *a3, _BYTE *a4)
{
  ...
  snprintf(byte_1A125, 0x50u, "/var/%s_assoc", a1);
  snprintf(&byte_1A125[80], 0x84u, "wl -i %s assoclist > %s", a1, byte_1A125);
  system(&byte_1A125[80]);
  *a3 = j_wlcsm_scanFileForMAC(byte_1A125, a2);
  unlink(byte_1A125);
  snprintf(byte_1A125, 0x50u, "/var/%s_autho", a1);
  snprintf(&byte_1A125[80], 0x84u, "wl -i %s autho_sta_list > %s", a1, byte_1A125);
  system(&byte_1A125[80]);
  *a4 = j_wlcsm_scanFileForMAC(byte_1A125, a2);
  unlink(byte_1A125);
  return 0;
}
```

Listing 4: Decompiled code of custom library function *wlcsm_wl_sta_assoc_auth_status* (simplified).

functions and all of them must be accurately labeled. The quality of the dataset determines the effectiveness of the model, and it is difficult to identify SLFs that have not been seen before in the dataset.

To overcome these challenges with low cost, LLM has shown tremendous potential. LLMs can analyze the propagation path of tainted data based on the semantics of function names and code without additional human intervention in a short time. Moreover, LLMs have stronger generalization and semantic understanding abilities than traditional deep learning techniques. With the continuous development of LLM technology, they can increasingly resemble experienced security experts in capturing the semantics of code.

Feasibility Analysis. With the rise of LLM, several studies [18, 19, 17] have evaluated the performance of LLMs in vulnerability detection. The results indicate that LLM is not good at analyzing large and complex code segments, but its automatic vulnerable pattern discovery capability can assist traditional vulnerability detection techniques, which is consistent with our requirements for detecting the risk of command injection in individual library functions. Therefore, in this section, we test the feasibility of applying LLMs to CI vulnerability detection.

Among existing LLM-based commercial products, ChatGPT 4.0 [20] has the strongest vulnerability detection capability according to a recent study [18]. To evaluate its performance, we sent the decompiled code of a sensitive library function that calls *system* (as shown in Listing 4) to ChatGPT 4.0 and asked it which input parameters of this function may cause CI vulnerabilities. The response of ChatGPT is shown below:

Analyzing the provided function **'wlcsm_wl_sta_assoc_auth_status'**, it's evident that there is a potential for command injection through its input parameters. Let's break down the key points:

Input Parameter 'a1': This parameter is directly used in the formation of system commands. It appears twice in the **'snprintf'** calls to format strings that are then executed by the **'system'** function.

System Commands: The **'system'** function executes the commands that are constructed using **'a1'**. Since **'a1'** is directly interpolated into these commands (**'wl -i %s assoclist > %s'** and **'wl -i %s autho_sta_list > %s'**), if it contains malicious content (like shell metacharacters or other command sequences), it could lead to command injection.

Lack of Sanitization or Validation: There is no apparent sanitization or validation of **'a1'** before its use in constructing the system commands. Without proper validation (such as ensuring that **'a1'** only contains safe characters), this poses a significant security risk.

Other Parameters ('a2', 'a3', 'a4'): These parameters do not directly influence the system commands. **'a2'** is used as a parameter in **'j_wlcsm_scanFileForMAC'**, and **'a3'** and **'a4'** are used to store results. They do not seem to pose a risk for command injection in the context of this function.

Risk Assessment: The primary risk here comes from the use of **'a1'** in the **'system'** calls. An attacker could exploit this by passing a specially crafted string as **'a1'**, potentially leading to arbitrary command execution on the system where

```c
int __fastcall agApi_iptable_nat_addUpnpRule(FILE *a1, int a2, int a3, int a4, int a5, int a6)
{
  ...
  v9 = (const char *)acosNvramConfig_get("lan_ifname");
  strcpy(v13, v9);
  j_arnonat_getWanInterface(v14);
  j_inet_ntoa_b(a5, (char *)v15);
  if ( a2 == 2 )
  {
    printf("%s(%d): proto=%d(%s), Wan(%d) IP(%s) Lan(%d)\n", "agApi_iptable_nat_addUpnpRule", 183, 2, "udp", a4, (const char *)v15, a6);
    sprintf(v16, "iptables -t nat -A FW_NAT_UPNP_PRIOTIY_2_3 -p udp -i %s --dport %d -j DNAT --to %s:%d", v14, a4, (const char *)v15, a6);
    if ( !a1 )
    {
      system(v16);
      sprintf(v16, "iptables -A FW_UPNP_PORTMAPPING -p udp -i %s -d %s --dport %d -j ACCEPT", v14, (const char *)v15, a6);
      v10 = system(v16);
      return j_get_new_ruleID(v10);
    }
    fprintf(a1, "%s\n", v16);
    sprintf(v16, "iptables -A FW_UPNP_PORTMAPPING -p udp -i %s -d %s --dport %d -j ACCEPT", v14, v15, a6);
  }
  ...
}
```

Listing 5: Decompiled code of custom library function *agApi_iptable_nat_addUpnpRule* (simplified).

this function is executed.

In summary, the parameter **'a1'** in the function **'wlcsm_wl_sta_assoc_auth_status'** can cause command injection due to its direct use in system command strings without any sanitization or validation. The other parameters (**'a2'**, **'a3'**, **'a4'**) do not directly contribute to this particular vulnerability in the context of this function.

In the response, ChatGPT accurately identified the taint sink function *system* and analyzed the propagation path of each input parameter to find parameters that can cause CI vulnerabilities, which has proven the taint analysis capability of LLMs for library functions.

To verify whether LLMs can capture the semantics of function names and code, we also sent the decompiled code of a sensitive library function that cannot be exploited (as shown in Listing 5) to ChatGPT. The response of ChatGPT is shown below:

The function **'agApi_iptable_nat_addUpnpRule'** builds command strings that include parameters for iptables rules. These parameters are mostly derived from the inputs **'a2'**, **'a4'**, **'a5'**, and **'a6'**. Let's inspect the manipulation and usage of these parameters:

1. ****'a2'**** - Examining its usage, **'a2'** seems to be responsible for determining which protocol (TCP or UDP) is being used by modifying the printed messages and iptables rule accordingly. However, it only chooses between predetermined strings ("tcp" or "udp") and does not take part in constructing potentially dangerous command strings.

2. ****'a4'**** **and** ****'a6'**** - These parameters are directly used in crafting the iptables command strings. They are used respectively as destination port values and target port/address values in the "DNAT" and "ACCEPT" rules commands. Although these parameters are integrated into shell commands, considering them as integers under typical usage suggests that even if passed directly, they would not pose a command injection risk without further manipulation or specific exploitative context associated with iptables or shell behavior that is not visible from this snippet.

3. ****'a5'**** - This parameter seems to be transformed via **'j_inet_ntoa_b()'**, which is typically used to convert a numeric IP address into a standard dot-decimal notation string. This conversion would typically sanitize the input to conform to an IP address format, reducing direct injection risks of shell commands hidden within the integer value.

