Leveraging Semantic Relations in Code and Data to Enhance Taint Analysis of Embedded Systems

Jiaxu Zhao, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences; Beijing Key Laboratory of Network Security and Protection Technology;* Yuekang Li, *The University of New South Wales;* Yanyan Zou, Zhaohui Liang, Yang Xiao, Yeting Li, Bingwei Peng, Nanyu Zhong, and Xinyi Wang, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences; Beijing Key Laboratory of Network Security and Protection Technology;* Wei Wang, *Institute of Information Engineering, Chinese Academy of Sciences; Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences; Beijing Key Laboratory of Network Security and Protection Technology;* Wei Huo, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences; Beijing Key Laboratory of Network Security and Protection Technology*

https://www.usenix.org/conference/usenixsecurity24/presentation/zhao

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Leveraging Semantic Relations in Code and Data to Enhance Taint Analysis of Embedded Systems

Jiaxu Zhao[1,2,3,4], Yuekang Li[5], Yanyan Zou[1,2,3,4]✉, Zhaohui Liang[1,2,3,4], Yang Xiao[1,2,3,4], Yeting Li[1,2,3,4]
Bingwei Peng[1,2,3,4], Nanyu Zhong[1,2,3,4], Xinyi Wang[1,2,3,4], Wei Wang[1,3,4], Wei Huo[1,2,3,4]✉
[1]*Institute of Information Engineering, Chinese Academy of Sciences, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*
[3]*Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, China*
[4]*Beijing Key Laboratory of Network Security and Protection Technology, China*
[5]*University of New South Wales, Australia*

Abstract

IoT devices have significantly impacted our daily lives, and detecting vulnerabilities in embedded systems early on is critical for ensuring their security. Among the existing vulnerability detection techniques for embedded systems, static taint analysis has been proven effective in detecting severe vulnerabilities, such as command injection vulnerabilities, which can cause remote code execution. Nevertheless, static taint analysis is faced with the problem of identifying sources comprehensively and accurately.

This paper presents LARA, a novel static taint analysis technique to detect vulnerabilities in embedded systems. The design of LARA is inspired by an observation that pertains to semantic relations within and between the code and data of embedded software: user input entries can be categorized as URIs or keys (*data*), and identifying their handling code (*code*) and relations can help systematically and comprehensively identify the sources for taint analysis. Transforming the observation into a practical methodology poses challenges. To address these challenges, LARA employs a combination of pattern-based static analysis and large language model(LLM)-aided analysis, aiming to replicate how human experts would utilize the findings during analysis and enhance it. The pattern-based static analysis simulates human experience, while the LLM-aided analysis captures the way human experts perceive code semantics. We implemented LARA and evaluated it on 203 IoT devices from 21 vendors. In general, LARA detects 556 and 602 more known vulnerabilities than SATC and KARONTE while reducing false positives by 57.0% and 54.3%. Meanwhile, with more sources and sinks from LARA, EMTAINT can detect 245 more vulnerabilities. To date, LARA has found 245 0-day vulnerabilities in 57 devices, all of which were confirmed or fixed with 162 CVE IDs assigned.

✉Corresponding Authors.

1 Introduction

The Internet of Things (IoT) has become an indispensable part of our daily lives, and the number of devices in use is expected to reach 27.1 billion by 2025 [2]. However, this growth has also resulted in an increase in vulnerabilities in embedded systems. According to recent statistics [6], weekly attacks on IoT devices have increased by 41% per organization in the first two months of 2023 compared to 2022. The threat of IoT vulnerabilities includes numerous vulnerabilities, some [4, 5, 47] of which are easily exploited and impact an average of 14% to 49% of organizations worldwide on a weekly basis. Hence, early detecting vulnerabilities in embedded systems has become crucial. IoT devices with web services are more susceptible to attacks compared to other IoT devices [10]. This is because while web services provide a convenient interface for device control and configuration, they also create opportunities for remote attacks if the underlying backend contains vulnerabilities. Thus, detecting vulnerabilities in IoT devices relying on web services is crucial.

Most existing techniques [7,8,55,59] have limited effectiveness in detecting vulnerabilities in the backend programs of the web services in IoT devices. The testing techniques suffer from low code coverage and only focus on memory-related vulnerabilities. In comparison, static taint analysis is more suitable for detecting a wider variety of vulnerabilities. Taint analysis tracks and analyzes the flow of tainted information in a program, using source, sink, and taint propagation [49]. The source is where user inputs are introduced, the sink is where potentially risky operations occur, and taint propagation analysis examines how tainted markers propagate along variable dependencies in the program. Currently, most researches focus on taint propagation analysis, which employs techniques such as multi-binary tracking or pointer analysis to obtain more efficient and accurate analysis results [12, 38]. However, their effectiveness on embedded systems is not significant. This is because traditional sources such as recv function are not effective in revealing the characteristics of user input entry identification and processing in embedded systems, thereby

missing many potential vulnerabilities.

To address the source identification challenge, a recent advancement in static taint analysis called SATC [10] proposes to leverage common input keywords between the frontend and backend to identify the user input handling code in the backend as sources. It can improve static taint analysis by providing more sources. Nevertheless, SATC misses certain sources (at least 82.5% according to our investigation) because some hidden user input handling code in the backend does not have frontend counterparts and some non-hidden user input entries are ignored due to the incomplete rules used for extracting the keywords. Additionally, SATC still suffers from high false positives (52.7%) for the detected sources due to a lack of awareness of dangling keywords with no/unreachable handling code. The falsely identified sources could affect the result of final vulnerability detection (missing over 87.3% with 75.2% false positives). In conclusion, static taint analysis shows promise in detecting vulnerabilities in IoT web services, but extracting sources systematically remains an open problem.

To mitigate false negatives and false positives in source identification, we conducted a comprehensive analysis of the web services in mainstream devices and made an observation relates to the characteristics of the user input entries. User inputs for the web services are typically encoded as key-value pairs organized by forms or form-like data [22]. Rather than treating all user input entries equally, separating them into URIs and keys enables better utilization of their mapping to identify corresponding backend handling codes and more sources effectively. Maintaining a mapping between them can facilitate the identification of their corresponding backend handling codes. These code of the functions, in turn, can reveal hidden URIs and keys, leading to the discovery of additional sources. However, how to identify sources based on the complex relations among URIs and keys and the corresponding pattern between them and the backend code remains a challenge. Additionally, the semantic information in the backend code can facilitate more precise pattern-based static analysis, such as inferring the purpose of a function. Effectively perform semantic-based analysis and combine it with pattern-based analysis is another challenge.

In this paper, we propose LARA[2], a novel static taint analysis technique for detecting vulnerabilities in embedded systems. LARA utilizes URIs and keys extracted from the frontend to determine their corresponding handling code in the backend. This process is achieved through a combination of pattern-based static analysis and large language model (LLM)-aided analysis, aiming to replicate how human experts perform the identification based on previous experience and code semantics. The pattern-based static analysis leverages predefined rules and pattern matching, which simulates human experience, to address the first challenge. Meanwhile, the LLM-aided analysis performs the identification from the code semantics aspect to address the second challenge, as LLMs have been shown effective for summarizing the semantics of functions [16, 25, 37]. LARA then combines the results obtained from both analyses to generate two sets of codes that handle URIs and keys, respectively. By analyzing the URI and key handling codes, LARA can also identify the other URIs and keys that are handled by the same functions. In this manner, LARA can systematically and precisely identify the key handling functions and use them to extract sources. Regarding sinks, LARA analyzes the primary program and its related shared libraries to identify calls to dangerous operations as sinks. Finally, using the identified sources and sinks, LARA performs the static taint analysis that supports inter-process analysis to detect potential vulnerabilities.

We implemented LARA as a static taint analysis framework and evaluated it on the dataset used by SATC, which includes 203 devices from 21 vendors such as DLink, Tenda, NetGear and others. The evaluation results indicate that LARA can detect significantly more vulnerabilities with fewer false positives than both SATC and KARONTE. Specifically, LARA can detect 556 and 602 more known vulnerabilities than SATC and KARONTE, respectively, while reducing false positives by 57.0% and 54.3%. Additionally, EMTAINT could detect 245 more vulnerabilities with the assistance of LARA. To comprehensively understand the capability of each component in LARA, we also conducted an ablation study and the results showed that the pattern-based static analysis, LLM-aided analysis, and sink extraction can all improve the overall performance. Last but not least, we applied LARA to the firmware dataset for vulnerability detection. In total, we have found 245 0-day vulnerabilities in 57 devices, with all confirmed or fixed by the developers and 162 CVE IDs assigned.

In summary, we make the following contributions:

- We tackle the technical challenges and transform the observation into a novel static taint analysis technique called LARA. LARA can capture sources with low false positive and false negative rates and thus is capable of detecting more vulnerabilities.
- We implemented LARA and comprehensively evaluated its performance in vulnerability detection, source identification and sink identification. The results show that LARA can significantly outperform the state-of-the-art IoT static taint analysis techniques by detecting more vulnerabilities with fewer false positives.
- We discovered 245 0-day vulnerabilities in 57 devices from 13 vendors. To date, all have been confirmed or fixed and 162 CVE IDs have been assigned.

We make the raw data, detailed information and source code of LARA available on the LARA-Site: https://sites.google.com/view/lara-data.

[2]Lara Croft is a fictional archaeologist and adventurer who can use her intuition and perception to spot hidden objects and secrets in tombs and ruins.

