# FirmAgent: Leveraging Fuzzing to Assist LLM Agents with IoT Firmware Vulnerability Discovery

Jiangan Ji*†, Chao Zhang†‡✉, Shuitao Gan§, Lin Jian*, Hangtian Liu*,
Tieming Liu*✉, Lei Zheng†, Zhipeng Jia*†
*Information Engineering University, †Institute for Network Sciences and Cyberspace, Tsinghua University,
‡JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.
§Laboratory for Advanced Computing and Intelligence Engineering

***Abstract*—The rapid proliferation of IoT devices has introduced substantial security vulnerabilities. Existing vulnerability detection techniques exhibit various weaknesses: static analysis solutions (including large language models, LLMs) suffer from high false positives and provide no PoC (proof-of-concept) samples, while dynamic analysis solutions (e.g., fuzzing) often have high false negatives. To address these challenges, we present `FirmAgent`, the first hybrid solution that leverages fuzzing to assist LLM agents in finding vulnerabilities in IoT firmware. Our design is motivated by the key observation that fuzzing can accurately identify input-related code points in firmware, while static analysis can thoroughly analyze program paths starting from those code points. `FirmAgent` utilizes fuzzing to collect runtime input points (i.e., taint sources) and reconstruct potential vulnerability paths. Then, it applies an LLM agent to perform context-aware taint analysis along the potential paths and another LLM agent to refine the fuzzing-generated testcase to generate PoC testcases. We evaluate `FirmAgent` on 14 real-world IoT firmware. It identifies 182 vulnerabilities with a precision of 91%, including 140 previously unknown vulnerabilities, 17 of which have been assigned CVE numbers. Our results demonstrate that `FirmAgent` substantially outperforms SOTA tools in both detection capability and precision.**

## I. Introduction

The proliferation of Internet of Things (IoT) devices has transformed modern life, enabling unprecedented levels of automation, connectivity, and convenience. From smart home appliances and industrial control systems to networked medical equipment, IoT devices now permeate nearly every aspect of society. However, this rapid adoption has also raised significant security concerns [1]. These devices often run on lightweight, customized firmware and are deployed in resource-constrained environments, frequently lacking comprehensive security mechanisms. Among them, IoT devices that expose web services are particularly vulnerable, as these services, while offering convenient interfaces for control and configuration, also significantly expand the attack surface. Notably, many of these devices run Linux-based firmware, making the security analysis of Linux-based IoT firmware a critical area of research.

To detect vulnerabilities in IoT firmware, researchers have developed both dynamic and static analysis approaches. Dynamic analysis, particularly fuzzing (including FirmAFL [2], SNIPUZZ [3], FirmFuzz [4], and Greenhouse [5]), has demonstrated strong capabilities in discovering exploitable vulnerabilities by automatically generating and executing test inputs. However, it suffers from limited code coverage, especially in IoT scenarios, where most conditional branches are closely tied to specific input values. As a result, many fuzzing tools leave large portions of the codebase untested. In contrast, static analysis (including SaTC [6], Emtaint [7], HermeScan [8], and OctopusTaint [9] offers broader coverage by examining program code without execution. Techniques such as symbolic execution and taint analysis can reveal latent vulnerabilities but often suffer from high false positive rates due to challenges such as inaccurate identification of source points, imprecise alias analysis, and a lack of semantic understanding.

To leverage the complementary strengths of both approaches, researchers have explored hybrid techniques, such as hybrid fuzzing (including SAVIOR [10], Driller [11], QSYM [12], and HyLLfuzz [13]), which aim to combine dynamic fuzzing with symbolic execution or LLM to bypass complex conditional checks and reach deeper execution paths. In the IoT domain, hybrid fuzzers such as FirmHybridFuzzer [14] and RSFuzzer [15] have been proposed for MCU and SMI processors, respectively. However, these approaches often suffer from significant overhead due to the need to record execution paths and solve constraints during fuzzing. Moreover, hybrid fuzzing for Linux-based IoT firmware remains particularly challenging due to the difficulty of collecting precise runtime constraints in emulated environments and using them to guide seed mutation.

In order to find a new hybrid solution, we have conducted an empirical study to understand the strengths of both techniques in the Linux-based IoT firmware. Our findings indicate that *fuzzing is highly effective at covering a wide range of input source points within firmware*. However, due to strict parameter checks and complex control logic, many security-sensitive

✉Corresponding authors: chaoz@tsinghua.edu.cn, fxliutm@163.com.

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231943
www.ndss-symposium.org

functions remain unreachable, resulting in numerous false negatives. Conversely, static analysis can bypass input constraints and perform thorough data-flow tracking from sources to sinks. The source is where user inputs are introduced, and the sink is where potentially risky operations occur. Notably, prior studies such as LATTE [16] and IRIS [17] have shown that leveraging LLMs for taint propagation enables significantly improved precision and recall compared to traditional static taint analysis techniques. However, identifying source points via static analysis often has high false positives, which may flag benign flows as vulnerable. Therefore, we propose *utilize fuzzing to accurately identify code points that accept external input and then initiate LLM-based taint analysis from these points to detect potential vulnerabilities*.

To implement this strategy, we face three key challenges: *C1: Limited Code Coverage.* In IoT firmware, critical web service logic is often implemented in individual service handler functions that are only reachable via specific URIs. Without knowledge of these URIs, fuzzing cannot effectively reach them. *C2: Source Points Identification's Accuracy and Efficiency.* Monitoring taint sources across the entire program address space during fuzzing can result in numerous redundant sources and significantly degrade performance. *C3: Static Taint Analysis' Precision and Verification Overhead.* Even with accurate sources, traditional taint analysis suffers from aliasing issues and limited capability of understanding code semantics, leading to high false positives. Moreover, it also cannot generate PoC for reported potential vulnerabilities (alert), resulting in a heavy manual verification burden.

In this paper, we propose `FirmAgent`, a novel hybrid solution specifically designed for vulnerability detection in IoT firmware. Unlike LLM-based hybrid fuzzers [13] that invoke LLMs during fuzzing to generate inputs satisfying complex constraints, `FirmAgent` leverages dynamic information collected during fuzzing to assist LLMs in reasoning over code and identifying potential vulnerabilities. This design eliminates the overhead of frequent LLM invocations during fuzzing and mitigates crash-induced disruptions commonly encountered in IoT fuzzing scenarios. `FirmAgent` requires that the target firmware can be rehosted using a single-service rehosting framework [5], ensuring that the network service of interest can be successfully launched and interacted. To overcome the difficulty in reaching diverse service handler functions in firmware and increase the code coverage of fuzzing, `FirmAgent` begins with a pre-fuzzing analysis to extract service handler functions, keywords, and distance metrics between sink points and basic blocks, which guides the fuzzing process to maximize coverage of potential source points. To address the inefficiencies associated with taint source identification during fuzzing, we implement a lightweight memory-based detection mechanism using QEMU [18], which enables efficiently identifying external inputs and dynamically completing call graphs. Thereby, it facilitates the generation of accurate and complete potential vulnerability paths. Given these source points and potential paths, we incorporate two LLM agents: the taint propagation agent module, which performs precise taint propagation analysis, and the PoC generation agent module, which automates the PoC generation. Together, these components validate the existence of vulnerabilities, significantly reducing the manual effort traditionally required for vulnerability verification.

To evaluate the effectiveness of `FirmAgent`, we conduct experiments on 14 real-world IoT firmware samples. We compare our tool with SOTA static analysis tools (HermeScan [8] and Emtaint [7]), SOTA dynamic analysis tool (Greenhouse [5]), and SOTA hybrid analysis tool (Hy-FirmFuzz). `FirmAgent` successfully identified 182 vulnerabilities with a precision of 91%, including 45 command injection and 137 buffer overflow vulnerabilities. Of these, 27 and 113 were previously unknown, respectively, and 17 have already been assigned CVE identifiers. In contrast, Emtaint detected only 10 vulnerabilities with a precision of 37%, HermeScan identified 71 with 33% precision, Greenhouse found 8 with 40% precision, and Hy-FirmFuzz detected 13 with 100% precision. To assess the contribution of individual components within `FirmAgent`, we conducted an ablation study. The results show that several key modules, such as source point identification, taint propagation agent, indirect call tracking, and directed fuzzing strategy, significantly enhance the overall detection effectiveness. We also evaluated the quality of the automatically generated PoCs. Results show that 91.8% of the PoCs were directly valid, confirming the practical value of our LLM-guided PoC generation process.

In summary, this paper makes the following contributions:

- We conduct a systematic analysis of the strengths and limitations of existing static and dynamic analysis techniques for IoT vulnerability detection. Based on this, we propose `FirmAgent`[1], the first hybrid solution that leverages fuzzing to assist LLM agents in discovering vulnerabilities by combining the strengths of both static and dynamic approaches.
- We design a lightweight fuzzing framework capable of accurately identifying source points within firmware execution. Building on this, we integrate two novel components, a taint propagation agent and a PoC generation agent, to enable precise vulnerability detection and automatic PoC generation.
- We perform evaluations on real-world IoT firmware samples. The results demonstrate that `FirmAgent` significantly outperforms SOTA in detection effectiveness, identifying 140 0-day vulnerabilities.

## II. Background and Motivation

### A. IoT Vulnerability Discovery Techniques

Current research efforts in IoT vulnerability discovery predominantly adopt either static or dynamic analysis techniques. To provide a comprehensive overview of the strengths and weaknesses of representative vulnerability detection tools, Table I compares existing approaches across three critical

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

TABLE I: **Comparison of SOTA Tools Based on Source Identification Method, Taint Propagation Methodology, and PoC Generation.** *DSE* represents Dynamic Symbolic Execution, *SSE* represents Structured Symbolic Expression, *AST* represents Abstract Syntax Tree, and *RDA* represents Reaching Definitions Analysis.

| Tool | Source Identification | Taint Propagation | PoC |
|---|---|---|---|
| **SaTC [6]** | Shared Keyword | DSE | × |
| **EmTaint [7]** | Predefined | SSE | × |
| **HermeScan [8]** | Shared Keyword + Filter | RDA | × |
| **Mango [19]** | Shared Keyword + Env | RDA | × |
| **Lara [20]** | Pattern + LLM | AST | × |
| **OctopusTaint [9]** | Shared Keyword + Env | RDA | × |
| **Greenhouse [5]** | × | × | Fuzzing |
| **FirmAgent** | Fuzzing | LLM | Fuzzing + LLM |

dimensions: source identification strategy, taint propagation methodology, and PoC generation.