From the data flow perspective, each user in-

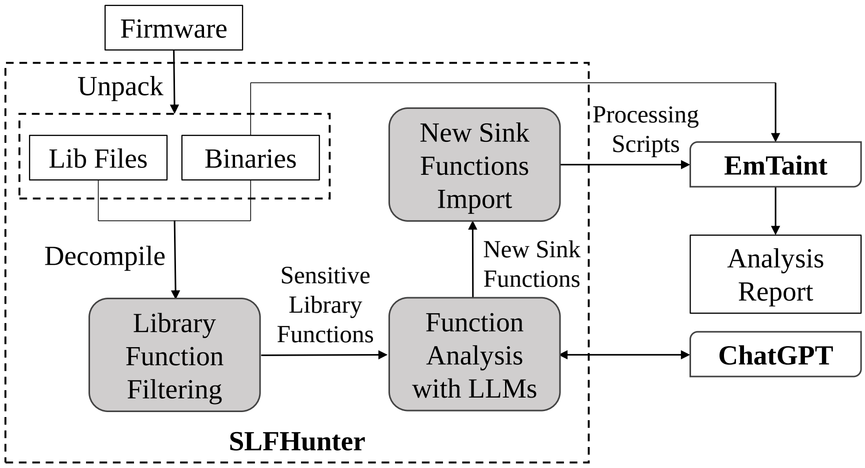

Figure 1: Overview of SLFHunter.

> put is either used in a limited context or transformed in such a way that arbitrary command execution through direct input manipulation would be unlikely. The function displays cautious use around the formats used in operations affecting system commands (**'sprintf()'** and **'system()'** calls), effectively mitigating typical command injection vectors unless there's an under-documented behavior of **'j_inet_ntoa_b'**, **'sprintf'**, or **'system'** regarding special inputs handling.

In this response, we can find that ChatGPT can infer the functionality of the library function *j_inet_ntoa_b* like an experienced security expert, which proves that LLMs can perform taint analysis accurately without inter-procedural analysis based on the semantics of function names.

3. Design

3.1. Overview

SLFHunter (Sink Library Functions Hunter) is an approach that leverages the taint analysis capability of LLMs to find potential taint sink functions that may cause CI vulnerabilities from DLLFs in Linux-based embedded firmware. It can assist existing single-binary static taint analysis tools (e.g., EmTaint [6]) in discovering more vulnerabilities. As shown in Figure 1, SLFHunter consists of three major components.

Library Function Filtering. Analyzing all library functions in all library files in the firmware is inefficient. This component follows several filtering rules to find out sensitive functions that may cause CI vulnerabilities from all DLLFs in the target binary based on their decompiled code extracted from external library files.

Function Analysis with LLM. This component constructs prompts with the decompiled code of sensitive library functions and sends them to the LLM. The LLM will return the input parameters that can be exploited for CI attacks in a fixed format. In this paper, we choose to employ ChatGPT 4.0, the best performing LLM product in vulnerability detection according to a recent study [18].

New Sink Functions Import. Finally, we need to import the information of SLFs into existing static taint analysis tools to tell them which input parameters of which library functions may be exploited. In this paper, this component can automatically generate processing scripts for SLFs and import them into EmTaint.

3.2. Library Function Filtering

To save resources of LLMs and time, we set several filtering rules as follows to remove library functions that are unlikely to cause CI vulnerabilities:

- We only focus on the DLLFs called by the target binary. Therefore, we obtain the names of loaded library files from ".dynamic" and ".dynstr" sections of the binary, and leverage a disassembler (e.g., IDA Pro [21]) to extract the decompiled code of library functions called by the binary from library files.

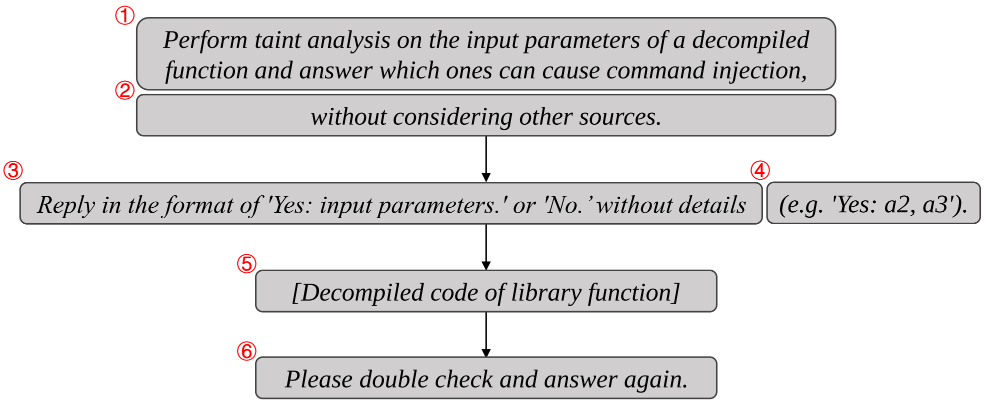

Figure 2: The structure of our prompts.

- Then, we find out library functions that call common library functions responsible for command execution (e.g., *system*, *popen*, and *execve*) because they are likely to be exploited for CI attacks.
- If a library function does not have any input parameters or all parameters are of type *int*, it is impossible for it to cause a CI vulnerability (we do not consider the mistakes of disassemblers or taint sources other than input parameters to reduce false positives). Therefore, we remove such functions.
- If the input parameters of all taint sink functions called by a library function are constant strings, we remove this library function.
- Format string functions (e.g., *do_system(char *a1, ...)*) are special cases. The format string "*a1*" is often a constant string and may be incorrectly identified as *int* type by IDA Pro. In this case, according to the above rules, this function will be removed, but parameters in "..." may be exploitable. Therefore, we keep all format string functions to avoid missing those whose input parameter types are incorrectly identified.

After filtering, we export the decompiled code of sensitive library functions from the disassembler for LLMs to analyze.

3.3. LLM Prompt Design

In Section 2.2, we have proven the feasibility of leveraging LLMs to perform taint analysis on the decompiled code of a library function, and identify input parameters that can be propagated to library functions responsible for command execution. Nevertheless, more detailed prompt design is still needed to standardize prompt generation and returned text parsing to ensure that this task is executed accurately by LLMs. Therefore, we designed a set of prompt templates as shown in Figure 2 to construct prompts automatically. The design details are as follows:

Prompt ①: In the process of testing prompts, we found that modifiers and explanation of terminology have little impact on the results of taint analysis because LLMs have already acquired the necessary knowledge. Therefore, we only need to clearly describe our requirements with "*Perform taint analysis on the input parameters of a decompiled function and answer which ones can cause command injection*".

Prompt ②: "*without considering other sources*" following Prompt ① is presented to require the LLM to ignore other potential taint sources (e.g., environment variables, and functions that read external data), as we are unable to provide the necessary information for analyzing them, and they would confuse the LLM and increase false positives.

