PANGOLIN: Fuzzing Multilingual IoT Firmware with LLM-Driven Code Analysis

Zhipeng Jia[1], *Xiaokang Yin*[1]*, *Shuitao Gan*[4], *Chao Zhang*[2,5]*, *Hangtian Liu*[3], *Jiangan Ji*[1], *Enzhou Song*[1], *Ruijie Cai*[1], *Jinglei Tan*[3], *Shengli Liu*[1]

[1]*Information Engineering University*
[2]*Institute for Network Sciences and Cyberspace, Tsinghua University*
[3]*State Key Laboratory of Mathematical Engineering and Advanced Computing*
[4]*Laboratory for Advanced Computing and Intelligence Engineering*
[5]*JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.*

Abstract

Multilingual IoT typically refers to the use of multiple languages to implement its web services, such as C, Python, Lua, etc. While some user-accessible interfaces are visualized through the frontend for interaction, a large number of interfaces remain hidden and are not exposed to the frontend in multilingual IoT. Additionally, their parameters often exhibit complex hierarchical structures. Effectively extracting interface specifications from multilingual devices for vulnerability discovery is an urgent problem that remains unresolved. In this paper, we present PANGOLIN, a novel fuzzing solution designed for multilingual IoT devices. First, we utilize LLMs to analyze API dispatching mechanisms and identify interfaces. Then, we introduce an LLM agent to perform cross-language analysis and generate input parameter specifications. Lastly, we utilize response-driven feedback to correct parameter specifications. This knowledge enables semantics-aware fuzzing that can explore deeper code paths and discover more vulnerabilities. PANGOLIN successfully discovered 68 previously unknown vulnerabilities, i.e., 2.96X more than SOTA tool LABRADOR. Notably, 45 of these vulnerabilities were found in hidden interfaces, whereas EAGLEYE was only able to identify 4 such cases. As of the time of writing, all vulnerabilities have been reported to vendors and acknowledged, with 31 vulnerability IDs assigned.

1 Introduction

Multilingual IoT firmware typically refers to the use of multiple languages, such as C, Python, Lua, etc., to implement its web management services. Unlike traditional monolithic binary backends, which struggle to accommodate the rapidly evolving and increasingly diverse functionality demands of modern IoT scenarios, multilingual backends greatly enhance the scalability of IoT devices, simplify updates and maintenance, and enable a unified interface style that facilitates interaction with various control terminals while promoting continuous development. Major companies such as Cisco [9], Huawei [14], and Xiaomi [30] have already adopted this design paradigm to develop feature-rich IoT devices.

Detecting vulnerabilities in multilingual IoT firmware is still a challenging task. Existing static analysis vulnerability discovery approaches (for example, SaTC [6], LuaTaint [28], HermeScan [12], and MangoDFA [13]) targeting IoT firmware are typically limited to a single language, binary or Lua, and cannot perform cross-language boundary analysis. Moreover, static analysis methods typically require extensive manual effort to verify the generated alerts. Another type of IoT firmware vulnerability discovery solution is fuzzing. Since testing IoT devices often depends on specific peripherals or configurations, various fuzzing approaches (e.g., Firm-AFL [36], Greenhouse [26], FirmFuzz [25], HouseFuzz [29]) focus on rehosting firmware to improve fuzzing efficiency, scalability, and vulnerability detection capabilities. Other fuzzing solutions (e.g. SmartTVs [1], Snipuzz [11], IoTFuzzer [5], ESRFuzzer [33], Labrador [20]) focus on guiding fuzzers to explore more code paths effectively, which typically rely on network traffic as initial seeds and leverage logs, code snippets, or response strings as feedback. However, for multilingual IoT firmware, these solutions still face significant limitations. Specifically, they fail to capture the full set of backend interfaces and parameters from network traffic alone. And, they struggle to handle the hierarchical parameter structures caused by deep parameter parsing and cross-language function calls, as well as complex parameter constraints.

To more effectively discover vulnerabilities in multilingual IoT firmware, we propose a novel fuzzing approach, named PANGOLIN, which leverages LLMs (large language models) to identify testing interfaces and parameters. We observe that multiple interfaces in one multilingual IoT device often share the same dispatch mechanism and LLMs could help recognize the patterns. Besides, LLMs could understand the high-dimensional semantics of code and significantly facilitate the generation of parameter specifications. Following the above observations, PANGOLIN analyzes the backend dispatch mechanism to extract all available interfaces and their

*Co-corresponding authors: Xiaokang Yin and Chao Zhang.

corresponding handler entry points, and leverages an LLM agent to generate multidimensional input parameter specifications, which are then used to guide the semantic-aware fuzzing process.

An efficient fuzzer should thoroughly explore the interfaces of IoT firmware as well as their parameters. To achieve this goal for multilingual IoT firmware, we must address three key challenges: ***C1: Construction of a Comprehensive Entry Map***. Each interface of an IoT firmware corresponds to a specific entry URI, which is accessible only when it can be parsed and dispatched to a backend handler. To thoroughly explore the interfaces, we need a complete map between interfaces and backend handlers. However, the diverse dispatch mechanisms, multi-level routing strategies, and indirect function calls commonly found in multilingual IoT device backends significantly complicate the parsing and modeling processes required to construct a comprehensive entry map. ***C2: Generation of Parameter Specifications via interprocedural analysis***. For each interface to be tested, identifying its parameters is also critical. However, the complete handler of an interface typically involves the collaboration of multiple functions, including multi-level hybrid calls between scripts and binaries. Therefore, the parameters are parsed in a layered manner, resulting in a complex hierarchical structure, which makes comprehensive analysis of parameters particularly challenging. ***C3: Parameter Specifications Correction and Application to Fuzzing***. Issues such as inaccurate function call resolution and LLM hallucinations can often lead to parameter specifications that do not align precisely with the actual code semantics. Moreover, traditional fuzzing approaches guide mutations without considering code semantics, making efficient vulnerability discovery difficult.

To address these key challenges, we propose our novel solutions from three key perspectives. In the first phase, PANGOLIN constructs a tree-based entry map by identifying the dispatch mechanism and extracting entry URIs along with their corresponding handler points. In the second phase, PANGOLIN adopts a novel data structure called the Multilingual Call Graph (MCG) to identify hybrid calls. Pruned MCG helps generate parameter specifications using an LLM agent. In the third phase, PANGOLIN employs a response-driven feedback mechanism to refine MCG and recover parameter specifications. It further adopts a specification-guided strategy to optimize fuzzing actions, including node selection, mutation operators, and energy allocation.

To demonstrate the effectiveness and performance of PANGOLIN, we evaluated it on 12 multilingual IoT devices from 8 major manufacturers. It discovered 2,856 backend interfaces, including 1,793 hidden from the frontend. In total, 68 0-day vulnerabilities were found, involving command injection (CI), cross-site scripting (XSS), information leak, denial of service (DoS) and arbitrary file operations.

In summary, the paper makes the following contributions:

- We presented a LLM-driven approach able to extract interfaces and parameter specifications of multilingual IoT firmware, and a response-based feedback to correct parameter specifications.
- We have implemented a prototype of PANGOLIN[1], a novel practical fuzzing method for physical devices that is effective at detecting vulnerabilities in complex multilingual IoT devices.
- We evaluated PANGOLIN on 12 multilingual IoT devices from 8 well-known vendors and successfully identified 68 0-day vulnerabilities. As of now, all vulnerabilities have been confirmed by vendors, and 31 have been assigned IDs.

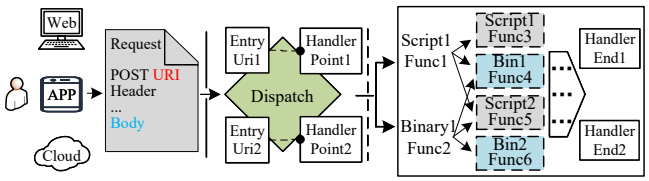


Figure 1: **An example abstract processing architecture in multilingual IoT devices.**

2 Background and Motivation

In this section, we provide an overview of the processing architecture in multilingual IoT devices, briefly and effectively illustrate the challenge in discovering vulnerabilities with two examples in multilingual IoT devices, and discuss the limitations of existing methods.

2.1 Multilingual IoT processing architecture

We conduct a detailed analysis of the processing architecture of multilingual IoT devices. Although the specific implementations vary across vendors, they can be modeled and represented as shown in Figure 1. When a user submits control request data to a multilingual IoT device through the management WebUI interface (local web pages, mobile applications, and cloud platforms), the backend will deal with the request data once it receives the data. The backend of IoT devices is designed to receive request packets from the frontend and perform dispatching based on specific values found in the URI or body fields of the request. The backend will match the corresponding URI (we define the path field of the URI together with the POST query field as the **entry URI**) for dispatch, passing the parameters to the backend functions for execution (we call the first function name in the backend that processes the body field as **`handler point`**.). Handler points in multilingual IoT devices contain script-based programs (e.g., programs implemented with Lua, Python and etc.)

[1] https://github.com/vul337/PANGOLIN.git

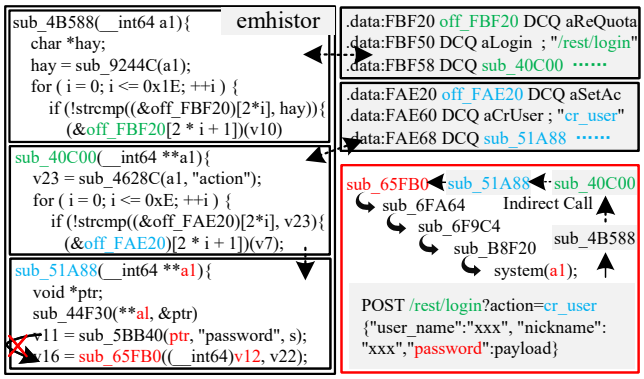


Figure 2: **An example of an entry URI indirect call combined with a parameter structure pointers.**

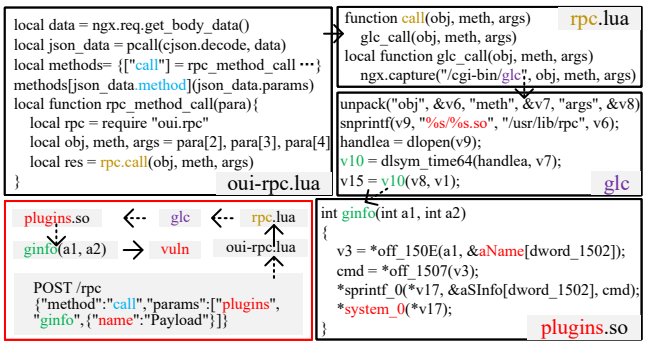


Figure 3: **An example of a hybrid call between scripts and binaries.** Fuzzing fails to detect vulnerabilities due to complex parameter specifications.

and binary-based programs (e.g., httpd, glc, plugins.so and etc.). Starting from these entry handler points, the backend executes script functions and binary functions with hybrid calls between script and binary components, to parse and apply the parameters of the request from the frontend.