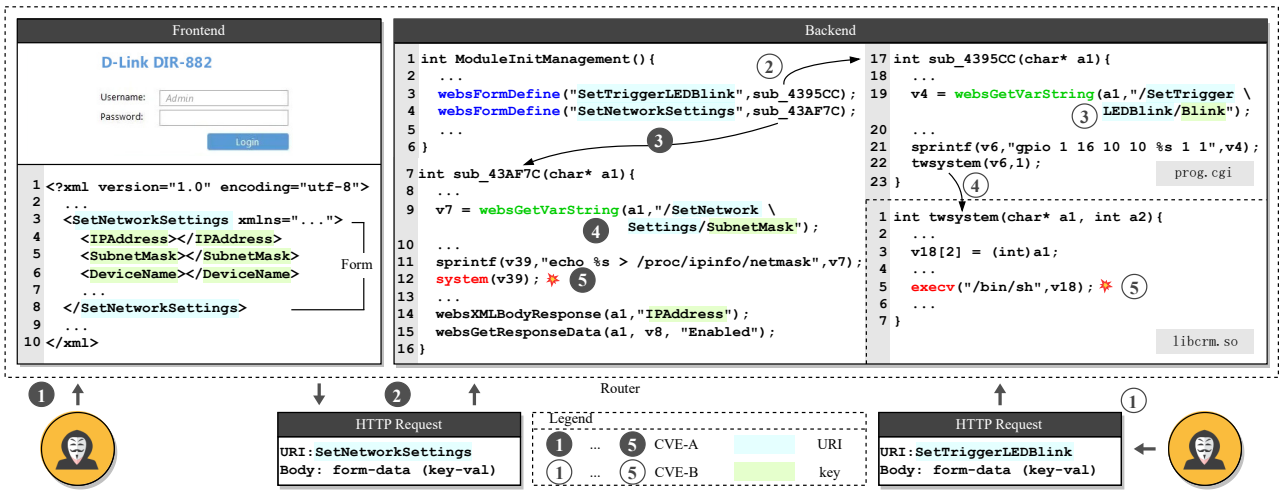

Figure 1: Motivating Example

2 Motivating Example

2.1 Threat Model

In this paper, we focus on attacks against WEB services in embedded systems. The threat model assumes that the attacker can obtain the corresponding firmware from the vendor website and access the target over a local (LAN) or wide area (WAN) network and send malicious HTTP requests to the services. On the service side, the backend programs, including middleware, CGI programs, and related shared libraries, handle these malicious data. Usually, the data in the HTTP request has corresponding labels in the frontend, but there are hidden data without frontend labels in the backend programs. The vulnerability CVE-B shown in Figure 1 is caused by such hidden data. By analyzing the backend program, attackers can obtain these non-hidden and hidden data and then send crafted HTTP requests that inject payloads to the vulnerable backend handling code, leading to consequences like denial-of-service (DoS) and remote-code-execution (RCE).

2.2 Observation

Figure 1 illustrates a motivating example that includes two vulnerabilities detected by LARA in the firmware of the D-Link DIR-882 router. The first vulnerability CVE-A [3] is triggered through the following steps: ❶ The attacker interacts with the web interface (frontend) of the router DIR-882 to configure the *networksettings*. Inside the SubnetMask field, the attacker inputs an injection payload such as ";rm –rf /;". ❷ The frontend generates an HTTP request using the form SetNetworkSettings and the *key-value* pairs encoding the form data. In this case, SetNetworkSettings is filled into the URI [52] field of the HTTP packet, while SubnetMask and the payload are filled into the body field as one of the *key-value* pairs ("SubnetMask=';rm –rf /;'") encoded as form–data [22]. ❸ When receiving the HTTP request, the backend of the router finds the corresponding function to handle it according to the URI value of the form. In this case, the function is sub_43AF7C. ❹ After sub_43AF7C receives the content of the HTTP request, it extracts the values of the keys with specific functions. In this case, the function is websGetVarString. ❺ The backend processes the extracted values as a command and invokes the function system to execute the command, allowing the attacker to execute arbitrary code on the router.

The second vulnerability, CVE-B, can be triggered with the following process: ① The form for this vulnerability is *hidden*, which means there is no corresponding frontend code. Hence, the attacker needs to generate the HTTP request and send it to the backend directly. In this case, the form used by the HTTP request is SetTriggerLEDBlink. ② According to the URI value of the form, the backend invokes the function sub_4395CC to handle the HTTP request. ③ After sub_4395CC receives the HTTP request, it extracts the value for the key Blink. ④ The extracted value is processed to form a command with sprintf and send to the function twsystem. It is worth noting that the function twsystem is not in the main binary of the router (prog.cgi), but instead resides in the shared library libcrm.so. ⑤ Inside the function twsystem, the user-controlled value is executed by execv and the vulnerability is triggered.

Existing techniques have difficulty in identifying vulnerabilities similar to both CVE-A and CVE-B due to omitted sources. To this end, we make an observation of why certain sources are omitted from the examples in Figure 1.

The ***observation*** relates to the characteristics of the user input entries. As shown in Figure 1, some user input entries are *non-hidden* as they have corresponding frontend handling logic (e.g., SubnetMask). In contrast, some user input entries are *hidden* as they do not have corresponding frontend han-

[3] CVE-A is CVE-2022-28896 and CVE-B is CVE-2022-28901.

dling logic (e.g., Blink). The backend handling codes of all the hidden and non-hidden user input entries can be used to extract sources for taint analysis. A state-of-the-art technique, namely SATC [10], suggests using the common keywords extracted from the frontend and backend to identify the sources precisely. Although using the common keywords can help with source identification to some extent, SATC fails to find the sources for both CVE-A and CVE-B. For CVE-A, since SATC uses predefined rules to capture the common keywords and the proposed rules are incomplete, SATC fails to locate the user input entry related to SubnetMask [4]. For CVE-B, since the user input entry lacks corresponding frontend handling code and is not related to a common keyword, SATC cannot detect it. To conclude, a systematic approach is needed to identify the sources, regardless of whether they are related to the hidden or non-hidden user input entries.

2.3 Findings based on the Observation

Based on the observation, we have uncovered two key findings that inspire the design of LARA to enhance taint-based vulnerability detection by identifying more accurate sources.

Finding 1. User input entries can be categorized as URIs or keys. Identifying their corresponding handling codes and relationships can help to identify both hidden and non-hidden user input entries and locate the taint sources.

The first finding pertains to identifying more sources. User inputs are often encoded as key-value pairs that are grouped by forms or form-like data [22]. URIs serve as unique identifiers for these forms, while the keys represent the names of the user input parameters under each form. Figure 2 illustrates the relationships between the URIs and keys in both frontend and backend. In the figure, u and k represent the URIs and keys in the frontend, while U and K represent those in the backend. Though the backend always contains the URIs and keys in the frontend, the inverse is not always true. There are four possible types of relationships between URIs and keys: ❶ both the keys and their corresponding URI are in both frontend and backend; ❷ the URI is in both frontend and backend, but some of the keys related to this URI are only in the backend; ❸ the URI and its related keys are only in the backend; and ❹ the keys do not have any related URIs.

Separating URIs and keys provides two advantages over treating all user input entries the same. Firstly, separating and recombining URIs and keys allows us to avoid counting keys in unreachable functions as sources leading to false postives. These keys are not reachable because they are not linked to any URIs (relation ❹ in Figure 2). Secondly, we can utilize the backend functions that handle URIs to identify the functions that handle keys. By identifying these functions, we

[4]Detailed analysis of why SATC fails to detect CVE-A is in the case study I on the LARA-Site

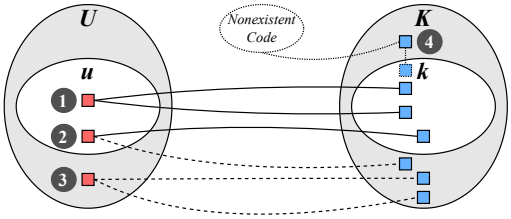

Figure 2: The relationships between the URIs and keys on the frontend and backend.

can uncover both more non-hidden URIs and keys (relation ❶) and more hidden URIs and keys (relation ❷ and ❸).

Although URIs and keys can be categorized as hidden and non-hidden according to whether they are labeled in the frontend, they share similar/same backend handling codes. In the example in Figure 1, we can get the URI SetNetworkSettings from the frontend. In the backend, we can infer that the function websFormDefine maps the URIs to their corresponding handling functions. Through websFormDefine, we can then identify SetTriggerLEDBlink as another URI even though it has no corresponding frontend form. Moreover, from the frontend, we can also see that SubnetMask is a key under the URI SetNetworkSettings. From the code of the function sub_43AF7C, we can see that webGetVarString handles the key SubnetMask, and identify it as a key handling function. Following this, inside sub_4395CC, we can identify Blink as a key under the URI SetTriggerLEDBlink. Therefore, we can use the backend functions and their relations to systematically extract the URIs and keys no matter whether they are hidden or not.

To this end, the rationale for how Finding 1 contributes to improved source identification is clarified, but the key problem in converting it into methodology is how to identify the backend URI and key handling codes. Intuitively, following the analysis of the rationale for Finding 1, we can conclude some patterns to identify these codes to address this challenge. The identification of non-hidden URIs and keys can help identify other hidden and non-hidden URIs and keys by revealing the corresponding handling codes. In the example in Figure 1, the pattern-based static analysis can help to identify websGetVarString (line 9 in prog.cgi) as a key handling function because it takes SubnetMask as a argument. Further, the hidden key Blink will be identified.

Finding 2. Due to the better understanding of code semantics provided by LLM and the differences in false positive sources between LLM-aided analysis and pattern-based static analysis, LLM-aided analysis effectively enhances the identification of more accurate sources.

The second finding pertains to identifying sources with fewer false positives. The results of only using the patterns are still worse than manual analysis because, besides previous experience (pattern), human experts can also understand the semantics of the functions according to the

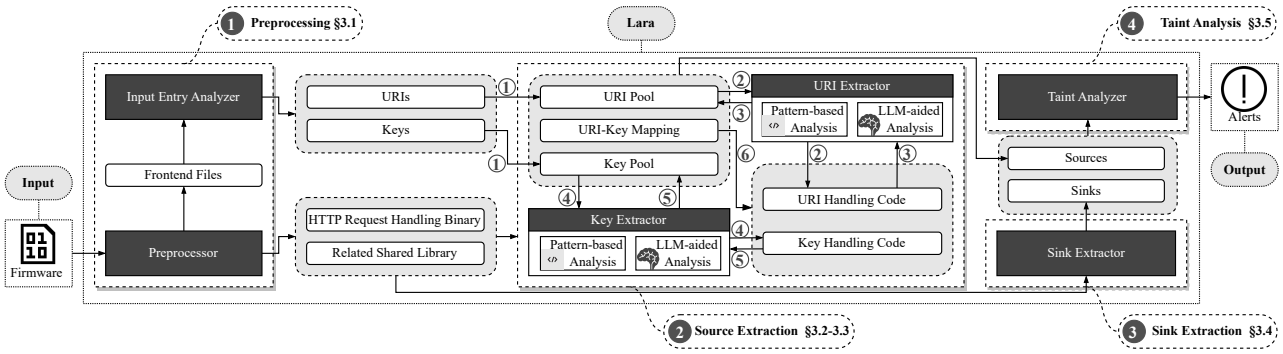

Figure 3: The Overview of LARA

decompiled code with meaningful symbols. For example, in Figure 1, the pattern-based analysis will falsely report websXMLBodyResponse (line 14 in prog.cgi) as a key handling function due to the appearance of IPAddress as a argument. However, through the name of the function, a human expert can easily grasp the semantic of it and realize that it is not a key handling function but a response generation function. Therefore, semantic analyses are necessary. In recent years, deep learning techniques have demonstrated great performance in understanding code semantics [26, 31, 45]. Specifically, the emerging LLMs are superior choices for this task [15, 17, 37, 44]. Therefore, combining the pattern-based static analysis and LLM-aided analysis to mimic how human experts work becomes the design choice of LARA.