Prompt ③: To automate the import of new sink functions, we employ "*Reply in the format of 'Yes: input parameters.' or 'No.' without details*" to have the LLM answer in a fixed format.

Prompt ④: Sometimes, the LLM cannot understand our meaning correctly and answer a lot of analysis details, which is why we present "*(e.g. 'Yes: a2, a3')*" as an example, and it did work.

Prompt ⑤: After describing the requirements, we send the decompiled code of sensitive library functions extracted from library files to the LLM.

Prompt ⑥: During our testing of prompts, we discovered an interesting trick that can improve the accuracy of the results, which we dubbed "double-check". Some-

```c
int __fastcall iqos_service(const char *a1)
{
  ...
  if ( !strcmp(a1, "start_iqos") )
  {
    ...
  }
  else
  {
    if ( !strcmp(a1, "stop_iqos") )
    {
      v5 = "bcmiqosd stop";
    }
    else if ( !strcmp(a1, "restart_iqos") )
    {
      v5 = "bcmiqosd restart";
    }
    else
    {
      if ( strcmp(a1, "pause_iqos") )
      {
        printf("%s(%d): action %s not supported \n", "iqos_service", 4883, a1);
        return 0;
      }
      v5 = "bcmiqosd pause";
    }
    system(v5);
  }
  return 0;
}
```

Listing 6: Decompiled code of custom library function *iqos_service* (simplified).

times, the LLM believes that input parameters that indirectly affect the parameters of library functions responsible for command execution can also be exploited for CI attacks. If we send "*Please double check and answer again*" to it, it will realize this mistake and provide the correct answer. For example, the initial response of ChatGPT 4.0 to the code shown in Listing 6 is "*Yes: a1.*", which is incorrect. After double-check, the response is shown below:

> Upon further review of the **'iqos_service(const char *a1)'** function, it appears that my initial response was incorrect. Here's a revised analysis:
>
> The function **'iqos_service'** checks the value of **'a1'** and compares it against specific strings like "start_iqos", "stop_iqos", "restart_iqos", and "pause_iqos" using **'strcmp'**. The outcome of these comparisons determines which hard-coded string is passed to the **'system'** function. For example, if **'a1'** is "stop_iqos", then "bcmiqosd stop" is passed to **'system'**.
>
> There is no direct concatenation of **'a1'** with any command string or execution of a command directly based on **'a1'**. The command strings are predefined and not dynamically constructed from **'a1'**. Considering this, **'a1'** does not appear to directly influence or construct the command strings executed by the **'system'** function in a way that could lead to command injection. The system calls are controlled and use predefined strings, not directly influenced by the contents of **'a1'** beyond selecting one of these predefined commands.
>
> Therefore, the correct answer is:
>
> No.

"Double-check" can be executed multiple times until the LLM responds the same determined result. Considering efficiency, we heuristically set the execution count to a threshold 2 based on our experience, but it can be changed as a parameter as LLMs evolve.

4. Implementation

We implemented SLFHunter as a module for EmTaint with about 500 lines of Python code. EmTaint leverages an IDA script to extract the control flow graph (CFG) and block information of the target binary, but this script only supports versions below IDA Pro 7.0. Therefore, we ported it to higher versions and added the feature of searching for loaded library files from the firmware filesystems, loading them into IDA Pro, and running the library function filtering script to extract the decompiled code of sensitive library functions. Then, this script combines our prompt templates designed in Section 3.3 with extracted decompiled code and call the API of GPT-4 to analyze these sensitive library functions. Finally, it automatically generates processing scripts for these functions based on the returned results. After that, we can add these scripts into EmTaint and run it with new sink functions and find more CI vulnerabilities.

Additionally, in the example shown in Section 2.1, we found a custom taint propagation function *web_get* in Wavlink routers. It is called to extract the values of parameters from the received user input. Similar functions may exist in other firmware images, but it is difficult to identify them automatically because they do not have obvious characteristics like taint sink functions of CI vulnerabilities. Therefore, we have to manually supplement the processing script for *web_get* into EmTaint to help it find more vulnerabilities in Wavlink firmware, just as its developers have done before. At the same time, we also fixed several small bugs that could cause errors in the code of EmTaint.

5. Evaluation

5.1. Dataset

To evaluate the performance of SLFHunter, we constructed a dataset consisting of 100 real-world Linux-based embedded firmware samples from 13 well-known IoT vendors, including ASUS, Cisco, D-Link, Linksys, LB-Link, Motorola, NETGEAR, TP-Link, Tenda, TOTOLink, TRENDnet, Wavlink and Xiaomi. Among them, 66 samples are selected from the datasets of state-of-the-art tools, including EmTaint [6], Karonte [13] and SaTC [14]. The remaining 34 samples are downloaded from official websites of vendors. The list of firmware samples in our dataset is shown in Appendix A.

Then, we extracted filesystems from these samples with Binwalk [22]. After that, to select the target binaries that are responsible for parsing user input and more likely to be vulnerable to CI from these filesystems, we borrowed the method of SaTC. For each filesystem, we extracted user input keywords from front-end files and matched the strings in back-end binaries with them. Finally, the binary with the maximum matched keywords was treated as the target binary.

5.2. Analysis of Sensitive Library Functions

In our dataset, Library Function Filtering Module found 120 sensitive library functions in 40 target binaries. The remaining 60 binaries do not call any sensitive library functions. Specially, none of the target binaries in the firmware images of Motorola, TP-Link and Xiaomi contains any custom DLLFs that call common library functions responsible for command execution. The reasons are discussed in Section 6.

Table 1: Results of ChatGPT-based taint analysis. For each sample, we list the analyzed binary, the accuracy of ChatGPT 4.0 before and after double-check (DC), and the parameter-grained false positive (#FP) and false negative (#FN) rates of SLF identification.