2.2 Motivating Example

The WebUI frontend typically provides management interfaces for the device along with parameter constraints. When hunting for vulnerabilities in IoT devices with fuzzing, researchers usually start by pulling URIs and parameter formats from the frontend of the device's WebUI. These details are then used to craft the test cases for fuzzing. However, during our investigation, we found that due to the lack of semantic understanding of the code, relying solely on information obtained from the WebUI frontend cannot effectively support vulnerability discovery. Meanwhile, a large number of interfaces are not explicitly defined or rendered in the WebUI frontend, which we refer to as **hidden interfaces**. These hidden interfaces may have a severe impact on the device.

Since all web requests must be processed by the backend programs, the backend programs must include all interfaces and perform parameter parsing, sanitization, and execution code segments. The intuitive idea is to extract hidden interfaces and their corresponding complete parameter specifications (including the overall parameter structure, parameter value types, individual parameter formats, and static values) from backend programs. However, since the backends of multilingual IoT devices contain both script-based and binary-based programs, extracting these hidden interfaces and parameter specifications remains a significant technical challenge. Next, we will use two specific motivation examples to illustrate this challenge [2].

Hidden Interfaces. Figure 2 illustrates an example of a multi-level dispatch mechanism where a hidden interface triggers a vulnerability. This interface *(/rest/login?action=cr_user)* consists of two parts separated by a "?". The first part (*/rest/login*) is used for first-level handler dispatching to determine the corresponding processing function, while the second part (*action=cr_user*) is used for second-level handler dispatching to determine the subsequent processing function. This hidden subinterface (*cr_user*) does not appear in the WebUI frontend, but it is processed by the backend. Specifically, the backend iteratively compares a data segment of size *0x1E*2* pointed to by *off_FBF20* based on */rest/login*, thereby locating the interface *sub_40C00* (indirect call). This device then processes *cr_user* by comparing a *0xE*2* data segment pointed to by *off_FAE20* to identify the handler at *sub_51A88* (indirect call). Parameters are passed through *a1* (i.e., *V12*), which points to a user information structure. Through five layers of function call, a command injection vulnerability is ultimately triggered due to insufficient validation of the *password* parameter.

Complex Parameter Specification. Figure 3 illustrates the customized complex parameter specification in multilingual IoT devices. After receiving the request packet, the backend routes it to *oui-rpc.lua* based on the URI path field */rpc*. It extracts the value *call* associated with the key *methods* and indirectly invokes the *rpc_method_call* function. Parameters are then passed to the binary program *glc* through underlying nginx communication. The function in *glc* retrieves the value list corresponding to the key *params* and, based on the first two string elements in the list (including *plugins* and *ginfo*), it indirectly invokes the *ginfo* function located in */usr/lib/rpc/plugins.so* with *v10*. The function *ginfo* then parses the third dictionary element in the list to extract the value *Payload* associated with the key *name* for further execution. Ultimately, due to insufficient validation of the *Payload* value, a command injection vulnerability is triggered.

From the above two examples, we observe that multilingual IoT devices contain hidden interfaces that rely on indirect calls and employ complex nested parameter specifications to transmit user requests. In processing these requests, both

[2]For security considerations, we have made special handling for function addresses and URIs while maintaining the original processing workflow.

script-based and binary-based programs are used to parse parameters, often involving deep call chains. These characteristics introduce significant challenges for extracting hidden interfaces and reconstructing parameter structures from backend programs.

2.3 The Limitations in Existing Approaches.

Current IoT static analysis tools, such as Karonte [24], SaTC [6], EmTaint [7], LuaTaint [28], SinkTaint [32], HermeScan [12], and MangoDFA [13], are designed for either binary or Lua code, and are difficult to deal with deep indirect call chains. Therefore, they cannot perform cross-language analysis and have poor effectiveness in discovering vulnerabilities in multilingual IoT devices.

Existing black-box fuzzing tools have made significant efforts in seed generation and mutation guidance. For example, Snipuzz [11] captures network packets and manually filters valid message sequences. IoTFuzzer [5] uses Monkeyrunner [2] to obtain initial seeds and performs data type classification based on taint sources and taint propagation, enabling only basic recognition of parameter types. SRFuzzer [34] and ESRFuzzer [33] utilize crawlers to collect input data and annotate parameters merely as numbers, fixed strings, or variable strings. These tools rely exclusively on frontend-visible seeds, ignoring backend interfaces that are not exposed through the frontend and overlooking hidden branches of exposed interfaces, leading to missed vulnerabilities. EAGLEYE [21] extracts routing tokens in advance and employs fuzzing to discover hidden interfaces that are not defined in the frontend. However, it performs poorly when handling hierarchical interfaces and indirect calls. Additionally, due to the absence of parameter specification recognition, it cannot effectively guide mutation strategies. VFuzz [18] leverages neural network models to predict the likelihood of vulnerability in functions and accelerates fuzzing accordingly. SmartTVs [1] uses logs to guide seed generation and mutation, while Labrador [20] infers program execution paths from network responses and uses distance metrics to guide mutation.

Current SOTA IoT fuzzing approaches lack semantic understanding of code. Mutation-based approaches alone are insufficient to meet the complex parameter constraints required to trigger vulnerabilities, resulting in missed vulnerabilities and reduced fuzzing efficiency. To address the limitation in the current approaches, we propose a novel code semantics-guided fuzzing framework to discover vulnerabilities in multilingual IoT devices.

3 Challenges and Solutions

To the best of our knowledge, there are currently no existing techniques that leverage code semantics to guide fuzzing for multilingual IoT devices. To achieve this, an intuitive workflow can be outlined as follows: ❶ Locate the dispatch code to extract the entry URIs and the corresponding `handler points`. ❷ Analyze multilingual call mechanisms and generate parameter specifications using LLMs. ❸ Semantics-guided fuzzing based on program code analysis. However, three main challenges arise with this workflow. In this section, we first discuss the challenges encountered (in §3.1) and then introduce key ideas of our solution (in §3.2).

3.1 Challenges

Challenge 1: Locate the dispatch code to extract the entry URI and the corresponding `handler points`. Each interface of an IoT device corresponds to a specific entry URI, which must match the backend definition to access the associated functionality. For heterogeneous backends written in multiple programming languages, the implementation of API dispatch mechanisms varies significantly. These systems often involve multi-level dispatching and indirect function calls, making it challenging to locate all relevant dispatch code and extract entry URI across a large backend codebase.

Challenge 2: Analyze multilingual call mechanisms and generate parameter specifications using LLM. The quality of parameter specifications directly influences the effectiveness of vulnerability discovery. The complete implementation of interface functionality associated with a given point often involves function calls across multiple programming languages and hierarchical parameter extraction. Merely identifying the code block corresponding to a specific point is insufficient. Accurately constructing the inter-language function call graph is inherently challenging due to the presence of hybrid calls between scripts and binaries, as well as indirect calls. Large language models exhibit strengths in understanding code semantics. However, their capabilities are constrained by limited context window sizes, necessitating effective pruning of the call graph to ensure relevant context is preserved. Enabling LLM-based agents to generate high-quality parameter specifications that can assist in overcoming test barriers, reducing false negatives, supporting the detection of multiple types of vulnerability, and guiding effective mutation remains a significant challenge.

Challenge 3: Semantics-guided fuzzing based on program code analysis. How can we ensure the quality of the generated parameter specifications? Poor quality may arise from incorrect pruning decisions or from the inability of LLMs to accurately understand program semantics, as well as from hallucinations during generation. Furthermore, certain vulnerabilities cannot be triggered by a single request alone; instead, they require a coordinated sequence of requests. Understanding the implicit relationships among different interfaces remains a significant challenge. To enable efficient vulnerability discovery, it is essential to determine three key aspects: *where to mutate*, *how to mutate*, and *how many times to mutate*. Identifying optimal strategies for these dimensions is critical to achieving effective and targeted fuzzing.

3.2 Solutions

To address these key challenges, we propose our novel solutions from three key perspectives: 1) Construction of entry map based on tree structures, 2) Generation of parameter specifications based on pruned MCG, and 3) Fuzzing guided by parameter specifications. We reformulate the challenge of cross-procedural analysis in multilingual IoT devices into the problem of recovering high-quality API specifications. By leveraging LLM agents and fuzzing techniques, we bridge the semantic gap across language boundaries to enable efficient and rapid detection of diverse types of vulnerabilities.

To address Challenge 1, we introduce a tree-structured entry map construction approach that identifies and locates backend dispatch logic, enabling the reconstruction of hierarchical mappings between entry URIs and corresponding `handler points`.

To address Challenge 2, we propose a novel data structure called the Multilingual Call Graph (MCG) to resolve the complexities of inter-language function calls in multilingual IoT firmware. Generate semantically accurate parameter specifications by leveraging LLM Agents in conjunction with the pruned MCG.

To address Challenge 3, we design a feedback-driven adaptive framework that dynamically adjusts MCG pruning and parameter specification generation with both response content and previous function index. Based on specifications, we perform API sequence identification, prioritize entry nodes, generate mutation operators, and allocate energy to optimize fuzzing effectiveness.

4 Design of PANGOLIN

In this section, we provide the design details of our approach, called PANGOLIN. Figure 4 illustrates the architecture of PANGOLIN, which consists of three key modules:

Entry Map Construction (§4.1). This module focuses on constructing the mapping between entry URIs and corresponding `handler points` by identifying and analyzing dispatch code based on a tree-structured approach.

Parameter Specification Generation (§4.2). This module focuses on performing reverse pruning of the MCG using LLMs and generating semantically accurate parameter specifications based on the pruned MCG through LLM Agents.

Correction and Specification Guidance (§4.3). This module regulates MCG pruning and parameter specification generation based on response content and previous feedback from the pruned MCG index. It then guides fuzzing with multidimensional parameter specifications.

4.1 Entry Map Construction

PANGOLIN is designed to detect vulnerabilities in multilingual IoT devices through fuzzing, with the first challenge being to address interface coverage. PANGOLIN first standardizes both scripts and binaries into a unified format. It then splits the entry URIs of public interfaces into independent strings and calculates their frequency of occurrence across functions, which allows it to locate dispatch-related functions. Based on the call relationships, these functions are organized into a tree structure. The LLM analyzes each path from the root to the leaf to generate the entry map. In addition, we define three dispatch mechanisms that are incorporated as few-shot examples to further guide the LLM analysis.

4.1.1 Format Standardizing for Script and Binary

As we mentioned above, the WebUI backend of multilingual IoT devices contains both script-based and binary-based programs. Since binary-based programs are very different from script-based programs, it is hard to analyze them with a unified approach directly. For LLMs, understanding binary pseudocode is significantly easier than assembly. Splitting long scripts by functions also enhances LLM versatility. So, to make the data easier for LLMs to handle, we start by standardizing it into a consistent format.

For the binary-based programs, we disassemble the program and subsequently obtain the decompiled pseudocode. We further optimized the decompiled output for code accessing the data segment. By applying extensive regular expression matching, we identified multiple data access patterns, including array indexing, function pointers, global variables, constant strings, and loop-based retrieval. The content corresponding to the identified addresses and sizes in the data segment was extracted and used to replace the original elements in the pseudocode. To facilitate the recognition of subsequent indirect calls in loop memory access, we transform the loop structure into a switch-case structure.

