EAGLEYE: Exposing Hidden Web Interfaces in IoT Devices via Routing Analysis

Hangtian Liu*†§, Lei Zheng†, Shuitao Gan†§✉, Chao Zhang†‡**, Zicong Gao*,
Hongqi Zhang*¶, Yishun Zeng†, Zhiyuan Jiang‖✉, Jiahai Yang†
*Information Engineering University †Institute for Network Sciences and Cyberspace (INSC), Tsinghua University
‡Zhongguancun Laboratory §Laboratory for Advanced Computing and Intelligence Engineering
¶Henan Key Laboratory of Information Security ‖National University of Defense Technology
**JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

***Abstract*—Hidden web interfaces, i.e., undisclosed access channels in IoT devices, introduce great security risks and have resulted in severe attacks in recent years. However, the definition of such threats is vague, and few solutions are able to discover them. Due to their hidden nature, traditional bug detection solutions (e.g., taint analysis, fuzzing) are hard to detect them. In this paper, we present a novel solution EAGLEYE to automatically expose hidden web interfaces in IoT devices. By analyzing input requests to *public* interfaces, we first identify `routing tokens` within the requests, i.e., those values (e.g., actions or file names) that are referenced and used as index by the firmware code (routing mechanism) to find associated handler functions. Then, we utilize modern large language models to analyze the contexts of such routing tokens and deduce their common pattern, and then infer other candidate values (e.g., other actions or file names) of these tokens. Lastly, we perform a hidden-interface directed black-box fuzzing, which mutates the routing tokens in input requests with these candidate values as the high-quality dictionary. We have implemented a prototype of EAGLEYE and evaluated it on 13 different commercial IoT devices. EAGLEYE successfully found 79 hidden interfaces, 25X more than the state-of-the-art (SOTA) solution IoTScope. Among them, we further discovered 29 unknown vulnerabilities including backdoor, XSS (cross-site scripting), command injection, and information leakage, and have received 7 CVEs.**

I. INTRODUCTION

IoT (Internet of Things) devices have been widely used in modern society. As reported [21], the number of IoT devices has reached 16.7 billion until 2023 and has been continuously growing. The explosive growth of IoT has been accompanied by increasingly serious security incidents [3].

Among various security issues of IoT devices, the hidden web interface issue is often overlooked due to its hidden nature. But it leaves undisclosed access channels to attackers and is very likely to cause serious incidents. For instance, the vulnerability CVE-2023-3519 in Citrix products [11] was confirmed as critical (with a CVSS score of 9.8), which comes with a typical stack overflow in a hidden interface, and could lead to unauthorized remote code execution in the device. In recent years, vulnerabilities and security incidents introduced by hidden web interfaces quickly grew, including but not limited to many serious vulnerabilities of network devices[1]. Thus, the security risks of hidden web interfaces in IoT devices demand special attention. However, few solutions address the hidden web interface threat, due to the following reasons.

First of all, the definition of such threats is vague. The recent solution IoTScope [39] argues that the hidden interface can lead to unauthenticated access but gives no clear definition. However, authenticated accesses could also have the hidden interface issue, i.e., hidden web interfaces after authentication can introduce significant security risks too. Instead, we define hidden web interfaces as *interfaces that are accessible but not disclosed to users in the devices' documentation*. Such hidden interfaces could be intentionally or unintentionally introduced by developers, and even inadvertently introduced without developers' awareness [35].

Second, it is difficult to discover hidden web interfaces, e.g., via traditional bug finding solutions. On one hand, there is no obvious pattern of hidden interfaces, making it hard to statically search such interfaces. On the other hand, there are no obvious consequences or feedback when the hidden interfaces are triggered, making it hard to dynamically test such interfaces (e.g., via fuzzing). For instance, fuzzing solutions [33], [45], [22], [16], [46], [25] are effective at finding bugs in software. Since the hidden interfaces are unknown, fuzzers cannot set such interfaces as targets to explore. But, fuzzers could mutate test cases and trigger hidden interfaces with a small probability. However, without an appropriate mutation strategy or runtime feedback mechanism, the probability of triggering such hidden interfaces is extremely low. Taint analysis is another type of popular solution to finding bugs in IoT [8], [31], [19], [10], [9], [43], [17]. But it can neither find hidden interfaces, since we can hardly define the taint source or taint sink related to hidden interfaces.

✉Corresponding authors: ganshuitao@gmail.com, jzy@nudt.edu.cn

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240399
www.ndss-symposium.org

[1]CVE-2024-3273 (D-link), CVE-2023-46805 (IVanti PCS), CVE-2022-0342 (Zyxel), CVE-2021-22893 (Pulse Secure), and CVE-2019-1653 (Cisco).

Lastly, since the hidden interfaces are not exposed, they are rarely explored or tested by regular bug finding solutions, making them more likely to be vulnerable to potential bugs and vulnerabilities.

To address this problem, we present a novel solution EAGLEYE to expose hidden web interfaces in IoT devices automatically. After conducting an extensive study of firmware from a variety of IoT devices (cf., Section II-B), we find that, an IoT device has multiple interfaces including public ones and hidden ones, which often share a similar pattern in the input requests, except the actions to perform or files to operate, which are denoted as *routing tokens* in the input requests. The root cause of this phenomenon is that, the underlying firmware code often has a specific *routing mechanism* to dispatch the input requests to different handler functions or services. For simplicity, these routing mechanisms often take certain values (i.e., the routing tokens) from the input requests and use them as the key or index to find associated handlers. As a result, we could analyze public interfaces to infer their routing tokens, then infer other candidate values from firmware, and finally generate new input requests by replacing these routing tokens with candidate values to explore other interfaces (including hidden ones). In this paper, we denote such a process as the *routing analysis*.

Specifically, routing tokens link input requests with the routing mechanism implemented in the firmware. Therefore, we first analyze input requests to *public* interfaces, and perform a token-based comparison to locate variable fields in the requests and identify routing tokens. Then, we analyze the contexts (including reference code and data) of these routing tokens and try to learn their common pattern, which is an intellectual task usually depending on expert experience and requiring much manual effort. To automate and streamline this process, we utilize the modern large language models (LLMs) to comprehend and analyze firmware code [38], [14] like humans. With the discovered pattern, we search for other similar token values within the firmware and mark them as candidate values for the routing tokens. These candidate values serve as a high-quality dictionary and can be used to replace the routing tokens and generate new input requests, able to explore new interfaces including hidden ones. Finally, we perform a hidden-interface directed black-box fuzzing, which explores hidden interfaces with these found routing tokens and tests the device to verify whether the explored interfaces are accessible and documented by the vendor. If an explored interface is accessible and undocumented, a hidden web interface will be reported. During the black-box fuzzing, we leverage responses from devices to assist our fuzzer in mutating seeds (i.e., requests) and catching exceptions during testing, promoting the fuzzing efficiency and finding vulnerabilities faster.

We have implemented a prototype of EAGLEYE and evaluated it on 13 different commercial IoT devices. EAGLEYE discovered 79 hidden interfaces on these devices, 25X more than the state-of-the-art (SOTA) solution IoTScope. Among these interfaces, we further found 29 unknown vulnerabilities including backdoor [2], XSS, command injection, and information leakage, and have received 7 CVEs.

In summary, this paper makes the following contributions:

- We present a novel solution EAGLEYE to automatically expose hidden web interfaces in IoT devices, which models the problem as a routing token searching problem and conducts a hidden-interface directed black-box fuzzing for efficient exploration.
- We present a novel routing analysis solution that integrates both traditional program analysis and large language models, to identify routing tokens, and infer their pattern and candidate values, which serves as a high-quality dictionary for the fuzzing campaign.
- We evaluated EAGLEYE on 13 different commercial IoT devices and discovered 79 hidden interfaces, among which 29 unknown vulnerabilities were found. Results showed that EAGLEYE significantly outperformed the SOTA baseline.

II. BACKGROUND AND MOTIVATION

A. Problem Explanation

Interface in the paper refers to the gateway for clients to access specific functionalities or services of a device. It establishes the rules for client-device interaction, effectively serving as a mutual agreement on how to request particular functionalities or services.

For optimal user experience, all device interfaces should be accessible to clients (e.g., browsers), through a centralized access portal. This portal acts as a unified gateway, offering a clear navigation structure that guides users to the various interfaces of the device, thereby defining what they can interact with and control. These interfaces are intended to be publicly accessible from the portal and are also documented in the device's user manual. This documentation ensures transparency and user familiarity with the device's capabilities. However, interfaces that are not included in the device's manual present a different scenario. These are the interfaces that remain invisible to the client, as they are neither documented nor traceable through standard web navigation. This lack of documentation and visibility renders them stealthy and elusive, which is the central focus of this paper. These are known as hidden interfaces, being undocumented and untraceable through web navigation, and thus offer no visual cues. Sometimes, hidden interfaces may be inadvertently exposed due to inadequate access controls, allowing access to internal functionalities or services. In other scenarios, functionalities or services may be outdated and concealed from the client by removing their references in the entry portal, yet they persist within the device. To formalize the discussion, we define interfaces documented in the device manual as **public interfaces**. [3] These are typically accessible through the entry portal. Conversely,

[2]Backdoor here refers to a hidden interface embedded within the firmware that allows unauthenticated access to control over the device.

[3]The device documentation may only reveal accessible functionalities or services, with corresponding interfaces classified as public.

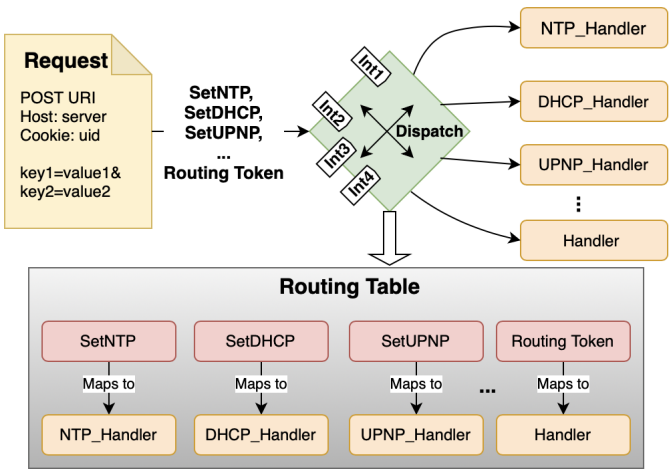


Fig. 1: An example abstract routing mechanism in IoT.

interfaces not documented but still accessible to clients are termed **hidden interfaces**. They are not listed in the entry portal, preventing navigation to them.

The hidden interface poses a significant risk to device security. These interfaces are often deeply embedded, being not visible to regular users and even developers. Since they are not subject to the same scrutiny and security measures as public interfaces, they may lack proper input validation, authentication, or other security controls. As a result, they can provide an avenue for unauthorized access, injection attacks, and other security breaches. Furthermore, hidden interfaces may not receive regular updates or patches, as they are often overlooked during maintenance. It means that any vulnerabilities in them may go unnoticed and unaddressed for extended periods, leaving devices long exposed to potential attacks.

B. Routing Mechanism in IoT Web

IoT devices always offer multiple functionalities or services, so there correspondingly exist multiple interfaces. For example, a VPN (Virtual Private Network) router usually supports multiple protocols (e.g., SSL, IPSec), thus it supports multiple interfaces to configure the device as demanded.

We conducted an extensive analysis of firmware from a diverse range of IoT devices, thoroughly examining their firmware code and observing their request-handling processes, which revealed that interfaces are ultimately managed by handlers within the firmware programs. When some interface is accessed, the corresponding handler is called into play. To correctly map interfaces to handlers, there exists a logical process called **routing mechanism** that parses the input requests from interfaces and delivers the data to the right handlers. The routing mechanism is generally integrated into modern web frameworks. Routing mechanisms in different frameworks vary widely.