Vendor	Model	Binary	Accuracy		SLF	
			Before DC	After DC	#FP	#FN
ASUS	RT-AX86U PRO	httpd	60% (6/10)	100% (10/10)	0% (0/16)	0% (0/16)
	GT-AC2900	httpd	33.3% (2/6)	100% (6/6)	0% (0/16)	0% (0/16)
Cisco	RV130X 1.0.3.44	httpd	50% (3/6)	100% (6/6)	0%(0/6)	0%(0/6)
	RV130X 1.0.3.55	httpd	60% (3/5)	100% (5/5)	0%(0/3)	0%(0/3)
D-Link	DIR-878	prog.cgi	50% (1/2)	100% (2/2)	0%(0/3)	0%(0/3)
NETGEAR	R7000P	httpd	75% (3/4)	100% (4/4)	0%(0/3)	0%(0/3)
	LAX20	httpd	50% (4/8)	75% (6/8)	57.1%(4/7)	0%(0/3)
	XR1000 v2	httpd	66.7% (6/9)	77.8% (7/9)	33.3%(2/6)	0%(0/4)
	R6700	httpd	80% (4/5)	80% (4/5)	40% (2/5)	0% (0/3)
	R7000	httpd	60% (3/5)	80% (4/5)	40% (2/5)	0% (0/3)
	XR300	httpd	66.7% (2/3)	100% (3/3)	0% (0/2)	0% (0/2)
	R7300	httpd	50% (2/4)	100% (4/4)	0% (0/2)	0% (0/2)
	R8000	httpd	50% (2/4)	100% (4/4)	0% (0/1)	0% (0/1)
	R8300	httpd	75% (3/4)	100% (4/4)	0% (0/2)	0% (0/2)
	R8500	httpd	75% (3/4)	100% (4/4)	0% (0/2)	0% (0/2)
	Other 25 samples		100% (41/41)	100% (41/41)	0% (0/56)	0% (0/56)
Total	-	-	73.3% (88/120)	95% (114/120)	7.4% (10/135)	0% (0/125)

Then, Function Analysis with LLM Module generated prompts with extracted sensitive library functions and sent them to ChatGPT 4.0. Because the results returned by LLMs are the input parameters that may cause CI vulnerabilities, we cannot say the result of an SLF is a true positive or false positive if some parameters are false positives and some are false negatives. Therefore, if the result of an SLF is strictly correct, we mark it as "accurate". At the same time, we refine the granularity to parameters when calculating false positive and false negative rates. For example, if the correct result of Function A(a1, a2, a3) is "a1, a2" and the result returned by the LLM is "a1, a3", we mark "a3" as a false positive and "a2" as a false negative. The results after manual check are shown in Table 1.

According to the results, ChatGPT 4.0 can identify SLFs with 100% accuracy in 25 samples. The results of 32 functions were inaccurate and 26 of them were corrected after double-check (introduced in Section 3.3). Among the 135 parameters returned by ChatGPT 4.0, 10 are false positives and no sensitive parameters were missed. Overall, the results have shown that ChatGPT 4.0 is able to identify SLFs based on the decompiled code of sensitive library functions with very high accuracy, low false positive rate and zero false negative rate.

False Positives. To figure out why the results of the

Table 2: Accuracy of ChatGPT-based taint analysis with different temperature settings. For each value of temperature parameter, we list the accuracy of results on 9 functions (including gen_jffs_backup_profile, upload_jffs_profile, bwdpi_maclist_db, bwdpi_monitor_info, get_wlan_radio_status, agApi_iptable_nat_addUpnpRule, get_wlan_mode, web_del_pptpd_user, and get_site_survey_result).

Test Number	Temperature			
	0	0.5	1	1.5
#1	77.8% (7/9)	77.8% (7/9)	88.9% (8/9)	55.6% (5/9)
#2	77.8% (7/9)	55.6% (5/9)	66.7% (6/9)	66.7% (6/9)
#3	77.8% (7/9)	55.6% (5/9)	33.3% (3/9)	44.4% (4/9)
#4	66.7% (6/9)	66.7% (6/9)	44.4% (4/9)	22.2% (2/9)
#5	55.6% (5/9)	44.4% (4/9)	44.4% (4/9)	55.6% (5/9)
#6	66.7% (6/9)	55.6% (5/9)	44.4% (4/9)	77.8% (7/9)
#7	66.7% (6/9)	66.7% (6/9)	66.7% (6/9)	55.6% (5/9)
#8	44.4% (4/9)	55.6% (5/9)	55.6% (5/9)	55.6% (5/9)
#9	66.7% (6/9)	44.4% (4/9)	66.7% (6/9)	55.6% (5/9)
#10	55.6% (5/9)	77.8% (7/9)	66.7% (6/9)	33.3% (3/9)
Average	65.6% (59/90)	60% (54/90)	57.8% (52/90)	52.2% (47/90)
Variance	0.99	1.16	2.18	2.01

remaining 6 functions were inaccurate, we rechecked them manually. All of them are false positives and include three different functions: *get_site_survey_result*, *agApi_iptable_nat_addUpnpRule* and *get_wlan_mode*. It is interesting that the initial results of the latter two were correct, but incorrect results were returned after double-check, which means double-check sometimes makes the LLM question its correct results and make mistakes. Executing double-check multiple times until the LLM responds the same determined result can alleviate this problem.

Then, we sent these three functions to ChatGPT 4.0 again and asked the analysis process. For *get_site_survey_result*, the LLM returned the correct result this time, which means that the performance of ChatGPT is fluctuating. For *agApi_iptable_nat_addUpnpRule*, all input parameters are of type *int* and cannot be exploited, but the LLM sometimes does not consider data types. For *get_wlan_mode*, the input parameters cannot be directly propagated to the sink function, but the LLM thought that one of these parameters can influence executed commands by buffer overflow. However, this way is not feasible after manual check.

The impact of temperature parameter. The temperature parameter of ChatGPT can control the creativity and diversity of returned results because it decides what sampling temperature to use. Higher values make results more random, while lower values make them more focused and deterministic. To overcome the instability mentioned above, we selected 9 sensitive library functions to evaluate the impact of temperature. These 9 functions are difficult for ChatGPT to analyze because the returned results of them were incorrect before or after double-check in the experiment introduced above, which means that the analysis results of ChatGPT on these functions have randomness and they are suitable as samples for this evaluation.

Table 3: Comparison with EmTaint. For each sample, we list the total number of alerts produced by EmTaint with and without SLFHunter (#Alerts), the total number of true positives verified by us (#TP) and the total time.

Vendor	Model	Binary	Emtaint			EmTaint with SLFHunter		
			#Alerts	#TP	Time (s)	#Alerts	#TP	Time (s)
ASUS	TUF-AX3000 V2	httpd	1	0	118.6	5	2	310.9
	RT-AC66U B1	httpd	0	0	75.6	4	4	134.4
	RT-AX86U PRO	httpd	2	0	208.2	8	4	418.1
	GT-AC2900	httpd	0	0	82.3	7	7	703.1
Wavlink	WN575A3	adm.cgi	0	0	4.4	8	8	20.2
	WN572HG3	adm.cgi	2	0	13.8	4	0	53
Tenda	AC9 V3.0	httpd	0	0	1933.4	6	5	1832.9
	AC9 V1.0	httpd	0	0	1209.1	3	2	1220.1
	AC18 V1.0	httpd	0	0	1389.8	3	2	1414.9
	AC15 V1.0	httpd	0	0	463.9	3	2	489.1
NETGEAR	R6700	httpd	2	1	761.1	4	3	822.2
	R7000	httpd	3	1	422.6	5	3	374.6
	XR300	httpd	3	1	252.4	5	3	295.7
Total	-	-	13	3	6935.2	65	45	8089.2

The range of temperature parameter is 0 to 2 and errors will happen sometimes when it is set over 1.8. To evaluate the impact of the values of temperature, we set it to 0, 0.5, 1, and 1.5 respectively and analyzed each sample with each value 10 times. The average and variance of the accuracy of 10 tests for each value are shown in Table 2. According to the results, the higher the value, the lower the accuracy, and the more unstable it tends to be. When the temperature is set to 0, ChatGPT performs best in taint analysis, but it is still not so accurate and stable for specific functions (e.g., get_wlan_mode and agApi_iptable_nat_addUpnpRule). Nevertheless, we believe that the development of LLM technology can alleviate these problems in the future.