For the script-based programs, each script file is divided into multiple files based on individual functions. We divide each script file into multiple files according to its functions and attach the global variables to the outer scope of each function. To address potential indexing errors caused by script package aliases, we identified all import statements and replaced every package name inside the functions with its actual package name.

4.1.2 Entry URI Dispatch Patterns in Multilingual IoT

To enable the LLM to better understand the task and generate accurate mappings between entry URIs and `handler points`, we summarize three dispatch mechanisms that cover the majority of multilingual IoT devices based on extensive empirical observations across a large set of firmware samples.

Function Segment. The correspondence between the entry URI and the `handler point` is established through a custom registration function, where both the entry URI and the `handler point` are passed as function parameters.

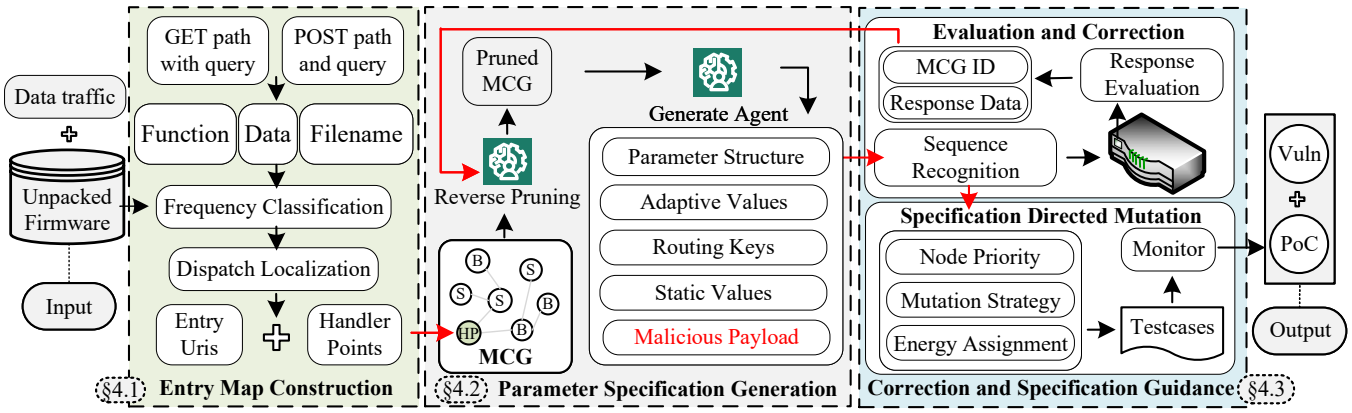


Figure 4: **Overview of PANGOLIN**

Data Segment. The correspondence between the entry URI and the `handler point` can also be realized through stored data. For script-based implementations, the entry URI typically serves as the key in a dictionary, while the handler point corresponds to the associated value. For binary-based implementations, both the entry URI and the handler point are usually represented as consecutive strings in the data segment.

Filename Segment: Dispatch can also be performed directly through file names, where the entry URI corresponds to the file name and the handler point is the main function within that file. We have crafted three illustrative examples in Appendix A to elucidate the dispatch mechanism.

The adoption of LLMs in entry map construction is mainly motivated by the challenge of multi-level and customized dispatch mechanisms prevalent in real-world firmware. Concrete implementations of these three dispatch patterns often vary across languages and vendors, involving nested or semantically dependent logic that cannot be adequately captured by regular expressions or heuristic rules. Traditional pattern-based matching methods struggle to generalize in such scenarios, as dispatch logic frequently embeds implicit semantics beyond syntactic patterns. In contrast, LLMs enable semantic reasoning for non-standard or various dispatch forms, thereby providing superior scalability and generality compared to conventional approaches.

4.1.3 LLM-Assisted Pattern Matching

The same device often employs similar mechanisms for dispatch. According to the principle of locality, dispatch code within the backend is usually distributed in one or several small clusters. Therefore, we assume that the entry URIs of public interfaces obtained from traffic can be leveraged to locate dispatch functions, which in turn allows the extraction of all URIs and their corresponding handler points. We collect all publicly defined frontend interface data from captured traffic, extracting the path field from all POST request entry URI entries, the query field if present, and the path field from a limited number of parameterized GET request packets. Although parameters may also possess dispatching functionality, we define it as the **routing value** to be addressed in §4.2.

Algorithm 1 provides a detailed description of how `PANGOLIN` constructs the entry map. We preprocess the extracted path values and POST request query values. For path values, we split them using the "/" delimiter; for query values, we split them using the "=" symbol and extract the string following the equal sign. These extracted components are converted into individual strings and deduplicated. Frequency statistics are then computed based on the occurrences of these strings within the preprocessed functions. To prevent identical strings from distorting the frequency distribution, we applied deduplication. We calculate the frequency of these individual strings within the functions obtained after format standardization. Functions are then classified into two categories based on abrupt changes in the frequency distribution, with the higher-frequency category considered as functions that contain backend dispatch code. We construct a tree structure from these functions according to their call relationships to capture multi-level dispatch hierarchies. The LLM performs a depth-first traversal over each path from root to leaf. To improve task comprehension, we construct a few-shot prompt using the three predefined dispatch patterns together with a small set of publicly exposed URIs from the frontend. This process ultimately generates the complete entry map.

4.2 Parameter Specification Generation

In this module, we introduce a novel structure called the Multilingual Call Graph (MCG) to represent the call relationships among functions in the backend of multilingual IoT devices. To generate parameter specifications with LLMs, pruning of the MCG is necessary due to the context length limitation of LLMs. However, constructing a complete MCG in advance and then pruning it is unnecessary. Instead, we adopt a reverse pruning strategy that incrementally identifies the required callees starting from the handler point, thereby producing a

Algorithm 1 Building entry map

Input: Requests R, Functions F
Output: Entry Map M (URIs $\rightarrow$ Handler points)

$T \leftarrow \emptyset, M \leftarrow \emptyset$
$T \leftarrow T \cup \text{SPLITENTRYURI}(R)$ ▷ Split entry URI
$Freq[f] \leftarrow \{f \in F \mid \sum_{t \in T} \text{COUNTOCCURRENCES}(f,t)\}$
$\tau \leftarrow \text{CALCFREQJUMPS}(Freq)$ ▷ Detect Frequency Jumps
$H \leftarrow \{f \in F \mid \text{Freq}[f] > \tau\}$ ▷ Clustering
$G \leftarrow \text{BUILDCALLGRAPH}(H)$ ▷ Build call graph
for root $\in \text{ROOTS}(G)$ **do** ▷ Start from the root dispatch
  $Q \leftarrow \text{CHILDREN}(G, \text{root})$ ▷ Get all children of root into queue
  $M \leftarrow M \cup (\text{uri}, hp) \leftarrow \text{LLMFEWSHOT}(Q)$ ▷ LLM Pattern Matching
end for
return M

Algorithm 2 Generate pruned MCG

Input: Handler point hp, Global index G
Output: Pruned MCG

$Q \leftarrow hp, S \leftarrow \emptyset$
while $Q \neq \emptyset$ **do** ▷ Completed when queue is empty
  $subid \leftarrow \text{DEQUEUE}(Q)$
  if $subid \notin S$ **then** $S \leftarrow S \cup \{subid\}$ ▷ Avoid repeated analysis
    $id \leftarrow \text{GETGLOBALID}(G, subid)$ ▷ Get global index
    $CalleeIds \leftarrow \text{LLMREVPRUNING}(G[id])$ ▷ Reverse pruning
    $purified \leftarrow \text{PURIFYCODE}(G[id])$ ▷ Purify long code
    $\text{ENQUEUEALL}(Q, CalleeIds)$ ▷ Add new id to queue
  end if
end while
return pruned MCG

pruned MCG used for parameter specification generation.

4.2.1 MCG Definition

First, we utilize graph notation to rigorously define the formal representation of the MCG.

Definition 1 (MCG). The MCG is a function-level call graph that represents the call relationships and parameter-passing dependencies among functions across multiple programming languages using a unified set of nodes and edges. We define them below.

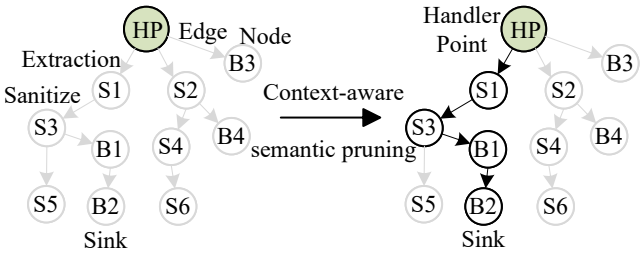
Definition 2 (MCG Node). The candidate node set N of the MCG contains all function names from the backend. To uniquely represent call relationships across multiple languages and processes, each node must have a unique identifier. For functions, including scripts and binaries, we assign a unique identifier in the format *path_name.file_name.function_name*.

Definition 3 (MCG Edge). Edges represent the call relationships among functions. To address the complexity of multi-language call interactions, we employ `Call Pruning Agent` for identification and define the recognized function names as node sub-identifiers.

4.2.2 Generate pruned MCG

We adopt a reverse pruning strategy that incrementally identifies the required callees starting from the handler point. Based on the semantic understanding of the code provided by LLMs, we only identify two types of calls: ❶ calls for parameter extraction and execution. ❷ calls of sanitizers. As shown in Figure 5, the code referenced by the handler point invokes *script1*, *script2*, and *binary3*. Through semantic analysis of the code, only *script1* is retained. Similarly, *script3*, *binary1*, and *binary2* are preserved, resulting in a pruned MCG that is simplified as much as possible without compromising the quality of the generated parameter specifications.

The pruning prompt is illustrated in Figure 6. We identify two categories of calls. When the LLM successfully finds all of them, we stop analyzing additional callees. If the discovery is incomplete and the callees may still contain these



(a) Initial Unconnected MCG (b) Pruned Connected MCG

Figure 5: **An example of context-aware semantic pruning.**

categories, because binaries lack actual function name information, the identified number of branches may become excessively large. To address this, we restrict the output to at most three sub-identifiers. When the LLM determines that no potential calls of these categories exist, we stop analyzing the corresponding callees. For direct and indirect calls between scripts and binaries, we output the function name as the sub-identifier. We also classify calls where parameters are processed by a script file or a binary file into this type, outputting *filename.main* as the sub-identifier. For indirect library calls, the function names with an in-degree of zero and containing explicit strings are used as sub-identifier.

The complete workflow of reverse pruning is illustrated in Algorithm 2. We store the identified sub-identifiers in a FIFO queue. Each sub-identifier is matched in reverse order against the identifiers of nodes in the Node set, and the content of the corresponding node is extracted for further identification. When the queue becomes empty, the pruning of the MCG for a `handler point` is complete. To address the issue of excessively large functions, we refine the code by preserving only the parts relevant to parameter handling, while keeping the semantics intact, thereby reducing and optimizing the code as much as possible. Since each `handler point` operates independently, we employ a multithreaded approach, which significantly accelerates the MCG pruning process.

Prompt Template

Task: Based on parameter input methods, identify two types of custom functions or dirs: (1) parameter extraction and execution; (2) sanitizers, follow these principles:

1) If found all, and the context indicates that the function terminates, output NULL.