Our preliminary analysis of code semantics with LLM reveals a noteworthy finding: Although false positives were present in the results, the sources of false positives in LLM-aided analysis results were distinct from those in pattern-based static analysis results. False positives in pattern-based static analysis are caused by code patterns, while false positives in LLM-aided analysis are due to misleading code semantics, such as symbolic data. For example, in Figure 1, the LLM-aided analysis will falsely report websGetResponseData (line 15 in prog.cgi) as a key handling function due to the function name and the arguments. However, both methods produce accurate results. By taking the intersection of their results, false positives can be eliminated. But effectively combining these two methods remains a challenge. To address this challenge, we designed LLM automatic interaction models and algorithms that combines pattern-based static analysis with LLM-aided analysis to identify more accurate sources.

3 Methodology

Figure 3 depicts the overview of LARA. The input for LARA is the firmware image, and the outputs are alerts for potential vulnerabilities. LARA works in four steps: ❶ ***Preprocessing.*** The preprocessor is in charge of extracting the frontend files and backend files. Mainly, backend files include the binary handling HTTP requests and the related shared libraries. The input entry analyzer extracts the URIs and keys from the frontend. ❷ ***Source Extraction.*** This step involves extracting the URIs from the backend and using the URI information to identify the keys and their handling code. The key handling code is then used to extract the source. For URI extraction, LARA first uses the URIs extracted from the frontend to identify the URI handling code (substeps ① – ② in Figure 3) and then enriches the URI pool (substep ③). Similarly, for key extraction, LARA first uses the keys extracted from the frontend to identify the key handling code (substeps ①,④) and then enriches the key pool (substep ⑤). In addition, LARA also uses the mapping between URIs and keys to filter out unreachable key handling code (substep ⑥). ❸ ***Sink Extraction.*** With a predefined dangerous functions list from system libraries, LARA detects their appearances in the main binary and the shared libraries. The caller functions of the dangerous functions are called the wrapper functions. LARA recursively finds all the wrapper functions, their callers, callers of the callers, and so on, and uses them as the sinks. It is worth noting that taint analysis is utilized to ensure that the parameters of the wrapper functions can affect the danger function calls. ❹ ***Taint Analysis.*** With the sources and sinks, LARA performs static taint analysis and reports alerts for vulnerabilities whenever it encounters connections between sources and sinks.

3.1 Preprocessing

Extraction of Files to be Analyzed. The files are mainly frontend files, backend programs, and related shared libraries. There are three types of frontend files in IoT devices: HTML files, XML files, and JavaScript files. The preprocessor traverses the file system of the firmware and matches file types to extract these frontend files. As for backend files, the preprocessor uses path matching, file type matching, and keyword matching to extract.

Extraction of Input Entries from the Frontend files. The input entry analyzer of LARA uses regular expressions to extract the input entries. Additionally, in LARA URIs and keys are categorized according to the tags or node labels. For

Algorithm 1: Extract URI from Backend Binary

Input: u, the set of URIs extracted form frontend
$\mathcal{B}$, the set of binaries handling HTTP requests
Output: UF, the corresponding set of URI and URI handling function

```
1  BindingTypeI, BindingTypeII, BindingTypeIII ← ∅;
2  UF ← ∅;
3  foreach u_i ∈ u do
4      if {b ∈ B | b.name = u_i.value} ≠ ∅ then
5          BindingTypeIII ←+ u_i;
6      else
7          address ← get_xref(u_i);
8          if address ∈ Segment(".data") then
9              BindingTypeII ←+ address;
10         if address ∈ Segment(".text") then
11             AST ← Decompiler(address);
12             r1 ← LLM_URIAnalysis(address);
13             r2 ← ∅;
14             while visit(AST) do
15                 if is_call(expr) ∧ u_i ∈ expr.arg then
16                     r2 ←+ expr.name;
17                 BindingTypeI ←+ Merge(r1, r2);
18 foreach F_i ∈ BindingTypeI do
19     foreach f_i ∈ get_xref_func(F_i) do
20         AST ← Decompiler(f_i);
21         while visit(AST) do
22             if is_call(expr) ∧ expr.name = f_i then
23                 [u, f] ← get_arg(expr);
24                 UF ←+ [u, f];
25 foreach add_i ∈ BindingTypeII do
26     if [add_i, add_i+offset] ∉ UF then
27         while [add_i, add_i+offset] match [String, Func] do
28             UF ←+ [add_i, add_i+offset];
29             add_i ← next_ins(add_i);
30 foreach b_i ∈ BindingTypeIII do
31     f ← ∅;
32     foreach f_i in b_i.func do
33         if no_xref(f_i) ∨ f_i = "Main" then
34             f ←+ f_i
35     UF ←+ [b_i, f];
36 return UF
```

example, in HTML files, URIs are identified by the action tag, and keys are identified by tags such as input.

3.2 Pattern-based Static Analysis

3.2.1 Extraction of the URI Set

Starting with the URIs extracted from the frontend, LARA extracts URIs from the backend program that handles HTTP requests. This involves two steps: identifying the functions that handle URIs in the program (substep ② in Figure 3) and using these functions to identify more URIs (substep ③). We studied the backend programs of mainstream firmware and found that the code to bind URIs with URI handling functions can be categorized into three types.

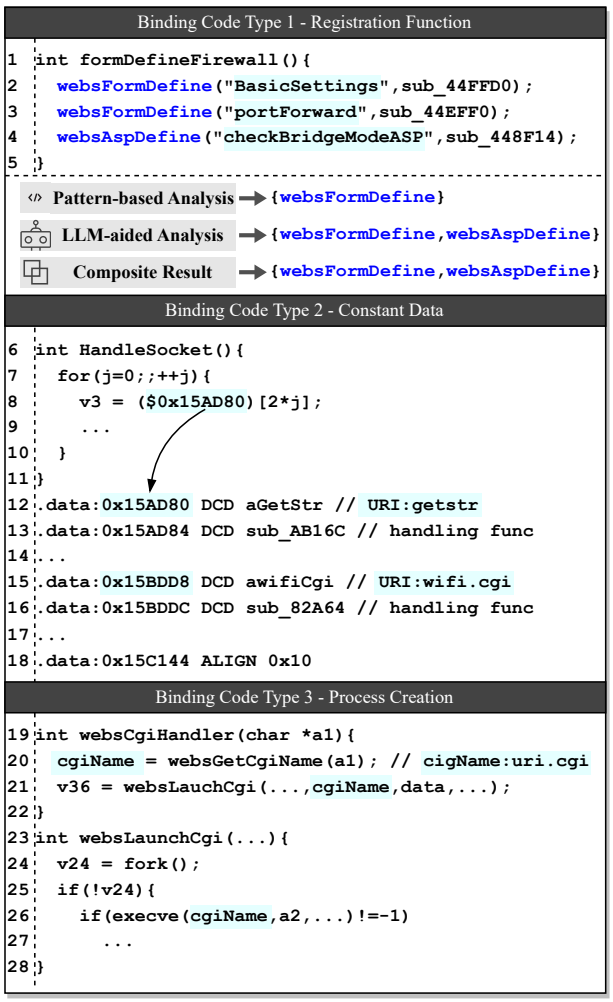

Figure 4: Examples of URI binding code types. The URIs and URI registration functions are marked with blue color.

Binding Code Type I — Registration Function. In line 2 of Figure 4, BasicSettings is an URI, which is an argument for the function websFormDefine, while the other argument is the entry address of the function sub_44FFD0. It means that the request body in the HTTP packet with the URI BasicSettings will be processed in function sub_44FFD0. We refer to functions such as websFormDefine and websAspDefine as *registration functions*, which are mainly used to bind the URIs with their corresponding handling functions.

Binding Code Type II — Constant Data. The Define Constant Data (DCD) instruction in lines 12 to 13 of Figure 4 allocates a continuous memory initialized with the address of the string getstr and the function sub_AB16C to handle the URI getstr. The URI and corresponding URI handling function are stored in pairs in the .data segment memory addresses range of the binary. When receiving an HTTP packet, the program matches the URI by traversing the address range and uses the content of next instruction as the corresponding URI handling function entry address to invoke (lines 6 to 11).

Binding Code Type III — Process Creation. As shown in lines 19 to 28 of Figure 4, the function websGetCgiName extracts the URI from HTTP requests and passes it to the function websLauchCgi as an argument. The function websLauchCgi launches a new process using fork and passes HTTP requests to the new process through an environment variable. This newly launched process handles the user input in HTTP requests, and the launched program is uri.cgi. The corresponding URI handling function resides in the launched program.

Algorithm 1 describes how LARA extracts URIs in detail. LARA utilizes the URIs extracted from the frontend to analyse URI handling code in the program. If an URI value matches a program name that handles HTTP requests, it is classified as the type III (line 5). If the instruction calling the URI is located in the .data segment, it is considered type II. However, as multiple *Address Ranges* may exist, each instruction that operates on the URI string must be recorded (line 9). If the instruction is in the .text segment, it is classified as type I. LARA decompiles the function containing the instruction to generate an abstract syntax tree (AST) (line 11). It then records the function with URI as an argument in AST traversal (lines 14 to 16). Simultaneously, LLM analyzes this code segment (line 12) and merges its results with the results obtained from pattern-based static analysis (line 17).

For type I, LARA decompiles each function containing call registration function instructions and then traverses the AST to resolve the parameters of the registration functions (lines 19 to 24). For type II, LARA traverses upwards and downwards from the instruction calling the URI string address (lines 27 to 29). For type III, the main function and functions without callers are considered URI handling functions empirically (lines 32 to 35). For the main function, the reason is that it's often the handler of the base URI (e.g., upgrade.cgi). For functions without callers, the reason is that some caller-callee relations are lost during the decompilation and we aggressively count functions without callers to avoid FNs.