There exists one key value called *routing token* in the request affecting the routing decision within the web. There are multiple routing tokens, each one indicating one specific interface. All routing tokens compose *routing table*, which defines the routing rules. Although routing mechanisms are diverse and lack a unified design, they all contain these two key elements. To be rigorous, we further define them as below:

- *Routing token* is one special value in the request, which indicates the specific interface accessed. It serves as a reference to make the routing decision, designating which handler should be called to deal with the data.
- *Routing table* contains the comprehensive set of routing tokens in the device. Every item in the table reflects the mapping relationship between a routing token and the corresponding handler.

The routing mechanism can be modeled and represented as Fig. 1: First, the client sends requests to the device and accesses its interfaces. The request carries the routing token to indicate the targeted interface. Then the request is parsed by web programs and matches the routing table to search for the right handler. Finally, the data in the request is handed on to the matched handler. It should be noted that the routing table is abstractly logical. In actual situations, it is not limited to the form of a real table. It can be implemented in various forms, generally distributed among programs, and organized through logical data structures.

The model depicted in Fig. 1 highlights the role of routing tokens in directing requests to the appropriate handlers. Confronted with the complexity and variability of firmware across different devices, this generalized model reveals that a majority of IoT devices employ a similar pattern in their routing mechanisms, can be modeled as two fundamental elements (i.e., routing token and routing table) and one process (i.e., dispatch).This insight is pivotal: Hidden interfaces, like their public counterparts, are very likely to undergo the same routing process, thus they may be discovered by examining this process, using clues provided by public interfaces.

C. Motivation

Why taint not applied. Discovering an interface means identifying a path from the input request to the processing handler. Ideally, to find hidden interfaces, it is required to trace the data flow and analyze the specific routing process. Taint analysis is popular to perform such tasks.

However applying taint analysis to uncover hidden interfaces is challenging due to: (1) It is difficult to precisely identify taint sources or sinks related to hidden interfaces, which are not explicitly defined; (2) Tracking the data flow along the interface is impeded by unresolved issues such as indirect calls and incomplete data recovery. Fig. 2 illustrates a real-world example of data flow along the interface. The routing token $actionName$ is embedded within the value of the $HTTP\_SOAP\_ACTION$ parameter. This token is subsequently used to index the appropriate handler in the $symTable$, a global variable in the runtime environment. The indirect nature of the handler call and the static recovery challenges of the table underscore the complexity. It is hugely challenging to identify which functions serve as handlers (i.e., sinks) and to trace the data flow due to the indirect call.

The complexity is further amplified in IoT web environments, where routing mechanisms are abundant and diverse. These mechanisms often involve multiple programs developed in various programming languages (such as C, Java, Lua, and PHP). The analysis of data flow across different programs and languages presents a formidable challenge.

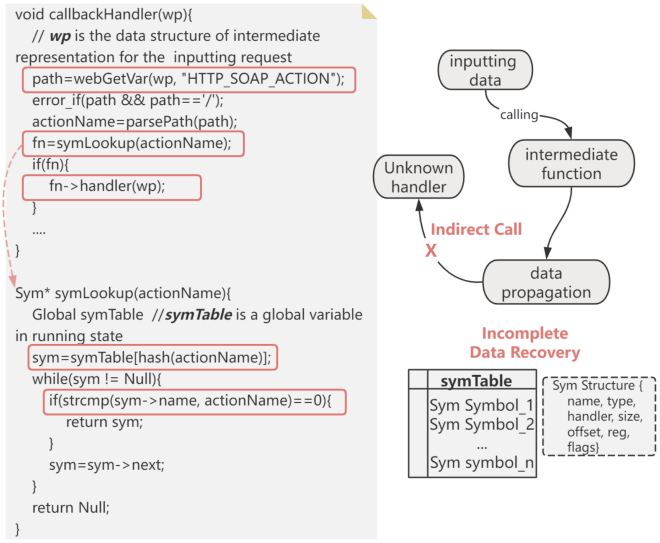


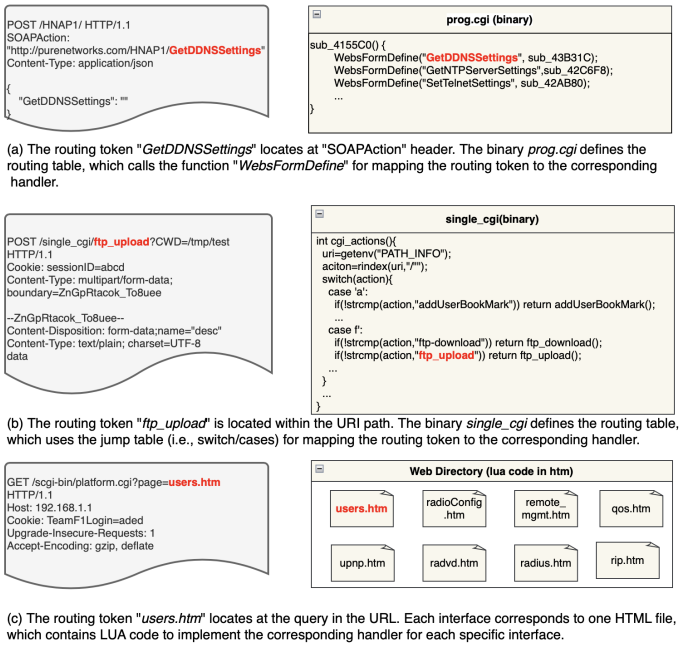
Fig. 2: A real-world example shows that it is hard to apply taint analysis to trace data flow along the interface, due to some open problems like indirect calls and incomplete data recovery.

Our intuition. The routing mechanism is closely related to the routing token in the request. We needn't analyze all data in the request. We could only pay attention to code slices related to the routing token. Through further research, we found most routing tokens share a similar pattern among their contexts. This provides a way to extract the routing table by pattern analysis, thus avoiding the burdensome data flow analysis.

Fig. 3 shows three different examples in the real world. In Fig. 3(a), the routing token $GetDDNSSettings$ is located at one header of the request. Meanwhile, it is referenced within one binary program *prog.cgi*. It is noticeable that several other string tokens are similarly referenced as the first argument to a function called $WebsFormDefine$. This function is utilized to establish the correlation between routing tokens and their corresponding handlers. Consequently, we can deduce the routing table by examining the invocation context of $WebsFormDefine$. In Fig. 3(b), the *routing token* ftp_upload is embedded within the URI path. It is also referenced within a binary program called *sing_cgi*. This token is positioned within a conditional judgment block of a jump table, which is structured as the switch/case statement. Each case within this table corresponds to a distinct interface. Other routing tokens are employed similarly across different case conditions. Therefore, by analyzing the structure and conditions of the jump table, we can effectively extract the routing table. In Fig. 3(c), the *routing token* $users.htm$ is found within the query of the URL in the request. Concurrently, it serves as the exact filename of an HTML file located within the web directory. This HTML file contains LUA code to manage the corresponding interface. The names of all these HTML files collectively form the routing table.

These cases reveal a consistent pattern within the context where the routing token is utilized in terms of code semantics and formatting. This consistency suggests a discernible methodology in how routing tokens are mapped to interfaces across different scenarios.

Beyond hidden interfaces' invisibility, they function similarly to public interfaces and can be accessed with prior knowledge.



(a) The routing token "*GetDDNSSettings*" locates at "SOAPAction" header. The binary *prog.cgi* defines the routing table, which calls the function "*WebsFormDefine*" for mapping the routing token to the corresponding handler.

(b) The routing token "*ftp_upload*" is located within the URI path. The binary *single_cgi* defines the routing table, which uses the jump table (i.e., switch/cases) for mapping the routing token to the corresponding handler.

(c) The routing token "*users.htm*" locates at the query in the URL. Each interface corresponds to one HTML file, which contains LUA code to implement the corresponding handler for each specific interface.

Fig. 3: Three real-world examples illustrate the intimate connection between the *routing token* in a request and the corresponding interface it triggers. In each case, the *routing table* reveals a consistent pattern, whether in terms of code semantics or formatting, within the context where the routing token is utilized.

D. Our solution and Main Challenges

As the routing token links the input test case (i.e. request) and the interface, we transform the problem of exposing hidden interfaces into the process of searching for hidden routing tokens. First, we identify routing tokens from public requests. Then we extract contexts among public routing tokens in programs, learning and deducing their common pattern. Next, with the learned pattern, we try to extract more similar ones and obtain the maximum approximate set of the routing table. Finally, with the help of the routing table as a dictionary, we perform a directed black-box fuzzing to mutate the field of the routing token. Black-box fuzzing is convenient and avoids the dependence on heavy instrumentation and rehosting.

This solution mainly faces three challenges:

1) **How to correctly identify the routing token in the request?** Routing token is the starting point of our analysis, it is important to identify the true one. However, the request contains a large number of tokens. Accurately identifying the routing token faces a lot of interference.
2) **How to extract the routing table in the firmware?** Though items in the routing table share a common pattern as depicted in Section II-C, it is difficult to summarize universal heuristic rules. Different serials of devices share different patterns, so there exists no unified standard pattern to extract the routing table.
3) **How to effectively perform the directed black-box fuzzing?** Although black-box fuzzing is convenient, it is ineffective due to lacking runtime feedback. In particular, it is difficult to generate high-quality test cases and is insensitive to exceptions without feedback as guidance.

III. Design

With a high-quality interface dictionary (i.e., the routing table) provided by relatively accurate routing analysis, the fuzzing campaign can focus on exposing hidden interfaces. The whole solution comprises two stages: intelligent routing analysis and directed black-box fuzzing, depicted as Fig. 4. Firstly, we conduct a token-based comparative analysis to identify public routing tokens. Given that the routing token is inherently the mutable field in requests, we look for variable tokens by comparing public requests. For multi-level interfaces (cf., Section III-A2), we additionally perform a hierarchy analysis to recover their data dependency. Secondly, we use identified routing tokens to extract their contexts in the firmware. Routing tokens with their contexts are then used to help LLM understand the task (i.e., learning their common pattern). We also design a correction feedback mechanism to guide LLM learning to the right results. Lastly, we carry out a directed black-box fuzzing toward hidden interfaces, avoiding heavy instrumentation and rehosting. To overcome the blindness, we combine the routing table and responses to guide the mutation, which distills compatible parameters to assemble requests (i.e., seeds). Concurrently, we catch hidden interfaces by judging the validity of the response via pattern.

A. Token-based Comparative Analysis

1) Variable Token Identification: Public interfaces stem from publicly available services, as detailed in the device's documentation. Consequently, we collect public requests consistent with these interfaces, adhering to the specifications outlined in the device documentation. The specific collection process of public requests is detailed in Appendix C.

Different interfaces correspond to different routing tokens. With collected public requests that access multiple public interfaces, the variable tokens can be located through comparative analysis which greatly reduces the searching scope for routing tokens.

Our solution is to use routing tokens to find the routing table in the vast code space. Correctly identifying the routing token from public requests is the basis of program analysis. The request is essentially a piece of text that conforms to the HTTP protocol specification and is rich in grammatical structure and semantics. Therefore, to identify the routing token from the request, it is necessary to consider the integrity of the grammatical structure and semantics, to avoid destroying the boundaries of the token. We first decompose the request into tokens, which is the basic granularity for implementing comparative analysis, ensuring semantic integrity in line with structural boundaries.