2) If incomplete discovery, and the code suggests that other callees may exist, provide up to three candidates.

3) If context suggests that no candidate callees, output NULL.

Output Format: Return the results in JSON format as {'result': {'functions': [''], 'dirs': ['']}}.

Figure 6: **The prompt of MCG pruning.**

4.2.3 Specification Generation based on pruned MCG

We perform control-flow and data-flow analysis on the pruned MCG using `Parameter Generation Agent` to generate parameter specifications, including the **structure**, **values**, and **labels** of the parameters.

As shown in Figure 7, `PANGOLIN` performs a holistic analysis of the parameter passing process in the pruned MCG, identifying how parameters are progressively extracted and parsed across nodes, and generates a structure that satisfies inter-parameter dependencies. To better ensure test cases reach the target code regions, we identify specific values from conditional checks, as well as type checks and format validations, to produce both fixed values and adaptive values that align with the code semantics. For parameters with dispatch functionality, whose values determine subsequent execution branches that each contain dangerous functions, we define such values as **routing values**. To reduce false negatives during fuzzing, we combine routing values with normal values to form multiple parameter specifications that conform to the parameter structure. We employ a few-shot learning approach to construct prompts. We define several strict sanitizers to enable the Agent to better identify unsafe sanitizers and then perform a lightweight analysis of the code to determine multiple vulnerability types. We describe its construction process in Appendix D. We mark parameter values as either non-mutable or mutable. For mutable values, if unsafe sanitizers are detected, we label their potential vulnerability types and generate payloads more likely to trigger vulnerabilities based on the corresponding default commands.

4.3 Correction and Specification Guidance

`PANGOLIN` performs vulnerability discovery based on the generated URIs and parameter specifications, but it first needs to identify the relationships among different URIs to satisfy the call sequences required for triggering specific vulnerabilities. In addition, the generated parameter specifications may contain errors, which must be adjusted accordingly.

Prompt Template

Task: Generate parameter specifications based on pruned MCG content corresponding to the entry URI, , follow these principles:

1) Confirm the request method (GET, POST, ...).

2) Trace the parameter parsing process to reconstruct the complete parameter structure.

3) Identify parameters with unique values, Label as NoMuta.

4) Generate adaptive values based on type checking and format validation. Label as NoMuta.

5) Identify parameter values corresponding to different sinks in a branch as routing values, and combine them with other parameter values to generate multiple result sets. Label as NoMuta.

6) Strong Sanitize few shot: Filter ' in '', filter all special characters (*;$'&"'*), parameter length limit, command whitelist, parameter type and format regex matching limit.

7) If a vulnerability exists, identify its type and mark it (CI, BOF, DoS, XSS, AFR, AFU) and generate payloads based on the default command. Label as Muta.

Output Format: Return the results in JSON format as {"result":{"normal_requests":[{"method":"","parameters":""}, {}], "label":[{"method":"","paramlabel":""}, {}]}}.

Figure 7: **The prompt of specification generation.**

4.3.1 URI Sequence Identification

In most cases, a single URI request is sufficient to trigger a vulnerability. However, certain vulnerabilities depend on multiple URI calls, with their execution closely tied to the order in which the calls occur. The entry URIs generated in §4.1 and the parameter specifications generated in §4.2 contain rich semantic information that reflects the dependencies among URIs. We use LLMs to analyze each entry URI and its parameter specification, constructing relationships among URIs and grouping dependent URIs into a testing unit. To help LLMs better understand the task, we define two types of URI sequence relationships: *shared resource* and *enable*. The shared resource relationship occurs when one URI temporarily stores input (such as a filename, configuration, or memory object) and another URI later consumes it for execution. The enable relationship occurs when access to certain requests becomes valid only after a specific request has enabled the corresponding functionality.

4.3.2 Evaluation and Correction

To mitigate the issue of inaccurate URIs recognition and parameter specifications caused by LLM hallucinations, erroneous MCG pruning and incorrect parameter inference, which hinders the effective generation of test cases and mutation guidance, we design a self-correction model to implement iterative adjustments. We employ pruned MCG nodes from the first N rounds, generated parameter specifications, and error outputs as feedback inputs to mitigate LLM failure modes.

We automatically generate valid request test cases based on the entry URIs and parameter specifications, then obtain the response status values and response content. To prevent excessively long response content from increasing the cost of feedback adjustment, we truncate the first 100 characters of the response content for evaluation. We pre-collected response and backend error information and used automated checks to identify obvious errors in the responses. For certain unexpected outputs, we employed the LLM to assist in evaluation. If an entry URI corresponds to a nonexistent request, for example, when a 404 not found error is returned, we remove it from the request pool. If the entry URI exists and its parameters require correction, we use the error output, MCG nodes, and parameter specifications as input to refine the reverse pruning of the MCG and the generation of parameter specifications. The maximum number of correction iterations is set to two. Across five replicate experiments, highly similar results were observed, which demonstrates the strong robustness of the method.

4.3.3 Specification Directed Mutation

Where to mutate. As we mentioned in §4.2, mutations applied to different parameter positions have varying impacts on vulnerability triggering. PANGOLIN assigns priority to parameters labeled in the generated specifications that can accelerate vulnerability triggering. Specifically, a mutation probability is set for each parameter based on its potential. The mutation probability is evaluated using the following factors: (1) Parameters labeled as mutable and associated with a vulnerability type are assigned the highest priority. By identifying sanitizers, such parameters are determined to be more likely to trigger vulnerabilities. (2) Parameter nesting depth. The deeper the nesting level, the higher the priority, as this indicates a longer parsing path in the MCG and may closer proximity to dangerous functions. (3) Parameters labeled as non-mutable are given the lowest priority, as these are often fixed values, and mutating them may reduce the likelihood of triggering vulnerabilities.

How to mutate. We built a static payload library containing both LLM-generated payloads and payloads capable of bypassing multiple vulnerability types and various sanitizers. we describe its construction process in Appendix D. LLM-generated payloads are prioritized, and when a potential vulnerability type is identified, payloads corresponding to that type are selected from the static library. PANGOLIN parses parameters holistically based on a hierarchical tree structure and, according to each parameter's mutation probability, applies mutation operators such as replacement and concatenation with the selected payloads, before restoring the original parameter structure.

How many times to mutate. For a given seed, its mutation energy is allocated according to the specification. In our experiments, we observed that 400 mutations were sufficient to produce the same alerts that 500 mutations (refs to LABRADOR [20]) would generate. So seeds with a vulnerability label are assigned a value of 400. Seeds without parameters are assigned a value of 200, as vulnerabilities may be triggered by API sequences, requiring direct requests without mutation. For other seeds without a vulnerability label, the energy is set to 300, and these seeds still require mutation because false negatives may arise from erroneous MCG pruning and inaccurate LLM analysis.

5 Implementation

We have developed a prototype of PANGOLIN, which consists of over 8,000 lines of Python code. The main components are as follows. We use binwalk [17] to extract the root file system from the firmware image. PANGOLIN collects the frontend requests from network traffic and decompiles the binary programs with IDA Pro. PANGOLIN locates the dispatch code through frequency distribution analysis and leverages the LLM mode DeepSeek-V3 [19] to analyze the dispatch code and generate the parameter specification with 20 threads, and we configure the temperature of DeepSeek-V3 with 0.7.

Collecting publicly defined interfaces. We only involved manual intervention when collecting the initial traffic data. Although we tried using automated crawlers to capture the WebUI, generating valid requests required complex parameter constraints and extensive customization, making the results significantly less effective compared to manually collecting request packets.

Automated Authentication. Most user interfaces require authentication checks, and sessions that pass verification are assigned a token with a defined lifespan. Therefore, proactive automatic token renewal is necessary. For devices from various brands in the dataset, PANGOLIN implements a generic token auto-refresh module that automatically updates tokens based on the brand and model once expiration is detected. Different models under the same brand often adopt similar authentication methods, and such similarities can also be observed across different brands. We categorize these device authentication methods and develop a token auto-refresh model for each category.

Fuzz-Breaking API filtering. When retrieving backend interfaces, it is inevitable to encounter APIs that modify the device IP address, power it off, or reboot it, which can interrupt fuzzing. Before commencing formal fuzzing, we conduct a preliminary testing round in which all seeds are iterated over while monitoring the device status, and any requests that disconnect the fuzzing engine from the device are removed.

Multiple Exceptions Detector. For command injection vulnerabilities, we execute specific commands targeting the local host and check whether a corresponding network request is received. For DoS and buffer overflow vulnerabilities, we determine device crashes by detecting whether the maximum predefined response time is exceeded. If the device

crashes, a smart plug automatically reboots the device. For arbitrary file read vulnerabilities, we set the default command to */etc/passwd* and monitor whether the response content contains output in the expected format. For XSS vulnerabilities, detection is performed by verifying whether the response contains the designated magic string.

6 Evaluation

To evaluate the effectiveness of PANGOLIN, we conducted a comprehensive set of experiments designed to address the following research questions:

- **RQ1:** How effective is PANGOLIN in detecting vulnerabilities within real-world multilingual IoT devices?
- **RQ2:** How does PANGOLIN vulnerability discovery performance compare to existing state-of-the-art tools?
- **RQ3:** How does each module in PANGOLIN contribute to vulnerability detection?
- **RQ4:** How does model choice affect the results and how efficient PANGOLIN is in performing the analysis?

6.1 Experiment Setup

Dataset. We selected 12 multilingual devices as the testing dataset, encompassing various types such as NAS, Camera, integrated router, wifi router, wireless controller and extender. Detailed information about the vendor and model of these devices is provided in Table 1. All devices are sourced from globally leading vendors, including HIKSEMI, Cisco, Linksys, Netgear, Xiaomi and TP-LINK, and cover both large enterprise devices and smart home devices. Among these devices, access control methods include cloud, mobile applications, local web, and client software. Their backend functionalities are implemented using multiple languages, including Lua, Python, and compiled Binaries. All testing is conducted on physical devices.

Table 1: **Summary of dataset.**

Vendor	Model	Device Type	Backend Mode
HIKSEMI	MAGE20PRO	NAS	Binary
TP-LINK	TL-IPC42A-4	Camera	Binary + Lua
Cisco	IOS-XE(C)	Wireless Controller	Binary + Lua + Python
	IOS-XE(Isr)	Integrated Router	Binary + Lua + Python
Linksys	E5600	WiFi Router	Binary + Lua
Netgear	EX8000	Wireless Extender	Binary + Lua
	EX6250	Wireless Extender	Binary + Lua
Xiaomi	R3A	WiFi Router	Binary + Lua
GL-iNet	AR300M16	WiFi Router	Binary + Lua
	MT300N-V2	WiFi Router	Binary + Lua
	XE300	Card Router	Binary + Lua
Ruijie	EG105GW	Gateway Router	Binary + Lua

Table 2: **Distribution of detected vulnerabilities (RQ1).**

Brand	Model	Type	Vuln	CVE/CNVD
HIKSEMI	MAGE20PRO	CI	5	2
		DoS	1	0
		leak	1	0
TP-LINK	TL-IPC42A-4	CI	2	0
Cisco	IOS-XE(C)	Arbitrary File Reads	1	0
	IOS-XE(Isr)	Arbitrary File upload	1	0
		CI	2	1
Linksys	E5600	CI	15	11
		XSS	4	1
Netgear	EX8000	CI	3	3
	EX6250	CI	3	3
xiaomi	R3A	CI	3	1
GL-iNet	AR300M16	CI	6	6
	MT300N-V2	CI	6	0
	XE300	CI	6	0
Ruijie	EG105GW	CI	9	3
Total			**68**	**31**

Baselines. LABRADOR [20] is a SOTA solution for IoT Devices Black-box Fuzzing and has demonstrated absolute advantages when compared with static-analysis-based SATC [6] and semi-white-box FIRM-AFL [36]. The mutation of test cases is guided by the distance between the response string and the sink point. EAGLEYE [21] is a SOTA solution to Exposing Hidden Web Interfaces in IoT Devices, which extracts candidate routing tokens via regular expressions, uncovering hidden interfaces through fuzzing.