3.2.2 Extraction of the Key Set

LARA uses a method similar to URI extraction to extract key. When the program parses the HTTP request body, the function that handles these data extracts the value corresponding to the key. For example, in Figure 5, the key time is extracted form the frontend. The function websGetVar extracts the value corresponding to the key time from the HTTP request represented by the wp variable in the URI handling function formSetSchedLed and assigns it to the local variable time_interval. LARA can also extract the hidden key ledCloseType, which cannot be found in the frontend file, through the same key handling function websGetVar.

LARA also needs to make sure that the extracted key is associated with the HTTP request. That is, the function takes the request packet as input and extracts the value matching the key from it, rather than simply comparing or concatenating the key string or extracting the *Key-Value* from other sources such as environment variables or configuration files. In Figure 5, string time is passed as an argument not only to websGetVar but also to strncmp. If whether the key time is relevant to the HTTP request is not checked, strncmp will also be treated as a function that handles keys, which will cause the string close to be incorrectly extracted as a key, maybe leading to a large number of false positives. The solution is straightforward as LARA needs to check if the parameters of the key handling function and the URI handling function have an intersection.

Algorithm 2 describes how LARA extracts keys in detail. At first, LARA decompiles the function which the instruction

Key Handling Code

```
1  int formSetSchedLed(wp,path,query){
2    time_interval = websGetVar(wp,"time","00:00-06:30"); // non-hidden
3    close_type = websGetVar(wp,"ledCloseType","allclose"); // hidden
4    if(!strncmp(old_sched_led_type,"time",4)&&!strncmp(sched_led_type,
5       "close",5)){
6      ResponseResultString = GetResponseResultString(v6);
7      ...
8      respCgiSendResp(v6,v7,"PWD_password",query);
9    }
10   get_mib_2cJSONString(root,"config","w15g.bss.wps.basic_have_config"
11                        ,&wifi_buf_entry);
12   ...
13   return sub_1BB34("PWD_password",query);
14 }
```

Pattern-based Analysis → {websGetVar,respCgiSendResp,strncmp,sub_1BB34}

LLM-aided Analysis → {websGetVar,get_mib_2cJSONString,GetResponseResultString}

Composite Result → {websGetVar}

Figure 5: An example of key handling code.The keys and key handling functions are marked with green color.

Algorithm 2: Extract Key from Backend Binary

Input: k for the key set extracted from frontend
UF for the corresponding set of URI and URI handling function
Output: $KFunc$ for the key handling function set
UK for the URI, key corresponding set

1 $KFunc \leftarrow \emptyset$;
2 $UK \leftarrow \emptyset$;
3 **foreach** $k_i \in k$ **do**
4 **foreach** $f \in get\_xref\_func(k_i)$ **do**
5 $r1, r2 \leftarrow \emptyset$;
6 $AST \leftarrow$ Decompiler(f);
7 $r2 \leftarrow$ LLM_KeyAnalysis(f);
8 **while** *visit(AST)* **do**
9 **if** *is_call(expr)* **then**
10 **if** $k_i \in expr.arg \wedge f.arg \cap expr.arg \neq \emptyset$ **then**
11 $r1 \xleftarrow{+} expr.name$;
12 $KFunc \xleftarrow{+} r1 \cap r2$;
13 **foreach** $[u_i, f_i] \in UF$ **do**
14 $AST \leftarrow$ Decompiler(f_i);
15 $\mathcal{K} \leftarrow \emptyset$;
16 **while** *visit(AST)* **do**
17 **if** *is_call(expr)* $\wedge$ $expr.name \in KFunc$ **then**
18 $\mathcal{K} \xleftarrow{+}$ getStringArg($expr$);
19 UK $\xleftarrow{+} [u_i, \mathcal{K}]$;
20 **return** $KFunc, UK$

calling the key and generates the AST for the function (line 6). LARA records the function that handles HTTP requests and takes the key as an argument by traversing the *AST* (lines 8 to 11). Simultaneously, LLM analyzes this code segment (line 7) and the intersection of its results with pattern-based static analysis is taken as key handling functions (line 12). Then, LARA extracts the key set using the key handling functions. The key set is extracted separately in each URI handling function and associated with its corresponding URI record. To extract the key, LARA traverses the AST generated by decompiling the URI handling function and analyzes the parameters called by the key handling function (lines 16 to 18).

3.3 LLM-aided Analysis

Large language model can effectively enhance the effectiveness of pattern-based static analysis through code semantics comprehension. It can help reduce false positives(FPs) and false negatives(FNs) in the extraction results of URI and key.

LLM-aided analysis can identify more URI registration functions for URI binding code type I with clear code structures. As shown in Figure 4, due to the limited URI extracted from the frontend (substeps ① in Figure 3), pattern-based static analysis can only identify websFormDefine. But LLM-aided analysis can effectively identify all registration functions, namely websFormDefine and websAspDefine.

LLM contributes more on reducing FPs when extracting keys. In Figure 5, function formSetSchedLed handles key time and ledCloseType, and the real key handling function is websGetVar. Pattern-based static analysis and LLM-aided analysis generate FPs in their respective results. Due to the presence of two strings password and PWD_password extracted in the frontend, so when functions respCgiSendResp, strncmp and sub_1BB34 conform to the code patterns, they will be mistaken for key handling functions by pattern-based static analysis. For LLM-aided analysis, some misleading symbols lead to false positives. Arguments wl5g.bss.wps.basic_have_config and wifi_buf_enty misled the judgment of LLM, leading the function get_mib_2cJSONString to be misidentified as key handling function. Additionally, the function GetResponseResultString was mistaken due to the function name. However, despite the misleading arguments PWD_password and query, the function sub_1BB34 is not considered as a key handling function. This is because LLM also takes into account the code structure and contextual information of the function. It is worth noting that FPs in both cases have inconsistent reasons, so intersecting the results can effectively eliminate them. As shown in §2.2 on the LARA-Site, the pattern-based static analysis extracted 1,032 key handling functions, the LLM-aided analysis extracted 1,206, but only 388 were in both sets.

We designed corresponding LLM interaction models for extracting URIs and keys based on the LMQL [3], enabling LLM-aided analysis to produce valid results without manual interactions. The model consists of three parts three components: the **query** clause, the **from** clause and the **where** clause. The Figure 6 shows the model for automatically identifying URI registration functions. The **query** clause serves as the core and models the process of interaction with LLM. In the query clause, the text [CoT Prompts] [51] marked with blue color serves as specific few-shot examples that can be referred to when LLM runs. The text marked with green color indicates the parsed answers retrieved from LLM or the calls on them. For unmarked text, the content within quotation marks serves as prompts for interacting with LLM, which prompts can be a combination of strings and previous results, while the rest consists of Python programming statements, which also makes use of control-flow and branching behavior. The **from** clause denotes which LMQL backend to use, like GPT-3 model available via the OpenAI API [35]. The **where** clause directly operates on tokens, constraining the LLM in what it can generate. Constraints can be an arbitrary conjunction or disjunction of condition expressions which allow comparison and membership checks between standard python expressions [3]. For the example on line 16 in Figure 6, STOPS_AT(A2, "YES") expresses when the answer A2 is decoded, the decoding should stop as the specified phrase "YES" is encountered. In summary, the process of LLM-aided analysis is controlled by the Python programming statements in the **query** clause. When sending the prompt, the LMQL backend specified by the **from** clause is selected, and the answer of LLM is parsed according to the constraint defined in the **where** clause. Figure 7 shows the model for automatically identifying key handling functions, with the same underlying concept but a different design.

LLM interaction model for extracting URI Registration Function

```
1  query:
2    [CoT Prompts]
3    f = Locate_function(URI_reference_address)
4    "Analyze the function f pseudocode and identify each called function"
5    funcs = []
6    "[FUNCs]"
7    funcs += (FUNCs.list())
8    result = []
9    for func in funcs:
10     "Whether the function func is with registration functionality?"
11     "[A2]"
12     if A2 == "YES":
13       result.append(func)
14 from "GPT-3"
15 where
16   FUNCs over OPTIONS.split(",") and
17   STOPS_AT(A2, "YES")OR STOPS_AT(A2, "NO")
```

Figure 6: The Model for extracting URI Registration Function.

Prompt Design. For URI registration function, prompts check if the code fragment referencing a URI includes a function with registration functionality. For key handling function, prompts first check if the function arguments originate from HTTP requests, then infers from the code implementation whether the primary purpose of the function is to retrieve some information and assign a value.

Merge Operation. For URI binding code type I, LARA first

```
int save_encrypted_data(char *a1, char *a2){
    memset(s, 0, 0x200);
    snprintf(s, 0x200, "echo -n %s | openssl ... -out %s",
    ↪ a1, a2);
    return popen(s, "r");
}
```

Listing 1: An Example of the Wrapper Function

takes the intersection of the results of pattern-based static analysis and LLM-aided analysis to identify potential URI registration functions. Any additional results identified by LLM-aided analysis are then evaluated to determine if they conform to the code patterns, i.e., having one string and one function as parameters. For type II and III, pattern-based static analysis is sufficient on its own, as LLM involvement offers minimal benefits. For key handling function, the intersection of LLM-aided analysis and pattern-based static analysis results is taken. While LLM-aided analysis provides additional results compared to pattern-based analysis, these additional results cannot be verified using valid code patterns and are therefore discarded. This approach leads to a small and acceptable number of FNs (explained in §C).

3.4 Sink Extraction

The sinks are operations that may violate security rules leading to vulnerabilities when supplied with distrusted inputs. We categorize these sinks into function-call sinks, which include dangerous functions from standard library and their wrapper functions, and non-function-call sinks.

Some dangerous functions may not be invoked directly. Instead, they are wrapped up by some wrapper functions in either the main binary or shared libraries. Wrapper functions may pertain to a straightforward but easily overlooked scenario where the dangerous function that triggers the vulnerability is not in the same binary as the code that handles user inputs. It is worth noting that this scenario differs from the multi-binary scenario that KARONTE [38] addresses, as there is no inter-process communication involved. Listing 1 illustrates a wrapper function, save_encrypted_data, defined in a shared library and used to save encrypted data by dangerous function popen, which poses a command injection risk. In the case of CVE-B in Figure 1, the absence of the corresponding sink twsystem is also one of the reasons why SATC cannot detect this vulnerability. To date, all existing techniques have overlooked this type of sinks, resulting in the failure to detect many vulnerabilities. Therefore, the program and its shared library should be analyzed when extracting sinks. The dangerous function calls can be detected via function name matching, while for wrapper functions, we can collect all related programs and shared libraries, then recursively extract the wrapper functions of the dangerous functions. When analyzing wrapper functions, there may be multiple layers wrapper functions. Therefore, we have designed two strategies to improve accuracy of extracting wrapper functions in the backend programs and its shared libraries recursively. ❶ Using taint analysis ensures that the input to the dangerous function is user-control. ❷ Filtering out the wrapper functions which do not include any URI or key handling code.