5.3. Comparison with EmTaint

For each sample with SLFs, we detected CI vulnerabilities leveraging EmTaint with and without SLFHunter. This experiment was conducted in the docker container provided by EmTaint in a Linux virtual machine with a 4-core CPU and 8 GB RAM.

Among these 40 samples with SLFs, SLFHunter can help EmTaint find more alerts in 13 samples with the help of SLFs. We verified these alerts by manual reverse-engineering. If the tainted data can be propagated from user input points that can be controlled by adversaries to functions responsible for command execution without proper constraints, we consider it a true positive bug. The results of these 13 samples are shown in Table 3.

According to the results, EmTaint with SLFHunter produced 65 alerts and 45 of them are true positives. SLFHunter helped EmTaint find 42 additional verified CI vulnerabilities. 25 of them are zero-day vulnerabilities and we are reporting them to the vendors. The details of these vulnerabilities are shown in Appendix B.

False Positives. The false positives in Function Analysis with LLM Module did not bring any false positives to the overall results. All of the 20 false alerts are caused by the limitations of EmTaint. The main reason is that EmTaint cannot rigorously check the constraints for command injection vulnerabilities, which means that some tainted data can be propagated from source to sink, but it cannot be exploited to run arbitrary commands due to sanitization. This is a common challenge for single-binary taint analysis tools because security checks may be implemented in custom library functions. Additionally, we find that EmTaint marks *getenv* as a common source function, but it is not easy to control a specific environment variable, which also caused two false alerts on our dataset.

In terms of efficiency, searching for sensitive functions and waiting for the results returned by ChatGPT requires additional time, which leads to EmTaint running slower after SLFHunter module is added. However, the maximum increase in time does not exceed 11 minutes and the average increase is only 89 seconds on these 13 samples, which is acceptable. For Tenda AC9 V3.0 and NETGEAR R7000, EmTaint with SLFHunter even costs less time due to shorter execution time of EmTaint after the addition of new sink functions.

In summary, SLFHunter has been proven to help existing tools find more CI vulnerabilities at an acceptable cost.

5.4. Combination with FITS

When checking the results of SLFHunter, we found it missing a vulnerability in NETGEAR XR1000v2. This sample calls a custom function *sub_29CD8* similar to *web_get* (shown in Listing 2) to process user inputs. This function is implemented in the target binary, but EmTaint cannot propagate tainted data correctly in this function because the data flow is too complex. Liu et al. [15] also noticed this challenge and proposed FITS to alleviate this problem. They name functions similar to *web_get* and *sub_29CD8* intermediate taint sources (ITSs). FITS can infer ITSs based on function behavior features and mark them as new taint sources to help existing taint analysis tools find more vulnerabilities.

Table 4: Results of combining FITS and SLFHunter. For each sample, we list the total number of alerts and true positives (TP/Alert) and the total time.

Vendor	Model	Binary	Emtaint with FITS TP/ Alert	Emtaint with FITS Time (hh:mm)	EmTaint with SLFHunter TP/ Alert	EmTaint with SLFHunter Time (hh:mm)	EmTaint with both TP/ Alert	EmTaint with both Time (hh:mm)
Cisco	RV130X 1.0.3.55	httpd	0/1	4:07	0/0	0:04	0/1	4:08
D-Link	DIR-878	prog.cgi	FITS crashed					
LB-Link	WR9000	goahead	FITS crashed					
Linksys	E9450	httpd	0/0	14:13	0/0	0:05	2/2	14:17
NETGEAR	R6700	httpd	1/4	17:48	3/4	0:14	3/6	17:49
	R7000P	httpd	0/3	22:10	0/1	0:10	0/3	22:11
	XR1000v2	httpd	FITS crashed					
Tenda	AC9 V1.0	httpd	0/0	6:13	2/3	0:20	14/19	6:13
	AC18 V1.0	httpd	0/0	5:27	2/3	0:24	14/19	5:28
Wavlink	WN575A3	adm.cgi	FITS crashed					
Total	-	-	1/8	69:58	7/11	1:17	33/50	70:06

To shorten the length of the data-flow path from the taint source to the sink, SFLHunter focuses on sink functions and FITS focuses on source functions, which means that new vulnerabilities found by them are different. Therefore, the combination of them may be interesting. To verify this idea, we incorporated FITS into SLFHunter and did a small-scale experiment on 10 samples randomly selected from our dataset. In this experiment, FITS was run in a server with a 8-core CPU and 64 GB RAM.

We run FITS to identify possible ITSs and import true positives into EmTaint manually first. Then, we run SLFHunter to identify SLFs and import them into EmTaint automatically to conduct taint analysis. The results are shown in Table 4. FITS crashed when analyzing 4 samples due to unknown bugs. Among the remaining 6 samples, FITS can help SLFHunter find additional CI vulnerabilities in 3 samples, 2 of which are similar. The details of them are shown in Appendix C.

According to the results, although EmTaint with FITS and SLFHunter discovered 26 more verified vulnerabilities than EmTaint with SLFHunter, the total time increased by over 68 hours, which leads to a significant decrease in efficiency. Additionally, even if FITS can finish analyzing Wavlink WN575A3, it cannot identify *web_get* as an ITS because it also ignores DLLFs and only analyzes functions implemented in the target binary.

False Positives. The main reason for false alerts caused by FITS is that the ITSs that are not executed are also marked as taint sources. FITS defines ITSs as custom functions that process user input received via common source functions and return a part of the input to be used by other functions. This definition is similar to taint propagation functions rather than taint sources. For

```c
1  int __fastcall gen_jffs_backup_profile(const char *a1, const char *a2)
2  {
3    ...
4    for ( i = (const char **)jffs_backup_profile_t; *i; i += 6 )
5    {
6      if ( !strcmp(a1, *i) )
7      {
8        snprintf((char *)&v6, 0x40u, "/jffs/%s", i[1]);
9        if ( !check_if_dir_exist(&v6) )
10         return 400;
11       snprintf((char *)&v8, 0x400u, "echo '%s' >> %s", i[3], "/jffs/exclude_lists");
12       system((const char *)&v8);
13       snprintf((char *)&v8, 0x400u, "tar czvf %s -C /jffs -X %s %s", a2, "/jffs/exclude_lists", i[1]);
14       system((const char *)&v8);
15       unlink("/jffs/exclude_lists");
16       return 200;
17     }
18   }
19   return 200;
20 }
```

Listing 7: Decompiled code of custom library function *gen_jffs_backup_profile* (simplified).

example, an abandoned function that is not executed by the target binary calls an ITS. FITS marks this ITS as a new taint source and can find a vulnerability in this function, but this vulnerability cannot be exploited because the user input cannot be propagated into this function. Therefore, the correct implementation should mark ITSs as taint propagation functions and can reduce 12 false positives.