Configurations. We conducted our experiments on a Kali system (version 6.8.11), equipped with an Intel Xeon Gold 6128 CPU at 3.40GHz and 48GB RAM.

6.2 RQ1: Effectiveness

To answer RQ1, we evaluate the effectiveness of PANGOLIN in discovering vulnerabilities across multilingual IoT devices.

Result Overview. Overall, PANGOLIN reported a total of 68 vulnerabilities, as shown in Table 2. Those vulnerabilities discovered by PANGOLIN cover several types, including CI, XSS, DoS, Leak and arbitrary file upload and download (AFU and AFD). Due to response delay, we preserved the first 30 mutated test cases that successfully triggered vulnerabilities and confirmed the final proofs of concept (PoCs) through direct execution. Appendix C contains the specific IDs.

Bug Disclosure. These detected vulnerabilities pose significant security threats to the target applications. All 68 vulnerabilities detected on physical devices will lead to severe security issues for the devices. Therefore, we promptly reported the vulnerabilities of the affected devices to the vendors, and some of these vulnerabilities have been timely addressed and fixed. As of now, all of them have been confirmed by the vendors, and 31 vulnerability IDs have been assigned. Among them, CNVD-2025-14455 and CNVD-2025-18467 received the maximum score of 10.0.

6.3 RQ2: Comparison

To answer RQ2, we evaluate the effectiveness of PANGOLIN in comparison with the LABRADOR [20] and EAGLEYE [21] across the entire dataset.

❶ **PANGOLIN Vs. LABRADOR.** Both PANGOLIN and LABRADOR leverage the response of the IoT device to construct a feedback mechanism. The most significant difference is that PANGOLIN directly generates initial seeds from backend code, leverages response-driven adjustments to refine parameter specifications, and guides fuzzing based on code semantics. In contrast, LABRADOR derives initial seeds from traffic and public documentation and guides fuzzing by measuring the distance from response strings to the target code.

Result Overview. Table 3 provides a detailed comparison of the effectiveness of PANGOLIN and LABRADOR across the entire dataset. Overall, PANGOLIN significantly outperforms LABRADOR in terms of both the number of vulnerabilities discovered and efficiency of static analysis. To be fair, run a 24-hour test following the LABRADOR configuration. LABRADOR discovered 23 vulnerabilities, while PANGOLIN uncovered 68 vulnerabilities, which is 2.96X more. In addition, the average time consumed for static analysis extraction was 19.1X faster than LABRADOR.

Vulnerability Count Analysis. PANGOLIN discovered 45 more vulnerabilities than LABRADOR. This is because LABRADOR obtains initial seeds from the frontend and public documents, without considering the scenarios where interfaces are defined only in the backend but not provided with access interfaces in the frontend. In contrast, PANGOLIN automatically extracts entry URIs and parameter specifications from the backend, enabling it to cover as many interfaces as possible. Through the analysis of the discovered vulnerabilities, the 45 vulnerabilities not found by LABRADOR are all interfaces not defined in the frontend. Moreover, LABRADOR requires parameter structures extracted from traffic as the basis for mutation, and it cannot generate the complex parameter structures of hidden interfaces.

Time Overhead Analysis. LABRADOR mainly involves three key areas where overhead concentrates: explicit string extraction, graph construction, and distance measurement. The time cost of PANGOLIN in static extraction consists of three parts: entry map construction, MCG pruning, and parameter specification generation. PANGOLIN is significantly more efficient than LABRADOR. In the static analysis stage, LABRADOR did not complete the static analysis of the Cisco device within 24 hours due to the large size of the service binary (IOSD). LABRADOR identifies target binaries based on shared strings between the frontend and backend, whereas PANGOLIN determines the dispatch code function directly from the entry URI. Our analysis shows that the primary functionality resides in backend scripts, so the binary was not analyzed. PANGOLIN preprocesses both scripts and binaries and leverages multithreading on the local file system, which greatly accelerates static extraction. In the fuzzing phase, Figure 8 shows the curve of vulnerability discoveries over time. PANGOLIN uncovers vulnerabilities in an explosive manner because it adopts specification-guided fuzzing. Compared with LABRADOR, which employs distance-guided directed fuzzing and relies on mutation to explore the vast input space of the backend, PANGOLIN is significantly more effective.

Table 3: **Comparison between PANGOLIN and LABRADOR (RQ2).**

Brand	Model	LABRADOR		PANGOLIN	
		Vuln	Static Time	Vuln	Static Time
HIKSEMI	MAGE20PRO	2	485.5min	7	23.7min
TP-LINK	TL-IPC42A-4	0	46.3min	2	2.7min
Cisco	IOS-XE(C)	0	>1,440min	1	4.6min
	IOS-XE(Isr)	0	>1,440min	3	2.2min
Linksys	E5600	9	17.3min	19	0.6min
Netgear	EX8000	2	24.2min	3	0.8min
	EX6250	2	23.8min	3	0.8min
Xiaomi	R3A	0	193.4min	3	9.7min
GL-iNet	AR300M16	2	164.5min	6	7.4min
	MT300N-V2	2	158.4min	6	7.2min
	XE300	2	166.2min	6	7.8min
Ruijie	EG105GW	2	87.8min	9	4.2min
Total		**23**	**1367.4min**	**68**	**71.7min**

❷ **PANGOLIN-H Vs. EAGLEYE.** EAGLEYE employs LLM-based code analysis with regular expressions to extract candidate values of routing tokens, leverages responses to supply necessary parameters, and uncovers hidden interfaces through fuzzing. To be fair, it is required to unify the comparison standards. We configured a variant of PANGOLIN, denoted as PANGOLIN-H, which removes the publicly extracted backend interfaces and focuses exclusively on uncovering vulnerabilities within hidden interfaces.

Table 4: **Comparison between PANGOLIN-H and EAGLEYE (RQ2).**

Brand	Model	EAGLEYE				PANGOLIN-H			
		HINT	B-A	A-A	vuln	HINT	B-A	A-A	vuln
HIKSEMI	MAGE20PRO	408	14	394	2	408	14	394	5
TP-LINK	TL-IPC42A-4	0	0	0	0	17	0	17	2
Cisco	IOS-XE(C)	119	0	119	0	119	0	119	1
	IOS-XE(Isr)	56	0	56	0	56	0	56	3
Linksys	E5600	0	0	0	0	25	0	25	10
Netgear	EX8000	11	0	11	1	11	0	11	1
	EX6250	10	0	10	1	10	0	10	1
xiaomi	R3A	271	0	271	0	271	0	271	3
GL-iNet	AR300M16	0	0	0	0	281	0	281	4
	MT300N-V2	0	0	0	0	272	0	272	4
	XE300	0	0	0	0	289	0	289	4
Ruijie	EG105GW	0	0	0	0	34	3	31	7
Total		**875**	**14**	**861**	**4**	**1,793**	**17**	**1,776**	**45**

HINT=the number of hidden interfaces, B-A=the number of hidden interfaces bypassing authentication, A-A=the number of hidden interfaces after authentication.

Result Overview. Table 4 provides a detailed comparison of the effectiveness of PANGOLIN and EAGLEYE across the entire dataset. On six devices, EAGLEYE identified the same number of hidden interfaces as PANGOLIN-H but detected only 4 vulnerabilities, whereas PANGOLIN uncovered 14 vulnerabilities. On the remaining six devices, EAGLEYE failed to

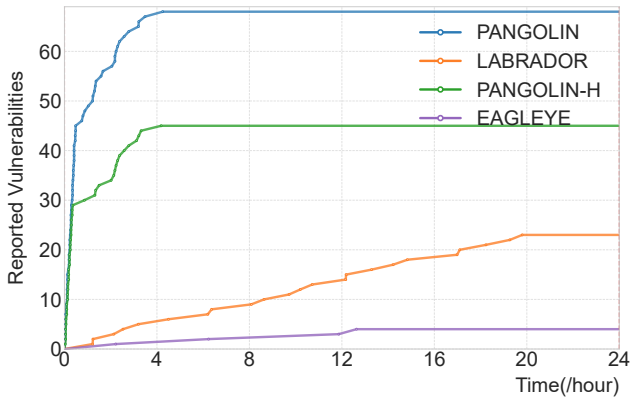


Figure 8: **Efficiency comparison between PANGOLIN and LABRADOR, PANGOLIN-H and EAGLEYE (RQ2).**

discover any hidden interfaces, while PANGOLIN identified 918 hidden interfaces and detected 31 vulnerabilities.

Hidden Interface Analysis. We analyzed why EAGLEYE identified hidden interfaces but failed to detect vulnerabilities. It relies on responses to obtain required parameters and depends on high-quality exception outputs generated during code development, which makes it difficult to handle complex parameter specifications and conditional checks. We also examined why EAGLEYE did not identify hidden interfaces on other devices. The core dispatch in these devices depends on multiple parameter fields and exhibits characteristics such as multi-level structures, cross-language interactions, dispersion, and cross-file dependencies. These properties make it difficult for simple mutations to satisfy value dependencies and parameter structures that do not appear in public interfaces.

6.4 RQ3: Ablation Study

To answer RQ3, we conduct an ablation study to demonstrate the necessity of each key component of PANGOLIN for the rapid and effective detection of vulnerabilities.

Variants Setup. We construct four variants of PANGOLIN, each of which disables a key component and uses the rest of the system as is. The details are as follows.

- *PANGOLIN-Entry* obtains entry URI directly from public interfaces and identifies the corresponding `handler point` without the PANGOLIN entry URI recognition component.
- *PANGOLIN-MCG* generates parameter specifications only from the initial code segment pointed to by `handler point` without the PANGOLIN component that generates parameter specifications based on pruned MCG.
- *PANGOLIN-Seq* performs purely sequential fuzzing on the requests without the PANGOLIN component for API sequence recognition.
- *PANGOLIN-FB* relies on the specifications generated in the first round to guide fuzzing without the PANGOLIN feedback regulation component.