In terms of source identification, most tools (including SaTC [6], Mango [19], HermeScan [8], and OctopusTaint [9]) rely on shared keyword matching combined with heuristic enhancements such as environmental modeling or filtering techniques. EmTaint uses a predefined list of source functions that offers precision but often overlooks many actual sources. Lara [20] employs a pattern-based approach augmented with LLMs. However, its lack of open source availability makes it difficult to evaluate its practical effectiveness. Overall, frontend-backend string matching remains the most widely adopted and empirically effective method for source identification in static analysis.

For taint propagation, traditional static techniques such as symbolic execution [6], structured symbolic expression-based on-demand alias analysis [7], reaching definitions analysis [8], and abstract syntax tree [20] traversal are commonly employed. Despite ongoing efforts to enhance these methods and reduce false positives, the absence of semantic understanding in traditional approaches limits their ability to accurately handle context-sensitive constructs such as sanitization functions, leading to persistent inaccuracies.

Regarding PoC generation, current static analysis tools face fundamental limitations, as they can identify potential vulnerability locations but lack the capability to automatically validate them. This often necessitates substantial manual effort to assess each reported alert, significantly increasing the time and resources required for vulnerability triage. In contrast, dynamic analysis, particularly fuzzing, is uniquely capable of simultaneously triggering vulnerabilities and producing actionable PoCs. Our proposed tool, `FirmAgent`, bridges this gap by combining fuzzing-generated test cases with LLM-guided parameter analysis to automatically construct complete PoCs. This approach addresses the high false-negative rate of dynamic tools and the manual verification burden of static tools. In the following sections, we analyze the key limitations of current static and dynamic vulnerability detection techniques in greater detail.

TABLE II: **Accuracy of Static Source Function Identification in Firmware Analysis.** SF = source function, C-SF = candidate source functions identified by HermeScan. CC and C-CC denote the actual and candidate vulnerability call chains, respectively.

| Vendor | Firmware | C-SF | SF | SF.ratio | C-CC | CC | CC.ratio |
|---|---|---|---|---|---|---|---|
| Trendnet | TEW800mb | 5 | 1 | 20.0% | 101 | 14 | 13.9% |
| | TEW_632BRPA1 | 6 | 1 | 16.7% | 2 | 2 | 100.0% |
| ASUS | FW_RT_G32 | 4 | 1 | 25.0% | 4 | 3 | 75.0% |
| Dlink | DIR_825 | 8 | 1 | 12.5% | 13 | 4 | 30.8% |
| Netgear | DC112A | 8 | 2 | 25.0% | 180 | 62 | 34.4% |
| Linksys | FW_WRT54Gv4 | 6 | 2 | 33.3% | 23 | 14 | 60.9% |
| **Average** | | 6.2 | 1.3 | **21.6%** | 53.8 | 16.5 | **30.7%** |

### *B. Limitations of Existing Approaches*

A key limitation of static analysis is its inability to accurately identify source points, leading to false positives. Existing static analysis tools typically employ shared keyword matching to determine candidate source functions. Functions that frequently contain certain predefined keywords are heuristically marked as source points, and all of their invocations are subsequently treated as taint entry points. However, our empirical evaluation on real-world firmware samples reveals a substantial discrepancy between these candidate functions and actual source functions. We selected HermeScan, the most accurate tool for identifying source points currently available, for our experimental evaluation. As shown in Table II, only 21.6% of the C-SF are true SF that genuinely accept external inputs. The remaining candidate sources result in the construction of C-CC, with an estimated 70% of such call chains being false positives. This imprecision severely undermines the reliability of static analysis results and limits its effectiveness in guiding downstream vulnerability detection efforts.

Moreover, static analysis suffers from false negatives due to indirect calls. Figure 1 demonstrates how indirect calls in static analysis can lead to taint loss. In this vulnerability, the first keyword of the call site contains both a URL and user-controllable input. When the URL matches *NTPSyncWithHost.cgi*, execution enters function *sub_F934*, which processes user-controllable content after the "?" character through string concatenation. This ultimately leads to a command execution vulnerability in the system function call for time configuration. However, since the vulnerability resides within a function pointer target function, SOTA static analysis tools struggle to reconstruct the calling relationships accurately. The target function is treated as an isolated node, disconnected from its predecessors, which causes the loss of taint relationships between the call site and the target function, ultimately resulting in a missed vulnerability.

On the other hand, dynamic analysis, particularly fuzzing, is often constrained by its limited ability to bypass complex conditional checks, thereby leaving a substantial portion of the codebase unexplored. This challenge becomes even more pronounced in IoT firmware, where numerous conditional branches rely on hardware states or configuration values, making it extremely difficult for fuzzing mutations alone to satisfy

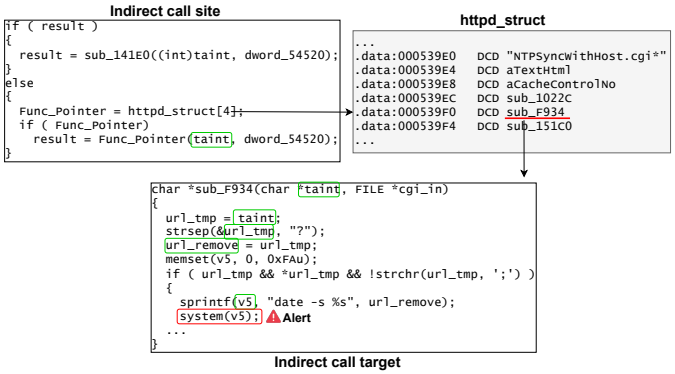


Fig. 1: **An Example of an Indirect Call in a Trendnet Router.** Static analysis fails to detect vulnerabilities due to taint propagation through parameters of the indirect call.

TABLE III: **Reachability of Source and Sink Points in Potential Vulnerability Paths via Fuzzing.** PP denotes the number of potential vulnerability paths identified, and R- counts those verified as reachable via fuzzing.

| Vendor | Firmware | R-Source | Source | Source.ratio | R-sink | Sink | Sink.ratio | R-PP | PP | PP.ratio |
|---|---|---|---|---|---|---|---|---|---|---|
| Trendnet | TEW800mb | 58 | 71 | 81.7% | 3 | 11 | 27.3% | 3 | 11 | 27.3% |
| | TEW_632BRPA1 | 11 | 11 | 100.0% | 3 | 5 | 60.0% | 3 | 5 | 60.0% |
| ASUS | FW_RT_G32 | 27 | 34 | 79.4% | 0 | 3 | 0 | 0 | 3 | 0 |
| Dlink | DIR_825 | 11 | 11 | 100.0% | 2 | 6 | 33.3% | 2 | 4 | 50.0% |
| Netgear | DC112A | 95 | 101 | 94.1% | 4 | 24 | 16.7% | 4 | 20 | 20.0% |
| Linksys | FW_WRT54Gv4 | 32 | 33 | 97.0% | 2 | 7 | 28.6% | 2 | 7 | 28.6% |
| **Average** | | 39.0 | 43.5 | **89.7%** | 2.3 | 9.3 | **25.0%** | 2.3 | 8.3 | **28.0%** |

all the required execution conditions. As shown in Table III, current fuzzing techniques can reach only approximately 25% of the sink points, which implies that a considerable number of potential vulnerabilities remain undetected, ultimately leading to persistently high false-negative rates.

Figure 2 illustrates how dynamic analysis can fail due to configuration dependent conditions. In this case, the vulnerability is triggered through user-controlled data via the *deviceName* keyword, but execution requires specific nvram values: either *enable_ap_mode* or *enable_sta_mode* must be set to *1*. This prerequisite configuration makes the command execution vulnerability challenging to trigger. Current state-of-the-art firmware fuzzing tools lack semantic understanding of these conditions, and mutation-based approaches struggle to satisfy such constraints, leading to missed vulnerabilities.

```
1. ...
2. websGetvar(a1, "deviceName", Name, 2048);
3. if ( acosNvramConfig_match("enable_ap_mode", "1")
4. || acosNvramConfig_match("enable_sta_mode", "1") )
5. {
6.   system("/sbin/rmmod br_dns_hijack");
7.   snprintf(v79, 0x80u,"/sbin/insmod...=%s",Name);
8.   system((const char *)v79); Alert
9. }
```

Fig. 2: **Decompiled Code of the `httpd` Binary from the DC112A Device.** Specific configuration checks hinder fuzzing from reaching the `system` function.

### C. Observation and Insights

While fuzzing struggles to reach sink points, this limitation arises primarily from input constraints imposed along the execution paths that lead to these sinks. Many of these constraints are difficult to bypass solely through fuzzing. In contrast, we observe that source points are typically located along shallow execution paths within handler functions. Consequently, fuzzing achieves high source coverage, as fewer conditional branches must be satisfied to reach these points.

To validate this observation, we conducted an empirical study across several firmware samples from different vendors. Specifically, we employed the fuzzing module of `FirmAgent` to dynamically assess the reachability of source and sink points within potential vulnerability paths. As shown in Table III, fuzzing was able to reach approximately 90% of source points on average, whereas only 25% of sink points were covered. This significant disparity in reachability suggests that a large portion of potential vulnerabilities may be missed when relying solely on traditional fuzzing techniques. In contrast, static taint analysis provides a more comprehensive exploration of potential paths, effectively mitigating the issue of false negatives. For instance, in the vulnerability illustrated in Figure 1, fuzzing was able to identify *url_tmp* as a tainted variable in *sub_F934*, effectively resolving the issue of missing taint propagation caused by indirect calls in static analysis. However, due to the specific input format in subsequent logic, existing fuzzers could not trigger the vulnerability. Similarly, in the case shown in Figure 2, fuzzing propagated the taint to *Name* (line 2) but failed to satisfy specific parameter constraints necessary to exploit the vulnerability. By leveraging LLM-based taint analysis starting from these tainted point, we were able to successfully identify the vulnerability.

These observations motivate an intuitive strategy: leverage fuzzing to identify program locations that accept external inputs and designate them as starting points (*Csource*) for subsequent static taint analysis. This hybrid approach enables a more thorough exploration of potential data flow paths and improves both the coverage and precision of vulnerability detection, thereby reducing false negatives and false positives.