It is worth noting that directly analyzing shared libraries during taint analysis and extracting wrapper functions from the shared libraries as sinks have the same effect. However, pre-analyzing shared libraries to extract wrapper functions can reduce the overhead caused by repetitive propagation and locating functions within the shared library.

For non-function-call sinks, vulnerabilities are caused by direct variable operations, which may affect the contents of variable values, the positions of array reads, the number of controlled loop iterations, and so on. Non-function-call sink can also exist within shared libraries, and it can be identified simultaneously during the process of extracting wrapper functions through taint analysis.

3.5 Taint Analysis

LARA uses a key-sensitive taint analysis engine to analyze the backend program that handles HTTP requests and tracks the data flow of the user input entries to detect potential vulnerabilities. The engine can decompile program binaries to produce AST and traverses the AST to propagate the taint information and detect security issues.

The Taint Source. During the analysis process, a taint source is considered to be the variable to which the value corresponding to the key is assigned. This variable can be an argument or a return value of the function.

Taint Tracking. Taint analysis begins at the URI handling function where the processing of the HTTP request body begins. Based on the AST, LARA can quickly obtain the type of each data and operation during the taint propagation process. We have defined a variable triplet consisting of the *taint property*, *allocated memory or value*, and *maximum allocatable memory* to record the state of each data. The taint property is categorized as *uninitialized*, *tainted*, and *not_tainted*. LARA leverages this triplet to record the memory size and taint property changes of each affected array, pointer, writable global variable, and data in classes and structures.

Taint Analysis. Taint propagation rules involve intra-function and inter-function propagation. Intra-function propagation is path-sensitive and utilizes ASTs and forward analysis to track data flow. It performs separate taint propagation for different branches and summarizes the results at branch convergence points. Taint status is assigned as tainted if a variable is tainted within a branch. Inter-function propagation follows the rules of analyzing only function calls with taint property parameters and categorizing functions into dangerous, imported, and general types for separate analysis. Dangerous functions, including standard library functions and their wrapper functions, are used as sinks and analyzed based on predefined rules. For

imported functions, LARA forcefully sets taint properties of return values and parameters to tainted. Although this may lead to errors in taint propagation, it also reduces the overhead caused by analyzing those functions in detail. For general functions, LARA decompiles them based on their entry addresses and performs taint analysis on the ASTs, propagating taint to variables corresponding to function parameters with taint attributes. And LARA determines the presence of vulnerabilities in the function based on contextual information before and after function calls and records whether other function parameters and return values are tainted.

Vulnerability Detection. LARA checks variable triplet and determines if a vulnerability can be triggered based on the sink models. Non-function-call sink models and dangerous standard library function sink models are predefined. Wrapper function models are extended based on the results of wrapper function extraction. LARA checks constraints when tainted data reaches a sink. For example, it assesses buffer overflow risks at strcpy by evaluating buffer constraints from functions like memset and stack length. Constraints from strncpy-like functions are also considered. Additionally, LARA evaluates other constraints, such as the impact of %s and %d on string formatting result, and only the first parameter of execv can lead to command injection vulnerability.

4 Evaluation

Implementation. We implemented the LARA prototype with over 10,200 non-comment lines of Python code. LARA leverages several other existing libraries or tools. The keyword analyzer is based on the standard XML processing library and JavaScript parsing library Js2Py [14]. All data flow analysis, including pattern-based static analysis, sink extractor and taint analyzer, is implemented using the disassembly engine of IDA Pro [20], which supports the analysis of programs across different architectures and the IDAPython [21] plugin. We implemented LLM-aided analysis using the LMQL model written in Python [42] and by invoking the GPT-3 model through the OpenAI library [35]. LARA focus on detecting memory-corruption vulnerabilities and command injection vulnerabilities, but also supports format string vulnerabilities, path traversal and other taint-style vulnerability detection.

Evaluation Questions. We evaluated LARA on real-world embedded systems to answer the following research questions (RQs):

- **RQ1.** How is the performance of LARA comparing with the state-of-the-art tools? (§4.2)
- **RQ2.** How effective each part of LARA is for discovering vulnerabilities? (§4.3)
- **RQ3.** Can LARA discover previously unknown real-world vulnerabilities in firmware? (§4.4)

Table 1: Experiment Configurations of LARA Variants.

Experiment Mode	Key-sensitive Taint Analysis	Key-pattern Analysis	URI-pattern Analysis	LLM-aided Analysis	Sink Extraction
LARA-Sink	✔	✗	✗	✗	✔
LARA-Key	✔	✔	✗	✗	✗
LARA-Pattern	✔	✔	✔	✗	✗
LARA-LLM	✔	✗	✗	✔	✗
LARA-Combined	✔	✔	✔	✔	✗
LARA	✔	✔	✔	✔	✔

4.1 Evaluation Setup

Dataset. To evaluate LARA, we collected samples from SATC [10], KARONTE [38], EMTAINT [12], FIRMAE [27] and other real-world devices as firmware dataset. The dataset comprises 203 firmware samples from 21 vendors, including 10 different device types, covering 80 Routers, 37 APs, 20 Switches, 19 IPCameras, 17 Firewalls, 11 Range Extenders, 6 VPNs, 6 Modems, 4 Bridges, 3 NAS. The selection process was guided by several crucial criteria, including popularity, frequency of being tested, development activeness, and functionality diversity. To have a fair ground truth, we collected all known buffer overflow vulnerabilities and command injection vulnerabilities that exist in these firmware samples from CVE records [13], while filtering out vulnerabilities without detailed information. For example, AXIS have no relevant vulnerabilities, whereas most of the vulnerability information for NetGear and Zyxel is not publicly available. In the end, we totally collected 646 known vulnerabilities, as illustrated in the fourth column in TABLE 2. Furthermore, we manually determined the URI binding code type for each vulnerability and checked whether the source was from hidden data. Our findings indicate that the dataset comprehensively covers all types of URI binding code, and that 84 of 645 vulnerabilities were attributable to hidden data.

Baselines. We compared LARA with three SOTA tools and various variants of LARA presented in TABLE 1. KARONTE [38], SATC [10] and EMTAINT [12] are the most relevant tools, while the variants of LARA are used to demonstrate the benefits of each technique in the paper.

- KARONTE, which focuses on vulnerabilities caused by interactions between multiple binaries. Its idea was integrated by SATC and LARA.
- SATC, which uses sources extracted from the frontend, and sinks are predefined functions.
- EMTAINT, which utilizes on-demand alias analysis to enhance taint tracking. Its sources and sinks are predefined.
- LARA-Sink, which uses the same sources with SATC and the sinks provided by sink extraction. Comparing LARA with this variant helps to evaluate the effectiveness of the sink extraction module (§3.4).
- LARA-Key, which considers key pattern to extract sources and uses the same sinks with SATC. Comparing LARA with this variant helps to evaluate the effectiveness of the key-pattern analysis module (§3.2.2).
- LARA-Pattern, which combines URI and key patterns to

Table 2: Vulnerability Detection Result of LARA and SOTAs on the Dataset. (x(y) means #All(#Hidden). "Prec." and "Reca."Indicate Precision and Recall, respectively. Time is overall analysis time which unitIs min.)

Vendor	Samples	URI Type	#Vuln	LARA						SATC						KARONTE					
				TP	FP	FN	Prec.	Reca.	Time	TP	FP	FN	Prec.	Reca.	Time	TP	FP	FN	Prec.	Reca.	Time
Tenda	19	I&II	233 (51)	231 (51)	28	2	89.2%	99.1%	4,306	64 (1)	89	169	41.8%	27.5%	8,903	12	23	221	34.3%	5.2%	9,120
TOTOLink	11	II&III	134 (13)	134 (13)	32	0	80.7%	100.0%	736	8	23	126	25.8%	6.0%	5,280	0	5	134	0.0%	0.0%	4,667
H3C	13	I&I&III	88	88	9	0	90.7%	100.0%	894	0	22	88	0.0%	0.0%	6,240	0	11	88	0.0%	0.0%	6,240
DLink	14	I&II&III	67 (11)	62 (11)	13	5	82.7%	92.5%	1,603	0	42	67	0.0%	0.0%	6,720	8	22	59	26.7%	11.9%	5,903
TRENDnet	13	I&II&III	23 (2)	23 (2)	11	0	67.6%	100.0%	1,103	3	10	20	23.1%	13.0%	5,080	0	3	23	0.0%	0.0%	5,914
Wavlink	3	III	21	21	9	0	70.0%	100.0%	96	1	6	20	14.3%	4.8%	903	0	2	21	0.0%	0.0%	1,003
NetGear	34	I&II&III	16 (1)	16 (1)	15	0	51.6%	100.0%	8,799	1	11	15	8.3%	6.3%	16,011	11	15	5	42.3%	68.8%	14,080
Cisco	4	II&III	10	10	8	0	55.6%	100.0%	153	1	16	9	5.9%	10.0%	603	3	6	7	33.3%	30.0%	667
Linksys	13	I&II&III	9 (3)	9 (3)	0	0	100.0%	100.0%	768	0	15	9	0.0%	0.0%	2,301	0	3	9	0.0%	0.0%	3,005
DrayTek	9	I&II&III	9	9	5	0	64.3%	100.0%	1,203	0	0	9	0.0%	0.0%	3,908	0	0	9	0.0%	0.0%	4,320
TPLink	8	I	9 (1)	8 (0)	3	1	72.7%	88.9%	803	1	0	8	100.0%	11.1%	1,608	0	0	9	0.0%	0.0%	1,303
Zavio	1	III	9	9	1	0	90.0%	100.0%	82	0	2	9	0.0%	0.0%	480	0	0	9	0.0%	0.0%	480
Motorola	2	I	6 (2)	6 (2)	2	0	75.0%	100.0%	147	0	3	6	0.0%	0.0%	960	0	1	6	0.0%	0.0%	960
Belkin	3	II&III	5	5	0	0	100.0%	100.0%	283	3	4	2	42.9%	60.0%	803	2	1	3	66.7%	40.0%	966
SonicWall	2	II&III	4	4	1	0	80.0%	100.0%	182	0	0	4	0.0%	0.0%	960	0	0	4	0.0%	0.0%	960
ASUS	18	I&II	1	1	0	0	100.0%	100.0%	2,108	0	3	1	0.0%	0.0%	5,166	0	3	1	0.0%	0.0%	3,806
QNAP	5	III	1	1	0	0	100.0%	100.0%	603	0	3	1	0.0%	0.0%	1,608	0	0	1	0.0%	0.0%	1,303
Fortinet	1	II	1	1	2	0	33.3%	100.0%	20	0	0	0	0.0%	0.0%	480	0	0	0	0.0%	0.0%	480
Zyxel	25	I&III	0	0	0	0	0.0%	0.0%	3,260	0	0	0	0.0%	0.0%	9,605	0	0	0	0.0%	0.0%	7,221
Mercury	3	I&III	0	0	3	0	0.0%	0.0%	228	0	0	0	0.0%	0.0%	1,440	0	0	0	0.0%	0.0%	920
AXIS	2	III	0	0	0	0	0.0%	0.0%	51	0	0	0	0.0%	0.0%	452	0	0	0	0.0%	0.0%	960
Total	203	—	646 (84)	638 (83)	142	8	81.8%	98.8%	27,428	82 (1)	249	563	24.8%	12.7%	79,511	36	95	609	27.5%	5.6%	74,278

extract sources and uses the same sinks with SATC. Comparing LARA with this variant helps to evaluate the effectiveness of the pattern-based static analysis module (§3.2).