Result Analysis. Table 5 provides the comparison results between PANGOLIN and its four variants. It is clear that these four key components are essential for effective vulnerability detection in Multilingual IoT devices. We also discuss dependencies among modules in Appendix E. A detailed analysis of the results is as follows:

❶ **PANGOLIN.** Overall, PANGOLIN identified 2,856 sets of parameter specifications, including 1,753 sets related to hidden interfaces. We pre-collected response and backend error information and used automated checks to identify obvious errors in the responses corresponding to the generated parameter specifications. For certain unexpected outputs, we leveraged the LLM for evaluation. In total, 2,253 parameters were recovered without causing display errors. To ensure the accuracy of the reported vulnerabilities, we employed a combination of automated monitoring and manual verification, resulting in the identification of 68 vulnerabilities.

❷ **PANGOLIN-Entry.** In this phase, we emphasize the importance of entry URI recognition by comparing the vulnerability detection results of PANGOLIN and PANGOLIN-Entry. Compared with PANGOLIN, PANGOLIN-Entry exhibits a 66% reduction in vulnerability discovery capability. Our analysis of the missed vulnerabilities shows that all of them originate from hidden interfaces.

❸ **PANGOLIN-MCG.** Overall, PANGOLIN-MCG detects 44 fewer vulnerabilities than PANGOLIN. For the vulnerabilities it fails to identify, our analysis shows that the relevant parameters are not fully represented in the `handler point` code but instead appear in subsequent multilingual calls with hierarchical structures. All the 23 vulnerabilities discovered by PANGOLIN-MCG, rely on parameter specifications extracted from traffic. These results demonstrate that parameter specification recovery based on pruned MCG provides a solid foundation for effective vulnerability discovery.

❹ **PANGOLIN-Seq.** Overall, PANGOLIN-Seq detects 8 fewer vulnerabilities than PANGOLIN. Our analysis of these missing vulnerabilities shows that they require more than one request to be triggered and involve two types of sequence relationships, namely shared resources and enable.

❺ **PANGOLIN-FB.** Overall, parameter specification R1 without feedback regulation discovers 25 fewer vulnerabilities within the same time frame. After one round of feedback regulation, R2 detects 19 fewer vulnerabilities. PANGOLIN performs fuzzing guided by the results of the second round. Based on the experimental data, we observe that the second round of feedback regulation provides greater improvement than the first.

Table 5: **Ablation study for four variants of PANGOLIN (RQ3).**

Brand	Model	Total	PANGOLIN-Entry			PANGOLIN-MCG			PANGOLIN-Seq			PANGOLIN-FB				PANGOLIN		
			Count	Prec	Vuln	Count	Prec	Vuln	Count	Prec	Vuln	Count(R1)	Vuln	Count(R2)	Vuln	Count	Prec	Vuln
HIKSEMI	MAGE20PRO	726	318	43.80%	2	213	29.34%	3	472	65.01%	7	394	3	416	3	472	65.01%	7
TP-LINK	TL-IPC42A-4	85	68	80.00%	0	0	0.00%	0	76	89.41%	2	67	2	79	2	76	89.41%	2
Cisco	IOS-XE(C)	314	195	62.10%	0	43	13.69%	0	244	77.71%	1	189	0	203	0	244	77.71%	1
	IOS-XE(Isr)	136	107	78.68%	0	38	27.94%	0	126	92.65%	3	113	2	117	3	126	92.65%	3
Linksys	E5600	33	21	63.64%	9	0	0.00%	9	33	100.00%	11	33	19	33	19	33	100.00%	19
Netgear	EX8000	36	25	69.44%	2	20	55.56%	2	32	88.89%	3	25	2	28	2	32	88.89%	3
	EX6250	34	24	70.59%	2	19	55.88%	2	30	88.24%	3	22	2	26	2	30	88.24%	3
xiaomi	R3A	410	139	33.90%	0	148	36.10%	0	357	87.07%	3	300	0	323	0	357	87.07%	3
GL-iNet	AR300M16	328	47	14.33%	2	0	0.00%	2	262	79.88%	6	241	3	249	4	262	79.88%	6
	MT300N-V2	317	45	14.20%	2	0	0.00%	2	256	80.76%	6	237	3	242	4	256	80.76%	6
	XE300	342	53	15.50%	2	0	0.00%	2	278	81.29%	6	245	3	251	4	278	81.29%	6
Ruijie	EG105GW	95	61	64.21%	2	20	21.05%	2	87	91.58%	9	52	4	63	6	87	91.58%	9
Total		2,856	1,103	38.62%	**23**	501	17.54%	**24**	2,253	78.89%	**60**	1,851	**43**	1,951	**49**	2,253	78.89%	**68**

Table 6: **Comparative experiments of different models (RQ4).**

Brand	Model	Total	GPT-4.1-nano			Cluade-3-5-haiku			DeepSeek-V3		
			Count	Prec	Vuln	Count	Prec	Vuln	Count	Prec	Vuln
HIKSEMI	MAGE20PRO	726	443	61.02%	7	461	63.50%	7	472	65.01%	7
TP-LINK	TL-IPC42A-4	85	76	89.41%	2	76	89.41%	2	76	89.41%	2
Cisco	IOS-XE(C)	314	244	77.71%	1	244	77.71%	1	244	77.71%	1
	IOS-XE(Isr)	136	126	92.65%	3	126	92.65%	3	126	92.65%	3
Linksys	E5600	33	33	100.00%	19	33	100.00%	19	33	100.00%	19
Netgear	EX8000	36	32	88.89%	3	32	88.89%	3	32	88.89%	3
	EX6250	34	30	88.24%	3	30	88.24%	3	30	88.24%	3
xiaomi	R3A	410	339	82.68%	3	351	85.61%	3	357	87.07%	3
GL-iNet	AR300M16	328	245	74.70%	6	256	78.05%	6	262	79.88%	6
	MT300N-V2	317	239	75.39%	6	250	78.86%	6	256	80.76%	6
	XE300	342	259	75.73%	6	270	78.95%	6	278	81.29%	6
Ruijie	EG105GW	95	87	91.58%	9	87	91.58%	9	87	91.58%	9
Total		2,856	2,153	75.39%	**68**	2,216	77.59%	**68**	2,253	78.89%	**68**

6.5 RQ4: Model Choice and Efficiency

In this experiment, to evaluate the impact of alternative models on PANGOLIN and to compare their performance, we incorporated the *gpt-4.1-nano* and *claude-3-5-haiku* models. Table 6 summarizes the comparison of the three models in terms of the number of identified parameter specifications, the number of specifications without obvious errors, and the number of discovered vulnerabilities. For each device, we conduct five runs and report the averaged outcomes. Although the models exhibit minor differences in the generated parameter specifications, the final number of identified vulnerabilities remains consistent across all models.

We also evaluate the performance of static extraction in PANGOLIN. The time cost of static extraction consists of three parts: entry map construction, MCG pruning, and parameter specification generation. Table 7 presents the details, showing that the performance primarily depends on the time consumed by MCG pruning. The average times for the three components are 13.3 seconds, 4.53 minutes, and 73.6 seconds, respectively. The results indicate that the overhead introduced by static analysis is fully acceptable for the practicality of PANGOLIN, and the high quality specifications generated provide multidimensional guidance for vulnerability discovery in multilingual IoT devices.

To facilitate a clearer understanding of PANGOLIN, we provide a detailed case study in Appendix B that illustrates the entire process, including entry map construction, reconstructed call chain analysis, parameter specification generation, and vulnerability triggering.

Table 7: **Efficiency of static extraction (RQ4).**

Brand	**Model**	**Entry.Time**	**MCG.Time**	**Gen.Time**	**Total.Time**
HIKSEMI	MAGE20PRO	52s	19.1min	230.0s	23.7min
TP-LINK	TL-IPC42A-4	5s	1.8min	49.0s	2.7min
Cisco	IOS-XE(C)	3s	3.2min	82.0s	4.6min
	IOS-XE(Isr)	2s	1.5min	40.8s	2.2min
Linksys	E5600	4s	0.4min	6.7s	0.6min
Netgear	EX8000	2s	0.7min	6.8s	0.8min
	EX6250	2s	0.7min	6.6s	0.8min
xiaomi	R3A	10s	7.3min	135.0s	9.7min
GL-iNet	AR300M16	25s	5.4min	97.2s	7.4min
	MT300N-V2	21s	5.3min	95.1s	7.2min
	XE300	27s	5.6min	105.6s	7.8min
Ruijie	EG105GW	7s	3.6min	28.5s	4.2min
Average		**13.3s**	**4.53min**	**73.6s**	**5.98min**

7 Discussion

Vulnerability Detection. Although PANGOLIN can effectively detect multiple types of vulnerabilities, its coverage still requires improvement, particularly for buffer overflow vulnerabilities. For devices whose main service runs as a single binary, detecting buffer overflows by monitoring the delay

of the main service is feasible. However, this approach does not apply to multilingual devices with binary submodules. A buffer overflow in a binary submodule does not affect the entire main service, which makes monitoring the delay of the main service ineffective. In future work, we plan to enhance our detection capabilities by monitoring multiple submodules, thereby improving our ability to identify buffer overflows within binary components.

Continuity of fuzzing Although PANGOLIN employs smart plugs to automatically reboot devices after crashes and removes APIs that may interrupt fuzzing, such as those for modifying the IP address, updating firmware, or rebooting, some user interfaces still cause instability. During fuzzing, devices often fail to reconnect properly after multiple reboots, and only a factory reset restores normal connectivity, which disrupts the continuity of fuzzing. In future work, we aim to analyze the causes of these issues, filter out additional dangerous APIs, and implement an automated factory reset mechanism to eliminate the need for manual intervention during the fuzzing process.

We also provide a detailed discussion in Appendix F on how obfuscated or encrypted firmware influences PANGOLIN and on how PANGOLIN scales to larger firmware images or more complex device ecosystems.

8 Related work

8.1 IoT Fuzzing

Gray-box Fuzzing. Many grey-box fuzzing solutions focus on emulating or rehosting IoT firmware to improve the efficiency and scalability of fuzzing as well as the capability of vulnerability detection. Firmadyne [4] and FirmAE [15] propose automated solutions for rehosting Linux-based firmware at scale. FIRM-AFL [36] integrates coverage-guided fuzzing with rehosting to uncover vulnerabilities. FirmFuzz [25] improves input validity by collecting initial test cases through crawlers. GreenHouse [26] introduces a user-space rehosting approach targeting individual firmware binaries. HouseFuzz [29] enables grey-box fuzzing across multiple interacting binaries using coverage feedback. However, for most IoT devices, especially enterprise-level ones, emulating or rehosting their firmware remains highly challenging. PANGOLIN performs fuzzing directly on physical devices, offering greater applicability and scalability.

Black-box Fuzzing. SmartTVs [1] infer input specifications from logs and collect feedback information to guide mutations. Snipuzz [11] leverages differences in responses to infer message fragments and improve mutation strategies. IoTFuzzer [5] uses mobile applications to send malformed payloads to physical devices and employs taint analysis to identify parameter types. SRFuzzer [34] gathers initial seeds through crawlers and labels parameters as Number, Fixed String, or Variable String. Labrador [20] infers execution traces from network responses and uses distance to guide seed mutations. IoTScope [31] focuses on potential unauthenticated hidden interfaces in IoT web systems. EAGLEYE [21] leverages LLMs to extract routing tokens and applies fuzzing to uncover hidden interfaces. Existing approaches in non-IoT scenarios demonstrate how LLMs can be applied to protocol and service fuzzing. ChatAFL [23] leverages LLMs to directly generate protocol grammar structures and adapts test cases through interactions, enabling guided protocol fuzzing. AutoRestTest [16] employs LLMs to learn OpenAPI [27] Specification documents and generate high-quality test cases, facilitating guided fuzzing of REST services. Overall, black-box fuzzing is constrained in its ability to explore broader code space and effectively trigger vulnerabilities due to the lack of code semantics guidance and limited handling of hidden interfaces. PANGOLIN bridges this gap.