## III. Challenges

Despite the advantages of combining fuzzing and static analysis for IoT vulnerability detection, significant challenges persist when applying these techniques to real-world firmware.

### A. Limited Code Coverage

In IoT firmware, numerous source points are embedded within various service handler functions. If the corresponding URIs and parameters are not provided during fuzzing, these service handlers may never be invoked, preventing necessary variable modifications and causing critical source points to be overlooked. Therefore, we need to refine our fuzzing strategy to systematically explore these code regions.

### B. Accurate and Efficient Identification of Source Points

Throughout the execution from entry points to sink points, numerous variables may receive external input. However, if all tainted variables are recorded during fuzzing, the number of identified source points will be excessively large, leading to redundant paths and an increased computational burden. Moreover, continuously tracking inputs from the initial fuzzing stage imposes significant overhead, potentially degrading fuzzing efficiency. Therefore, designing a precise and computationally efficient method for source point identification remains a critical challenge.

### C. Accuracy of Static Taint Propagation Analysis and Manual Verification Overhead

Once the taint analysis starting points are identified, ensuring accurate taint propagation is crucial. Traditional taint analysis engines often suffer from pointer aliasing issues, difficulty in recognizing sanitization mechanisms, and other limitations that contribute to both false positives and false negatives. Furthermore, conventional approaches typically mark a potential attackable sink address as an alert, requiring manual intervention to analyze the execution paths and construct a PoC. This process is not only time-consuming but also requires considerable expertise, posing a significant barrier to efficient vulnerability validation.

## IV. Methodology

### A. Overview

We present `FirmAgent`, a fuzzing-assisted taint analysis framework designed for efficient and precise vulnerability detection in IoT firmware. As illustrated in Figure 3, the framework operates in two main phases: *Fuzzing-driven Information Collection* and *Taint-to-PoC Agent*. This design effectively leverages the taint authenticity verification and test case reachability obtained during dynamic analysis to enhance the subsequent taint propagation analysis accuracy of LLMs.

*1) Fuzzing-Driven Information Collection:* This phase combines lightweight static preprocessing with runtime instrumentation to monitor taint behavior and program execution flows. It provides precise source points and comprehensive call graphs for subsequent taint propagation analysis.

To address Challenge C1 (Section III-A), we conduct pre-fuzzing analysis to extract three key elements: (1) URIs registered by the web service, (2) input keywords that can accept external data, and (3) sink function address ranges, along with a mapping of basic blocks to their distances from these sinks. These components provide critical semantic and structural information to guide fuzzing. Specifically, the URIs and keywords serve as a mutation dictionary for constructing precise inputs, while sink address ranges define instrumentation boundaries to reduce overhead and focus monitoring. The basic block distance mapping enables a distance-oriented mutation strategy, helping the fuzzer reach deeper execution paths with more *Csource* points.

With this static knowledge, we employ a directed fuzzing strategy that systematically mutates request templates by injecting potentially malicious values into keyword fields. To address Challenge C2 (Section III-B), we extend QEMU with custom instrumentation to perform taint detection on memory writes and control flow operations. During execution, the instrumented QEMU monitors sink scope regions and detects propagation of tainted data. We define the observable influence of tainted input on program behavior as a *Csource* point. Additionally, we collect the resolved targets of indirect calls, which enables the reconstruction of a complete call graph for downstream taint propagation analysis.

*2) Taint-to-PoC Agent:* To address Challenge C3 (Section III-C), we conduct a precise and automated LLM-based vulnerability analysis using information provided from the fuzzing process. After obtaining all potential paths from *Csource* to sink points, we deploy our taint propagation agent to perform comprehensive data flow analysis on the decompiled code along these paths. This agent addresses several limitations inherent in static analysis. These include handling inaccuracies in IDA decompilation output, managing intra-procedural and inter-procedural taint propagation, performing alias analysis, and sanitization checks encountered in real-world scenarios. For each vulnerability alert reported by the taint propagation agent, our PoC generation agent systematically constructs concrete inputs capable of triggering the identified vulnerable behavior. This agent employs an approach that combines constraint information collected during taint propagation analysis with reachable test cases obtained from the fuzzing phase to generate precise PoCs. These generated PoCs are utilized to validate whether each alert represents a genuine vulnerability.

### B. Fuzzing-Driven Information Collection

In this section, we detail our fuzzing-driven information collection framework that combines static pre-analysis and dynamic runtime information collection, serving as the foundation for subsequent vulnerability detection. The whole workflow is depicted as Algorithm 1.

*1) Pre-fuzzing Analysis:* To establish a foundation for effective fuzzing and vulnerability discovery in IoT firmware, we first conduct a comprehensive pre-fuzzing analysis. This analysis aims to extract critical information from firmware binaries without execution, enabling more targeted and efficient fuzzing. Our approach focuses on three key components that guide subsequent fuzzing operations: *Service Handler Detection* to identify potential entry points for each handler function, *Keyword Dictionary Analysis* to identify and extract parameters associated with external input, and *Sink Scope and Distance Calculation* to prioritize code paths leading to security-critical operations. These components collectively form a knowledge base that significantly improves code coverage to find more *Csource*.

*a) Service Handler Detection:* Effective fuzzing relies on triggering the most backend service handlers, yet identifying

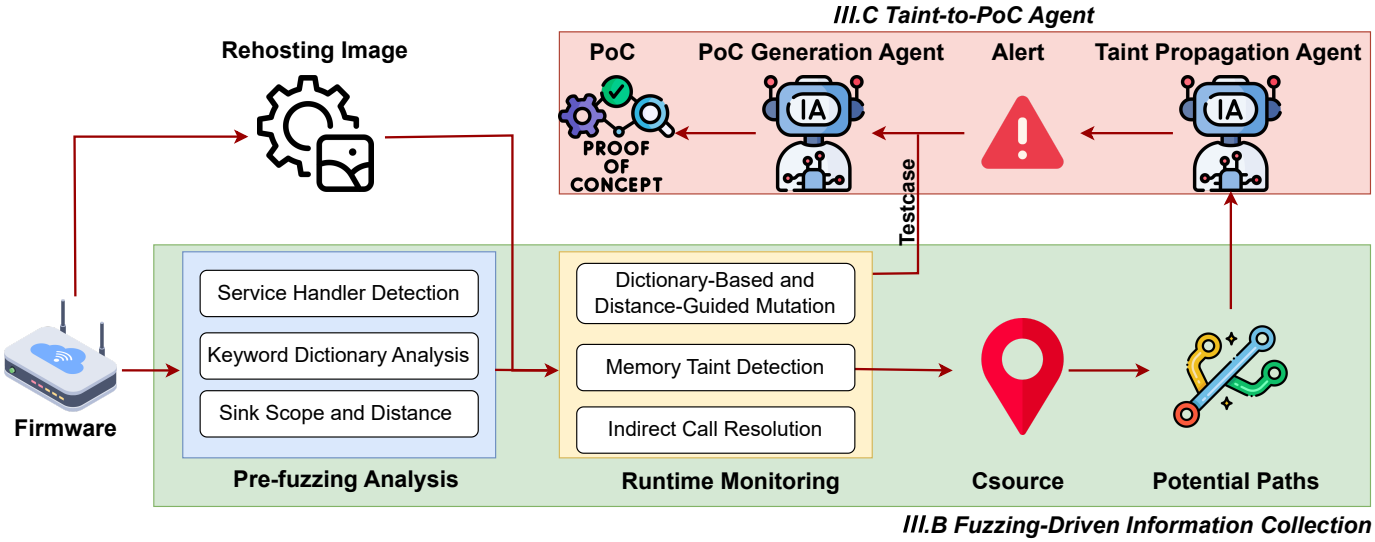


Fig. 3: **Overview of `FirmAgent`**

**Algorithm 1** Fuzzing-Driven Information Collection

**Input:** Firmware $B$, Request Template $S$, Timeout $T$
**Output:** Potential Paths $P$, Reachable Testcases $R$

$H \leftarrow \text{EXTRACTSERVICEHANDLERS}(B)$
$K \leftarrow \text{BUILDKEYWORDDICTIONARY}(B)$
$Sinks, CallGraph \leftarrow \text{STATICEXTRACT}(B)$
$Scope, DistMap \leftarrow \text{BACKWARDANALYSIS}(B, Sinks)$
$emu \leftarrow \text{INITIALIZEEMULATOR}(B)$
$P, R, Csource, IndCalls \leftarrow \emptyset$
**for** each $h \in H$ **do**
  $S_h \leftarrow \text{REPLACEURI}(S, h)$
  $startTime \leftarrow \text{NOW}$
  **for** each $k \in K$ **do**
    **if** $\text{NOW} - startTime > T$ **then**
      **break**
    **end if**
    $Input \leftarrow \text{TAINTINJECT}(S_h, k)$
    $Score \leftarrow \text{COMPUTESCORE}(Input, DistMap)$
    $Mutated \leftarrow \text{GUIDEDMUTATE}(Input, Score)$
    $\text{EXECUTE}(emu, Mutated)$
    **if** $\text{ISTAINTOBSERVED}(emu, Scope)$ **then**
      $Csource \leftarrow Csource \cup \text{TAINTADDR}(emu)$
      $R \leftarrow R \cup \{Mutated\}$
    **end if**
    $Targets \leftarrow \text{RESOLVEINDIRECTCALLS}(emu)$
    $IndCalls \leftarrow IndCalls \cup Targets$
    $CallGraph \leftarrow CallGraph \cup IndCalls$
    $P \leftarrow P \cup \text{GETPATH}(CallGraph, Csource, Sinks)$
  **end for**
**end for**

these handlers is challenging due to incomplete documentation, frontend-backend mismatches, and legacy or hidden code paths. Many handlers, including those for URI patterns, SOAP operations, and HANP messages, lack corresponding request patterns in public interfaces, leading to limited code coverage and missed vulnerabilities. To overcome this, we propose a context-aware handler detection framework that combines static analysis with LLMs. We first extract initial request patterns from exposed interfaces such as web server configs, SOAP definitions, and API docs. These serve as seeds to locate handler code blocks via string references and control flow analysis. Next, we collect contextual information around these handlers and use LLMs to learn generalized request handling patterns. This enables us to discover undocumented handlers by identifying structural similarities in the code. Compared with conventional static approaches, our LLM-guided method achieves enhanced handler coverage while maintaining high precision through systematic pattern validation.