The routing token's field in the request (i.e., location) is high relative to the specific implementation. It may appear in URI, query, header, or body. Fig. 5 shows a specific example HTTP request. URI, which denotes the precise pathways to web resources, frequently features routing tokens. These tokens can be embedded within multi-tiered path structures, with the routing token potentially situated at any level. To effectively navigate this, we segment the URI into its constituent path tokens, and assign a path depth to each token, indicating its position within the hierarchical structure. Query, indicated by a $'?'$ following the URI, optionally carries client parameters and may contain routing tokens. They are parsed into key-value pairs, with keys identifying token fields. Header, which provides additional information in the request, is formatted as key-value pairs and can also include routing tokens. We similarly parse headers into key-value form. Body, used primarily for POST/PUT methods and optional in requests, carries the bulk of data and can be in various formats like JSON and XML. We transform body parameters into key-value pairs for token field identification. Nested parameters are handled recursively to create multi-level key-value pairs, allowing for the tagging of tokens within complex structures.

When decomposing a request into tokens based on its grammatical structure, we compare tokens within the same fields to identify variables, as shown in algorithm 1 line 3. Variable tokens are then considered as potential routing tokens. Several token types, such as session tokens, timestamp tokens, and normal tokens, can be variable and may interfere with routing token identification, necessitating further elimination, as indicated in algorithm 1 line 4. Session tokens are typically stable within a session and change only upon session switch. Timestamp tokens, used for client-server synchronization, vary over time. Leveraging these temporal characteristics allows us to exclude them from consideration. Normal tokens, often user-submitted data, are controllable and can be identified using magic strings by traffic analysis.

2) Hierarchy Analysis: Some interfaces are hierarchical. For example in Fig. 5, the $diagnostic$ interface provides a functionality to check network state, which is more specifically divided into two functions including $PING$ and $TRACEROUTE$, and further divided into IPv4 and IPv6 according to IP version. For this interface, $diagnostic$ is the first-level interface, which contains two second-level sub-interfaces, $PING$ and $TRACEROUTE$. Each sub-interface

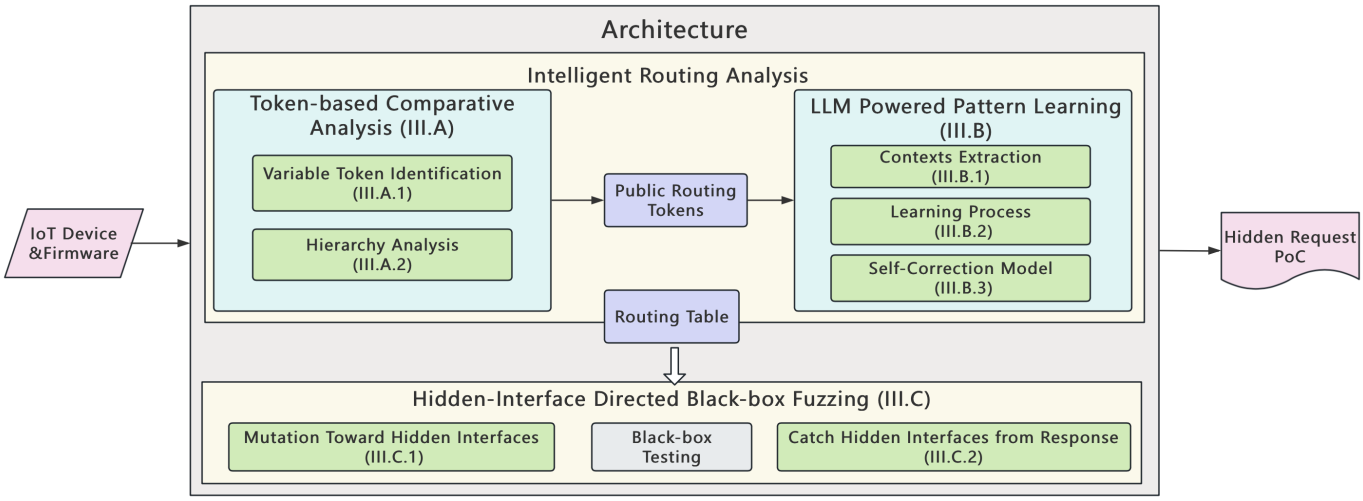


Fig. 4: Workflow Overview.

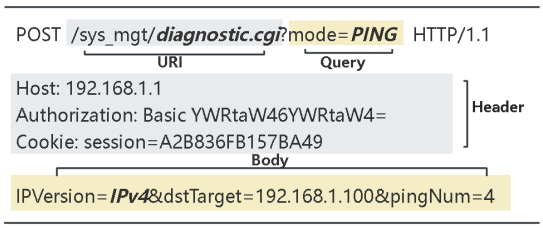


Fig. 5: A request example to illustrate grammatical structure in HTTP.

Algorithm 1: Token-based Comparative Analysis

Input: public requests $PubReqs$
Output: routing token set $RToken$, hierarchy $Hier$ for multi-level routing tokens.

1 **foreach** $req \in PubReqs$ **do**
2 $tokens \leftarrow$ `Parse(`req`)` `/* Parsing request into tokens, which are tagged by their field locations. */`
3 $VTokens \leftarrow$ `SelVarToken(`$tokens$`)` `/* Identify tokens varying with interfaces. */`
4 $RToken \leftarrow$ `Filter(`$VTokens$`)` `/* Filter interference tokens. */`
`/* Analyze hierarchy for multi-level tokens */`
5 **if Size**$(RToken \geq 2)$ **then**
6 **foreach** $field : token \in RToken$ **do**
7 $reqs \leftarrow$ `SelReqs(`$token$`)` `/* Select requests involving the routing token. */`
8 $subRToken \leftarrow$ `Search(`$reqs$`)` `/* Search subordinate routing tokens. */`
9 **if Size**$(subRToken) \geq 1$ **then**
10 $Hier \leftarrow$ `Edge(`$field : token, subRToken$`)`

is further divided into two versions, i.e., $IPv4$ and $IPv6$. For hierarchical interfaces, there exist correspondingly multi-level routing tokens. Multi-level routing tokens form data dependencies, that is, lower-level routing tokens rely on higher-level ones. When accessing a sub-interface, the upper-level interface is required to be selected correctly. Hierarchy among interfaces decides the data dependency among multi-level routing tokens, which is beneficial to direct the fuzzing.

With identified routing tokens, we try to analyze and recover the hierarchy among them. For multi-level tokens, the upper one dominates several lower ones. We first cluster the requests according to each specific routing token. If a cluster collects more than one request, it means multiple interfaces share the same routing token, depicted as algorithm 1 line 7. We then set up partial order between the upper routing token and the lower ones, depicted as algorithm 1 line 10. In some cases, there are conflicts between the analysis results. Taking $diagnostic$ interface as the example, we cannot distinguish whether $PING/TRACEROUTE$ dominates $IPv4/IPv6$ based on clustering results alone because whether $PING$ or $TRACEROUTE$, they both include the two IP versions. To solve these conflicts, we further recover the hierarchy according to the field's region. The routing token located in URI, is mostly dominated; Fields within header are more dominant than body. If two routing tokens are located in the same region, the front dominates the back. [4]

B. LLM-Powered Pattern Learning

Either public interfaces or hidden interfaces in one device, are generally dispatched in the same way. Therefore, the contexts of routing tokens including public and hidden ones, generally share a similar code pattern. We leverage LLM to learn the common pattern among public routing tokens' contexts. Then we use the learned pattern to analyze web programs and try to find hidden ones. This LLM-based approach boasts superior generalization capabilities. It circumvents the need for numerous heuristics and their inherent limitations, thereby maximizing the LLM's code analysis potential. Instead of adopting traditional techniques to trace data flow during the

[4]Distinct regions reflect varying levels of depth within the hierarchical interfaces.

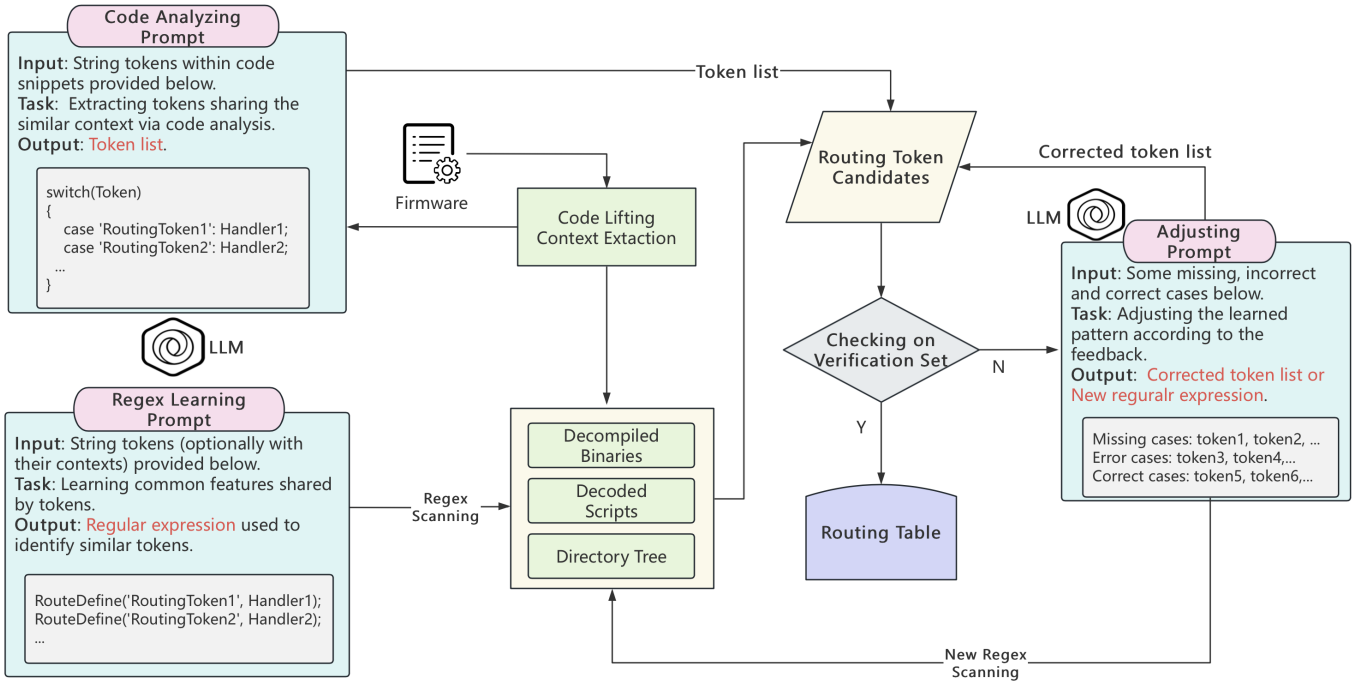


Fig. 6: Two distinct prompts have been crafted to guide the LLM in comprehending the task. An additional adjusting prompt is employed, incorporating feedback from the verification process, to refine and correct the outcomes learned by the LLM.

routing process, our solution is more intelligent and avoids heavy program analysis, suitable for a wide variety of IoT devices.

1) Contexts Extraction: Web programs are developed by multiple programming languages. Text-based programs (e.g., PHP, Python, LUA) are readable for LLM, while low-level programs like binary or bytecode, are not directly understood by LLM. For low-level codes, it is required to lift them to high-level program languages. Specifically, we decompile binary code to C/C++ code instead of to assembly code, because C/C++ has higher information density. For bytecode, we decompile it to corresponding high-level code. We extract routing tokens' contexts within lifted high-level codes, allowing LLM to effectively analyze and interpret the functionality of code and structure of data.