8.2 Static Analysis in IoT Firmware

Static taint analysis has been extensively adopted for vulnerability detection in Linux-based firmware, primarily comprising two phases: source identification and taint propagation tracking. For source identification, SaTC [6] identifies sources through shared keywords between the frontend and backend. LARA [35] further enhances SaTC with LLM-driven recognition of semantic relations in code and data, uncovering more sources that were previously undetectable. FITS [22] clusters functions based on their behavioral features to identify sources. For taint propagation tracking, KARONTE [24] pioneered cross-component static dataflow analysis to detect vulnerabilities. Emtaint [8] resolves indirect calls using structured symbolic expressions to reconstruct complete execution paths. HermeScan [12] improves detection efficiency through optimized reaching dataflow analysis. LuaTaint [28] performs taint analysis on OpenWRT [10] Lua scripts. MangoDFA [13] introduces a sink-to-source strategy that prunes unreachable paths, enabling taint analysis across binaries with acceptable overhead. Despite these developments, existing static analysis tools for IoT firmware often suffer from high false positive rates, require significant manual validation, and cannot perform cross-language boundary analysis when limited to a single language, which severely restricts their practical applicability in real-world IoT devices.

9 Conclusion

In this paper, we present a new solution, PANGOLIN, which leverages an LLM agent and fuzzing to address the challenge of cross-process analysis in multilingual IoT devices and to bridge the gaps across language boundaries. At the same time, PANGOLIN conducts testing directly on physical devices, avoiding issues caused by the lack of simulation environments or the low fidelity of emulation. It extends the applicability of the tool and enables efficient and rapid vulnerability discovery

in Large devices. We evaluate PANGOLIN on 12 multilingual IoT devices from 8 globally recognized vendors. PANGOLIN uncovers 68 0-day vulnerabilities, and 31 vulnerability IDs have been assigned.

Ethical Considerations

In this paper, we analyze publicly available IoT firmware and do not involve humans, animals, private user data, environmental systems, healthcare, or military applications. Given that IoT devices are widely used in everyday life and work, if attackers were to exploit vulnerabilities to gain control of such devices, it could lead to severe financial losses and information leakage. Therefore, we believe it is necessary to promptly identify vulnerabilities in IoT devices and facilitate their timely remediation to prevent harm to a broader range of organizations and individuals.

Ethical Principles. We recognize that vulnerability research on IoT devices involves certain potential ethical concerns. In light of these possible risks, we carefully considered the ethical implications of our work and implemented mitigation strategies throughout the research process. Our research adhered to the principles outlined in the Menlo Report [3], including Beneficence, Respect for Persons, Justice, and Respect for Law and Public Interest. Below, we evaluate the primary stakeholders and detail how each principle was implemented throughout the research process.

Stakeholders. We identify the following stakeholders: IoT end users, device manufacturers, the broader security community, potential adversaries, and researchers. During the research process, we maintained respect for individuals, avoided using discovered vulnerabilities to interfere with end users, and did not disclose any private data from manufacturers. We also complied with relevant laws and upheld the public interest. No testing was conducted on affected targets in public environments. We ensured fairness in constructing the dataset and did not target any specific manufacturers. Our work aims to proactively identify potential vulnerabilities in IoT devices to assist manufacturers in remediation and to maintain the overall security of the IoT ecosystem.

Potential Impacts. Our findings contribute to improving IoT security by enabling manufacturers to patch vulnerabilities promptly, allowing end users who update their firmware in a timely manner to avoid potential attacks. Researchers can build upon our work to further enhance community-wide security. However, improper handling of vulnerability data could cause real-world harm, including financial loss, privacy breaches, or facilitating attacks. Automated vulnerability discovery also raises dual-use concerns: while researchers and manufacturers can leverage our work to secure IoT devices, adversaries could potentially exploit it to harm end users.

Mitigation Measures. We follow a strict responsible disclosure workflow: all vulnerabilities are first reported to manufacturers and only disclosed to CVE or CNVD after patches are applied. We intentionally avoid publishing proof-of-concept exploits, payloads, or sensitive technical details that could substantially lower the barrier for malicious use. End users who update their firmware promptly are protected from potential attacks. Residual dual-use risks cannot be completely eliminated, but the defensive benefits outweigh the remaining risks.

Decision process to publish. The goal of this research is to effectively identify vulnerabilities in IoT devices and to encourage manufacturers to remediate them promptly. After carefully weighing the potential ethical harms against the benefits, we decided to conduct this study with the aim of protecting individual interests and enhancing system security. We chose to publish our findings because they provide actionable defensive value to manufacturers and the research community while avoiding details that could pose operational risks. We determined that the benefits of responsible disclosure and publication outweigh the residual risks. We confirm that we have complied with all ethical guidelines outlined in the CFP and ensure that our research and disclosure decisions were made with full consideration of all stakeholders, potential impacts, and the assumptions underlying our findings. While in some cases the most ethical course of action might be to refrain from conducting or publishing the research, in this instance the application of ethical principles supports both conducting the study and sharing its results.

Open Science

The source code and dataset of this work are available at: https://doi.org/10.6084/m9.figshare.30904379.

Acknowledgment

We would like to thank all the reviewers for their insightful suggestions that helped us to improve this paper. We also thank Shepherd and Ethics Shepherd for their tireless efforts in accepting our paper. We also appreciate the valuable comments and recognition from the artifact evaluation reviewers. This work is supported by the National Natural Science Foundation of China under grant U24A20337, and Joint Research Center for System Security (JCSS), Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

References

[1] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. Android smarttvs vulnerability discovery via log-guided fuzzing. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2759–2776. USENIX Association, 2021.

[2] Android Open Source Project. *monkeyrunner*, 2024.

[3] Michael Bailey, David Dittrich, Erin Kenneally, and Doug Maughan. The menlo report. *IEEE Security & Privacy*, 10(2):71–75, 2012.

[4] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, volume 1, pages 1–1, 2016.

[5] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Network and Distributed System Security Symposium*, pages 1–16, 2018.

[6] 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 (USENIX Security 21)*, pages 303–319, 2021.

[7] 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. ISSTA 2023, page 360–372, 2023.

[8] 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 *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 360–372, 2023.

[9] Cisco Community. Rest api basics, 2020. https://community.cisco.com/t5/crosswork-automation-hub-knowledge-articles/rest-api-basics/ta-p/3635342.

[10] Florian Fainelli. The openwrt embedded development framework. In *Proceedings of the Free and Open Source Software Developers European Meeting*, volume 106, 2008.

[11] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minghui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2021.

[12] Zicong Gao, Chao Zhang, Hangtian Liu, Wenhou Sun, Zhizhuo Tang, Liehui Jiang, Jianjun Chen, and Yong Xie. Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis. In *NDSS*, 2024.

[13] Wil Gibbs and Raj. Operation mango: Scalable discovery of taint-style vulnerabilities in binary firmware services. In *USENIX Security Symposium*, pages 312–326, 2024. https://www.usenix.org/system/files/sec24fall-prepub-1634-gibbs.pdf.

[14] Huawei Technologies Co., Ltd. Wireless access controller (ac and fit ap) v200r023c00 cli-based configuration guide. Online, 2024.

[15] 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 *Proceedings of the 36th Annual Computer Security Applications Conference*, pages 733–745, 2020.

[16] Myeongsoo Kim, Tyler Stennett, Saurabh Sinha, and Alessandro Orso. A multi-agent approach for rest api testing with semantic graphs and llm-driven inputs. *arXiv preprint arXiv:2411.07098*, 2024.

[17] ReFirm Labs. Binwalk: Firmware analysis tool, 2021. https://github.com/ReFirmLabs/binwalk.

[18] Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen, Qinchen Gu, Chunming Wu, and Raheem Beyah. V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. *IEEE transactions on cybernetics*, 52(5):3745–3756, 2020.

[19] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[20] Hangtian Liu, Shuitao Gan, Chao Zhang, Zicong Gao, Hongqi Zhang, Xiangzhi Wang, and Guangming Gao. Labrador: Response guided directed fuzzing for black-box iot devices. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 127–127. IEEE Computer Society, 2024.

[21] Hangtian Liu, Lei Zheng, Shuitao Gan, Chao Zhang, Zicong Gao, Hongqi Zhang, Yishun Zeng, Zhiyuan Jiang, and Jiahai Yang. EAGLEYE: exposing hidden web interfaces in iot devices via routing analysis. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[22] Puzhuo Liu, Yaowen Zheng, Chengnian Sun, Chuan Qin, Dongliang Fang, Mingdong Liu, and Limin Sun. Fits: Inferring intermediate taint sources for effective

vulnerability analysis of iot device firmware. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 138–152, 2023.

[23] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, volume 2024, 2024.

[24] 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 *IEEE Symposium on Security and Privacy*, pages 1544–1561. IEEE, 2020.

[25] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias 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*, pages 15–21, 2019.

[26] Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, et al. Greenhouse:{Single-Service} rehosting of {Linux-Based} firmware binaries in {User-Space} emulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5791–5808, 2023.

[27] Aimilios Tzavaras, Nikolaos Mainas, and Euripides GM Petrakis. Openapi framework for the web of things. *Internet of Things*, 21:100675, 2023.

[28] Jiahui Xiang, Lirong Fu, Tong Ye, Peiyu Liu, Huan Le, Liming Zhu, and Wenhai Wang. Luataint: A static analysis system for web configuration interface vulnerability of internet of things devices. *IEEE Internet of Things Journal*, 12(5):5970–5984, 2025.

[29] Haoyu Xiao, Ziqi Wei, Jiarun Dai, Bowen Li, Yuan Zhang, and Min Yang. HouseFuzz: Service-Aware Grey-Box Fuzzing for Vulnerability Detection in Linux-Based Firmware . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3801–3819, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[30] Xiaomi Inc. *Xiaomi IoT Developer Platform: Control Application Development*. Xiaomi Inc., December 2023. Document: Control Application Development.

[31] 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 *Proceedings of the ACM Web Conference 2022*, pages 524–532, 2022.

[32] Xiaokang Yin, Ruijie Cai, Xiaoya Zhu, Qichao Yang, Enzhou Song, and Shengli Liu. Precise discovery of more taint-style vulnerabilities in embedded firmware. *IEEE Transactions on Dependable and Secure Computing*, 22(2):1365–1382, 2025.

[33] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Longquan Liu, Yanyan Zou, Chao Zhang, and Baoxu Liu. Esrfuzzer: an enhanced fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities. *Cybersecur.*, 4(1):24, 2021.