*b) Keyword Dictionary Analysis:* To address the limitations of existing keyword identification methods like SaTC [6], which rely on frontend-backend string matching and miss backend-only logic, we propose a systematic keyword extraction framework that leverages both network traffic and program analysis. We begin by extracting the initial seed from observed network traffic through manual interaction with the rehosted firmware. Using these seeds, we analyze the binary to locate functions that interact with them, rank these functions by invocation frequency, and filter out generic string operations (e.g., strcpy, strcmp). The top three functions are then selected as candidate keyword handlers. To construct a comprehensive keywords dictionary, we extract all parameters passed to candidate keyword handler functions. When the argument is a hardcoded string, we directly retrieve the corresponding parameter at the call site as a keyword. For arguments represented by variables, we perform backward data-flow analysis to identify their origins. These cases fall into three main categories: (1) *Direct assignment from string constants:* If the variable is directly assigned a hardcoded string, backward analysis can retrieve the string value, which is then recorded as a keyword. (2) *Initialization from the .data section:* If the variable originates from a global or static variable stored in the .data segment, we locate its base address and extract all associated string values as potential keywords.

(3) *Dynamic string construction:* In cases where the variable is generated through string concatenation functions (e.g., sprintf, strcat), we identify such operations during backward analysis and continue to trace the sources of all string components involved in the construction. We provide concrete examples in Appendix Figure 8.

*c) Sink Scope and Distance Calculation:* This stage aims to extract the address scope reachable by sink points and compute the distance between each basic block and its nearest sink. This facilitates more efficient and targeted fuzzing in later phases. Our sink extraction process targets security-critical APIs that are commonly associated with taint-based vulnerabilities. Since each type of vulnerability in firmware typically corresponds to specific sensitive APIs, we perform comprehensive pattern matching over the binary to identify these APIs as sink functions.

For each identified sink function, we perform backward reachability analysis to determine all basic blocks that may feasibly execute paths leading to that sink. This analysis helps delineate the relevant address space where tainted data could propagate. By narrowing down the required instrumentation scope to these reachable paths, we significantly reduce runtime overhead without sacrificing detection effectiveness. This targeted analysis ensures that our dynamic taint detection remains focused on the most security-sensitive code regions, thereby improving fuzzing efficiency.

To further prioritize fuzzing efforts, we compute the shortest-path distance from each basic block to the nearest sink using the Dijkstra algorithm on the reversed CFG. Each basic block n is then assigned a score based on both its CFG depth and its proximity to a sink point, as defined by Eq. (1):

$$\text{score}(n) = \text{depth}(n) + w \cdot \text{dis}(s) \tag{1}$$

Here, *depth(n)* denotes the structural depth of block n in the CFG, *dis(s)* represents the shortest distance to a reachable sink *s*, and *w* is a weight parameter that balances exploration depth and sink-directed focus. This scoring metric is later used to guide mutation strategies during fuzzing, enabling a more targeted and effective vulnerability discovery.

*2) Runtime Monitoring:* Prior to fuzzing, we leverage user-space emulation to rehost firmware web services. Subsequently, we collect critical information during the fuzzing process. It consists of two complementary components: a *Mutation Strategy* that leverages insights from pre-fuzzing analysis to generate high-quality test cases, and a *Information Collection* mechanism that efficiently captures runtime behavior with minimal overhead. Together, these components enable accurate source identification and reconstruct the call graph for subsequent taint propagation analysis.

*a) Mutation strategy:* Our grey-box fuzzing approach is designed to overcome the limitations of traditional mutation strategies when applied to firmware web binaries. Techniques such as bit flipping and simple splicing often yield low coverage, primarily due to the architectural nature of firmware: both source and sink points are typically located within the service handler routines. As a result, effective fuzzing must maximize coverage of these service handlers to identify as many source points as possible. To achieve this, mytool integrates insights from the pre-fuzzing analysis phase into its mutation strategy, effectively combining dictionary-based and distance-guided seed mutations to maximize *Csource* coverage.

Leveraging specific service request patterns and a keyword dictionary extracted during the pre-fuzzing phase, we systematically generate test cases by replacing parameters in input requests. Each keyword is annotated with a customized taint tag to facilitate tracking during sanitization checks and to minimize false positives. These taint tags allow us to precisely monitor how each input flows through the program and determine whether it reaches critical sink points without being properly sanitized.

During fuzzing, we dynamically monitor the current execution path by recording the basic block addresses visited. Using a precomputed mapping between basic blocks and their distance-based scores (as described in Section IV-B1c), we calculate the score of each test case. This score reflects the proximity of the current execution path to known sink locations. Code segments closer to sink points typically exhibit a higher possibility of source points. This score is then used to guide both seed prioritization and input mutation. Specifically, inputs that achieve higher scores are prioritized for further mutation, while parameter values in low-scoring inputs are selectively replaced with alternative keywords from the dictionary. This feedback-driven strategy enables the fuzzer to explore a broader range of source points effectively.

*b) Information Collection:* To enable accurate and efficient collection of runtime information during fuzzing, we design a selective instrumentation mechanism guided by a precomputed sink scope. Specifically, during the TCG (Tiny Code Generator) execution phase of QEMU, we monitor only those instructions whose addresses fall within the sink scope. For each such instruction, our framework inspects the memory state after execution to detect whether any memory location transitions from an untainted to a tainted state. When such a transition is observed, we record the corresponding instruction address as a *Csource* point. This selective instrumentation strategy allows us to precisely capture taint propagation sources while significantly reducing the overhead associated with comprehensive monitoring.

Beyond memory state detection, we also collect dynamic control flow information, specifically focusing on indirect call targets encountered during fuzzing. To achieve this, we instrument basic blocks with lightweight callback functions. Within these callbacks, we analyze branching instructions to identify indirect call sites and record the corresponding *(caller, callee)* address pairs. This runtime data enables the reconstruction of a more complete call graph of the binary program. With precise identification of *Csource* points and a complete call graph, we can construct more accurate and complete potential vulnerability paths in relation to predefined sink points. This enriched path information provides a solid foundation for subsequent taint propagation analysis.

### C. Taint-to-PoC Agent

In our system, fuzzing is tightly integrated with real-time taint analysis. We continuously extract *Csource* points and construct potential paths to predefined sink points. These paths are then subjected to taint propagation analysis using taint propagation agent to identify potential vulnerabilities. Since this process is a static analysis, its accuracy is paramount. Compared to traditional taint analysis methods, such as symbolic execution, on-demand alias analysis, and reaching definitions analysis, which often suffer from pointer aliasing and limited semantic understanding. These issues can lead to a high rate of false positives. LLM-based taint analysis [16], [17] has shown significant advantages in reducing both false positives and false negatives. Our empirical findings further validate that LLM can reason effectively about taint flows at the semantic level of decompiled code, demonstrating strong potential for taint-style vulnerability detection. Importantly, traditional taint analysis also lacks the ability to automatically generate PoC to verify vulnerabilities. In contrast, LLMs exhibit robust code reasoning capabilities [21], [22], enabling them to analyze decompiled code and infer the concrete parameter values required to trigger vulnerabilities. This capability allows LLMs to augment fuzzing-generated inputs with the specific data needed to construct effective PoC test cases. Motivated by these insights, we introduce LLMs as the central component of the final vulnerability detection and verification phase in `FirmAgent`. This phase consists of two specialized agents: taint propagation agent, responsible for semantic-level taint analysis, and PoC generation agent, which focuses on validating the existence of vulnerability.

*1) Taint Propagation Agent:* After extracting potential paths between *Csource* and sink points, we leverage LLMs to perform taint analysis on decompiled code. However, static decompilation tools such as IDA sometimes produce imprecise outputs, including missing function parameters, missed return value assignments, and incorrect control/data flow structures, which can significantly impair the effectiveness of LLM-based reasoning. To address these limitations, we apply an LLM-driven refinement process to enhance the quality of the decompiled code along the extracted potential paths (detailed in Appendix B).

Following this refinement, we conduct precise data flow analysis to determine whether tainted data originating from *Csource* points can reach sensitive sink locations through valid execution paths. Our analysis is performed at the function level. As illustrated in Figure 4, we define a prompt template that supplies the LLM with the decompiled function, along with the relevant *Csource* and sink points. The LLM is then asked to determine whether a taint flow exists between the given *Csource* and sink. When unknown functions are encountered during analysis, we will provide function decompilation code in an interactive way to taint them. Then we will continue the analysis based on LLM functional interpretation. For inter-procedural analysis where *Csource* and sink reside in different functions, the sink point is defined at the callsite. The process

**Prompt Template**

Decompiled Code:{**decompiled_code**}
Sources: {**related_source**}
Sink: {**related_sink**}

Perform a comprehensive taint analysis to determine whether data from the source can propagate to the sink and cause a vulnerability. Stop analysis if unknown functions appear in the taint propagation path, and request the decompiled code in JSON format:{"function_def":function_name, ...}.

If taint propagation can be directly determined and cause a vulnerability, report vulnerabilities as [('alert', source_addr, sink_addr), ...] or [] if none found.

Fig. 4: **The Prompt of Taint Propagation Agent.**

begins with intra-procedural analysis of the source function, followed by an additional prompt that guides the LLM to identify the positions of tainted parameters at the callsite. These identified parameters are then treated as taint sources in the callee function for further intra-procedural analysis. This iterative approach continues until the sink function is reached. Furthermore, due to the LLM's insufficient understanding of firmware taint analysis, certain false positives occur. Through our analysis, we identified that these false positives primarily stem from three scenarios: improper handling of sanitization logic, misinterpretation of indirect data dependencies, and the incorrect treatment of values read from system files as taint sources. To address this issue, we incorporated an alert verification module after obtaining alerts from taint analysis. We apply few-shot prompting to validate each alert generated during taint analysis (detailed in Appendix D). Finally, to reduce redundant analysis of the same function across multiple call chains, we implement a function-level caching mechanism. This cache stores LLM-inferred taint behaviors, enabling the system to reuse previously inferred results and avoid repetitive analysis, thereby improving overall efficiency.

*2) PoC Generation Agent:* Once the taint propagation agent identifies a potential vulnerability and raises an alert, the PoC generation agent is invoked to synthesize an input that concretely triggers the vulnerability. During taint analysis, semantic constraints such as conditional branches, sanitization logic, and specific input formats are required to reach the vulnerable sink and are implicitly captured during the process. These captured constraints are then extracted and formalized into an intermediate vulnerability abstraction, which encodes the essential preconditions necessary for successful PoC generation (detailed in Appendix C).