The context of a routing token encompasses the code and data that are intimately connected with it. We consider both the code that references the token and the data that includes it as part of its context. This context is vital as it offers critical insights into the token's purpose and usage. We extract a relatively complete unit containing the routing token, such as function, class, or module, which better preserves the context related to the routing token in a limited input space. Specifically, we assess locality based on the programming paradigm: for procedural programs, we consider the function; for object-oriented programs, the class; and for functionally organized programs, the module. When a routing token corresponds directly to a filename, acting as an identifier for accessing static resources, we include the entire file path as part of the token's context. This inclusion ensures that the LLM can comprehend the broader scope of the routing token's function and application.

2) Learning Process: LLM has demonstrated remarkable capability in comprehending and analyzing code. One intriguing idea is to harness LLM's analytical prowess to examine the routing mechanism within the device, thereby inferring the routing table. However, two challenges need to be addressed to achieve this objective.

Firstly, while low-level code is elevated to a higher level, it is still difficult to revert to the source code, hindering full understanding of the code's functionality. During the compilation process, certain information from the source code is lost, making it arduous to recover even through decompilation. For instance, in the case of binary, symbols from the source code are discarded, resulting in obscure and hard-to-comprehend decompiled code. Additionally, indirect calls present in binary code obscure the control flow and disrupt the original data structure, resulting in a decompiled high-level code that lacks the logical connections and relationships found in the source code, thereby impeding the complete recovery of data flow information. Secondly, LLM has limitations on the length of input text it can process. Given the vast number of programs that may be present within the device, it becomes challenging to feed all of them to LLM for analysis.

To mitigate these challenges, we apply the principle of program locality, including spatial and temporal aspects, which influence coding practices by clustering related variables together to enhance readability, maintainability, and performance [12]. This principle suggests that most contexts relative to one variable are often concentrated in a small portion of the

code. By focusing our analysis on the immediate context surrounding the routing tokens, rather than attempting a wholesale examination of the entire program, we can more efficiently leverage LLM's analytical capabilities to infer the routing table. This targeted approach allows for a more manageable and effective analysis, despite the inherent limitations.

We devise two tailored prompts for LLM to analyze the extracted contexts. On one hand, we utilize the LLM to analyze the code contexts surrounding routing tokens. The characteristics of the routing table may be concealed within the program's control flow or data flow (e.g., the examples (a) and (b) in Fig. 3). We guide the LLM to understand the program semantics and, based on this comprehension, extract the routing tokens, as shown in the upper left of Fig. 6. On the other hand, due to the challenges mentioned earlier regarding the use of LLM for program analysis, it is difficult to comprehensively extract the routing table. To further reduce the false negatives in identifying routing tokens, we guide the LLM to learn the probable formatting patterns among the routing tokens and to generate appropriate regular expressions, as shown in the lower left of Fig. 6. These regular expressions are then used to scan programs and extract similar string tokens. In the case of using a file as the routing token (e.g., Fig. 3 (c)), we supplement the directory tree of the local file system into the searching scope.

We offer a concrete example to demonstrate the intelligent extraction of the routing table by LLM in Appendix A.

3) Self-Correction Model: To mitigate the possibility of initial errors and guide LLM to learn the correct pattern, a self correction model is designed to implement iterative adjustments that facilitate continuous error correction, eventually learning the correct result. The model leverages both positive and negative samples to assist LLM in error correction during the prompt process. We allocate a subset of public routing tokens for verification, while the remainder forms the corpus that feeds into the LLM, enhancing its task comprehension. Upon receiving each LLM output, EAGLEYE evaluates it against the verification set to identify missed and correct instances. Negative samples, which are non-routing tokens within the firmware, are utilized to pinpoint incorrect cases. Although the LLM might initially falter, it can progressively refine its understanding and improve performance through self-correction. Generally, through finite times of iteration, LLM can master the pattern (right in Fig. 6).

We have crafted an illustrative example in Appendix B, to elucidate the specific process of self-correction.

C. Hidden-Interface Directed Black-box Fuzzing

With the routing table, we use it to generate high-quality seeds and conduct directed black-box fuzzing toward hidden interfaces. To conquer blindness, we leverage responses to supply necessary parameters and catch clues for hidden interfaces. The whole workflow is depicted as algorithm 2.

1) Mutation Toward Hidden Interfaces: We utilize public requests as templates and the routing table as a fuzzing dictionary to generate high-quality seeds, which allows EAGLEYE to navigate through different interfaces. For hierarchical interfaces, where one interface is subject to another one, we carefully orchestrate corresponding routing tokens, accounting for data dependencies among them.

A valid request requires other parameters matched with the routing token. The device sends responses to the client, providing some necessary information for the desired functionalities. Within responses, there generally exists a list of parameters (e.g., forms), that the client needs to answer when interacting with the device. The client fills in these parameters to assemble a request before the next visit. This process gives us valuable clues to infer parameters matching the routing token.

Throughout the fuzzing campaign, we continuously collect parameters from responses, which are then assembled to generate new requests. To guide this assembly process, we design three specific mutation policies. Firstly, if we observe a correlation between the response and interface, that is, the response contains one routing token, we incorporate the parameters extracted from this response into the request targeted at that interface. This correlation suggests that the parameters identified within the response are pertinent to the interface in question, and adding them to the request could improve the quality of mutation. Secondly, we observed some devices respond with hints for missing parameters, which provide crucial insights about the parameters required. During interactions with the device, we continuously add these indicated parameters from the response into subsequent mutated requests, making mutation increasingly towards completeness and accuracy. Lastly, to maintain the diversity of mutation, we randomly select parameters from collected responses. It ensures a broad and varied exploration of potential mutations, combining information from real-time responses with the knowledge gathered from public interfaces.

2) Catch Hidden Interfaces from Response: Hidden interfaces, similar to their public counterparts, operate without any discernible anomalies or irregularities. In the context of black-box testing, the response emitted by the device serves as a crucial clue to determine the validity of the seed (i.e., mutated request), which indicates a potential hidden interface. By subjecting mutated requests with a hidden routing token, we catch the presence of one hidden interface according to the validity of the response. Responses from interfaces can vary significantly in format, which complicates the direct extraction of features that indicate their validity. However, it is a common observation that invalid responses from a specific device often share common attributes.

We analyze and obtain the distinctive characteristics of invalid responses as Eq. (1). Invalid responses typically provide feedback indicating unsuccessful attempts to access or interact with a device. Common reasons for such failures include unauthenticated access, unauthorized access, or incorrect parameters. While these distinct failure causes may introduce minor variations in the invalid responses, they typically adhere to a general pattern of uniformity. To identify the potential manifestations of invalid responses, we employ a union oper-

Algorithm 2: Hidden-Interface Directed Black-box Fuzzing

Input: Testing device $\mathcal{P}$ with black-box environment, routing table $RTable$
Output: Hidden interfaces PoC

```
SeedsPool ← ∅
ParasPool ← ∅
foreach rtoken ∈ RTable do /* Generate initial seeds using the routing table. */
    SeedsPool ← SeedsPool ∪ Generate(rtoken)
repeat
    seed ← Pop(SeedsPool)
    seed' ← Mutate(seed, ParasPool)
    res ← Interact(P, seed)
    if Validity(res) then /* Check the validity of response */
        PoC ← PoC ∪ seed
        continue
    param ← Distill(res) ;   /* Distill parameters from the response. */
    ParasPool ← ParasPool ∪ {param}
    if Augment(ParasPool) then /* Check if find any new parameter. */
        SeedsPool ← SeedsPool ∪ seed'
until SeedsPool ≡ ∅;
```

ation. This operation enables us to aggregate and scrutinize the diverse forms of invalid responses that can emerge as a result of varying failure reasons. For example, devices frequently denote failed attempts through specific status codes, with a 401 status code commonly signifying unauthorized access and a 403 status code indicating unauthenticated access. The invalid response always reflects restricted failures via some string snippets like status codes. Thus we can identify the common pattern shared among invalid responses. It is important to note that both valid responses and invalid responses, may share some common string snippets, which can lead to confusion and impede precise identification of the pattern that is unique to invalid responses. Therefore, it is imperative to refine the identified pattern by eliminating the shared string snippets.

$$InvPat = \bigcup_{sp \in InvSnip} Norm(sp) - \bigcap_{sp \in ValSnip} Norm(sp) \quad (1)$$

Given that responses are predominantly textual, the pattern distinguishing invalid responses is often characterized by the presence of unique string snippets. We initially decompose the response into a series of discrete snippets and then conduct a targeted search to isolate those snippets that are exclusively present in the invalid responses. To exclude tiny deformations, we normalize the snippet to remove redundant placeholders such as tabs or spaces at the header and tail.

IV. Evaluation

We have implemented a prototype of EAGLEYE, with the selection of LLM as GPT [40] (gpt-3.5-turbo [28]), and have integrated a verifier to ascertain whether the interface is protected by authentication. We have conducted a comprehensive evaluation of the prototype, and in this section, we present the findings and outcomes of our experimental analysis.

TABLE I: Testing Devices

Vendor	Model	Version	Device Type	Web Type
Netgear	SRX5308	4.3.5-5	SSL VPN	Bin+LUA
	R7000	1.0.11.136	WiFi Router	Bin+HTM
Cisco	RV-042	4.2.3.14	VPN Router	Bin+HTM
Motorola	CX2L	1.0.1	WiFi Router	Bin
TrendNet	TEW-811DRU	1.0.10.0	Wifi Router	Bin+ASP
Linksys	WRT54GL	4.30.18	Wifi Router	Bin+ASP
Ubiquiti	EdgeRouterX	2.0.9	Ether Router	Bin+Python
DLink	DNS-320	1.11B01	NAS	Bin+PHP
	DIR-823G	V1.0.2B05	Wifi Router	Bin
Mercury	MNVR816	2.0.1.0.5	Video Recorder	Bin+LUA
ZTE	C520	2.1.6T3	IP Camera	Bin+LUA
Zyxel	WSQ50	V2.20	Wifi Router	Bin+LUA
TPLink	Archer A7	V5_1.2.1	Wifi Router	Bin+LUA

A. Experiment Setup

Testing IoT devices. We selected 13 commercial devices as the testing dataset, comprising multiple types (e.g., VPN, wifi router, NAS, video recorder, and IP camera). They all source from leading manufacturers such as Netgear, Cisco, Zyxel, and TrendNet, representing the market's mainstream offerings. Detailed information about devices like model and version is shown in Table I. Among these devices, the implementations of their web are disparate. Their web servers are generally binaries, originating from popular embedded web schemes such as lighttpd, minihttpd, and goAhead. Devices' web applications (i.e., implementing various business functionalities) are usually developed by multiple programming languages. As Table I shows, applications in some devices are compiled and released as binaries, while some use various types of scripts (e.g., LUA, PHP, ASP, and HTM).

SOTA solutions for comparison. The research in this area is relatively insufficient. IoTScope is a state-of-the-art solution, focusing on URL-based and unauthenticated hidden interfaces, represents the latest progress in this field to some extent. We set head-to-head experiments to compare EAGLEYE with IoTScope, and fairly exhibit their capability for exposing hidden interfaces.

Effectiveness metrics. We evaluated the effectiveness of routing analysis and black-box fuzzing separately, to fully demonstrate the performance of EAGLEYE. Specifically, we collected the key procedural data to indicate the correlation between strategies and the final result. We further evaluated the severity of hidden interfaces by discovering vulnerabilities among them.

Experiment environment. We conduct each experiment on one Kali system (Linux version 5.18.0), equipped with an Intel Core i7 (2.6GHz) processor and 16G RAM. All testing devices were restored to factory settings and used in the initial configuration before the experiment.