In conclusion, combination with FITS can alleviate the limitation of EmTaint and help SLFHunter discover more vulnerabilities, but there is still room for improvement. This interesting exploration not only proves the scalability of SLFHunter, but also demonstrates the feasibility of shortening the path between taint sources and sinks as a possible research direction of taint analysis.

6. Discussion

In this section, we discuss various caveats and limitations of our technique, and future directions.

Incorrect Data Type. ChatGPT 4.0 is not suitable for directly analyzing binaries, so we send the decompiled code to it for analysis, which leads to the analysis results also relying on the performance of decompilers. However, due to information loss during the compilation process, even mainstream disassemblers (e.g., IDA Pro [21], Ghidra [23]) may make mistakes in identifying parameters of functions and their data types (such as identifying *char ** as *int*), which can affect the accuracy of our approach. Therefore, although we found that ChatGPT sometimes does not consider data types when performing taint analysis, we did not add relevant restrictions in the prompt. Hopefully, it's possible for LLMs to analyze assembly code directly in the near future [24, 25].

External Data Sources. To reduce false positives, we require LLMs to ignore external data sources other than the input parameters, which means that we assume that other sources in the library functions cannot be controlled by users by default. For example, in the custom library function *gen_jffs_backup_profile* shown in Listing 7, input parameter *a1* can indirectly influence **i* (line 4-6), which is part of *jffs_backup_profile_t* array. If users can control the content of *jffs_backup_profile_t*, *a1* can cause CI. However, due to the missing information of *jffs_backup_profile_t*, the response of ChatGPT 4.0 is "*Yes: a2.*" after double-check, which is a more conservative result.

Heterogeneity of Firmware. Different firmware images from different vendors implement web services with various frameworks. During the experiment, we found that most firmware images do not contain web service binaries that call many custom DLLFs. Their developers tend to implement the features of web services with common library functions, rather than defining and calling custom library functions. The heterogeneity makes our approach only effective in the firmware of specific vendors (e.g., ASUS, Wavlink, Tenda and NETGEAR).

Fortunately, SLFHunter is a scalable approach that will be enhanced continuously with the development of LLM technology, static taint analysis, and decompilation techniques.

Future Work. In addition to CI sink functions, leveraging LLMs to search for potential taint propagation functions (e.g., *web_get* in Wavlink routers) and taint sink functions for other types of vulnerabilities from custom DLLFs is also an interesting idea. However, they are more complex because they do not have significant and unique features for quick filtering or cannot be judged for security solely by analyzing the code of a single function. The identification method based on function behavior similarity proposed by FITS may be another way to find these functions. Based on the findings of SLFHunter and FITS, we believe that shortening the path between taint sources and sinks can be one of improvement directions of taint analysis in the future.

7. Related Work

Static Taint Analysis for Embedded Firmware. Karonte [13] can efficiently detect multi-binary buffer overflows and denial-of-service vulnerabilities by modeling and tracking multi-binary interactions, but did not consider CI vulnerabilities. FITS [15] can infer functions that process user inputs based on function behavior similarity, and mark them as new taint sources to shorten the data-flow path from taint sources to sinks. Similar to our work, this method can also help existing tools find more vulnerabilities, but it focuses on custom taint source functions and ignores DLLFs. SaTC [14] can infer taint sources based on keywords shared between the front-end and back-end of web services and detect dangerous use of untrusted input, but may miss taint sources whose keywords cannot be found in front-end files. Cheng et al. [26] proposed a binary firmware data flow analysis approach that generates data dependency through identifying interprocedural data flows, pointer alias and similarity of the data structure layout to eliminate the influence of complex binaries. In 2023, they proposed another static approach for taint-style vulnerabilities detection in Linux-based embedded firmware named EmTaint [6]. This approach can implement indirect call resolution and accurate taint analysis by SSE-based on-demand alias analysis. EmTaint can find more vulnerabilities with high accuracy in less time than state-of-the-art tools such as Karonte and SaTC, but cannot find those caused by custom DLLFs because it only analyzes the target binary.

Command Injection Vulnerability Detection in Embedded Firmware. IoTCID [27] improved fuzzing for CI vulnerabilities by generating fuzzing samples based on the logic analysis of front-end files and intelligent feedbacks from back-end programs. Yu et al. [28] discovered higher-order CI vulnerabilities (HOCIVs), which means that adversaries abuse an interface to store the injection payload and later use it in a command interpreter through another interface. They presented an approach combining fuzzing with dynamic data flow tracking named ReLink, which can identify HOCIVs by detecting data stores that would be transferred to command interpreters. However, the efficiency of these two tools is limited because they rely on real-world devices. In contrast, CINDY [29] is a sink call-site classification method to accelerate the CI vulnerability discovery in embedded firmware with static backtracking analysis. It can improve the efficiency of SaTC by removing secure sink functions whose parameters are derived from constant strings. However, this tool leverages a fixed sink list, which makes it impossible to find CI vulnerabilities caused by custom library functions.

Application of LLMs in Vulnerability Detection. Several previous studies [18, 19, 17] have evaluated the performance of LLMs in vulnerability detection and confirmed the feasibility of this direction. Zhang et al. [30] studied the impact of prompts on the performance of ChatGPT in software vulnerability detection. Li et al. [31] utilized ChatGPT to create summaries of functions automatically to enhance existing tool UBITect. Similar to our work, Liu et al. [32] tried to leverage LLMs to perform taint analysis. However, they entrust LLMs to identify taint sources, identify sink functions, and perform taint analysis, which results in significant resource consumption of LLMs, especially for large binaries with thousands of functions. Our work only hands over time-consuming steps in traditional taint analysis to LLMs to minimize costs and improve efficiency. They also take DLLFs into consideration, but choose to judge their security just based on function names, which is not rigorous.

8. Conclusion

In this paper, we propose SLFHunter, an approach that leverages LLMs to find new taint sink functions that may cause CI vulnerabilities from library functions to assist existing static taint analysis tools in discovering more vulnerabilities. We implemented SLFHunter as a ChatGPT-based module for EmTaint and evaluated it with 100 real-world Linux-based embedded firmware samples from 13 well-known IoT vendors. The results show that our prompts can guide ChatGPT 4.0 to identify SLFs with 95% accuracy, and our approach can help EmTaint, a state-of-the-art static taint analysis tool, find 42 additional CI vulnerabilities at an acceptable cost on our dataset. As a scalable approach, the performance of SLFHunter can continue to be enhanced with the development of LLM technology and static taint analysis by combining with state-of-the-art methods such as FITS. We hope our work can shed some light on detecting vulnerabilities caused by custom DLLFs and inspire researchers to combine LLM with traditional vulnerability detection techniques.

Acknowledgments

This work was supported by the National Key Research and Development Program of China with 2019YFB2101704.

References

[1] J. Howarth, 80+ Amazing IoT Statistics (2024-2030), https://explodingtopics.com/blog/iot-stats (2023).