Meanwhile, the fuzzing engine explores program behavior and produces concrete test cases that reach the identified *Csource* points. The final step involves bridging the gap between the syntactically valid input structures discovered by fuzzing and the semantic exploit conditions inferred through LLM-based reasoning. As illustrated in Figure 5, we provide the PoC generation agent with both the fuzzing-derived test case and the constraints obtained during taint analysis. Based

**Prompt Template**

Reachable test case: {**testcase**}
Input validation constraints: {**constraints**}
Decompiled code: {**decompile code**}
Please analyze the decompiled code in conjunction with the input validation constraints to infer the expected values for each input parameter. Then, complete the partial test case by filling in the appropriate values to construct a valid and effective PoC. The resulting input should satisfy all semantic conditions and ensure that the taint can propagate from the identified source to the sink.

Fig. 5: **The Prompt of PoC Generation Agent.**

on the decompiled code and these inputs, the PoC generation agent infers the required values for each relevant parameter and augments the original test case accordingly, ultimately producing a complete and effective PoC input.

## V. Implementation

We implemented a prototype of `FirmAgent`, consisting of over 4,000 lines of Python and 1,000 lines of C code. As a first step, we use `binwalk` to extract the root file system from firmware images and identify the web binary for subsequent analysis. It does not require access to the source code or symbols. In the pre-fuzzing analysis stage, we employed custom IDAPython scripts within IDA Pro [23] to extract critical binary program information, including function call patterns, keyword dictionaries, and distance metrics. This static analysis output forms the basis for both input generation and dynamic analysis. To enable firmware emulation, we adopted the Greenhouse framework [5], which supports user-space rehosting of firmware components. During fuzzing, a custom QEMU [18] plugin was used to capture indirect call relationships and monitor memory state transitions. Furthermore, we integrated DeepSeek-R1 [24], configured with a temperature of 0.7, into both the taint propagation agent and PoC generation agent. `FirmAgent` is designed as a fully automated framework: once fuzzing is initiated on the rehosted environment, the system continuously detects input source points in real-time and forwards their corresponding addresses to the taint propagation agent for further processing.

## VI. Evaluation

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

**RQ1. How does `FirmAgent` vulnerability discovery performance compare to existing state-of-the-art tools?**
**RQ2. How accurate and complete is `FirmAgent` in identifying *Csource* points by fuzzing?**
**RQ3. How does each module in `FirmAgent` contribute to vulnerability detection?**
**RQ4. To what extent can `FirmAgent` generate practical PoCs for real-world vulnerability?**

**Dataset:** Since `FirmAgent` requires fine-grained runtime information collection, we selected firmware samples that could be successfully emulated. Our dataset consists of firmware from major vendors, including Netgear, D-Link, Tenda, Trendnet, Linksys, ASUS, and TOTOLINK, covering 14 firmware samples that span a diverse range of architectures and functionalities. This selection ensures a representative evaluation of our approach across different IoT ecosystems.

**Baseline Comparison:** To benchmark the effectiveness of `FirmAgent`, we compare it against SOTA analysis tools across static, dynamic, and hybrid approaches. Specifically, we evaluate the static analysis tools EmTaint [7] and HermeScan [8], the dynamic analysis tool Greenhouse [5], and the hybrid analysis tool Hy-FirmFuzz. Hy-FirmFuzz extends FirmFuzz [4] by integrating a concolic execution engine built upon angr [25] and replacing the original Firmadyne [26] emulation framework with FirmAE [27] to support emulation across our firmware dataset. The evaluation focuses on the capability to detect buffer overflow and command injection vulnerabilities, using key metrics such as detection precision, analysis time, and the ability to discover previously unknown vulnerabilities. In addition, we consider other representative tools, such as SaTC [6], LARA [20], Mango [19], and OctopusTaint [9]. However, we exclude these tools from direct comparison for the following reasons: a) SaTC: Prior research has demonstrated that HermeScan and EmTaint outperform SaTC in vulnerability detection. b) LARA: The tool is not publicly available, making it impossible to reproduce its results. c) Mango: Since Mango and HermeScan both utilize reaching definitions analysis for taint tracking and achieve comparable results, we select only one of them for comparison. d) OctopusTaint: We found that its open-source code lacks essential components, such as source point identification, making it unsuitable for testing on new firmware.

**Vulnerability Confirmation:** To evaluate the effectiveness of our tool in confirming vulnerabilities, we compare `FirmAgent` with representative SOTA approaches, including EmTaint, HermeScan, and Greenhouse. For traditional static analysis tools such as EmTaint and HermeScan, manual verification is required to confirm each reported vulnerability. This typically involves analyzing the corresponding binary code and crafting a tailored PoC to validate the authenticity of each alert. In contrast, both Greenhouse and `FirmAgent` generate executable PoCs, allowing us to directly test and confirm vulnerabilities on real-world IoT devices.

**Experiment environment:** All of our experiments were conducted on an Intel Xeon Platinum 8358 CPU with 128 logical cores running at 2.60GHz. The machine was equipped with 2.0T RAM and was running on Ubuntu 20.04 LTS. To ensure consistency in evaluation, each firmware sample was analyzed in an isolated environment using docker-based rehosting, and all experiments were repeated four times to mitigate potential measurement bias.

### A. Comparison with the SOTA Tools

Table IV presents the vulnerability detection results of the evaluated tools on our dataset. EmTaint, HermeScan, Greenhouse, Hy-FirmFuzz, and FirmAgent reported 27 alerts, 215 alerts, 20 crashes, 13 crashes, and 200 alerts, respectively. Among these, FirmAgent achieved a 91% precision and successfully confirmed 182 real vulnerabilities, comprising 45 command injection and 137 buffer overflow vulnerabilities. Notably, 27 command injection and 113 buffer overflow vulnerabilities were previously unknown.

In comparison, EmTaint, HermeScan, Greenhouse, and Hy-FirmFuzz achieved precision of 37%, 33%, 40%, and 100%, respectively, confirming 10, 71, 8, and 13 real vulnerabilities. These tools identified subsets of 7, 51, 4, and 5 zero-day vulnerabilities, respectively. FirmAgent demonstrated superior performance with **18.2X**, **2.6X**, **22.8X**, and **14X** more vulnerabilities discovered compared to EmTaint, HermeScan, Greenhouse, and Hy-FirmFuzz, respectively. Furthermore, we analyzed the overlap in vulnerabilities detected by each tool, as illustrated in Figure 6. We found that FirmAgent contains all vulnerabilities reported by the other tools.

We conducted a detailed analysis of the false positives and false negatives produced by each tool to better understand their limitations. For FirmAgent, all 18 false positives were related to buffer overflow vulnerabilities. Through log analysis, we found that these inaccuracies primarily stemmed from limitations in the LLM reasoning process. Specifically, in some cases, the relevant variables were defined outside the current function scope (e.g., global variables), making it difficult for the LLM to accurately determine the relationship between the actual input and the target buffer. As a result, LLM erroneously flagged these cases as vulnerabilities. In contrast, for command injection detection, the LLM achieved higher accuracy. This is largely because such vulnerabilities do not require reasoning about buffer sizes, and the number of relevant sink points in firmware is typically much lower than for buffer overflows.

For EmTaint, false positives were mainly caused by over-aliasing, which resulted in the unintended tainting of unrelated variables. Furthermore, EmTaint lacks the capability to correctly handle in-program sanitizers, which further leads to its false positives. False negatives arose primarily due to EmTaint inability to identify customized source points in the firmware and its limited support for analyzing certain firmware samples. As a result, many potential execution paths remained unanalyzed. HermeScan exhibited false positives due to misidentification of source points and a similar inability to process sanitizer logic. Its false negatives were largely due to limitations in its reaching definition analysis, which led to taint loss in complex data flow scenarios. Greenhouse false positives were mainly attributed to crashes caused by exceeding memory constraints or incomplete emulation by QEMU, which were incorrectly interpreted as genuine vulnerabilities. Its false negatives resulted from its lack of support for hidden URIs and parameter dictionaries, as well as program-specific constraints that limited its code coverage. Consequently, many sink points were not tested during analysis. Hy-FirmFuzz exclusively tests URLs and their corresponding parameters that are exposed in the frontend interface. However, the majority of functional interfaces within the firmware are not displayed in the frontend, resulting in significant false negatives. Since Hy-FirmFuzz performs real-time detection of various vulnerability types during runtime, it does not produce false positives.

Regarding time efficiency, FirmAgent performs taint analysis in real-time during fuzzing, which constitutes its total analysis time. In our evaluation, we constrained fuzzing time to one hour per firmware sample. This is relatively higher than the average analysis time of EmTaint (approximately 3 minutes) and HermeScan (approximately 8 minutes). In comparison, Hy-FirmFuzz and Greenhouse required 78 minutes and 24 hours, respectively. While FirmAgent incurs higher time overhead than static analysis tools, it compensates with significantly greater precision and automation in vulnerability confirmation, including the generation of concrete PoCs. Moreover, since fuzzing and analysis are independent per firmware, FirmAgent supports parallel processing to improve overall efficiency.

**Conclusion**: Compared with SOTA static, dynamic, and hybrid analysis tools, FirmAgent effectively addresses several key challenges in vulnerability detection, including inaccurate source point identification, false positives caused by sanitizer misinterpretation, and insufficient coverage during dynamic fuzzing. As a result, both the false positive and false negative rates in vulnerability detection are significantly reduced.

### B. Source Point Identified by Fuzzing

Table V presents the source point identification results of FirmAgent on our firmware dataset. Experimental results show that all source points identified by FirmAgent are actual sources capable of receiving external input, yielding an accuracy of 100%. This high accuracy stems from the fact that during dynamic analysis, only memory locations that exhibit taint propagation are selected as *Csource* points. Therefore, no false positives were observed in source identification.