B. Overall Findings

Results. EAGLEYE overall found 79 hidden interfaces, including 27 ones bypassing authentication and 52 ones after authentication, showing in Table II. Within the dataset of

devices, over 90% have hidden interfaces, and over 60% have unauthenticated interfaces. Traditional solutions typically pay attention to hidden interfaces that can bypass authentication because they pose obvious security threats to IoT devices while ignoring those hidden interfaces after authentication. However, the hidden interface after authentication also easily breeds security issues, and the vulnerabilities that occur on them could cause server problems such as unauthorized operations and DoS attacks, causing huge threats to the device too. Hidden interfaces after authentication are more common than those bypassing authentication. As Table II shows, the former is approximately twice the number of the latter.

Correctness. We quantify whether there are false positives or false negatives for exposed hidden interfaces. A false positive here means that EAGLEYE wrongly reports one hidden interface that is truly public. We thoroughly scrutinized each hidden interface by manual inspection and a double-check using an LLM. We dissected the functionality of these hidden interfaces and cross-referenced them with the relevant documentation. To mitigate the risk of oversight, we also utilized the LLM to perform a secondary analysis, feeding it the hidden requests, corresponding responses, and documentation for a comprehensive check. For this purpose, we selected Kimi [1], an LLM known for its capability to digest long, lossless contexts [29], rendering it ideal for managing extensive manuals. After rigorous examination, we found no false positives. This is attributed to the EAGLEYE's design. Our methodology ensures that each reported interface remains undisclosed, meaning it is not explicitly defined in the device's official documentation. Firstly fuzzing ensures the authenticity of every report. Secondly the detection logic for hidden interfaces is robust, distinguishing the regularities of hidden interfaces from normality. Regarding false negatives—instances where EAGLEYE fails to detect actually hidden interfaces—there is a theoretical possibility of omissions. For example, routing analysis may not uncover routing tokens for hidden interfaces that deviate from common patterns. EAGLEYE's goal is not to exhaustively list all hidden interfaces but to reveal some of them. Addressing and reducing the occurrence of false negatives is an area for future improvement.

Comparison with SOTA. IoTScope is able to only expose hidden interfaces bypassing authentication. To be fair, it is required to unify the comparison standards. IoTScope generates a URL list through firmware analysis and then tries to identify unauthenticated interfaces via differential analysis based on responses. The generated URL list is critical to enumerate interfaces. Based on the original capability, we add the ability to expose authenticated hidden interfaces to IoTScope. Specifically, we directly fed the seeds same as IoTScope, using the URL list it generated, to our fuzzer. We leverage EAGLEYE to update the authentication credentials within seeds and check the response to catch hidden interfaces after authentication. In comparison, IoTScope only found 3 hidden interfaces, of which one is unauthenticated and the other two are authenticated, showing in Fig. 7.

TABLE II: Hidden interfaces exposed by EAGLEYE. #HINT=the number of hidden interfaces, #B-Authen=the number of hidden interfaces bypassing authentication, #A-Authen=the number of hidden interfaces after authentication.

Vendor	Model	#HINT	#B-Authen	#A-Authen
Netgear	SRX5308	3	0	3
	R7000	3	2	1
Cisco	RV-042	9	1	8
Motorola	CX2L	34	6	28
TrendNet	TEW-811DRU	1	0	1
Linksys	WRT54GL	6	3	3
Ubiquiti	EdgeRouterX	0	0	0
DLink	DNS-320	11	7	4
	DIR-823G	6	6	0
Mercury	MNVR816	1	1	0
ZTE	C520	3	0	3
Zyxel	WSQ50	1	1	0
TPLink	Archer A7	1	0	1
Total	-	79	27	52

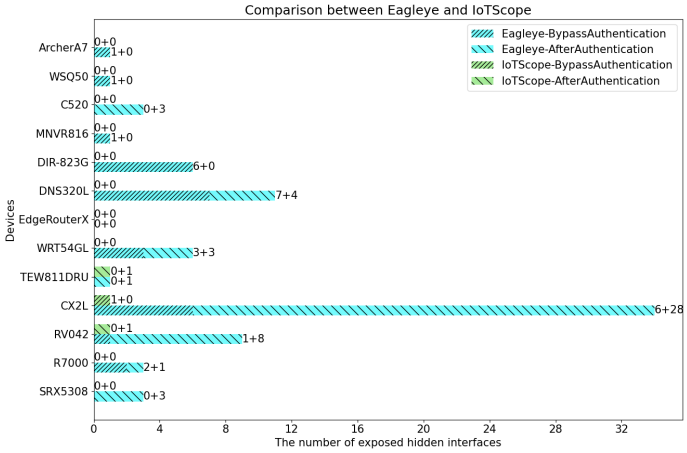


Fig. 7: Comparison with IoTScope. EAGLEYE outperforms IoTScope, exposing 25X more hidden interfaces.

C. Routing Analysis Effectiveness

To correctly identify the routing token in the request, EAGLEYE does a token-based comparative analysis, which first identifies all variable tokens and then filters noise inferences. As shown in Table III, EAGLEYE found an average of 4.5 variable token fields and successfully filtered out 2.0 inferring token fields in them. [5] At the same time, we can observe that different devices have a wide distribution of the routing token, including URI, query, header, and body, of which the URI is the most common location. The routing token is essentially a parameter that can be located anywhere in the request. Six devices exist hierarchical interfaces, and EAGLEYE successfully recovered the hierarchy of their multi-level routing tokens. It can be observed that interfaces in most devices usually have shallow layers, likely a design choice by manufacturers for clarity and straightforward service provision. It should

[5]We only counted session tokens and timestamps as filtered tokens here. Normal tokens have been marked in advance and can be accurately identified, and the number of them is much more than the other two. To intuitively reflect the statistics, they are not included here.

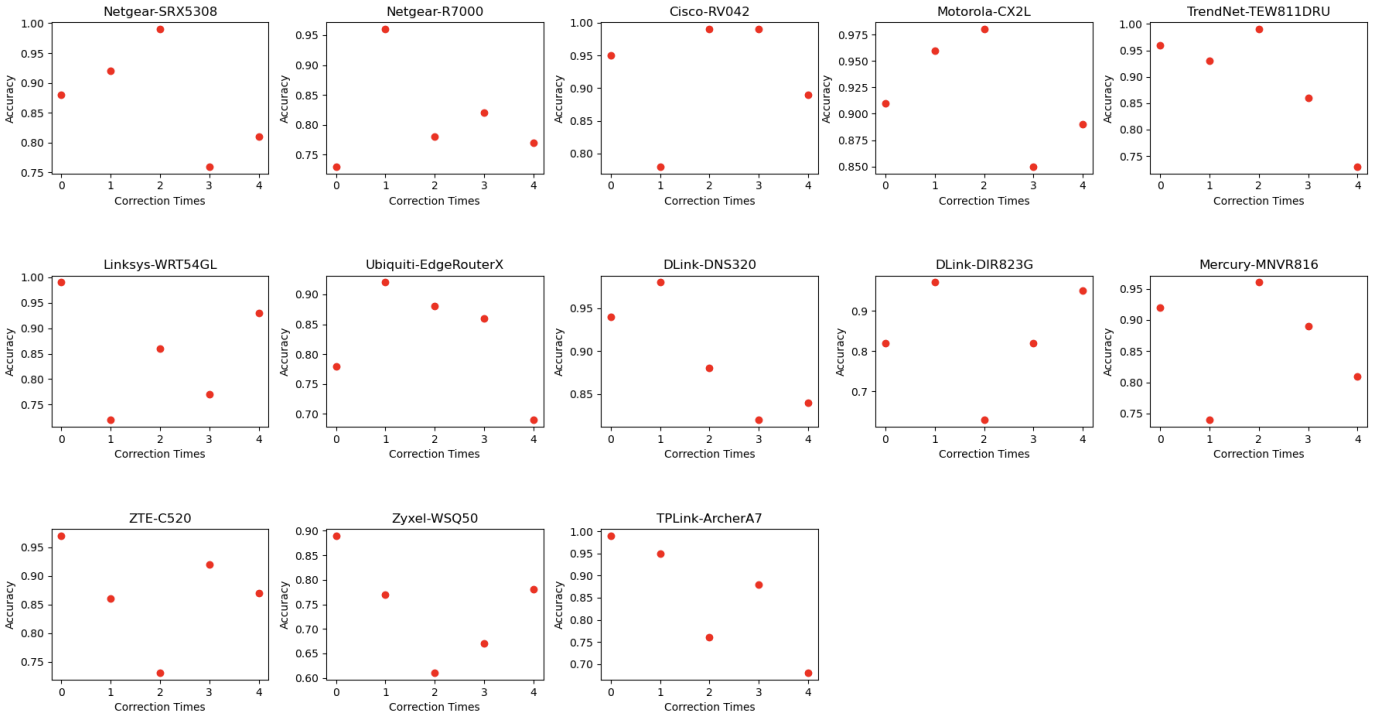


Fig. 8: Accuracy of the pattern learned by LLM varies with the number of corrections. Within limited adjustments, the LLM can learn the correct features among the routing table.

be noticed that even the non-hierarchical interface may have multiple routing token fields, often corresponding to requests with different methods. This is why #VTF may be more than 1 larger than #FTF.

The common pattern of routing tokens varies greatly depending on the device, thus EAGLEYE use LLM instead of the human to learn this pattern. The benefits are obvious, not only is automation achieved to improve efficiency, but LLM has clear advantages in pattern learning. As shown in Fig. 8, the accuracy of the pattern learned by LLM varies with the number of corrections. LLM could learn the right pattern within the limited times to correct errors according to interactive feedback. In most cases, LLM could achieve the highest accuracy with 1 or 2 adjustments. Especially it learns the right pattern just the first time on some devices such as Linksys WRT54GL and ZTE C520. However, the more interactions, not better the accuracy. On the contrary, after the accuracy reaches its peak, it decreases sharply as the times of correction increase. This may be due to the overfitting during learning iterations. There may be some corner cases, which are very different from other routing tokens, disturbing LLM to understand the common pattern. Therefore, several adjustments are deemed more effective than a forced fitting, particularly as the latter may not yield desired results over times of adjustment.

With the learned pattern by LLM, EAGLEYE then extracts the routing table from firmware, averaging 138.9 routing tokens per table, as shown in Table III. In most cases, the routing table is distributed among the firmware. In some cases (e.g., Motorola CX2L and DLink DNS320), part of the routing table appears locally aggregation.

TABLE III: Effectiveness of routing analysis. #VTF=the number of variable token fields, #FTF=the number of filtered token fields, #LoC is where the routing token is located, #Hier=the max level of hierarchy for multi-level routing tokens, #Table=the size of the routing table, layout of routing table: DIS=Distributed, AGG=Aggregated.