[2] M. Yu, J. Zhuge, M. Cao, Z. Shi, L. Jiang, A Survey of Security Vulnerability Analysis, Discovery, Detection, and Mitigation on IoT Devices, Future Internet 12 (2) (2020) 1–23.

[3] M. Intelligence, IoT (Internet of Things) Security Market Size & Share Analysis - Growth Trends & Forecasts (2023 - 2028), https://www.mordorintelligence.com/industry-reports/iot-security-market (2023).

[4] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, A Large-scale Analysis of the Security of Embedded Firmwares, in: USENIX Security Symposium (USENIX Security), 2014, pp. 95–110.

[5] Dusun, Linux IoT and Distributions, Why Use it for IoT Deployment?, https://www.dusuniot.com/blog/linux-iot-and-distributions-why-use-it-for-iot-deployment/ (2023).

[6] K. Cheng, Y. Zheng, T. Liu, L. Guan, P. Liu, H. Li, H. Zhu, K. Ye, L. Sun, Detecting Vulnerabilities in Linux-Based Embedded Firmware with SSE-Based On-Demand Alias Analysis, in: ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'23), 2023, pp. 360–372.

[7] D. D. Chen, M. Egele, M. Woo, D. Brumley, Towards Automated Dynamic Analysis for Linux-Based Embedded Firmware, in: Network and Distributed System Security Symposium (NDSS'16), 2016, pp. 1–16.

[8] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, L. Sun, Efficient Greybox Fuzzing of Applications in Linux-Based IoT Devices via Enhanced User-mode Emulation, in: ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22), 2022, pp. 417–428.

[9] K. Liu, M. Yang, Z. Ling, H. Yan, Y. Zhang, X. Fu, W. Zhao, On Manually Reverse Engineering Communication Protocols of Linux-Based IoT Systems, IEEE Internet of Things Journal (IoT-J) 8 (8) (2021) 6815–6827.

[10] X. Feng, X. Liao, X. F. Wang, H. Wang, Q. Li, K. Yang, H. Zhu, L. Sun, Understanding and Securing Device Vulnerabilities through Automated Bug Report Analysis, in: USENIX Security Symposium (USENIX Security), 2019, pp. 887–903.

[11] D. Wang, M. Jiang, R. Chang, Y. Zhou, H. Wang, B. Hou, L. Wu, X. Luo, An Empirical Study on the Insecurity of End-of-Life (EoL) IoT Devices, IEEE Transactions on Dependable and Secure Computing (TDSC) (2023) 1–14.

[12] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, A. A. Clements, Challenges in Firmware Re-Hosting, Emulation, and Analysis, ACM Computing Surveys (CSUR) 54 (1) (2021) 1–36.

[13] N. Redini, A. MacHiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, G. Vigna, Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware, in: IEEE Symposium on Security and Privacy (S&P'20), 2020, pp. 1544–1561.

[14] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, Z. Xue, Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems, in: USENIX Security Symposium (USENIX Security), 2021, pp. 303–319.

[15] P. Liu, Y. Zheng, C. Sun, C. Qin, D. Fang, M. Liu, L. Sun, FITS: Inferring Intermediate Taint Sources for Effective Vulnerability Analysis of IoT Device Firmware, in: ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), 2023, pp. 138–152.

[16] X. Jin, K. Pei, J. Y. Won, Z. Lin, SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings, in: ACM SIGSAC Conference on Computer and Communications Security (CCS'22), 2022, p. 1631–1645.

[17] M. D. Purba, A. Ghosh, B. J. Radford, B. Chu, Software Vulnerability Detection Using Large Language Models, in: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'23), 2023, pp. 112–119.

[18] Z. Gao, H. Wang, Y. Zhou, W. Zhu, C. Zhang, How Far Have We Gone in Vulnerability Detection Using Large Language Models (2023). arXiv:2311.12420.

[19] A. Cheshkov, P. Zadorozhny, R. Levichev, Evaluation of ChatGPT Model for Vulnerability Detection (2023). arXiv:2304.07232.

[20] OpenAI, GPT-4, https://openai.com/research/gpt-4 (2023).

[21] Hex-Rays, IDA Pro, https://hex-rays.com/ida-pro (2023).

[22] devttys0, Binwalk, https://github.com/ReFirmLabs/binwalk (2014).

[23] NSA, Ghidra, https://ghidra-sre.org (2023).

[24] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, C. Zhang, jTrans: Jump-Aware Transformer for Binary Code Similarity Detection, in: ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22), 2022, p. 1–13.

[25] K. Pei, Z. Xuan, J. Yang, S. Jana, B. Ray, Learning Approximate Execution Semantics From Traces for Binary Function Similarity, IEEE Transactions on Software Engineering (TSE) 49 (4) (2023) 2776–2790.

[26] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, Z. Liang, DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware, in: Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18), 2018, pp. 430–441.

[27] H. Chen, J. Ma, B. Cui, J. Fu, IoTCID: A Dynamic Detection Technology for Command Injection Vulnerabilities in IoT Devices, International Journal of Advanced Computer Science and Applications (IJACSA) 13 (10) (2022) 7–14.

[28] L. Yu, H. Wang, L. Li, H. He, Towards Automated Detection of Higher-Order Command Injection Vulnerabilities in IoT Devices, International Journal of Digital Crime and Forensics (IJDCF) 13 (6) (2021) 1–14.

[29] X. Yin, R. Cai, Y. Z. Zhang, L. Li, Q. Yang, S. Liu, Accelerating Command Injection Vulnerability Discovery in Embedded Firmware with Static Backtracking Analysis, in: International Conference on the Internet of Things (IoT'22), 2022, pp. 65–72.

[30] C. Zhang, H. Liu, J. Zeng, K. Yang, Y. Li, H. Li, Prompt-Enhanced Software Vulnerability Detection Using ChatGPT (2023). arXiv:2308.12697.

[31] H. Li, Y. Hao, Y. Zhai, Z. Qian, Assisting Static Analysis with Large Language Models: A ChatGPT Experiment, in: ACM Joint European Software Engineering Conference and

Symposium on the Foundations of Software Engineering (ES-EC/FSE'23), 2023, pp. 2107–2111.
[32] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Li, L. Sun, Harnessing the Power of LLM to Support Binary Taint Analysis (2023). arXiv:2310.08275.

A. Tested Firmware Samples

Table A1: Results of tested firmware samples.

