# CGIFuzz: Enabling Gray-Box Fuzzing for Web CGI of IoT Devices

Cheng Shi, Jiongchi Yu, Ziming Zhao, *Member, IEEE*, Jiongyi Chen, and Fan Zhang, *Member, IEEE*

***Abstract*—Fuzz testing for Internet of Things (IoT) devices has become a critical area of research, as these devices play an increasingly vital role in modern networks and infrastructure. While significant efforts have been made, the Common Gateway Interface (CGI) programs that serve as an important component within these devices remain underexplored. Despite their extensive use in IoT web services, the specific characteristics of CGI programs have posed technical challenges to existing fuzzing infrastructures. To address these gaps, we propose CGIFuzz, the first gray-box fuzzing framework tailored for CGI programs in Linux-based IoT devices. CGIFuzz initially enables dynamic instrumentation of CGI programs through *Relay-Pass Instrumentation*, then leverages Large Language Models (LLM) for assisting high-quality fuzz test input generation. Furthermore, CGIFuzz devises oracles for detecting command injection and memory corruption vulnerabilities by leveraging multiple critical features during program execution. Our evaluation of CGIFuzz on ten popular IoT devices demonstrates superior coverage exploration and vulnerability detection capabilities compared to the state-of-the-art fuzzers. Notably, CGIFuzz discovered 69 vulnerabilities, including 13 previously unknown ones for which 9 CVEs were assigned.**

***Index Terms*—Gray-box fuzzing, Internet of Things, common gateway interface, command injection vulnerabilities, large language models.**

## I. Introduction

As INTERNET of Things (IoT) devices become ubiquitous, their web-based management interfaces have emerged as a primary attack surface [1], [2]. These web systems, typically accessible via ports 80 or 443, consist of two main components: a web server and multiple Common Gateway Interface (CGI) programs [3], [4], as shown in Fig. 1. While the web servers often employ well-tested open-source software (e.g., *Apache httpd*, *GoAhead*, and *Nginx*) [5], [6], [7], the CGI programs present a starkly different security posture. Typically developed in-house by manufacturers, distributed as closed-source binaries, and granted high privileges for system operations, these custom CGI programs are frequently insecure and serve as a critical entry point for compromising devices [8], [9]. Therefore, proactively discovering vulnerabilities in these CGI binaries is crucial for securing the IoT ecosystem.

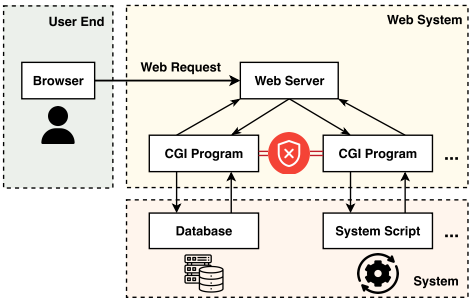


Fig. 1. Workflow of CGI-based IoT web system.

To this end, fuzz testing is a widely adopted and effective technique for identifying software vulnerabilities, with several tools [10], [11], [12], [13], [14], [15], [16] having been developed for IoT devices. These approaches can be broadly categorized as black-box or gray-box, yet both fall short when applied to CGI programs. Some tools employ black-box techniques, which require no knowledge of the program's internals [10], [11], [16]. While simple, their lack of runtime feedback severely limits their ability to explore complex program states. This is particularly detrimental for fuzzing CGI programs, whose behavior often depends on intricate combinations of input parameters. Furthermore, as black-box scanners can only infer bugs from outputs, they are prone to high false-positive rates and struggle to detect critical threats like command injection vulnerabilities, one of the most prevalent types in IoT devices [14]. Gray-box fuzzers, in contrast, leverage runtime feedback such as path coverage to guide input generation, demonstrating significant advantages over black-box methods [12], [13], [14], [15], [17]. However, these solutions are still not directly applicable to CGI programs. The reason lies in the unique, two-stage validation process within IoT web systems. Specifically, an incoming request must first pass the web server's *syntactic* validation (e.g., protocol structure). Only then is it dispatched to the target CGI program, which performs its own *semantic* verification (e.g., parameter logic and value constraints) before executing its core functions. Consequently, the internal logic of a CGI program is triggered only when an input satisfies both criteria. We term this interaction model a *shadow calling pair*, as the CGI program is invoked transiently and is opaque to the end-user. This unique execution model poses fundamental challenges that existing fuzzing frameworks fail to address