- LARA-LLM, which only adopts LLM-aided analysis to extract sources and uses the same sinks with SATC. Comparing LARA with this variant helps to evaluate the effectiveness of the LLM-aided analysis module (§3.3).
- LARA-Combined, which combines pattern-based static analysis with LLM-aided analysis to extract sources and uses the same sinks with SATC. Comparing LARA with this variant helps to evaluate the effectiveness of the entire source extraction module.

Configurations. LARA and its various variants were executed on a host machine with an 8-core Intel Xeon Processor and 128GB of RAM running the Ubuntu 18.04 operating system. Firmware analysis was conducted and any analysis exceeding 8 hours was recorded as 8 hours.

4.2 RQ1: Comparison with the SOTA tools

We conducted a comparative analysis between LARA, SATC and KARONTE, focusing on their capabilities in finding known vulnerabilities, extracting source and sink information, and incurring time overhead.

Overall Results. TABLE 2 presents the results of LARA, SATC and KARONTE on firmware samples. LARA identified 780 potential vulnerabilities, with a precision of 81.8%. It missed 8 vulnerabilities, having a recall of 98.8%. On the other hand, SATC reported 331 potential vulnerabilities with a precision of 24.8%. Moreover, SATC achieved a low recall of 12.7%. KARONTE reported 131 potential vulnerabilities with a precision of 27.5% and a recall of 5.6%. In particular, all vulnerabilities detected by SATC and KARONTE are covered by LARA. In summary, LARA significantly outperformed SATC and KARONTE concerning precision and recall in detecting vulnerabilities in firmware samples; i.e., it improved the precision of SATC by 57.0% and KARONTE by 54.3%, while improving recall by 86.1% and 93.2%.

Regarding detecting of vulnerabilities triggered by hidden data, LARA successfully identified 83 of 84 vulnerabilities, while SATC only caught one. Upon further manual analysis, we discovered that SATC only coincidentally identified the vulnerability. This is because the key which caused the vulnerability was also in the frontend file, but with a different corresponding URI. SATC only analyzed the key, resulting in the detection of the hidden vulnerability.

FP Analysis for LARA. We analyzed all false positives in LARA, and summarized two reasons. 105 false positives occurred due to LARA disregarding critical key operations between the source and sink, like filtering or format-checking functions. For example, a string check on user input filters out certain special characters to avoid command injection vulnerabilities. However, incomplete filtering can be bypassed leading to command injection vulnerabilities. Vulnerability CVE-2022-45996 that found on Tenda W20E by LARA allows for command injection using the character $, despite the code checking for characters ; \ &. Additionally, incorrect identification of sinks leads to 37 false positives. This includes cases where conditional checks are used in wrappers functions to prevent data from reaching dangerous functions.

FN Analysis for LARA. 8 vulnerabilities were not detected by LARA due to either the complex inter-process communication (IPC) method or problems with the disassembly engines itself. Some embedded systems use complex IPC communication methods without keywords, leading to untraceable inter-process data flows and resulting in 3 false negatives. In addition, some functions cannot generate pseudo-code due to the disassembly engine itself, specifically missing function parameters, which prevents regular data flow analysis. After manual intervention and testing, the pseudo-code was

Table 3: Overall Comparison Results of LARA with Baselines.

Tools	Vulnerability Detection				URI Registration Fun.			URI			Key Handling Fun.			Key		
	#Alert	TP	Prec.	Reca.	#	TP	Prec.	#	TP	Prec.	#	TP	Prec.	#	TP	Prec.
SATC	331	82	24.8%	12.7%	—	—	—	5,201	4,329	83.2%	—	—	—	34,081	17,959	52.7%
LARA-Sink	269	168	62.5%	26.0%	—	—	—	5,201	4,329	83.2%	—	—	—	34,081	17,959	52.7%
LARA-Key	782	498	63.7%	77.1%	—	—	—	—	—	—	1,032	383	37.1%	168,952	102,597	60.7%
LARA-Pattern	685	498	72.7%	77.1%	203	70	34.5%	24,969	24,969	100.0%	1,032	383	37.1%	155,664	102,433	65.8%
LARA-LLM	690	504	73.0%	78.0%	188	122	64.9%	27,823	27,714	99.6%	1,206	388	32.2%	133,266	102,611	77.0%
LARA-Combined	603	504	83.6%	78.0%	132	122	92.4%	27,781	27,714	99.8%	388	383	98.7%	102,905	102,597	99.7%
LARA	780	638	81.8%	98.8%	132	122	92.4%	27,781	27,714	99.8%	388	383	98.7%	102,905	102,597	99.7%

Table 4: Vulnerability Detection Result of EMTAINT.

Tools		TP	FP	FN	Prec.	Reca.
EMTAINT	Prototype	14	275	632	4.8%	2.2%
EMTAINT+LARA	EnhanceI	212	162	434	56.7%	32.8%
	EnhanceII	259	168	387	60.7%	40.1%
EMTAINT +SATC	EnhanceI	45	226	601	16.6%	7.0%
	EnhanceII	47	231	599	16.9%	7.3%

fixed, and LARA could detect these five previously missed vulnerabilities. An in-depth investigation of the URIs and keys responsible for these vulnerabilities revealed that LARA had identified them already.

FP and FN Analysis for SATC. The false positive is caused by imprecise identification of key in the backend binary and harmless processing operations. First, the key extracted by SATC is called at multiple locations, where only one represents a controllable user input while the others are uncontrollable strings. A path from an uncontrolled string to a dangerous function may lead to a false positive. Second, some keys are in unreachable functions. However, for LARA, these false positives can be avoided by separating and recombining URIs and keys and extracting the key through key handling code. Additionally, Our manual analysis of false negatives for SATC found two primary reasons: SATC failed to extract some non-hidden or hidden keys and lacked certain sinks. Out of 563 vulnerabilities not found compared with LARA, 441 were caused by failure to extract corresponding keys, 105 were due to the lack of sinks, and 17 were attributed to the absence of both keys and sinks.

FP and FN Analysis for KARONTE. KARONTE focuses on the shared data between binaries but ignores on the entry points of the user input. Thus uncontrolled sources (such as fgets) lead to most false positives. The dataset contained a total of 58 IPC vulnerabilities, out of which KARONTE failed to detect 22. Among these, 13 were attributed to unrecognized IPC paradigms (such as apmib_set and apmib_get), while the remaining 9 were due to a lack of sources.

Time Overhead. The analysis time overhead of KARONTE, SATC and LARA is presented in TABLE 2. It shows that LARA requires less time than SATC and KARONTE. Overall, the total analysis time was reduced by 65.5% and 63.1%.

Source Extraction. TABLE 3 presents an overview of the URI and key extraction results obtained by LARA in the dataset. LARA extracted a total of 27,781 URIs and 102,905 keys from the dataset with a precision of 99.8% and 99.7%,

Table 5: Results of Sink Extraction in the Dataset by LARA.

Sample	Type	#Sink Func.	Max Wrap Times	FP
DLink DIR816	Router	9	2	1
Tenda AC9	Router	12	3	0
TOTOLink A7100RU	Router	31	3	2
NetGear EX6200	AP	42	3	4
H3C H100	AP	23	3	1
TRENDnet TV-IP110WN	IPCamera	2	2	0
Cisco RV110W	FireWall	18	2	0
Draytek G1282	Switch	21	3	2
Total	—	**158**	—	**10**

including 9,299 hidden URIs and 15,411 hidden keys. In contrast, SATC identified 5,201 URIs and 34,081 keys with a precision of 83.2% and 52.7%. Additionally, LARA detected 23,385 more URIs and 84,638 more keys than SATC, representing an increase of 5.4 times and 4.7 times. This indicates that some non-hidden data is not being detected by SATC, due to the incomplete keyword matching rules employed.

Sink Extraction. TABLE 5 presents the Sink extraction results of 8 firmware samples. The samples were selected based on the number of known vulnerabilities collected make sure that they come from different vendors or device types. LARA identified 158 such functions in 8 firmware samples, with 10 false positives identified with manual analysis, indicating a precision rate of 93.7%. These false positives are caused by ignoring data checks, i.e., the wrapper functions may check the data, leading to the user-controlled data not reaching the dangerous operations. The maximum number of wrapper function calls for these dangerous functions was 3, highlighting the significance of identifying them.

Enhanced EMTAINT Evaluation. To better assess the impact of additional sources and sinks, we conducted an analysis of EMTAINT that utilizes SSE-based on-demand alias analysis technique to enhance the analysis in embedded systems. As shown in TABLE 4, EMTAINT could find 14 known vulnerabilities. The uncontrolled sources, disregarding sanitizer operations cause false positives, while incomplete recognition of indirect calls and IPC methods lead to false negatives. With more sources from LARA and SATC, EMTAINT-EnhanceI could detect 212 and 45 vulnerabilities. Meanwhile with more sources and sinks from LARA and SATC, EMTAINT-EnhanceII could detect 259 and 47 vulnerabilities. Note that EMTAINT only used the provided sources and sinks in the enhanced evaluation. The results proves that our source and sink extraction methods are more effective than SATC, which leads to higher precision and higher recall.