Vendor	Model
ASUS (4)	TUF-AX3000 V2*, RT-AC66U-B1*, RT-AX86U PRO*, GT-AC2900*
Cisco (3)	RV130X 1.0.3.55•, RV130X 1.0.3.44•, RV32X°
D-Link (17)	DIR-825B1•, DIR-878•, DIR-836L°, DIR-822A1°, DIR-842†, DCS-935L°, DIR-826LA1°, DIR-895L°, DWR-118°, DIR-868L°, DIR-880°, DIR-885L°, DIR-890LA1°, DIR-823G†, DCS-5020L°, DAP-1860°, DIR-859°
LB-Link (1)	WR9000•
Linksys (4)	E9450•, EA8100 V2°, E900°, EA4500V3°
Motorola (2)	C1†, M2†
NETGEAR (23)	R7000P•, LAX20•, XR1000v2•, R6700*, R7000*, XR300*, AC1450•, R7500†, WNR3500Lv2•, XR500†, R6200v2•, R6300•, R6400v2•, R7300•, R7800†, R7900•, R8000•, R8300•, R8500•, R8900†, R9000†, DGN1000°, DGN2200•
Tenda (11)	AC9V3.0*, AC9V1.0*, WH450A•, AC18V1.0*, AC15V1.0*, RX9Pro†, AC6V1.0°, FH1201•, FH1206•, G1V1.0•, W20EV4.0•
TOTOLink (4)	A950RG°, A3700R•, A8000RU°, T10°
TP-Link (20)	TL-WR940N†, Archer_C7v2°, TL-WR841Nv14°, Archer_C5v2°, TD_W9970°, TL-MR3040_V2°, TX-VG1530°, Archer_C2°, Archer_C20°, Archer_C50°, Archer_C3200°, Archer_D2, C2600†, TD-W8968_V4†, TD-W8980°, TL-MR3020°, TL-WA701ND_V2°, TL-WA830RE_V2°, TL-WR1043ND_V3°, TL-XTR7880°
TRENDnet (5)	TEW632BRP°, TEW-823DRU°, TEW-752DRU°, TEW-818DRU•, TEW827DRU†
Wavlink (3)	WN575A3*, WN572HG3•, WN579X3C†
Xiaomi (3)	4C†, AX9000°, Redmi AX5400°

Legend: † means no ".dynamic" or ".dynstr" section and SLFHunter cannot analyze it.
° means SLFHunter cannot find any SLFs.
• means SLFHunter cannot find new vulnerabilities based on SLFs.
* means SLFHunter can find new vulnerabilities based on SLFs.

B. Vulnerabilities Found by SLFHunter

Table B1: The vulnerabilities found by SLFHunter. We list the custom taint propagation and sink library functions found by SLFHunter, and the number of 0-day vulnerabilities and the CVE IDs of n-day vulnerabilities discovered based on these functions.

Vendor	Model	Added taint functions	0-day	N-day
ASUS	TUF-AX3000 V2	sqlite_Stat_hook	-	CVE-2023-41348
		get_traffic_hook	1	-
	RT-AC66U B1	sqlite_Stat_hook	-	CVE-2023-41345, CVE-2023-41346, CVE-2023-41347
		get_traffic_hook	1	-
	RT-AX86U PRO	sqlite_Stat_hook	-	CVE-2023-41345, CVE-2023-41346
		get_traffic_hook	1	-
		bwdpi_monitor_info	1	-
	GT-AC2900	sqlite_Stat_hook	-	CVE-2023-41345 (3)
		get_traffic_hook	-	CVE-2023-39780
		bwdpi_monitor_info	-	CVE-2023-41347
		bwdpi_monitor_ips	-	CVE-2023-41346
		bwdpi_monitor_nonips	-	CVE-2023-41348
Wavlink	WN575A3	web_get	2	-
		web_get + do_system	6	-
Tenda	AC9 V3.0	doSystemCmd	4	CVE-2022-37810
	AC9 V1.0	doSystemCmd	1	-
		doShell	1	-
	AC18 V1.0	doSystemCmd	1	-
		doShell	1	-
	AC15 V1.0	doSystemCmd	1	-
		doShell	1	-
NETGEAR	R6700	devName_setDeviceName	1	PSV-2023-0109
	R7000	devName_setDeviceName	1	PSV-2023-0109
	XR300	devName_setDeviceName	1	PSV-2023-0109
Total	-	-	25	17

C. Vulnerabilities Found by SLFHunter and FITS.

Table C1: The vulnerabilities found by SLFHunter and FITS. We list ITSs found by FITS, SLFs found by SLFHunter, and the number of 0-day vulnerabilities and the CVE IDs of n-day vulnerabilities discovered based on these functions.

Vendor	Model	Version	Added ITSs	Added SLFs	0-day	N-day
Tenda	AC9 V1.0& AC18 V1.0	15.03.05.15 15.03.05.05	sub_2B9D4	doSystemCmd	1	CVE-2022-25441, CVE-2024-30891 (4), CVE-2024-2854, CVE-2024-3880, CVE-2024-28545, CVE-2023-40839 (2), CVE-2023-40837
Linksys	E9450	1.1.00.064	sub_3341C	rut_doSystemAction	2	-
Total	-	-	-	-	3	11

Junjian Ye is a PhD Student in Information Security at Nanjing University of Posts and Telecommunications. He has participated in several projects related to the security of smart homes and wireless routers. His research fields concern IoT security.

Xincheng Fei is a junior in Information Security at Nanjing University of Posts and Telecommunications. He is skilled in reverse engineering. He has found and reported many vulnerabilities in IoT devices, and has won multiple awards in related competitions.

Xavier de Carné de Carnavalet is a Lecturer at the Hong Kong Polytechnic University. He received his Ph.D. degree in Information and Systems Engineering in 2019 from Concordia University, Montreal, QC, Canada. He was a post-doctoral fellow from 2019 to 2020 at Carleton University, Ottawa, ON. He holds a Dipl-Ing. from École Supérieure d'Informatique, Électronique et Automatique, Paris, France, and a M.A.Sc. in Information Systems Security from Concordia University. His research aims to measure and improve user privacy and security from threats posed by software, web technologies and services.

Lianying Zhao is an Assistant Professor at the School of Computer Science, Carleton University, Ottawa, Canada, prior to which he had 6 years of experience working mainly on mainframes at IBM. He received his Ph.D. degree from Concordia University, Montreal in 2018 and was an NSERC postdoctoral research fellow at the University of Toronto in 2019. His primary research interests include hardware/architectural security support (in particular, trusted computing), systems/platform security (firmware, hypervisor and OS), authentication, privacy preservation and data protection, which has lead to publications at security venues such as NDSS, CCS, TIFS, FC and RAID.

Lifa Wu was born in 1968 and received the Ph.D. degree from Nanjing University in 1998. He is currently a Professor with Nanjing University of Posts and Telecommunications. His research fields concern network security and software security.

Mengyuan Zhang is an assistant professor at the Vrije Universiteit Amsterdam, Netherlands. She received the Ph.D. degree in information and systems engineering from Concordia University, Montreal, Canada. Previously, she was an research assistant professor with the Department of Computing at the Hong Kong Polytechnic University, Hong Kong. She also worked as an experienced researcher at Ericsson Research, Montreal, Canada. Her research interests include security metrics, attack surface, cloud computing security, and applied machine learning in security. She has published several research papers and book chapters on the aforementioned topics in peer-reviewed international journals and conferences such as the IEEE Transactions on Information Forensics and Security, IEEE Transactions on Dependable and Secure Computing, CCS, and ESORICS.