Received 15 February 2025; revised 28 June 2025, 25 August 2025, and 3 October 2025; accepted 6 October 2025. Date of publication 10 October 2025; date of current version 24 October 2025. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB3106800 and in part by the Natural Science Foundation of China under Grant 62302508. The associate editor coordinating the review of this article and approving it for publication was Prof. Mika Ylianttila. *(Corresponding authors: Fan Zhang; Ziming Zhao.)*

Cheng Shi, Ziming Zhao, and Fan Zhang are with Zhejiang University, Hangzhou 310027, China (e-mail: shicheng@zju.edu.cn; zhaoziming@zju.edu.cn; fanzhang@zju.edu.cn).

Jiongchi Yu is with the School of Computing and Information Systems, Singapore Management University, Singapore 188065 (e-mail: jcyu.2022@phdcs.smu.edu.sg).

Jiongyi Chen is with the National University of Defense Technology, Changsha 410073, China (e-mail: chenjiongyi@nudt.edu.cn).

Digital Object Identifier 10.1109/TIFS.2025.3620120

### A. Enabling the Universality of CGI Path Coverage Collection

Some gray-box tools [12], [14], [15] use emulation-based instrumentation to collect coverage data. However, emulation often struggles with the intricate hardware and software dependencies of IoT devices, leaving many real-world devices unsupported [13]. To address this, debugger-based approaches [13], [17] extend applicability by enabling instrumentation on both simulated and physical hardware devices. Despite these advancements, debugger-based methods are ill-suited for handling transient and dynamically invoked CGI programs. Their requirement to launch the target program under a debugger is fundamentally incompatible with the CGI execution model, where programs are dynamically spawned by a web server process, not the fuzzer. Therefore, a novel approach is necessary to collect path coverage for CGI binaries universally and efficiently on live devices.

### B. Generating Effective Test Input for CGI

The second major challenge stems from the fact that CGI programs do not operate in isolation. Unlike typical fuzzing where the fuzzer interacts directly with the target binary, a fuzzer for CGI programs must communicate through an intermediary: the web server. This creates a stringent, two-stage validation gauntlet that any test input must pass. First, the input must be *syntactically* valid to be accepted and parsed by the web server (e.g., conforming to HTTP specifications). Second, it must be *semantically* correct to satisfy the internal logic of the CGI program itself (e.g., containing the right parameter names, value formats, and dependencies). Generating effective test inputs that can consistently satisfy both layers of validation is a significant hurdle for automated fuzzing tools.

### C. Detecting CGI Vulnerabilities Accurately

The final challenge lies in vulnerability detection. The oracles in most existing fuzzers are designed primarily to detect memory corruption vulnerabilities by watching for process-ending signals like segmentation faults. However, command injection, a critical and prevalent vulnerability class in CGI programs, often does not cause a crash. Instead, a successful injection may result in the silent execution of malicious commands, making it completely invisible to conventional crash-based oracles. This highlights the urgent need for a more sophisticated oracle capable of reliably identifying the subtle execution pattern anomalies associated with command injection attacks in IoT environments.

In this paper, we present CGIFuzz, the first gray-box fuzzing framework designed to effectively detect vulnerabilities for CGI programs of IoT devices. Specifically, CGIFuzz introduces new solutions across feedback collection, test case generation, and vulnerability detection. The framework consists of three core components: the *Relay-Pass Dynamic CGI Instrumentation*, which leverages a Debugger-Based CGI Wrapper to dynamically collect runtime basic block coverage data of CGI programs; the *LLM-Assisted Test Input Generator*, which ensures syntactically and semantically valid fuzzing inputs through a Semantic Packet Collector and an RFC-Conformant Test Input Filtration; the *Command Injection Enhanced Vulnerability Detector*, enabling the detection of memory corruption and command injection vulnerabilities in IoT devices. We implement a prototype of CGIFuzz and evaluated it on ten popular IoT devices. Our experiments demonstrated that CGIFuzz achieves 184.91% higher basic block coverage compared to the state-of-the-art gray-box fuzzer GDBFuzz and 119.25% higher compared to the state-of-the-art black-box tool BooFuzz. It successfully detected 69 vulnerabilities, including 13 previously unknown vulnerabilities, 9 of which were assigned CVEs.