Summary to RQ1: The performance superiority of LARA is evident. Compared with SATC, LARA detected 556 more vulnerabilities, extracted 5.4 times more URIs and 4.7 times more keys, and incurred 65.5% less time overhead. Compared with KARONTE, LARA detected 602 more vulnerabilities and incurred 63.1% less time overhead. With more sources and sinks from LARA, EMTAINT could detect 245 more vulnerabilities.

4.3 RQ2: Ablation Study

Contribution of Source Extraction. To evaluate the contribution of source extraction, we compared variants of LARA with SATC. The detailed results are represented in TABLE 3. Specifically, compared with SATC, LARA-Combined (with the same sink) detected an additional 422 vulnerabilities, thus achieving a 58.8% and 65.3% improvement in precision and recall, respectively. This can be attributed to the fact that LARA-Combined extracted 23,385 more URIs and 84,638 more keys than SATC while maintaining a higher precision rate. The detection of additional vulnerabilities was facilitated by both pattern-based static analysis and LLM-aided analysis, which contributed to an increase in precision. Specifically, with pattern-based static analysis enabled, LARA-Pattern identified 498 true vulnerabilities, achieving a precision rate of 72.7%. Similarly, with LLM-aided analysis enabled, LARA-LLM detected 504 true vulnerabilities, albeit with a precision rate of 73.0%. However, by combining these two methods, the precision rate improved significantly to 83.6%. Meanwhile, extracting key based on URI has been shown to enhance the precision of vulnerability detection, as evidenced by the data presented in the third and fourth row in TABLE 3. Specifically, compared to LARA-Key, LARA-Pattern improved the precision by 9.0% with regards to vulnerability detection and 5.1% in terms of key identification.

Contribution of LLM-aided Analysis. LLM-aided analysis contributes more on reducing the FPs than reducing the FNs. TABLE 3 shows that compared with LARA-Pattern, LARA-Combined reduces the FPR of vulnerability detection by 10.9%, the FPR of URI registration function identification by 57.9%, the FPR of key handling function identification by 61.6%, and the FPR of keys by 33.9%. Meanwhile, LARA-Combined reduces the FNs by detecting 6 more vulnerabilities, 52 more URI registration functions, 2,745 more URIs, and 164 more keys. Detailed data as show in §2.2 at LARA-Site, LLM-aided analysis detects more registration functions, leading to more URIs extracted in 44 firmware samples. Combining pattern-based static analysis and LLM-aided analysis, the key handling functions of 196 firmware samples were manually confirmed to be completely accurate.

FP Analysis for Source Extraction. There are 5 FPs in the combined results of key handling functions, which leads to 308 FPs of keys. Further analysis of the misstated keys showed that they lead to no FPs in vulnerability detection.

Table 6: 0-Day Vulnerabilities Discovered by LARA.

Vendor	#Series	#0-Day Vuln			#Hidden	#Wrapper Func.
		LARA	SATC	KARONTE		
Tenda	8	81	26	1	5	3
TOTOLInk	16	48	0	0	6	23
DLink	4	33	0	0	13	16
H3C	3	21	0	0	4	0
TRENDnet	5	15	7	4	1	3
Linksys	3	13	3	5	2	2
QNAP	3	12	0	0	0	0
Draytek	4	9	0	0	1	5
TPLink	4	4	0	0	0	0
ASUS	3	3	0	1	0	0
Cisco	2	3	0	0	0	0
Zavio	1	2	0	0	0	0
NetGear	1	1	0	0	0	0
Total	**57**	**245**	**36**	**11**	**32**	**52**

Contribution of Sink Extraction. Each sink has the potential to lead to a vulnerability with user-controllable data . The results presented in TABLE 3 indicate that LARA-Sink has identified 86 additional vulnerabilities in comparison to SATC. Moreover, LARA has identified 134 more vulnerabilities compared to LARA-Combined. This proves that sink extraction can effectively improve the ability to discover vulnerabilities. Despite a slight decrease in precision of 1.8%, it's an acceptable trade-off due to improved vulnerability detection.

Summary to RQ2: The evaluation proves the effectiveness of every component of LARA. Source extraction and sink extraction significantly contributes to detecting 556 additional vulnerabilities. Furthermore, the utilization of LLM-aided analysis and URI-pattern analysis helps to minimize FPs and FNs in the extraction of URI and key, resulting in a notable improvement in precision by 57.0%.

4.4 RQ3: Real-world Vulnerabilities

We also applied LARA to detect unknown vulnerabilities of the firmware dataset in the wild. Each vulnerability was verified on the latest version of the firmware sample. As illustrated in TABLE 6, LARA uncovered a total of 245 previously unknown vulnerabilities. So far, all 245 vulnerabilities have been confirmed by the corresponding vendors, and 162 of them have been assigned CVE IDs following responsible disclosure. Among these confirmed vulnerabilities, 32 (13.1%) were due to hidden data, and 52 (21.2%) were caused by dangerous wrapper functions.

Meanwhile, SATC and KARONTE were applied to analyze these firmware samples. The results were disappointing, with 36 and 11 vulnerabilities detected. Furthermore, after analysis of the vulnerabilities that not detected by SATC, LARA detected 32 more vulnerabilities caused by hidden data, at least 125 more vulnerabilities caused by non-hidden data and 52 more vulnerabilities caused by wrapper functions.

Summary to RQ3: LARA identified 245 vulnerabilities on different devices, including vulnerabilities caused by hidden or non-hidden data and multi-layer wrapper functions.

5 Discussion and Limitation

Adaptation for Other Protocols. Our evaluation focuses on detecting vulnerabilities caused by the HTTP protocol in the embedded systems. We also analyzed vulnerabilities caused by other key-value pairs-based protocols. LARA can detect vulnerability CVE-2021-27239 caused by the SSDP protocol and vulnerabilities [28] caused by the DHCP options field using methods similar to pattern-based static analysis.

Sanitizer for Taint Analysis. Some false alarms are often caused by incomplete consideration of user input legitimacy checks. Some operations, such as type conversion and length check, have been considered. However, some checks can be bypassed, such as CVE-2022-45996 mentioned in §4.2.

Unique Advantages of Taint Analysis. For fuzzing, black-box testing targeting real devices faces challenges in code coverage and crash monitoring. The web program of some devices restarts quickly after crashing, making it difficult to judge the crash through response packets. Additionally, some devices cannot be accessed by shell, which also brings difficulties to monitoring. Meanwhile, emulation-based fuzzing is limited by the success rate of emulation [53]. FIRMAE [27] successfully emulated 892 out of 1,124 firmwares. Out of 102 randomly selected firmwares from 1,124, FIRMAE determined that 80 were successfully emulated, but after manually confirmation, only 45 met the fuzzing requirement. Furthermore, the setup of specific complex functions is necessary before they can be tested. For symbolic execution, it requires a lot of time to solve constraints and explore paths. However, Time overhead is also an important criterion for evaluating the efficiency of static analysis. TABLE 2 shows that SATC still has significant time costs even when analysis paths are provided for the symbolic execution engine. Taint analysis is more efficient for complex data flows and diverse data structures, and IDA pro [20] provides a well-structured AST to support pseudocode analysing.

Threat to Validation. First, LARA only supports backend programs developed with C and C++. Second, the precision of the disassembly engine may impact the outcomes of data flow analysis for intricate program. Third, there may be potential errors in the labeling/ground truth manually confirmed.

Responsible Disclosure. We have responsibly disclosed all vulnerabilities we found. We provided detailed information and PoC to the vendors for each vulnerability, facilitating them to confirm and reproduce the vulnerability.

6 Releated Work

Static Analysis. Various static analysis techniques [30, 32, 57] have been developed to detect IoT vulnerabilities. Karonte [38] tracks interactions between multiple binaries, but overlooks user input resulting in many false positives. SATC [10] extends Karonte [38] by focusing on the frontend user input, but it is not enough. DTaint [11] detects taint-style vulnerabilities in firmware using pointer aliases, inter-procedural data flow, and data structure layout similarities, but also overlooks the backend logic. EmTaint [12] comes up with indirect call resolution and accurate taint analysis scheme based on a structured symbolic expression-based on-demand alias analysis technique. However, EmTaint only strengthens the analysis of function pointers and cannot pinpoint precise source in the program to guide the discovery of more vulnerabilities. Firmalice [41] uses concolic execution and program slicing to detect authentication bypass vulnerabilities in firmware, but the constraint solver limits its effectiveness. NeuTaint [40] tracks information flow using neural program embeddings and performs better than other taint analysis tools. They all miss many sources and sinks, which leads to numerous undetectable vulnerabilities.

Dynamic Analysis. Many works are based on fuzzing techniques [50, 55, 58] to detect vulnerabilities in IoT devices. SRFuzzer [59] and IoTFuzzer [8] employ black-box fuzzing to detect vulnerabilities. Simulation-based fuzzing has also become a research trend to reduce costs, with FirmAFL [60] and Fuzzware [39] improving on the simulation method of Firmadyne [7]. Static analysis techniques have been combined with fuzzing to improve testing efficiency [9, 33, 43]. Other works [18, 23, 54, 56] focus on firmware rehosting or verifying specific vulnerabilities. However, they suffer from low code coverage and only focus on memory-related vulnerabilities.

LLM for Program Analysis. With the popularity of LLMs, many researchers have begun to focus on their applications in program analysis, particularly in vulnerability detection [1, 34, 46, 48], program repair [24, 36], software testing [16, 17, 29] and program comprehension [19]. In addition, the work [37] has surveyed the assistance of GPT models in reverse engineering and reports promising results.

7 Conclusion

In this paper, we proposed LARA, a novel static taint analysis technique for detecting vulnerabilities in embedded systems by leveraging semantic relations in code and data, which is motivated by two key findings. First, user input entries can be classified into URIs or keys. Analyzing their relations and correspondence between the frontend and backend can systematically locate more sources. Second, due to the distinct sources of false positives between LLM-aided analysis and pattern-based static analysis results, LLM-aided analysis based on code semantic comprehension can effectively help to identify more accurate sources. The evaluation indicates that, in comparison to other state-of-the-art tools, LARA has a clear advantage in detecting more vulnerabilities with fewer false positives. Moreover, LARA identified 245 vulnerabilities in popular real-world embedded system firmwares, including 162 CVE IDs.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is partly supported by National Key R&D Program of China under Grant #2022YFB3103900, Strategic Priority Research Program of the CAS under Grant #XDCO2030200 and Chinese National Natural Science Foundation (Grants #62032010, #62202462, #62302500).