However, the coverage of source points is not always complete. Through manual verification of candidate source points identified by static analysis tools, we determined all external input acceptance points (All-Source). On average, FirmAgent was able to identify 94.2% of the true source points reachable via fuzzing. The remaining undetected sources fall into two main categories. First, some source points reside in functions that are dynamically unreachable due to legacy or dead code in the firmware. Since these functions are never invoked during execution, excluding them from analysis actually helps reduce false positives without impacting vulnerability detection. Second, certain source points lie on complex inter-procedural paths, where reaching the corresponding sink requires specific input conditions to be met in earlier execution stages. As a result, fuzzing may fail to trigger these paths, leaving some source points unobserved. Since FirmAgent utilizes LLMs to analyze code segments in the call chain, including those missing source points, we guide the LLM to

TABLE IV: **Vulnerability Detection Results of Emtaint, HermeScan, Greenhouse, Hy-FirmFuzz, and `FirmAgent`**. A "/" denotes cases in which a tool failed to run. prec = $\frac{\text{alert}}{\text{vuln}}$.

| Vendor | Firmware | Emtaint | | | HermeScan | | | Greenhouse | | | Hy-FirmFuzz | | | `FirmAgent` | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Alert | Vuln | Prec | Alert | Vuln | Prec | Crash | Vuln | Prec | Crash | Vuln | Prec | Alert | Vuln | Prec |
| **Trendnet** | TEW800mb | 14 | 10 | 71.4% | 10 | 5 | 50.0% | 0 | 0 | 0 | 0 | 0 | 0% | 25 | 23 | 92.0% |
| | TEW_632BRPA1_FW1.10B31 | 4 | 0 | 0 | 1 | 1 | 100.0% | 2 | 1 | 50.0% | 1 | 1 | 100% | 10 | 8 | 80.0% |
| | TEW673GRUA1_FW100B40 | 1 | 0 | 0 | 2 | 1 | 50.0% | 3 | 1 | 33.3% | 2 | 2 | 100% | 9 | 8 | 88.9% |
| **ASUS** | FW_RT_G32_C1_5002b | 0 | 0 | 0 | 5 | 3 | 60.0% | 0 | 0 | 0 | 0 | 0 | 0% | 3 | 3 | 100.0% |
| **DLink** | DIR_825_REVB_2.03 | 3 | 0 | 0 | 3 | 1 | 33.3% | 9 | 3 | 33.3% | 2 | 2 | 100% | 10 | 8 | 80.0% |
| | DIR_601_REVA_1.02 | 5 | 0 | 0 | 2 | 0 | 0 | 2 | 1 | 50.0% | 0 | 0 | 0% | 5 | 4 | 80.0% |
| | DCS_934L_REVA_1.04.15 | / | / | / | 15 | 6 | 40.0% | 0 | 0 | 0 | 0 | 0 | 0% | 12 | 11 | 91.7% |
| **Netgear** | DC112A_V1.0.0.64 | 0 | 0 | 0 | 57 | 1 | 1.8% | 0 | 0 | 0 | 0 | 0 | 0% | 42 | 40 | 95.2% |
| | JNR3300_V1.0.0.34PR | 0 | 0 | 0 | 18 | 9 | 50.0% | 2 | 0 | 0 | 1 | 1 | 100% | 12 | 11 | 91.7% |
| | XWN5001_V0.4.1.1 | 0 | 0 | 0 | 18 | 11 | 61.1% | 0 | 0 | 0 | 0 | 0 | 0% | 14 | 12 | 85.7% |
| **Tenda** | W6_S_v1.0.0.4_510_en | / | / | / | 9 | 4 | 44.4% | 2 | 2 | 100.0% | 3 | 3 | 100% | 14 | 12 | 85.7% |
| | HG7_HG9_HG10re_300001138 | 0 | 0 | 0 | 26 | 16 | 61.5% | 0 | 0 | 0 | 0 | 0 | 0% | 17 | 16 | 94.1% |
| **Linksys** | FW_WRT54Gv4_4.21.5 | / | / | / | 27 | 3 | 11.1% | 0 | 0 | 0 | 3 | 3 | 100% | 15 | 14 | 93.3% |
| **TOTOLINK** | LR350_v9.3.5 | 0 | 0 | 0 | 22 | 10 | 45.5% | 0 | 0 | 0 | 1 | 1 | 100% | 12 | 12 | 100.0% |
| Total | | 27 | **10** | | 215 | **71** | | 20 | **8** | | 13 | **13** | | 200 | **182** | |
| Average | | 2.5 | 0.9 | **37.0%** | 15.4 | 5.1 | **33.0%** | 1.4 | 0.6 | **40.0%** | 0.9 | 0.9 | **100%** | 14.3 | 13.0 | **91.0%** |

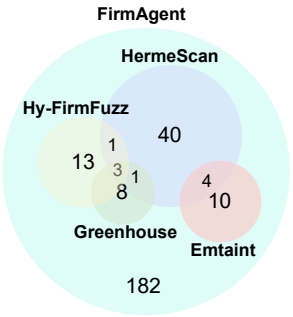


Fig. 6: **Vulnerability Containment Relationships.**

TABLE V: **The *Csource* Identification Results of `FirmAgent`.** *Csource* denotes external input point identified by fuzzing, T-Source denotes external input points, All-Source denotes Total external input points. Src.Ratio = $\frac{\textit{Csource}}{\text{All-Source}}$.

| Firmware | *Csource* | Source | All-Source | Src.Ratio |
|---|---|---|---|---|
| **TEW800mb** | 58 | 58 | 71 | 81.7% |
| **TEW_632BRPA1_FW1.10B31** | 11 | 11 | 11 | 100.0% |
| **TEW673GRUA1_FW100B40** | 64 | 64 | 64 | 100.0% |
| **FW_RT_G32_C1_5002b** | 27 | 27 | 34 | 79.4% |
| **DIR_825_REVB_2.03** | 11 | 11 | 11 | 100.0% |
| **DIR_601_REVA_1.02** | 21 | 21 | 24 | 87.5% |
| **DCS_934L_REVA_1.04.15** | 37 | 37 | 41 | 90.2% |
| **DC112A_V1.0.0.64** | 95 | 95 | 101 | 94.1% |
| **JNR3300_V1.0.0.34PR** | 129 | 129 | 133 | 97.0% |
| **XWN5001_V0.4.1.1** | 148 | 148 | 153 | 96.7% |
| **W6_S_v1.0.0.4_510_en** | 306 | 306 | 321 | 95.3% |
| **HG7_HG9_HG10re_300001138** | 52 | 52 | 56 | 92.9% |
| **FW_WRT54Gv4_4.21.5** | 32 | 32 | 33 | 97.0% |
| **LR350_v9.3.5** | 433 | 433 | 458 | 94.5% |
| **Average** | 101.7 | 101.7 | 107.9 | 94.2% |

TABLE VI: **Ablation Study on the Contribution of Each Component in `FirmAgent` to Vulnerability Detection.** *Firm-Directed* removes the directed strategy; *Firm-Indirect* disables the use of indirect call information collected during dynamic execution; *Firm-Cut* removes the LLM-based decompiled code refinement and alert verification components; *Firm-Emt* and *Firm-Her* replace `FirmAgent` taint propagation module with Emtaint and HermeScan, respectively.

| Tools | Alert | TP | FP | TPR | FPR |
|---|---|---|---|---|---|
| **Firm-Directed** | 194 | 176 | 18 | 90.7% | 9.3% |
| **Firm-Indirect** | 187 | 169 | 18 | 90.4% | 9.6% |
| **Firm-Cut** | 232 | 172 | 60 | 74.1% | 25.9% |
| **Firm-Emt** | 34 | 15 | 19 | 44.1% | 55.9% |
| **Firm-Her** | 120 | 71 | 49 | 59.2% | 40.8% |
| **`FirmAgent`** | 200 | 182 | 18 | 91.0% | 9.0% |

treat these source functions as source points to compensate for parts not reached during dynamic analysis. Consequently, no false negatives were observed in vulnerability detection due to missing source points.

**Conclusion**: All source points identified by `FirmAgent` are real and externally controllable, effectively eliminating the false positives common in traditional static approaches. Although a small portion of source points may be missed, this does not lead to missed vulnerabilities, and can even help reduce false positives.

### *C. Ablation Study*

To evaluate the effectiveness of key components in `FirmAgent`, namely *Csource*, the taint propagation agent module, the directed fuzzing strategy, and the indirect call resolution, we conducted an ablation study:

**Effectiveness of *Csource* and Taint Propagation Agent.** To assess the advantage of our *Csource* and taint propagation agent module over traditional taint analysis approaches, we replaced `FirmAgent` taint analysis with those from Emtaint (Firm-Emt) and HermeScan (Firm-Her), while keeping the *Csource* consistency. We then assessed the improvements brought by our *Csource* selection when integrated with Emtaint and HermeScan. Finally, we compared their overall effectiveness against the `FirmAgent` to highlight the benefits of our taint analysis strategy. As shown in Table VI, Firm-Emt reported 34 alerts, of which 15 were true positives, achieving a precision of 44.1%. This represents an improvement of five additional vulnerabilities compared to the original Emtaint, primarily due to `FirmAgent` use of dynamically extracted custom source points, many of which are not covered by Emtaint predefined source list. However, Firm-Emt exhibited higher false negatives due to its limited instruction support in certain firmware and higher false positives due to excessive alias analysis. Firm-Her reported 120 alerts with 71 true positives, achieving a precision of 59.2%. While the number of detected vulnerabilities was consistent with HermeScan results, the false positive rate decreased from 67.0% to 40.8%. This reduction stems from our refined source point selection, which filters out candidate source points that do not accept external input. Nevertheless, false negatives persisted due to inaccurate alias analysis that led to lost taint.

**Effectiveness of the Directed Fuzzing Strategy.** To evaluate the contribution of the directed fuzzing strategy, we disabled this component in `FirmAgent`. As shown in the Table VI, Firm-Directed reported 194 alerts, including 176

true vulnerabilities, six fewer than the FirmAgent. These missed vulnerabilities correspond to deep path source points that were not reached without the guidance of the directed fuzzing strategy, highlighting its importance in improving path exploration depth.

**Effectiveness of Indirect Call Resolution.** To assess the impact of indirect call resolution, we disabled the use of dynamically collected indirect call targets. As shown in Table VI, Firm-Indirect reported 187 alerts, of which 169 were true positives, 13 fewer than FirmAgent. Our analysis revealed that these missed vulnerabilities involved tainted parameters passed through indirect calls to sink functions. Without dynamic resolution of indirect calls, these control-flow paths could not be constructed, leading to false negatives.