Vendor	Model	Routing Token Identification				Routing Table Extraction	
		#VTF	#FTF	#LoC	#Hier	#Table	Layout
Netgear	SRX5308	4	1	Query	2	211	DIS
	R7000	5	3	URI	1	376	DIS
Cisco	RV-042	3	2	URI	1	197	DIS
Motorola	CX2L	4	3	Header	1	270	DIS& AGG
TrendNet	TEW-811DRU	3	1	URI	1	31	DIS
Linksys	WRT54GL	3	2	URI	1	101	DIS
Ubiquiti	EdgeRouterX	9	4	URI& Body	3	34	AGG& DIS
DLink	DNS-320	5	1	URI& Query& Body	2	137	DIS& AGG
	DIR-823G	4	2	URI& Header	1	166	DIS& AGG
Mercury	MNVR816	7	3	URI& Body	2	152	DIS
ZTE	C520	4	2	URI	1	58	DIS
Zyxel	WSQ50	3	1	URI& Body	2	45	DIS
TPLink	Archer A7	4	1	URI	2	28	DIS
Average	-	4.5	2.0	-	1.5	138.9	-

D. Black-box Fuzzing Effectiveness

EAGLEYE extracts parameters in the response which are then matched with the routing table and assembled into new

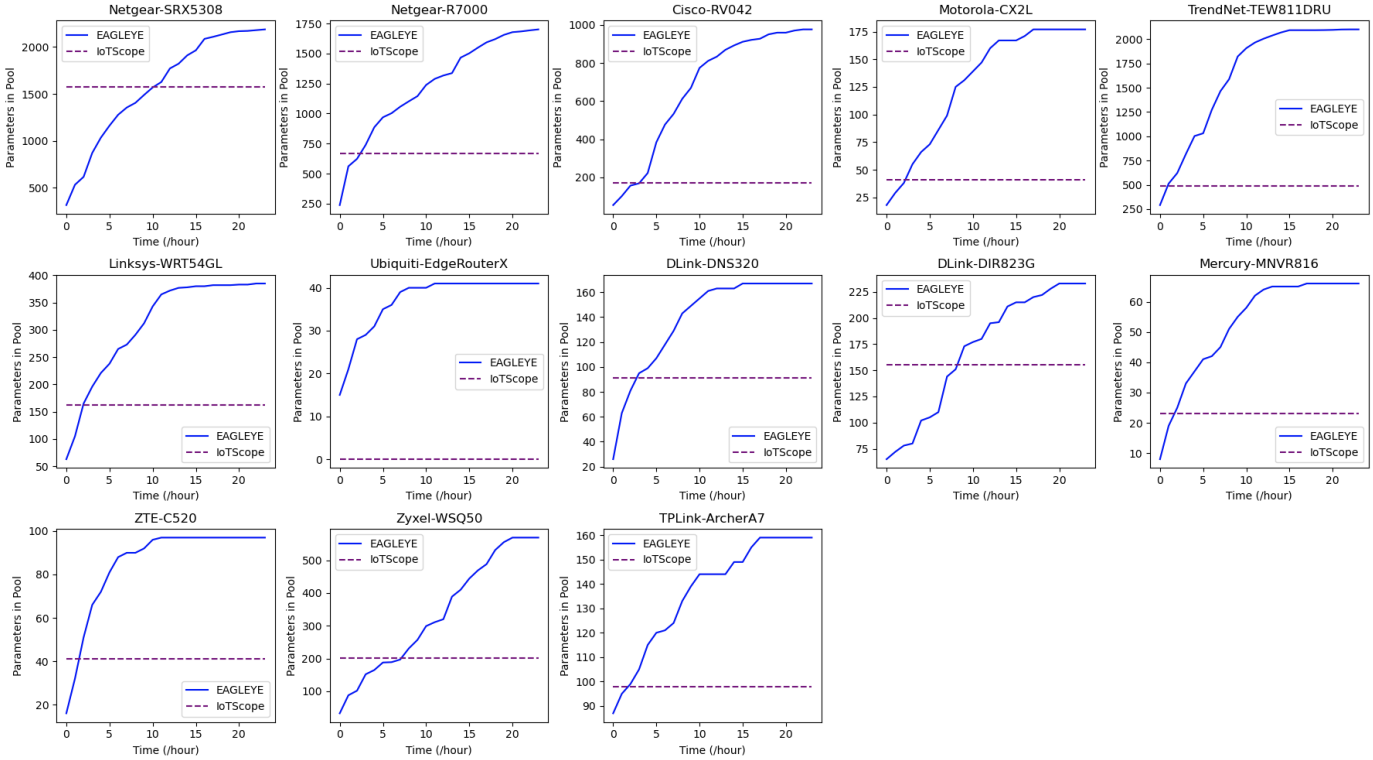


Fig. 9: Growth trend of parameters in the pool. Compared with IoTScope, EAGLEYE can not only obtain more parameters but also the parameters are more accurate. To a certain extent, this trend reflects the continuous improvement of EAGLEYE's mutation quality with the continuous supplement of parameters from responses.

seeds (i.e., mutation process). With the fuzzing campaign progressing, more and more parameters are added to the pool. EAGLEYE's growth trend of parameters in the pool is shown as the blue line in Fig. 9. The trend indicates that as the test proceeds, EAGLEYE triggers more interfaces, guiding the device to respond with more parameters. To some extent, it reflects the effectiveness of EAGLEYE's mutation strategy, which can continuously produce better seeds.

In comparison, IoTScope also distills parameters from firmware and stores them in a database. During the fuzzing campaign, one parameter is selected from the database and assembled into the request. Therefore, the parameter database in IoTScope is extracted statically in advance and does not change during the fuzzing test. We used a dotted purple line in Fig. 9 to represent the number of parameters IoTScope extracts. In all cases, EAGLEYE obtained many more parameters than IoTScope. In the case of Ubiquiti-EdgeRouterX, IoTScope even distilled no parameter. Meanwhile, the parameters extracted by EAGLEYE are more accurate because they are extracted from dynamic responses.

E. Severity of Hidden Interface

To evaluate the security risks introduced by hidden interfaces, we further try to discover vulnerabilities on them. Unauthenticated ones are classified as one kind of vulnerability breaking the access control. For others after authentication which don't directly result in the vulnerability, it is required to further analyze their security. We prioritize security audits for GET requests that include queries and POST requests that contain data payloads. These requests encompass parameter-laden regions—specifically, the query parameters for GET requests and the body content for POST requests. An increased number of parameters equates to a higher volume of user-submitted data, which consequently elevates the associated security risks. As a result, we discovered a total of 29 unknown vulnerabilities. The causes of these vulnerabilities are not tangled, but they are too deeply hidden to be found. All

TABLE IV: Discovered vulnerabilities on hidden interfaces. #VUL=the number of vulnerabilities, #CVE=the number of assigned CVEs, #ID=detailed CVE ID EAGLEYE obtained.

Vendor	Model	#VUL	#CVE	#ID
Netgear	R7000	3	2	CVE-2024-1430, CVE-2024-1431
Cisco	RV-042	2	1	CVE-2024-20362
Motorola	CX2L	6	1	CVE-2024-25360
Linksys	WRT54GL	3	3	CVE-2024-1404, CVE-2024-1405, CVE-2024-1406
DLink	DNS-320L	5	0	-
	DIR-823G	6	0	-
Mercury	MNVR816	1	0	-
ZTE	C520	1	0	-
Zyxel	WSQ50	1	0	-
TPLink	Archer A7	1	0	-
Total	-	29	7	-

the vulnerabilities we discovered have been reported to their vendors, and 7 of them have been assigned CVE numbers, as shown in Table IV.

We present 3 typical cases to illustrate the discovered vulnerabilities on hidden interfaces as follows.

Case Study 1: Backdoor Bypassing Authentication. There is a hidden interface in one device of Table I that allows an unauthenticated user to gain the highest privilege of the system and take over the device. As shown in Fig. 10 (A), this hidden interface could provide access to the device as a backdoor, making attacks easy to carry out. Specifically, this vulnerability opens the device's telnet service without authentication, which is publicly unknown and disabled by default. By contrast, the device only opens the web to authorized users to manage the device. The root permission of the underlying operating system could be directly obtained through this vulnerability.

Case Study 2: Command Injection Escalating Privilege. There exists a command injection on one hidden interface in another device of the dataset. This vulnerability injects the crafted command in one parameter when requesting a hidden functionality. As shown in Fig. 10 (B), any authenticated user can execute arbitrary system commands with the highest privilege through this vulnerability. Compared to normal usage, the vulnerability escalates the privilege, allowing malicious users to remotely operate the underlying operating system and control the device.

Case Study 3: XSS Attacking Clients. There exists an XSS on one hidden interface in another device. Different from the above two cases, XSS allows malicious users to attack the client instead of the device, resulting in code execution on users' machines. This vulnerability injects the crafted code into an unknown parameter carried by one hidden request. Worse, the malicious payload is permanently saved in the back end of the device. As shown in Fig. 10 (C), any access to web pages containing the malicious payload would trigger the XSS and cause harm to clients.

F. Digging into Hidden Interface

With exposed hidden interfaces, we try to search their potential features especially those different from public ones. We classified hidden interfaces according to their functionalities. We found the majority of hidden interfaces are typically concentrated on debugging operations, device status, system information, and network settings, as shown in Table V. Generally, these functionalities are intended for operations personnel or developers, not directly for end-users.

We delved deeper into why these hidden interfaces are not documented in the manual:

(1) Legacy Code: During the development, certain interfaces may initially serve purposes such as debugging, environment setup, and status monitoring. We found traces of development in some firmwares, such as the existence of SVN (i.e., an open source code version control system) records for code management in a certain device, which is due to the incomplete separation of the development and production environment. If the separation between development and production environments is not thorough, these interfaces might be mistakenly deployed into the production environment. If they are not thoroughly cleaned up before the product is released, they will be left behind in the final product as hidden interfaces.

(2) Permission Management Flaws: There may be flaws in the permission management mechanism that either fail to correctly restrict access to privileged interfaces or inadvertently bring internal interfaces to light. For instance, a hidden Telnet service interface on one device, due to the lack of permission verification, allows unauthenticated users to gain low-level access to the device's operating system and potentially take control of the device.

(3) Security and Privacy Concerns: Some vendors may choose not to detail the description of services unrelated to the user in the manual for security and privacy reasons, to prevent potential misuse. For example, the TR069 interface for remote management that we found hidden in one device is primarily intended for operators instead of end-users to use, providing the ability to automatically configure and monitor

TABLE V: Classification of hidden interfaces according to their functionalities. #HINT=the number of hidden interfaces.

Device	Category	#HINT	Authentication
Netgear SRX5308	Remote Management	1	Authenticated
	User Information	1	Authenticated
	Traffic Statistics	1	Authenticated
Netgear R7000	Configuration Information	1	Unauthenticated
	Debug Information	1	Unauthenticated
	Network Settings	1	Authenticated
Cisco RV-042	Network Settings	5	4 Authenticated + 1 Unauthenticated
	System Information	3	Authenticated
	Device Status	1	Authenticated
Motorola CX2L	Remote Management	1	Unauthenticated
	Network Settings	2	1 Unauthenticated + 1 Authenticated
	Traffic Statistics	1	Authenticated
	Device Status	1	Unauthenticated
	System Information	6	2 Unauthenticated + 4 Authenticated
	System Settings	5	1 Unauthenticated + 4 Authenticated
	Debug Information	2	Authenticated
	Configuration Information	16	Authenticated
TrendNet TEW-811DRU	Network Settings	1	Authenticated
Linksys WRT54GL	Device Status	1	Authenticated
	System Information	3	Unauthenticated
	Network Settings	2	Authenticated
DLink DNS-320	System Information	4	Unauthenticated
	File Transfer	7	4 Authenticated + 3 Unauthenticated
DLink DIR-823G	System Information	4	Unauthenticated
	Remote Management	2	Unauthenticated
Mercury MNVR816	File Downloading	1	Unauthenticated
ZTE C520	System Information	1	Authenticated
	Network Settings	1	Authenticated
	Device Status	1	Authenticated
Zyxel WSQ50	Remote Management	1	Unauthenticated
TPLink Archer A7	Remote Management	1	Authenticated