References

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. Fixing hardware security bugs with large language models. *CoRR*, abs/2302.01215, 2023.

[2] IoT Analytics. Iot connections market update. https://iot-analytics.com/number-connected-iot-devices/, 2022.

[3] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

[4] CheckPoint. Mvpower cctv dvr remote code execution (cve-2016-20016). https://www.checkpoint.com/advisory/cpai-2017-0863/, 2017.

[5] CheckPoint. Dasan gpon router authentication bypass (cve-2018-10561). https://advisories.checkpoint.com/advisory/cpai-2018-0459/, 2018.

[6] CheckPoint. The tipping point: Exploring the surge in iot cyberattacks globally. https://blog.checkpoint.com/security/the-tipping-point-exploring-the-surge-in-iot-cyberattacks-plaguing-the-education-sector/, 2023.

[7] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *23rd Annual Network and Distributed System Security Symposium(NDSS)*, 2016.

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *25th Annual Network and Distributed System Security Symposium(NDSS)*, 2018.

[9] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, and Zhi Xue. Sfuzz: Slice-based fuzzing for real-time operating systems. In *2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[10] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *30th USENIX Security Symposium*, 2021.

[11] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: Detecting the taint-style vulnerability in embedded device firmware. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[12] Kai Cheng, Yaowen Zheng, Tao Liu, Le Guan, Peng Liu, Hong Li, Hongsong Zhu, Kejiang Ye, and Limin Sun. Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA)*, 2023.

[13] CVE. Common vulnerabilities and exposures. https://cve.mitre.org/, 1999.

[14] Piotr Dabkowski. Js2py: Javascript to python translator. https://github.com/PiotrDabkowski/Js2Py, 2020.

[15] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. Pentestgpt: An llm-empowered automatic penetration testing tool, 2023.

[16] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023.

[17] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA)*, 2023.

[18] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. Sok: Enabling security analyses of embedded systems via rehosting. In *ACM Asia Conference on Computer and Communications Security(ASIACCS)*, 2021.

[19] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *46th International Conference on Software Engineering, ser. ICSE*, 2024.

[20] Hex-Rays. The interactive disassembler pro is a computer software disassembler which generates assembly language code from machine-executable code. https://hex-rays.com/ida-home/, 2005.

[21] Hex-Rays. An ida plugin which makes it possible to write scripts for ida. https://www.hex-rays.com/products/ida/support/idapython_docs/, 2019.

[22] IBM. Examining form data in an http request. https://ibm.com/docs/en/cics-ts/5.4?topic=applications-examining-form-data-in-http-request, 2019.

[23] Evan Johnson, Maxwell Bland, Yifei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium*, 2021.

[24] Harshit Joshi, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. Repair is nearly generation: Multilingual program repair with llms. *ArXiv*, abs/2208.11640, 2022.

[25] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging, 2023.

[26] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval, 2023.

[27] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *20th Annual Computer Security Applications Conference(ACSAC)*, 2020.

[28] Vu Thi Lan. The last breath of our netgear rax30 bugs - a tragic tale before pwn2own toronto 2022.

[29] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *45th International Conference on Software Engineering, ser. ICSE*, 2023.

[30] Amit Kr Mandal, Pietro Ferrara, Yuliy Khlyebnikov, Agostino Cortesi, and Fausto Spoto. Cross-program taint analysis for iot systems. In *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing*, 2020.

[31] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. Ai-based blackbox code deobfuscation: Understand, improve and mitigate, 2021.

[32] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: Pipelined symbolic taint analysis. In *24th USENIX Security Symposium*, 2015.

[33] Mattia Monga, Roberto Paleari, and Emanuele Passerini. A hybrid analysis framework for detecting web application vulnerabilities. In *ICSE Workshop on Software Engineering for Secure Systems(SESS)*, 2009.

[34] Marwan Omar. Detecting software vulnerabilities using language models, 2023.

[35] OpenAI. The openai python library provides access to the openai api written in the python language. https://pypi.org/project/openai/, 2020.

[36] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[37] Hammond Pearce, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Brendan Dolan-Gavitt. Pop quiz! can a large language model help with reverse engineering?, 2022.

[38] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy(SP)*. IEEE, 2020.

[39] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *31st USENIX Security Symposium*, 2022.

[40] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy(SP)*, 2020.

[41] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Annual Network and Distributed System Security Symposium(NDSS)*, 2015.

[42] ETH Zurich SRI Lab. Lmql is a programming language for large language models based on a superset of python. https://github.com/eth-sri/lmql, 2022.

[43] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: Automated iot firmware introspection and analysis. In *2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, 2019.

[44] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. When gpt meets program analysis: Towards intelligent detection of smart contract logic vulnerabilities in gptscan, 2023.

[45] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A. Laredo, and Alessandro Morari. Towards reliable ai for source code understanding. In *the ACM Symposium on Cloud Computing*, 2021.

[46] Mohammad Reza Taesiri, Finlay Macklon, Yihe Wang, Hengshuo Shen, and Cor-Paul Bezemer. Large language models are pretty good zero-shot video game bug detectors, 2022.

[47] Tenable. Netgear dgn remote unauthenticated command execution. https://www.tenable.com/plugins/nessus/104128, 2018.

[48] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. Transformer-based language models for software vulnerability detection, 2022.

[49] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI)*, 2009.

[50] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. Rpfuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing. *KSII Trans. Internet Inf. Syst.*, 7(8):1989–2009, 2013.

[51] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[52] wikipedia. Uniform resource identifier. https://en.wikipedia.org/wiki/Uniform_Resource_Identifier, 2023.

[53] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 2021.

[54] Wei Xie, Jiongyi Chen, Zhenhua Wang, Chao Feng, Enze Wang, Yifei Gao, Baosheng Wang, and Kai Lu. Game of hide-and-seek: Exposing hidden interfaces in embedded web applications of iot devices. In *The ACM Web Conference(WWW)*, 2022.

[55] Zelin Xu, Wei Huang, Wenqing Fan, and Yixuan Cheng. Fiotfuzzer: Response-based black-box fuzzing for iot devices. In *22nd IEEE/ACIS International Conference on Computer and Information Science(ICIS)*, 2022.

[56] Yao Yao, Wei Zhou, Yan Jia, Lipeng Zhu, Peng Liu, and Yuqing Zhang. Identifying privilege separation vulnerabilities in iot firmware with symbolic execution. In *24th European Symposium on Research in Computer Security(ESORICS)*, 2019.

[57] Xiaokang Yin, Ruijie Cai, Yizheng Zhang, Lukai Li, Qichao Yang, and Shengli Liu. Accelerating command injection vulnerability discovery in embedded firmware with static backtracking analysis. In *12th International Conference on the Internet of Things(IoT)*, 2022.

[58] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. Poster: Fuzzing iot firmware via multi-stage message generation. In *the 2019 ACM SIGSAC Conference on Computer and Communications Security(CCS)*, 2019.

[59] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. Srfuzzer: an automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities. In *35th Annual Computer Security Applications Conference(ACSAC)*, 2019.

[60] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium*, 2019.

A Case Study

LARA-Site provides a detailed analysis of previously undisclosed vulnerabilities from various sources, including non-hidden data, hidden data, and multi-layer wrapper functions.

B Rules Adopted by SATC and LARA

The URI and key identification rules adopted by LARA and SATC are specifically listed to illustrate their differences and clarify the gaps.

- **URI identification rules**: For URI binding code I, SATC introduces a URI-like concept but doesn't link URIs and keys for vulnerability detection or analyze backend code for them. LARA take this into account. For identification rules defined in URI binding code II and III, they are unique to LARA.
- **Key identification rules**: SATC only uses the backend location of the shared keywords extracted from the frontend as the source. LARA first identifies the key handling function and then extracts the key in the backend, taking into account the corresponding pattern of URIs and keys.
- **LLM-aided analysis rules**: LLM-aided analysis identifies registration functions as URI handling functions and identifies functions that retrieve information from HTTP requests as potential source functions. This feature is unique to LARA.
- **Handle unreachable keywords through URIs**: This rule is adopted both by LARA and SATC, but the difference is that SATC only applies to certain URIs of URI binding code I.
- **Exclude key handling functions that share no arguments with URI handling functions**: This rule is unique to LARA.

C Detail in Compare URI and Key Extraction

The detailed data is presented in §2.2 on the LARA-Site, showing the results of URI and key extraction in each firmware sample by LARA and SATC in the dataset.

And since LLM-aided are not used for URI binding code type II and III, so URI extracted by LLM-aided and Combined are used from P.A. It means *27,823 = 17,138 + 10,685*.

In two firmware samples (G0 and G0-POE), there were 2 key handling functions missed , namely cJSON_GetDouble and cJSON_GetObjectItem. While LLM-aided analysis was able to extract these two functions (total four functions for two samples), they were not extracted by key-pattern analysis. However, since we took the intersection of the analysis results, we missed these two functions, resulting in 14 fewer keys being extracted. Further analysis of the missed key showed that they did not lead to any vulnerability.

In two firmware samples (BS228FX and BS252FX), IDA Pro does not support analysis of the eh_frame section, which makes it impossible to extract URIs.

D The Model for extracting key Handling Function.

LLM interaction model for extracting key handling function

```
query:
 [CoT Prompts]
 "Analyze the following C pseudocode and identify each called function"
 funcs = []
 "[FUNCs]"
 funcs += (FUNCs.list())
 result = []
 for func in funcs:
  "Whether the function func has the ability to retrieve information?"
  "[A2], [info]\n"
  if A2 == "YES":
   "Whether the source of info comes from an HTTP request?"
   "[A3]\n"
   if A3 == "YES":
    code = Decomplie([func])
    "Analyze the pseudocode code and speculate on the usage environment
, intended purpose and detailed function of func respectively"
    "[A4]\n"
    "Based on A4 judge whether the main purpose of function func is re-
lated to retrieving information"
    "[A5]"
    if A5 == "YES":
     result.append(func)
from "GPT-3"
where
 FUNCs over OPTIONS.split(",") and
 STOPS_AT(A2, "YES") or STOPS_AT(A2, "NO") and
 STOPS_At(info, "\n") and
 STOPS_AT(A3, "YES") or STOPS_AT(A3, "NO") and
 STOPS_AT(A4, "\n") and
 STOPS_AT(A5, "YES") or STOPS_AT(A5, "NO")
```

Figure 7: The Model for extracting key Handling Function.