**Effectiveness of LLM Refinement and Verification Modules.** To evaluate the effectiveness of the decompiled code refinement and alert verification components in taint propagation agent, we conducted an ablation study by disabling them. As shown in Table VI, Firm-Cut reported 232 alerts, with 172 true positives, ten fewer than FirmAgent. It also introduced 42 additional false positives, reducing the precision to 74.1%. Through analysis, we found that the false negatives were caused by missing critical parameters in functions within the IDA decompiled code, leading to taint loss. The false positives resulted from the LLM's inability to handle sanitization processes accurately, recognize non-propagating taints, and identify system file sources.

**Conclusion.** The ablation study demonstrates that each component of FirmAgent plays a critical role in enhancing the overall effectiveness of vulnerability detection. The *Csource* points identified through fuzzing significantly reduce false positives commonly encountered in traditional taint analysis tools. The taint propagation agent module improves the precision of taint propagation, the directed fuzzing strategy increases the likelihood of reaching deep *Csource* points, and the indirect call resolution bridges critical gaps in control flow analysis. Collectively, these modules work synergistically to reduce both false negatives and false positives, thereby enabling more accurate and comprehensive vulnerability discovery in IoT firmware.

### D. PoC Effectiveness

To evaluate the practical effectiveness of the PoCs generated by FirmAgent, we conducted effectiveness testing on real-world devices without any manual intervention. PoCs that could be successfully executed in this fully automated setting were categorized as E-PoC. For the remaining PoCs that failed to trigger vulnerabilities automatically, we manually analyzed the program logic, adjusted the input parameters, and considered those that could be successfully verified through minor manual effort as H-PoC. The remaining cases were deemed false positives (F-PoC).

As shown in Figure 7, excluding 18 false positives, among the remaining 182 PoCs generated by FirmAgent, 167 were directly effective without requiring any modification and 15 could be made effective with minimal manual adjustments.

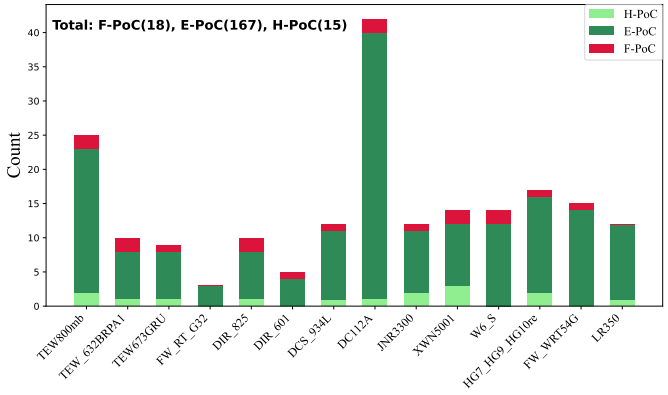


Fig. 7: **The Result of PoC Effectiveness.** E-PoC denotes directly effective PoCs, and H-PoC refers to PoCs that become effective after manual modification, and F-PoC represents false positives that result in ineffective PoCs.

Most of the H-PoC were associated with command injection vulnerabilities. The primary reason for failure stems from the diversity of command injection payloads, as different execution contexts often require distinct malicious input to successfully trigger the vulnerability. This increases the complexity of reasoning required by LLMs, making it difficult to consistently generate valid payloads. For instance, some firmware employs blacklist-based input filtering mechanisms, which restrict the use of special characters commonly used in command injection (e.g., ';', '|', '&'). However, these filtering mechanisms are often incomplete or circumventable. Successfully triggering them typically requires expert knowledge to understand the execution context and craft an effective PoC accordingly. Our findings demonstrate that this category of vulnerabilities remains a challenge for fully automated triggering by current LLM-based approaches.

**Conclusion:** FirmAgent achieved an effective PoC generation rate of 91.8%, significantly reducing the manual effort required for vulnerability verification compared to traditional static analysis tools. Analysts only need to assess a small fraction of the remaining PoC, which streamlines the overall process of confirming vulnerabilities.

## VII. Discussion

### A. Limitations of Firmware Rehosting

The capability of FirmAgent is fundamentally dependent on the underlying firmware rehosting framework. While Greenhouse is one of the state-of-the-art solutions that support rehosting user-space components from firmware samples of mainstream router vendors, its success rate remains low, especially when dealing with newer firmware versions. Furthermore, Greenhouse is currently limited to emulating a single service at a time. As a result, FirmAgent can only detect vulnerabilities within that binary, potentially overlooking vulnerabilities that span across binaries or require full-system emulation. We anticipate that advancements in firmware rehosting technologies will expand the range of

supported devices and improve compatibility with more recent firmware versions. In future work, we will try to improve the generality of our system by integrating multiple rehosting strategies to support multi-binary and more comprehensive firmware emulation.

### B. False Positives in `FirmAgent`

While `FirmAgent` achieves a significantly lower false positive and false negative rate compared to prior approaches, it is not entirely immune to inaccuracies. Our analysis reveals that most false positives arise from buffer overflow detection. Specifically, the LLM occasionally fails to account for the actual size relationships between input data and target buffers. This often results in secure code locations being misclassified as vulnerable due to hypothetical overflow scenarios that are infeasible in practice. To mitigate this issue, future iterations of `FirmAgent` may incorporate more precise data modeling techniques or leverage retrieval augmented generation (RAG) and fine-tuning strategies to enhance the LLM understanding of memory semantics and size constraints in vulnerability contexts. This would improve the overall precision of buffer overflow detection.

## VIII. Related Work

### A. Firmware Rehosting and Fuzzing

Firmware rehosting provides a foundation for dynamic firmware analysis by enabling embedded system execution in a controlled and analyzable environment. Early approaches such as QEMU and Avatar [28] offered general-purpose emulation and hardware-in-the-loop capabilities. Building on these foundations, frameworks like Firmadyne [26] and FirmAE [27] proposed automated solutions for rehosting Linux-based firmware at scale, enabling large-scale vulnerability analysis across heterogeneous devices. To improve service-level analysis granularity and scalability, Greenhouse [5] introduced a user-space rehosting approach targeting individual firmware binaries. Rehosting enables further dynamic analysis, particularly fuzzing. FirmAFL [2] integrated greybox fuzzing with rehosting to discover vulnerabilities in user-space firmware components. EM-Fuzz [29] and IoTHunter [30] embedded real-time checkers into the emulation environment to enhance fuzzing effectiveness by providing runtime feedback. FirmFuzz [4] employs static analysis to generate valid input seeds and extracts contextual information about potential vulnerabilities, thereby enhancing dynamic analysis for uncovering deep vulnerabilities in embedded firmware. For firmware that cannot be rehosted, researchers have explored black-box fuzzing techniques. IoTFuzzer [31] uses mobile applications to send malformed payloads to physical devices, while SRFuzzer [32] leverages heuristic-guided mutation strategies to target IoT web interfaces. LABRADOR [33] leverages response-based execution trace inference and IO-oriented distance measurement to guide efficient fuzzing.

### B. Static Analysis in IoT Firmware

Static analysis of IoT firmware enables vulnerability detection without requiring execution, making it suitable for large-scale or safety-critical applications. KARONTE [34] pioneered cross-component static dataflow analysis to detect vulnerabilities that arise from interactions among firmware modules and linked libraries. SaTC [6] identified that developers often reuse keywords between front-end interfaces and back-end binaries, which can be exploited to localize potential sources of user input. EmTaint [7] proposed a precise taint analysis method that resolves indirect calls using a structured symbolic expression and an on-demand alias analysis scheme. HermeScan [8] introduced a lightweight, context-sensitive Reaching Definitions Analysis built upon an expanded control-flow graph, significantly reducing the overhead compared to symbolic execution-based approaches. Mango [19] enhanced scalability and efficiency through a novel *sink-to-source* analysis combined with an *Assumed Non-impact* optimization technique. OctopusTaint [9] introduced a sanitization inspection mechanism and a recursive tracing algorithm, both of which substantially reduce false positives and improve overall accuracy in vulnerability detection.

### C. LLMs for Vulnerability Detection

Recent advances in LLMs have opened up new possibilities for vulnerability detection by enabling deep semantic understanding and reasoning over code. ChatAFL [35] leverages LLMs to infer protocol formats and generate structurally valid test cases, thereby enhancing code coverage during the fuzzing process. It represents an LLM-assisted fuzzing framework, where LLMs are integrated directly into the fuzzing loop to improve input quality and effectiveness. In contrast, `FirmAgent` adopts a fundamentally different approach: it is a fuzzing-assisted static analysis framework that utilizes runtime information collected during fuzzing to guide LLM-based taint analysis and PoC generation.

Lara [20] combines traditional pattern-based static analysis with LLM-assisted reasoning to identify key elements such as URIs and keywords in firmware, improving detection accuracy. LATTE [16] leverages LLMs to perform taint analysis directly on binary code, significantly improving both the automation and accuracy of vulnerability detection. IRIS [17] proposes a hybrid approach that fuses LLM-inferred specifications with the CodeQL [36] static analysis framework, enabling the detection of previously overlooked vulnerabilities. Moreover, IRIS uses LLMs contextual reasoning ability to filter out false positives, thereby improving precision. However, these existing approaches are purely static and lack targeted design for firmware taint analysis, resulting in suboptimal analysis accuracy. In contrast, `FirmAgent` collects runtime data from firmware execution to identify source points and indirect calls that assist in taint analysis. Furthermore, when employing LLMs for taint analysis, we incorporate specialized strategies including refined decompilation, intra-procedural analysis, and alert verification to enhance detection accuracy.

## IX. Conclusion

In this paper, we systematically analyze the limitations of existing vulnerability detection techniques in IoT firmware and observe a critical gap: dynamic analysis reaches many sources but fails to propagate taint to sinks (causing false negatives), while static analysis explores all source-to-sink paths but struggles to identify true sources (causing false positives). To address these complementary shortcomings, we propose `FirmAgent`, a novel framework that integrates the strengths of both dynamic and static analysis. Specifically, we employ lightweight fuzzing to dynamically identify source points in the binary. These are then used as starting points (*Csource*) for taint propagation agent to trace data flows toward potential sinks. Finally, our framework utilizes a PoC generation agent to confirm whether the identified alerts represent exploitable vulnerabilities. We evaluate `FirmAgent` on 14 real-world firmware samples. The framework successfully identifies 182 vulnerabilities with a precision of 91%. Compared to existing state-of-the-art static and dynamic analysis tools, `FirmAgent` significantly reduces both false positives and false negatives, demonstrating its effectiveness and practicality in IoT vulnerability detection.