In summary, this paper makes three contributions.

- We conduct an empirical vulnerability study that reveals the prevalence and critical severity of CGI program vulnerabilities in IoT devices, which highlights an aspect that has been overlooked in existing research. A comprehensive review of related research, along with the identified shortcomings in testing CGI programs is discussed.
- We design CGIFuzz, a novel framework that enables gray-box fuzzing for CGI programs of IoT devices. The framework introduces three key techniques: (1) a universal dynamic instrumentation method for CGI binaries on live IoT devices, (2) the leveraging of Large Language Models (LLMs) to generate syntactically and semantically valid test inputs, and (3) a specialized oracle for the accurate detection of both memory corruption and command injection vulnerabilities.
- We implement a prototype of CGIFuzz and evaluate it on ten popular IoT devices. Our experiments demonstrate the superior performance of CGIFuzz as it achieves 184.91% higher basic block coverage compared to the state-of-the-art gray-box fuzzer GDBFuzz and 119.25% higher compared to the state-of-the-art black-box tool BooFuzz. CGIFuzz detects a total of 69 vulnerabilities, among them 13 are previously unknown vulnerabilities, with 9 assigned CVEs.

## II. Background and Related Works

### A. Fuzz Testing for IoT Devices

Fuzz testing [18] has proven to be an effective method for detecting software vulnerabilities by sending randomized test inputs to the target software. Compared to black-box fuzzing, gray-box methods leverage execution feedback to enhance vulnerability detection effectiveness [19], [20], [21], [22], [23], [24], [25], [26]. While fuzzing has been applied in the IoT domain, existing methodologies are ill-suited for the unique challenges posed by Web CGI programs.

General-purpose fuzzers, such as AFL [27] and libFuzzer [28], are powerful for standalone binaries but inherently lack the mechanisms to handle the structured network communication required by the HTTP protocol. Similarly, advanced fuzzers designed for other specific IoT protocols, like LLMIF [29] for Zigbee, are methodologically incompatible. Their design is deeply coupled with the data formats and state machines of their target protocols (e.g., Zigbee frames), making them unsuitable for the distinct request-response architecture of HTTP-based web services. This fundamental mismatch prevents them from effectively interacting with a web service to test backend CGI applications.

Even fuzzers designed specifically for web protocols struggle when confronted with the unique execution model of CGI programs, which imposes a stringent “two-stage validation” process. An input must first pass the web server’s syntactic protocol check and then satisfy the CGI program’s internal

semantic logic. Black-box web fuzzers [10], [11], [30], [31], which typically use predefined packet templates, face a significant efficiency bottleneck against this dual barrier. Their mutation strategies generate a high volume of test cases that fail the initial syntactic validation, and the few that pass are unlikely to meet the semantic requirements of the CGI program. Consequently, this approach fails to achieve deep path exploration, resulting in very low testing efficiency.

Gray-box web fuzzers also face significant challenges. Applying them to IoT devices is complicated by proprietary source code, limited computational resources, and restricted permissions. Existing approaches often rely on emulators like QEMU [32] to gather coverage data [12], [15], [33], [34], [35]. However, the intricate dependencies between hardware and software in IoT devices make full-system emulation challenging, often resulting in environments that lack crucial real-world functionalities and thus limiting universality. Alternatively, debugger-based approaches like GDBFuzz [13], while more versatile, are incompatible with the "shadow calling" model of CGI execution. These fuzzers require direct control over the target process's lifecycle, which is impossible for transient CGI programs that are dynamically spawned by a web server. This inability to attach to and monitor the target CGI process prevents the collection of the very code coverage feedback essential for a gray-box fuzzer to operate effectively. These limitations highlight the need for a specialized framework designed to overcome the specific challenges of CGI fuzzing.