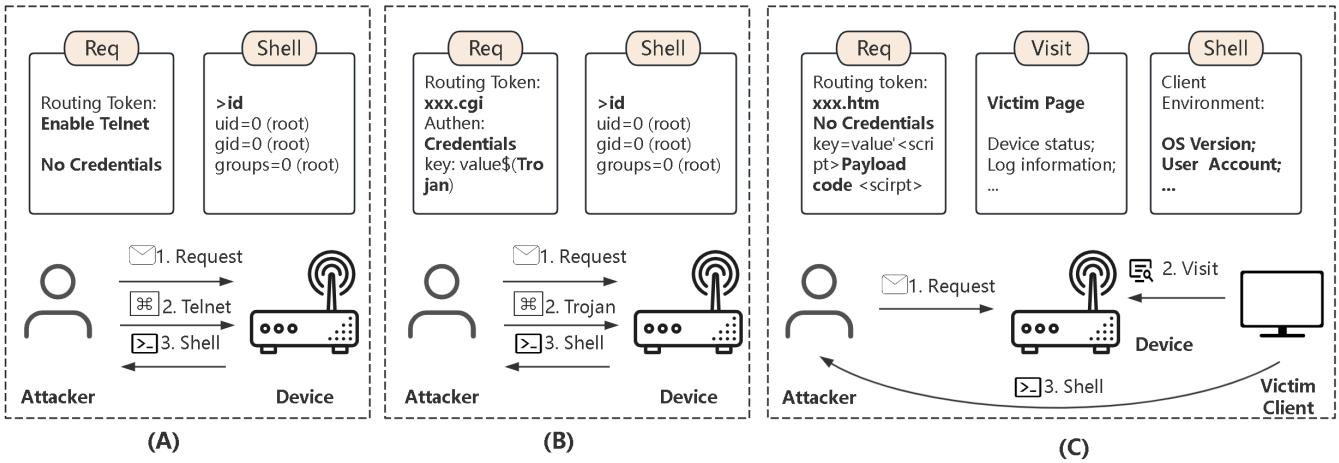


Fig. 10: Three typical vulnerabilities in discovered hidden interfaces. (A) A backdoor bypassing authentication: opens the telnet and gains a shell with the highest privilege. (B) A command injection escalating privilege: allows attackers with normal credentials to execute OS commands with the highest privilege. (C) An XSS attacking victim's clients: injects malicious code into one parameter saved in the web, and then the users who visit the victim pages will be infected.

the status of the device remotely.

(4) Hidden Default Configurations: The device is designed to work with default settings in most use cases, so vendors may not provide additional configuration options in the manual. Especially some advanced settings may be intended only for experienced users or administrators, and vendors may believe that ordinary users do not need to access these settings. Exposing certain critical configuration options to inexperienced users could affect the normal running of the device and network security. For example, a hidden interface found in one device for configuring TCP/UDP protocol parameters hugely affects the network. These parameters are usually set to default and generally do not require changes after the product is finalized. More notably, the routing token in this interface contains the word "hidden," directly indicating that it is hidden.

V. Discussion

Scope of routing analysis. Routing analysis extracts the routing table via its common pattern. However, for dynamic routing, its routing token exists dynamically, and its routing table has no entity in the firmware, thus it cannot be extracted through static analysis. For example, in the URI '/users/:userId', *userId* is a dynamic routing token that matches a user ID. It is produced and consumed dynamically and doesn't land. EAGLEYE cannot handle this situation at present. While it seems that dynamic routing is widely used in cloud scenarios and rarely used in IoT web [18]. As dynamic routing table is created on-the-fly, the key is to identify the code responsible for generating dynamic routing tokens. Once located, LLMs can analyze this code to extract the token generation rules, which then inform the fuzzer to craft seeds adhering to the interfaces' specifications. In the future, we will further study this mechanism to enrich the application scenarios of routing analysis.

Accuracy of the routing table. Although LLM performs well in learning the pattern, there may still be false negatives and false positives in the extracted routing table. In essence, our solution tries to strike a balance between precision and recall, minimizing missed routing tokens (i.e., false negatives) while avoiding the introduction of too many false routing tokens (i.e., false positives). However, it is still quite challenging to improve the accuracy of the routing table. Our primary goal is to uncover as many hidden interfaces as possible. To minimize false negatives, we consider expanding the training data on hidden interfaces, thus enhancing the LLM's analytical capabilities through fine-tuning. This is an avenue we intend to pursue in future.

Limitation of hierarchy analysis. Hierarchy analysis is grounded in real-world case studies, where we manually dissected device interfaces. For those with multi-level structures, their hierarchies clearly correlate with the routing tokens' location. Evaluations indicate that this analysis could effectively bolstered fuzzing efforts and yielded positive outcomes. Admittedly, there may be exceptions with intricate, deeply nested interfaces that could be overlooked. It's where we need to enhance in the future.

VI. Related Work

Hidden Interface and Broken Access Control. The related research is comparatively inadequate at present. IoTScope [39] exposes hidden interfaces based on URL in IoT web. It constructs probing requests via light firmware analysis and is dedicated to finding unauthenticated interfaces leading to information leakage and device settings. However, its scope is limited and neglects authenticated hidden interfaces. Stringer [34] intends to find undocumented functionalities and backdoors within binaries of devices via static data comparison function which affects the program flow. While this method limits the scope of the application and compromises analysis accuracy

due to open challenges such as indirect calls. Chen et al [7] analyze the process of remote binding between IoT devices and users, exploring the possible questionable practices related to authentication and authorization.

Many previous works pay more attention to hidden interfaces in mobile applications and focus on the problem of broken access control. APIScope [36] revealed that many supper mobile applications (e.g., WeChat and TikTok) contain undocumented APIs just for their 1st-party mini-apps which may be exploited by malicious 3rd-party mini-apps. It is committed to identifying these hidden APIs and evaluating their security issues. LeakScope [48] tries to identify potential data leakage in the cloud back-end services by analyzing mobile applications. Saint [4] performs static analysis to identify sensitive data flows with the source code of mobile applications for IoT. AuthScope [50] strives to find authorization problems in online services from the view of mobile applications. SmartGen [47] aims to expose server URLs in mobile applications, with selective symbolic execution to solve user input constraints. Autoforge [49] forges encrypted conformance messages from mobile applications to test server-side problems. These works provide us with good inspirations and offer beneficial references for EAGLEYE.

IoT Fuzzing Test. IoT devices are usually closed-source, closely running, not supporting third-party programs, and lacking debugging means, which makes black-box fuzzing still prevalent and play an important role [27], [13]. IoTFuzzer [6] and DIANE [30] conduct black-box fuzzing through mobile applications sending crafted requests. Snipuzz [15] tries to improve mutation policy by inferring snippets in messages via differences among responses. SRFuzzer [44] generates seeds in the format of key-value pair and sequences the seeds by the configuration-reading order. One significant drawback of black-box fuzzing is blindness. Labrador [25] provides code coverage and distance feedbacks to direct the mutation, generating a gray-box model under black-box settings.

Gray-box fuzzing with feedback guidance has demonstrated a powerful capability [24], [26]. Though with the above limitations in IoT devices, it still attracts relentless enthusiasm for research. FIRM-AFL [45] proposes a high-throughput fuzzing for IoT devices based on the classical solution AFL [42] via augmented process emulation. IoTHunter [22] and FirmFuzz [32] implant real-time checkers in the emulation environment to assist fuzzing. TriforceAFL [20] tests OS kernel via emulation environment to provide code coverage feedback. It can be observed that gray-box fuzzing has a strong demand for rehosting, which emulates IoT devices on general computing platforms [37]. Firmadyne[5] and FirmAE[23] try to emulate the firmware under the system mode of QEMU [2]. To mitigate high failure rates, avatar [41] switches to running on real devices when the emulation lacks hardware support. Greenhouce [33] proposes an efficient rehosting based on the user mode of QEMU, for one common class of binaries providing single-service in the firmware.

VII. CONCLUSION

In this paper, we explain the significant problem of hidden web interfaces in IoT devices and give a series of clear definitions. To automatically expose hidden interfaces, we propose a novel solution EAGLEYE, which models the problem as a searching process. Specifically, we revealed the similarity of the routing pattern between hidden interfaces and public interfaces. Accordingly, an adaptive approach, routing analysis, is presented to search the routing table in firmware intelligently by LLM. Further, EAGLEYE conducts a hidden-interface directed black-box fuzzing with the routing table as a dictionary, which focuses the testing energy on exploring interfaces. To conquer blindness, EAGLEYE leverages the response to guide the mutation and catch hidden interfaces. We evaluated EAGLEYE on 13 commercial IoT devices and successfully exposed 79 hidden interfaces, on which 29 unknown vulnerabilities including backdoor, command injection, XSS, and information leakage were found, and 7 have been assigned CVEs.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful suggestions on our paper. This work is supported in part by the National Natural Science Foundation of China (U24A20337), National Key R&D Program of China (2021YFB2701000), the National Natural Science Foundation of China (62402509), and the Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

REFERENCES

[1] M. AI, "Kimi," Available at https://kimi.moonshot.cn, 2023. 10
[2] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, 2005, p. 46. 15
[3] Bitdefender, "The 2023 iot security landscape report," Online, Bitdefender, Apr. 2023. [Online]. Available: https://www.bitdefender.com/files/News/CaseStudies/study/429/2023-IoT-Security-Landscape-Report.pdf 1
[4] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity {IoT}," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018. 15
[5] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *Network and Distributed System Security Symposium*, 2016. 15
[6] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *Network and Distributed System Security Symposium*, 2018. 15
[7] J. Chen, C. Zuo, W. Diao, S. Dong, Q. Zhao, M. Sun, Z. Lin, Y. Zhang, and K. Zhang, "Your iots are (not) mine: On the remote binding between iot devices and users," in *49th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019. 15
[8] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems," in *30th USENIX Security Symposium*, 2021, pp. 303–319. 1
[9] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: detecting the taint-style vulnerability in embedded device firmware," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 430–441. 1

[10] K. Cheng, Y. Zheng, T. Liu, L. Guan, P. Liu, H. Li, H. Zhu, K. Ye, and L. 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*, 2023. 1

[11] Citrix Systems, Inc. (2023) Citrix ADC and Citrix Gateway Security Bulletin for CVE-2023-3519, CVE-2023-3466, CVE-2023-3467. Citrix Systems, Inc. Accessed: 2024-04-23. [Online]. Available: https://support.citrix.com/article/CTX561482 1

[12] P. J. Denning, *The Locality Principle*, 2006. 7

[13] M. Eceiza, J. L. Flores, and M. Iturbe, "Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems," *IEEE Internet of Things Journal*, 2021. 15

[14] J. Eom, S. Jeong, and T. Kwon, "Covrl: Fuzzing javascript engines with coverage-guided reinforcement learning for llm-based mutation," *arXiv preprint arXiv:2402.12222*, 2024. 2

[15] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, 2021, pp. 337–350. 15

[16] J. Gao, Y. Xu, Y. Jiang, Z. Liu, W. Chang, X. Jiao, and J. Sun, "Em-fuzz: Augmented firmware fuzzing via memory checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020. 1

[17] Z. Gao, C. Zhang, H. Liu, W. Sun, Z. Tang, L. Jiang, J. Chen, and Y. Xie, "Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis," in *Proceedings 2024 Network and Distributed System Security Symposium*, 2024. 1

[18] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter, and M. Williams, "Backrest: A model-based feedback-driven greybox fuzzer for web applications," *arXiv preprint*, 2021. 14

[19] I. Gotovchits, R. Van Tonder, and D. Brumley, "Saluki: finding taint-style vulnerabilities with static property checking," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, vol. 2018, 2018. 1