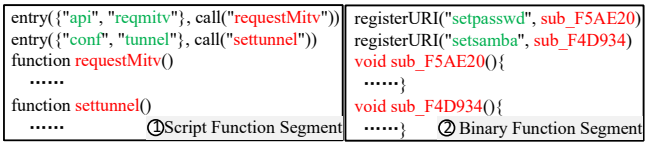
[34] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. Srfuzzer: An automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 544–556, 2019.

[35] Jiaxu Zhao, Yuekang Li, Yanyan Zou, Zhaohui Liang, Yang Xiao, Yeting Li, Bingwei Peng, Nanyu Zhong, Xinyi Wang, Wei Wang, et al. Leveraging semantic relations in code and data to enhance taint analysis of embedded systems. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7067–7084, 2024.

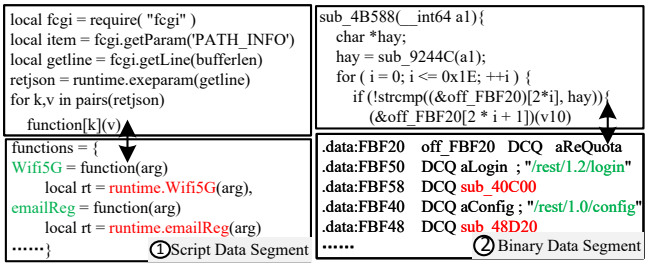
[36] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium*, pages 1099–1114, 2019.

A Entry uri dispatch pattern

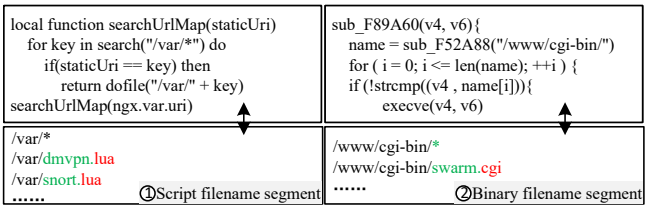
Figure 9 illustrates three modes of entry URIs dispatch in multilingual IoT devices: function dispatch, data dispatch, and filename dispatch. To provide a clear and intuitive explanation, we mark URIs in green and handler points in red within the figure. Figure 9a illustrates the correspondence between entry URIs and handler points established through custom registration functions, with both elements passed as function parameters. For example, accessing */conf/tunnel* invokes the handler point *settunnel*. Figure 9b illustrates the correspondence between URIs and handler points established through data association. For script-based implementations, this relationship is typically stored in a dictionary, with URIs serving as keys and handler points as values. For binary-based implementations, the mapping is usually maintained in contiguous data segments, with the correspondence explicitly defined in the code. For example, the entry */rest/1.0/config*



(a) Function dispatch pattern



(b) Data dispatch pattern



(c) Filename dispatch pattern

Figure 9: **Entry URI dispatch patterns**

maps to the handler point *sub_48D20*. Figure 9c illustrates the filename-based dispatch mechanism. In this approach, entry URIs are typically not displayed in either the code segment or the data segment. Instead, programs are dynamically loaded by indexing file names within a folder that match the code semantics. In this case, both the entry URI and the handler point correspond to the file name. While they may not encompass all potential real-world scenarios, PANGOLIN remains extensible to accommodate new mechanisms.

B Key Study

We present one simplified typical case to illustrate the complete process of PANGOLIN as follows. As shown in Figure 10, PANGOLIN locates several dispatch functions such as *UriHandler* and *confsys* from interface information extracted from network traffic and constructs their call relationships to obtain the entry map. Within this map, the `/rest/confsys?action=change_sys` interface appears as a hidden interface, and its corresponding handler point is *ChangeSysConf*. Starting from *ChangeSysConf*, LLMs analyze the call chain and identify the parameter extraction, execution, and sanitization functions, including the main, sub_02, sub_07, is_valid_ip, sub_09 and sub_16 of the */sbin/sys*. Then, PANGOLIN generates the corresponding parameter specifications and applies corrections when the response indicates the need for adjustment. Finally, PANGOLIN generates test cases based on the parameter specifications, as shown in Figure 11. Since the precise specifications capture the code semantics and successfully bypass the sanitizers, and the labels prevent redundant ineffective mutations during fuzzing, the payload ultimately reaches the sink point and triggers the vulnerability.

C Vulnerability Identifiers

Table 8: Vulnerabilities discovered by PANGOLIN

Brand	Model	Type	Vuln	IDs	CVE/CNVD
HIKSEMI	MAGE20PRO	CI	5	2	**CNVD-2025-14195**
		Dos	1		**CNVD-2025-14455**
		leak	1		
TP-LINK	TL-IPC42A-4	CI	2	0	
cisco	IOS-XE(C)	AFR	1	0	
	IOS-XE(ISR)	AFU	1	0	
		CI	2	1	**CNVD-2025-18467**
Linksys	E5600	CI	15	11	**CVE-2025-29223**
					CVE-2025-(29226∼29231)
		XSS	4	1	**CVE-2025-(45487∼45491)**
Netgear	EX8000	CI	3	3	**CVE-2025-(45492∼45493)**
					CVE-2025-50526
	EX6250	CI	3	3	**CNVD-2025-(09131∼09132)**
					CNVD-2025-12271
xiaomi	R3A	CI	3	1	**CNVD-2025-13974**
GL-iNet	AR300M16	CI	6	6	**CNVD-2025-14616**
					CNVD-2025-19408
					CNVD-2025-19973
					CNVD-2025-(20412∼20414)
	MT300N-V2	CI	6	0	
	XE300	CI	6	0	
Ruijie	EG105GW	CI	9	3	**CNVD-2025-05920**
					CNVD-2025-08689
					CNVD-2025-09549
Total			**68**	**31**	

D Sanitizers and Payload Generation

Although we obtained the pruned MCG that contains sanitizers along with the corresponding code snippets, our tests show that directly feeding the code into the LLM without additional constraints leads to poor vulnerability detection. The model often misidentifies bypassable sanitizers as non-bypassable and misidentifies non-bypassable sanitizers as bypassable. For some bypassable sanitizers, the model can recognize the presence of a vulnerability but fails to generate payloads that can effectively bypass the sanitizers. We define a set of strict and non-bypassable sanitizers to support LLM-based analysis. Using command injection as an example:

1. When a parameter is enclosed by single quotes, the parameter is treated as a pure string that cannot be inter-

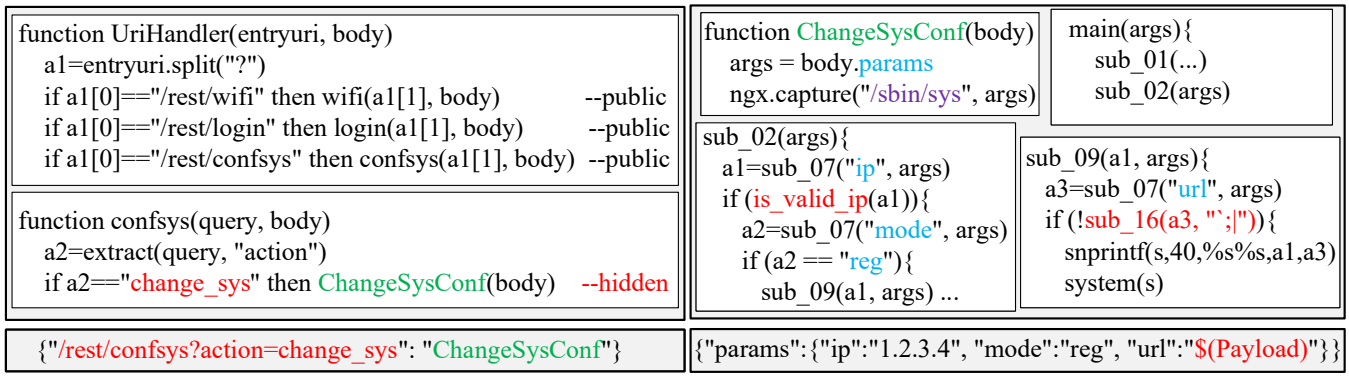


Figure 10: **An example of the complete process, including entry map construction, reconstructed call chain analysis, parameter specification generation, and vulnerability triggering.**

```
POST /rest/confsys?action=change_sys
Header{...}
{"params":{"ip":"1.2.3.4", "mode":"reg", "url":"$(Payload)"}}
{"params":{"ip":"NoMuta", "mode":"NoMuta", "url":"CI"}}
```

Figure 11: **Test case with label and payload.**

preted as a command, and every single quote inside it is filtered or escaped to prevent premature closure.

2. When a parameter is enclosed by double quotes, special characters such as `;$'&"\` are filtered.

3. Commands or characters are restricted through whitelists that permit only specific strings.

4. The parameter is subject to restrictions on length, type, and format regex matching.

At the same time, we rely on the security principles of these strict sanitizers to craft methods that circumvent unsafe sanitizers, which enables the LLM to generate payloads that bypass multiple sanitization mechanisms:

1. Constructing a payload that contains single quotes triggers premature closure of the surrounding quotes.

2. Using special characters such as `;$'&"\` enables bypass during double quote enclosure, and using `${IFS}` bypasses whitespace checks.

LLM-generated results contain both false positives and false negatives. Fuzzing eliminates vulnerabilities that cannot be triggered, and with high quality parameter specifications, allows a limited number of mutations to mitigate false negatives produced by the LLM.

E Dependencies Among Modules

The entry map construction module provides handler points for the MCG module and supplies the sequence identification module with a maximally comprehensive set of backend URIs. Removing this module will reduce the handler points processed by the MCG, limit the completeness of URI Sequence Identification, and decrease the number of generated parameter specifications. The MCG serves as the medium for feedback regulation. Removing this module will restrict feedback to regulating parameter specifications for handler point code only, and it will be unable to process subsequent multilingual calls with hierarchical structures. Sequence Identification orders entry URIs based on parameter specifications. Removing this module will cause missed vulnerabilities that require multiple requests to trigger. The feedback module regulates entry URI extraction, MCG pruning, and parameter specification generation. Removing this module will prevent the removal of invalid entry URIs, hinder the correction of pruning errors introduced by incorrect LLM recognition, and leave errors in parameter specifications caused by LLM hallucinations unresolved.

F Scalability in complex firmware and the impacts of obfuscated/encrypted firmware

The main challenges in testing larger firmware images and complex device ecosystems lie in the complexity of their dispatch mechanisms and the substantial code size. However, these systems still expose Web services and rely on protocols such as HTTP or HTTPS for management and configuration. `PANGOLIN` locates backend dispatch functions through entry URI information from public interfaces, constructs call relationships for multi-level dispatch logic, recognizes diverse dispatch patterns that carry semantic information with the help of the LLM, and generates semantically valid test cases through a pruned MCG with feedback-driven refinement. These capa-

bilities enable effective testing and provide strong scalability and applicability for larger firmware images and more complex device ecosystems.

For obfuscated firmware, deobfuscation is required. Obfuscation appears frequently in software but occurs infrequently in IoT contexts. For encrypted firmware, analysts can locate encryption and decryption algorithms and keys in critical firmware versions or in firmware extracted directly from storage chips, and then decrypt the encrypted content. Analysts can also extract filesystem images from memory during normal device operation to recover the unencrypted firmware.