### B. Debugger-Based Instrumentation

Debuggers such as GDB are commonly used to monitor program execution and step through code at specific breakpoints. For targets with limited computational resources, debugging often involves setting up a debugger server, such as *gdbserver*, on the remote device and configuring a debugger client, such as *GDB*, to control program execution more flexibly [36]. This setup allows developers to test programs and apply universal debugging scripts with greater ease. Additionally, debuggers support cross-platform instruction sets, including MIPS, ARM®, and PowerPC, making them particularly well-suited for testing IoT devices.

Existing research [13], [17] has explored leveraging software debugging interfaces for fuzzing embedded systems. These techniques use breakpoint information from debuggers to collect code coverage data, enabling more efficient testing of embedded firmware. However, this approach encounters a fundamental conflict when applied to CGI programs. Debugger-based fuzzers require control over the target's lifecycle, typically by launching the program directly under the debugger's supervision. This is incompatible with the CGI execution model, where the program is dynamically spawned and managed by a web server in response to a request, which is the very "shadow-calling" feature we identified. Additionally, IoT devices typically lack accessible debugging options, making it difficult to extract internal data from hardware devices. As a result, debugging-based testing methods are generally limited to devices where debugging capabilities are enabled by default.

### C. LLM-Assisted Fuzz Testing

Recent studies have explored the potential of Large Language Models (LLMs) to enhance fuzz testing methodologies. Leveraging their exceptional performance in coding tasks, LLMs have been utilized to generate fuzz drivers by producing driver code based on provided code information and specific requirements. For instance, research such as TitanFuzz [37] and PromptFuzz [38] has demonstrated the use of LLMs to generate fuzz drivers.

LLMs have also been applied to the generation of test inputs or input generators. Specifically, GPTCombFuzz [39] employs LLMs to create suitable fuzz seeds to enhance the performance of AFL. CovRL-Fuzz [40] uses LLMs as test input generators while incorporating reinforcement learning to optimize mutation strategies. However, these approaches, while demonstrating the promise of LLMs, primarily focus on generating inputs for general-purpose libraries or standalone programs. We aim to address the significant, open problem of generating inputs for network-facing applications like CGI programs, which must satisfy both strict protocol syntax and complex, stateful application semantics.

TABLE I

STUDY OF CGI VULNERABILITY PREVALENCE AND SEVERITY ACROSS MAJOR IOT MANUFACTURERS

| Manufacturer | Total Vulns | CGI Vulns | CGI Proportion (%) | CGI Avg CVSS |
|---|---|---|---|---|
| Dlink | 1337 | 258 | 19.30 | 7.87 |
| ASUS | 400 | 59 | 14.75 | 7.57 |
| Linksys | 141 | 35 | 24.82 | 6.85 |
| TOTOLink | 691 | 164 | 23.73 | 8.55 |
| Trendnet | 145 | 44 | 30.34 | 8.46 |
| Belkin | 60 | 15 | 25.00 | 8.65 |
| QNAP | 336 | 11 | 3.27 | 8.81 |
| Synology | 375 | 50 | 13.33 | 7.15 |
| **Total** | 3486 | 636 | 18.24 | 7.99 |

### D. Vulnerability Study of CGI in IoT Devices

*1) CGI-Based IoT Web Server:* As shown in Fig. 1, a typical IoT web system consists of a web server and multiple CGI binary programs. The web server is responsible for parsing incoming HTTP requests and, based on the request, dispatching them to the appropriate CGI program for processing [3], [4]. This interaction follows a two-stage validation: the web server first ensures the request is syntactically correct according to protocol formats [41], after which the CGI program validates the semantic content, such as required parameters and input formats, before executing its core logic [42]. Executing with high privileges, these CGI programs then interact directly with the underlying operating system, making them a critical component whose security posture defines the overall security of the device.