[20] J. Hertz and T. Newsham, "Project triforce: Run afl on everything," *NCC Group, Tech. Rep.*, 2016. 15

[21] IoT Analytics. (2023) State of iot 2023: Number of connected iot devices growing 16% to 16.7 billion globally. [Online; accessed 2024-04-22]. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices 1

[22] P. Khandait, N. Hubballi, and B. Mazumdar, "Iothunter: Iot network traffic classification using device specific keywords," *IET Networks*, vol. 10, no. 2, pp. 59–75, 2021. 1, 15

[23] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 733–745. 15

[24] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, pp. 1–13, 2018. 15

[25] H. Liu, S. Gan, C. Zhang, Z. Gao, H. Zhang, X. Wang, and G. Gao, "Labrador: Response guided directed fuzzing for black-box iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 127–127. 1, 15

[26] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019. 15

[27] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." in *Network and Distributed System Security Symposium*, 2018. 15

[28] OpenAI, "Gpt-3.5 turbo," Available at https://platform.openai.com/docs/models/gpt-3-5-turbo, 2021. 9

[29] R. Qin *et al.*, "Mooncake: Kimi's kvcache-centric architecture for llm serving," *arXiv preprint arXiv:2407.00079*, 2024. 10

[30] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021. 15

[31] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1544–1561. 1

[32] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "Firmfuzz: Automated iot firmware introspection and analysis," in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, 2019, pp. 15–21. 15

[33] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, F. Dong, Z. Smith *et al.*, "Greenhouse:{Single-Service} rehosting of {Linux-Based} firmware binaries in {User-Space} emulation," in *32nd USENIX Security Symposium*, 2023. 1, 15

[34] S. L. Thomas, T. Chothia, and F. D. Garcia, "Stringer: Measuring the importance of static data comparisons to detect backdoors and undocumented functionality," in *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*. Springer, 2017, pp. 513–531. 14

[35] C.-W. Tien, T.-T. Tsai, I.-Y. Chen, and S.-Y. Kuo, "Ufo - hidden backdoor discovery and security verification in iot device firmware," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018. 1

[36] C. Wang, Y. Zhang, and Z. Lin, "Uncovering and exploiting hidden apis in mobile super apps," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2471–2485. 15

[37] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys (CSUR)*, 2021. 15

[38] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. 2

[39] W. Xie, J. Chen, Z. Wang, C. Feng, E. Wang, Y. Gao, B. Wang, and K. Lu, "Game of hide-and-seek: Exposing hidden interfaces in embedded web applications of iot devices," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 524–532. 1, 14

[40] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen *et al.*, "A comprehensive capability analysis of gpt-3 and gpt-3.5 series models," *arXiv preprint*, 2023. 9

[41] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *Network and Distributed System Security Symposium*, 2014. 15

[42] M. Zalewski, "AFL: American Fuzzy Loop," Google, Tech. Rep., 2013. [Online]. Available: https://lcamtuf.coredump.cx/afl 15

[43] L. Zhang, J. Chen, W. Diao, S. Guo, J. Weng, and K. Zhang, "{CryptoREX}: Large-scale analysis of cryptographic misuse in {IoT} devices," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 151–164. 1

[44] Y. Zhang, W. Huo, K. Jian, J. Shi, H. Lu, L. Liu, C. Wang, D. Sun, C. Zhang, and B. Liu, "Srfuzzer: an automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities," in *The 35th annual computer security applications conference*, 2019. 15

[45] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "{FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1099–1114. 1, 15

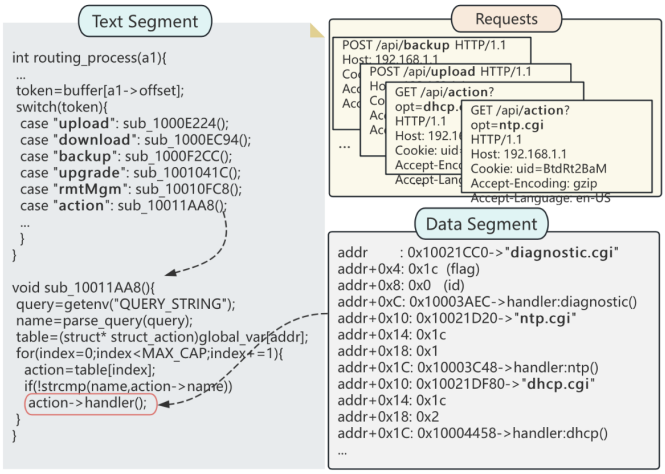
[46] Y. Zheng, Z. Song, Y. Sun, K. Cheng, H. Zhu, and L. Sun, "An efficient greybox fuzzing scheme for linux-based iot programs through binary static analysis," in *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2019. 1

[47] C. Zuo and Z. Lin, "Smartgen: Exposing server urls of mobile apps with selective symbolic execution," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 867–876. 15

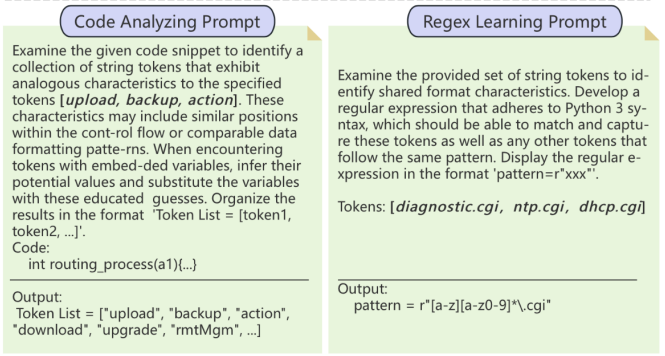
[48] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1296–1310. 15

[49] C. Zuo, W. Wang, Z. Lin, and R. Wang, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services." in *Network and Distributed System Security Symposium*, 2016. 15

[50] C. Zuo, Q. Zhao, and Z. Lin, "Authscope: Towards automatic discovery of vulnerable authorizations in online services," in *2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. 15



(a) An illustrative program demonstrating a specific routing process that incorporates two-level routing tokens, distinctly positioned within the URI and the query string.



(b) Two carefully designed prompts have been crafted to assist the LLM in identifying routing tokens effectively. These prompts work in tandem to maximize the discovery of potential routing tokens.

Fig. 11: An illustrative example to demonstrate the specific learning process of LLM.

Appendix A
LLM Learning Demonstration

We show an illustrative example to demonstrate the specific learning process of LLM. The Fig. 11(a) exhibits the specific routing process in one device. It incorporates two-tiered routing tokens, where the first level uses the routing token (e.g., "upload", "backup" and "action") located in the URI of the request to specify the targeted interface. The first-level handlers are invoked through a jump table. Under the *action* interface, there are subordinated interfaces (e.g., "diagnostic.cgi", "ntp.cgi" and "dhcp.cgi") indicated in the request's query. The second-level handlers are indirectly called via a routing table in the data segment. It should be noted the complexity in determining the sources for taint analysis within the function *routing_process* due to the intricate structure of the input data. However, for routing analysis, the process can directly pinpoint where the routing tokens are referenced, enabling the LLM to analyze the code and discern the underlying patterns. Consequently, routing analysis does not require the definition of sources and sinks as taint analysis does, allowing for a more focused examination of the routing process.

Two carefully crafted prompts are designed to facilitate the learning process of LLM to recognize common patterns among routing tokens. The first prompt is aimed at directly analyzing the code to identify similar string tokens functioning as routing tokens. The second prompt is focused on learning the common patterns within their formatting, with the objective of outputting a regular expression. This regular expression can then be utilized to scan and identify such tokens within firmware. As shown in Fig. 11(b), through code analysis by LLM, the first-level routing tokens are distilled from the jump table, but the second-level routing tokens cannot be analyzed due to the problem of indirect call and unknown data structure. Fortunately these second-level routing tokens exhibit a similar formatting pattern, which the LLM can learn and express as a regular expression. Utilizing this regular expression, we have successfully extracted subordinate routing tokens in the data segment.

Appendix B
Self-Correction Demonstration

We present an illustrative example to demonstrate the specific self-correction process for one device. As depicted on the left side of Fig. 12, the initial output of the LLM may omit certain correct tokens (e.g., 'GetSysStatus', 'AddPortMapping') and incorrectly identify others (e.g., 'Reboot'). To address this, we have designed a correctness-checking operation for the LLM's results, employing an adjusting prompt to guide the LLM in correcting its previous output, as illustrated on the right side of Fig. 12. This adjusting prompt includes the correct cases, missing cases, and error cases, all of which are extracted from the LLM's most recent output.

The correctness-checking process is visualized in the center of Fig. 12. The EAGLEYE distills a set of public routing tokens to serve as the verification set and prepares a set of non-routing tokens as the negative token set. By comparing the verification set with the LLM's output tokens, we can identify the missing and correct cases. The error cases are determined by comparing the negative token set with the LLM's output tokens. The checking results are then incorporated into the adjusting prompt to obtain new adjusted tokens, which are subsequently verified for correctness. This iterative adjustment process continues until the correct rate reaches the highest possible level.

Appendix C
Public Requests Collection

To comprehensively gather public interfaces, we employ an interactive collector functioning as a man-in-the-middle proxy to intercept network traffic. As clients interact with devices according to the documentation, triggering events and navigating through pages, the collector logs all traffic with duplication. This method allows EAGLEYE to capture all standard webpage transitions.

It is noted that to thoroughly explore public interfaces, manual operation of the device following the provided documentation is necessary, particularly for services or function-

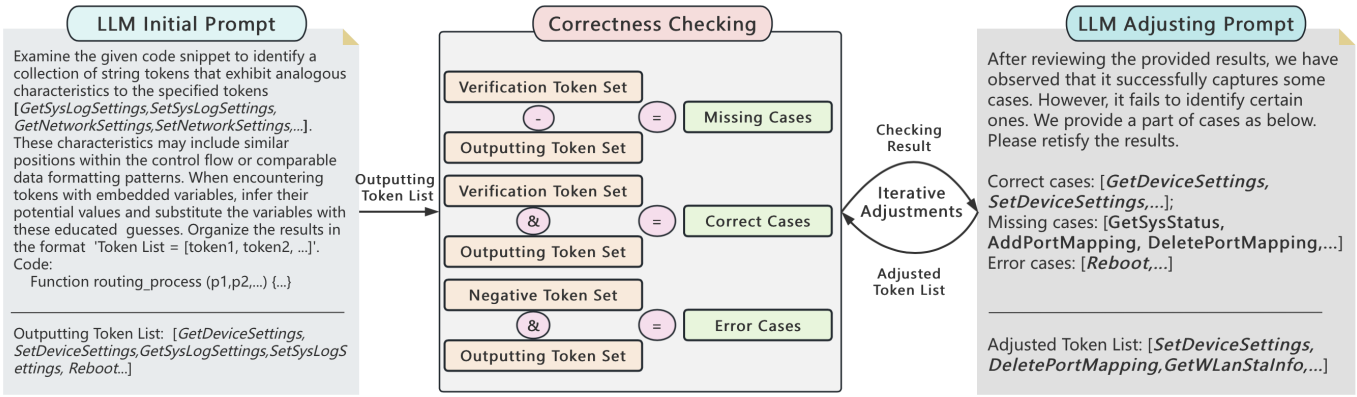


Fig. 12: An illustrative example to demonstrate the specific self-correction process for LLM.

alities contingent upon orderly configurations. This approach guarantees a complete inventory of public interfaces. While the approach necessitates manual intervention, it is a one-time procedure that does not demand extensive labor. As a precursor step, this collection process does not impeding the automation of subsequent processes.