## Ethics Considerations

This research focuses on analyzing publicly available IoT firmware and does not involve any interaction with human subjects, personal data, or deployed systems. All vulnerabilities discovered during our experiments were responsibly disclosed to the corresponding vendors, following established responsible disclosure practices. Our work strictly adheres to ethical research guidelines, and no attempts were made to publicly release any identified vulnerabilities.

## Acknowledgment

We would like to sincerely thank all the reviewers for their insightful suggestions that helped us to improve this paper. This work is supported by the National Natural Science Foundation of China under grant U24A20337 and 62276091, Natural Science Foundation of Henan Province of China (No.242300420698), Ant Group, and Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

## References

[1] S. Sinha, "Iot connections market update." *IoT Analytics*, 2024. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

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

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

[4] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "Firmfuzz: Automated iot firmware introspection and analysis," in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, ser. IoT S&P'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 15–21. [Online]. Available: https://doi.org/10.1145/3338507.3358616

[5] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, F. Dong, Z. Smith *et al.*, "Greenhouse:{Single-Service} rehosting of {Linux-Based} firmware binaries in {User-Space} emulation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5791–5808.

[6] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 303–319.

[7] K. Cheng, Y. Zheng, T. Liu, L. Guan, P. Liu, H. Li, H. Zhu, K. Ye, and L. Sun, "Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 360–372.

[8] Z. Gao, C. Zhang, H. Liu, W. Sun, Z. Tang, L. Jiang, J. Chen, and Y. Xie, "Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis," in *NDSS*, 2024.

[9] A. Qasem, M. Debbabi, and A. Soeanu, "Octopustaint: Advanced data flow analysis for detecting taint-based vulnerabilities in iot/iiot firmware," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2355–2369.

[10] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1580–1596. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00002

[11] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[12] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[13] R. Meng, G. J. Duck, and A. Roychoudhury, "Large language model assisted hybrid fuzzing," 2024. [Online]. Available: https://arxiv.org/abs/2412.15931

[14] L. Situ, C. Zhang, L. Guan, Z. Zuo, L. Wang, X. Li, P. Liu, and J. Shi, "Physical devices-agnostic hybrid fuzzing of iot firmware," *IEEE Internet of Things Journal*, vol. 10, no. 23, pp. 20 718–20 734, 2023.

[15] J. Yin, M. Li, Y. Li, Y. Yu, B. Lin, Y. Zou, Y. Liu, W. Huo, and J. Xue, "Rsfuzzer: Discovering deep smi handler vulnerabilities in uefi firmware with hybrid fuzzing," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2155–2169.

[16] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Xu, Z. Li, P. Di, Y. Jiang *et al.*, "Llm-powered static binary taint analysis," *ACM Transactions on Software Engineering and Methodology*, 2025.

[17] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=9LdJDU7E91

[18] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 41–46. [Online]. Available: http://www.usenix.org/events/usenix05/tech/freenix/bellard.html

[19] W. Gibbs, A. S. Raj, J. M. Vadayath, H. J. Tay, J. Miller, A. Ajayan, Z. L. Basque, A. Dutcher, F. Dong, X. Maso *et al.*, "Operation mango: Scalable discovery of {Taint-Style} vulnerabilities in binary firmware services," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 7123–7139.

[20] J. Zhao, Y. Li, Y. Zou, Z. Liang, Y. Xiao, Y. Li, B. Peng, N. Zhong, X. Wang, W. 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)*, 2024, pp. 7067–7084.

[21] R. Widyasari, J. W. Ang, T. G. Nguyen, N. Sharma, and D. Lo, "Demystifying faulty code with llm: Step-by-step reasoning for explainable fault localization," *arXiv preprint arXiv:2403.10507*, 2024.

[22] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 2024, pp. 103–111.

[23] I. Guilfanov, "Ida pro disassembler and debugger," 2022. [Online]. Available: https://hex-rays.com/

[24] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, and S. M. et al., "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[25] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[26] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *NDSS*, vol. 1, 2016, pp. 1–1.

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

[28] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, vol. 18, 2018, pp. 1–11.

[29] J. Gao, Y. Xu, Y. Jiang, Z. Liu, W. Chang, X. Jiao, and J. Sun, "Em-fuzz: Augmented firmware fuzzing via memory checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3420–3432, 2020.

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

[31] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *NDSS*, 2018.

[32] Y. Zhang, W. Huo, K. Jian, J. Shi, H. Lu, L. Liu, C. Wang, D. Sun, C. Zhang, and B. Liu, "Srfuzzer: An automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities," in *Proceedings of the 35th annual computer security applications conference*, 2019, pp. 544–556.

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

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

[35] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[36] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "QL: object-oriented queries on relational data," in *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, ser. LIPIcs, S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:25. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

# APPENDIX

## A. Keyword Extraction

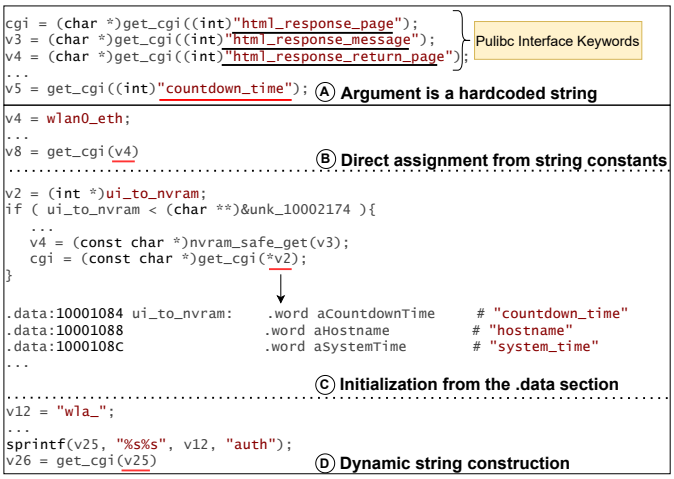


Fig. 8: **Four Common Patterns for Extracting Keywords.**

## B. LLM-Based Refinement of Decompiled Code

The prompt used for LLM-based decompilation refinement (as illustrated in Figure 9) is designed to address the limitation of traditional static analysis tools: the generation of imprecise decompiled code due to issues such as missing function parameters, omitted return value assignments, and inaccurate control or data flow structures. These deficiencies can significantly impair the accuracy of subsequent taint analysis.

Figure 10 presents a real-world example in which IDA fails to reconstruct the correct high-level logic due to architectural and contextual limitations. A key reason for IDA failure to preserve the return value of *get_cgi("date")* lies in its incorrect interpretation of the *system()* function signature. Consequently, the return value of *get_cgi("date")* is mistakenly considered unused, leading IDA to optimize away the function call. Such omissions can result in false negatives during taint analysis, as essential data flow relationships are lost. In contrast, our LLM-based refinement approach leverages semantic context understanding and pattern recognition capabilities to recover missing data flows and accurately reconstruct argument construction processes, thus preserving essential taint propagation relationships and improving overall detection fidelity.

**Prompt**

Decompiled Code:{**decompiled_code**}
Please carefully review the decompiled code and produce a more accurate, semantically correct version of the C code. Your output should restore meaningful variable names, preserve the original logic, and infer any implicit behaviors such as string formatting, argument construction, or return value assignments.

Fig. 9: **The Prompt of Decompiled Code Refinement.**

```
1. get_cgi("date");
2. v7 = (char *)get_cgi("timeTag");
3. if ( !strcmp(v7, "manual") ) {
4.   system("date -s %s ");
5. }
```
**Before Refinement**

```
1. char *date = (char *)get_cgi("date");
2. char *timeTag = (char *)get_cgi("timeTag");
3. if ( !strcmp(timeTag, "manual" )) {
4.     char command[64];
5.     snprintf(command, sizeof(command), "date -s '%s'", date);
6.     system(command);
7. }
```
**After Refinement**

Fig. 10: **A Real-World Example of LLM-Based Decompiled Code Refinement.** *Before Refinement* represent the original decompilation result produced by IDA, *After Refinement* represent the refined version generated by LLM-based approach.

### C. PoC Generation and Security Impact

The PoCs generated by `FirmAgent` are HTTP-format request packets. Since the fuzzing phase already provides test cases that reach the source point (including URL and headers), we need to analyze the parameter constraints between the source and sink points to identify concrete input values. For example: {NtpDstEnable: "1", NtpDstOffset: 'A'*300, ...}. These values are used to complete the request body and form a full PoC. The generated PoCs can trigger two primary vulnerability types: buffer overflows, which can cause program crashes, and command injections, which allow arbitrary command execution.

### D. Alert Verification

```
Code:
int sub_E96C()
{
  ...
  Value = cgiGetValue(dword_42D0C, "rtick");
  v1 = 0;
  while (Value[v1])
  {
    if ((unsigned __int8)Value[v1] - 48 > 9)
    {
      syslog(149, "... rtick=%s", Value);
      exit(1);
    }
    ++v1;
  }
  sprintf(v5, "'%s'.gif '%s'.txt", Value, Value);
  system(v5);
  ...
}
Analysis: The input is validated within the loop to
ensure that only numeric characters are allowed. Any
invalid input triggers an exit, rendering malicious
user-constructed strings ineffective.
Result: NO
```

(a) Example of the sanitizer process identifying validation logic that eliminates malicious inputs, such as numerical character checks.

```
Code:
if (*taint)
{
  if (!strcmp(taint, "2"))
  {
    v4 = "factory_hm info fat 2";
  }
  else
  {
    v4 = "factory_hm info fat 1";
  }
  system(v4);
}
Analysis: The taint is only used in conditional
statements and does not directly influence the value
of v4, which is always assigned a constant.
Result: NO
```

(b) Example of non-propagating taint where the tainted data is used only for conditional checks without affecting sink parameters.

```
Code:
int sub_463F0()
{
  ...
  FILE *f = fopen("/etc/passwd", "r");
  fgets(buf, sizeof(buf), f);
  system(buf);
  ...
}
Analysis: The data is directly read from a system
file (/etc/passwd) and passed to a sink. However,
since the file contents are not user-controllable,
they cannot be exploited by users.
Result: NO
```

(c) Example of source originates from system files.

Fig. 11: Illustrative examples of mitigating false positives across three scenarios: (a) sanitizer processes, (b) non-propagating taint, and (c) system file sources.