*2) Severity of CGI Vulnerabilities:* To quantify the security risk posed by CGI programs in IoT devices, we conducted an empirical study of their vulnerabilities. We selected eight popular IoT manufacturers: D-Link, ASUS, Linksys, TOTOLink, Trendnet, Belkin, QNAP, and Synology. We collected a total of 3,486 CVEs reported for these manufacturers up to September 28, 2023, from the MITRE CVE database [43], [44], [45]. We then jointly performed an in-depth manual analysis of these reports to identify all vulnerabilities rooted in CGI programs. We also calculated the average Common Vulnerability Scoring System (CVSS) score for the identified CGI vulnerabilities and compared it against the overall average. The results are presented in Table I.

**Finding:** CGI programs represent a significant and severe threat vector in IoT devices. Our study reveals that **18.24%** of all analyzed vulnerabilities are related to CGI programs. Furthermore, these CGI-related vulnerabilities are substantially

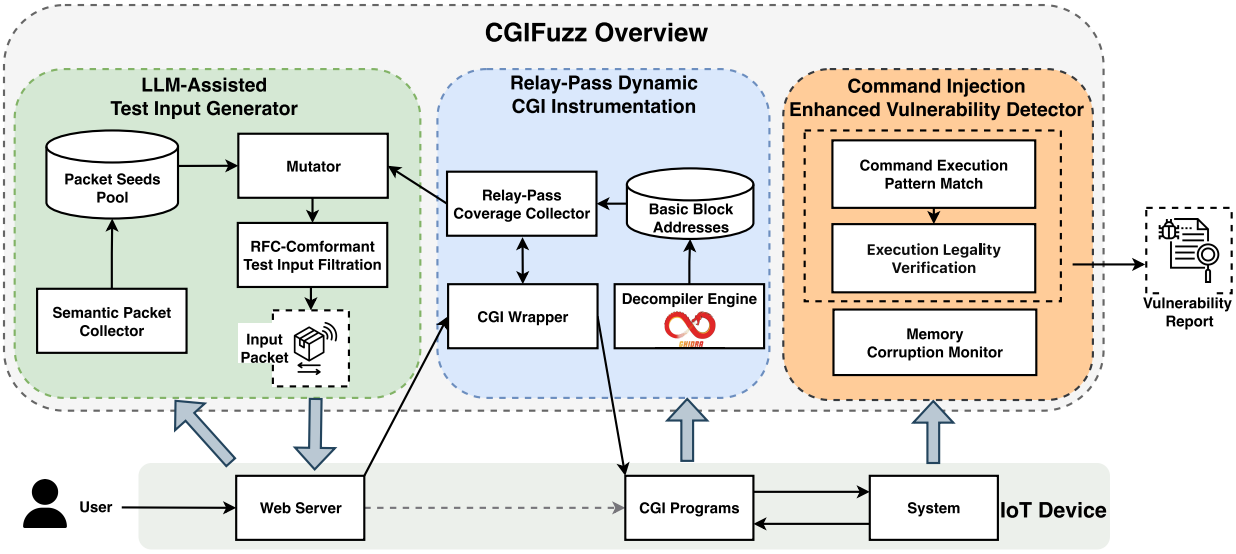


Fig. 2. Overview of CGIFuzz.

more severe, with an average CVSS score of **7.99**, indicating a high-risk profile [46].

Our findings indicate that CGI programs are responsible for nearly one-fifth of the security vulnerabilities in the IoT devices studied, underscoring their critical role in IoT security. The significantly higher average CVSS score is largely attributed to the high-privilege context in which these programs operate. This risk is compounded by several factors. Unlike open-source web servers that benefit from a large community for maintenance and extensive testing, CGI programs are typically developed in-house by device manufacturers. These programs are often closed-source, receive less rigorous security testing, and are execute with elevated privileges, increasing the risk of severe vulnerabilities like command injection stemming from inadequate input validation [47].

*3) Vulnerability Types in CGI Programs:* While many vulnerability types exist, memory corruption and command injection are among the most common and dangerous in IoT CGI programs. Memory corruption, such as a buffer overflow, occurs when a program writes data beyond an intended buffer, which can lead to a system crash. In contrast, command injection vulnerabilities [48] arise when untrusted user input is passed to sensitive system functions like *system()* or *popen()*, allowing an attacker to execute arbitrary commands, control the system, or exfiltrate data.

Detecting command injection is a known challenge. For instance, Witcher [14] attempts detection by replacing system binaries to transform errors into crashes, a method unsuitable for the read-only filesystems common in IoT devices. Black-box tools like BooFuzz [10] analyze device responses, but this is less effective for command injection, which may not produce an overt abnormal response. Other tools like SRFuzzer [49] inject specific payloads (e.g., using *ping* or *wget*) and monitors for their effects. However, triggering these vulnerabilities is often non-trivial due to constraints like input length limits and character filtering, making it difficult to uncover exploitable vulnerabilities with these methods.

## III. Design of CGIFuzz

In this section, we introduce the design details of CGIFuzz. The high-level architecture of CGIFuzz is illustrated in Fig. 2. CGIFuzz overcomes the challenges of applying coverage-guided fuzzing to CGI programs by introducing three key components. To ensure universal instrumentation of CGI programs, CGIFuzz implements the *Relay-Pass Dynamic CGI Instrumentation* (blue components in Fig. 2). To generate valid inputs for CGI programs, CGIFuzz implements the *LLM-Assisted Test Input Generator* (green components in Fig. 2). Finally, to facilitate the detection of command injection, CGIFuzz implements the *Command Injection Enhanced Vulnerability Detector* (brown components in Fig. 2). CGIFuzz is a gray-box web vulnerability scanner that uses a coverage-guided mutation fuzzer to drive the automated exploration of CGI programs in Linux-based IoT devices. Particularly, CGIFuzz collects basic block coverage data without relying on software simulation, making it widely applicable to both simulations and hardware devices. This overall design of interacting directly with the live web server, rather than simulating the CGI environment, ensures that CGIFuzz can test all parameter-passing mechanisms with high fidelity, whether they are standard or proprietary.

### A. Relay-Pass Dynamic CGI Instrumentation

The *Relay-Pass Dynamic CGI Instrumentation* provides coverage information by implementing dynamic instrumentation for CGI programs in three stages: enabling debugging functionality through the Debugger Configurator; deploying a CGI Wrapper to intercept program invocations; and using the Relay-Pass Coverage Collector to gather basic block data.

*1) Debugger Configurator:* As most IoT devices do not have GDB debugging functionality enabled by default, the Debugger Configurator employs a two-step process to enable debugging on IoT devices. The first step involves acquiring command execution capabilities on the device. To ensure broad applicability across various IoT devices, we employ a systematic, tiered strategy that proceeds with escalating invasiveness. We first attempt Software-Based Exploitation, the least invasive method, by leveraging known, publicly disclosed vulnerabilities. If software exploits are unavailable or ineffective, we proceed to Hardware-Based Access. This involves identifying and utilizing standard debugging interfaces,

such as UART or TTL ports, on the device's printed circuit board (PCB) to establish a shell session, a method that is common practice in IoT security research. As a final resort, we employ direct Firmware Modification, a process that involves physically extracting the firmware from its flash memory chip, modifying it to enable a root shell, and then re-flashing it onto the device. While this final method is the most invasive, its broad applicability is grounded in the typical security posture of our target devices. The vast majority of consumer and SOHO IoT devices are subject to significant cost and time-to-market pressures, which often results in a lack of sophisticated hardware security measures such as secure boot or robust firmware encryption [50], [51]. Large-scale studies have confirmed that unencrypted firmware is commonplace in the IoT landscape, ensuring that direct flash modification remains a highly viable last resort for gaining access even when other software and hardware interfaces are locked down [52]. Once command execution privileges are obtained, the subsequent step is to determine the device's instruction set architectures and upload the precompiled *gdbserver* binaries for the corresponding architecture to the device using built-in tools such as *wget*, *ftp*, or *tftp*. This process enables debugging functionality on the device.

*2) CGI Wrapper:* Most existing instrumentation methods for IoT devices are limited in their applicability due to their reliance on software simulation of the device. A novel instrumentation approach for embedded systems based on GDB is introduced in GDBFuzz [13]. GDBFuzz requires that CGI programs be launched under *gdbserver* attached to facilitate instrumentation. However, CGI programs on IoT devices are typically launched dynamically by web servers, involving complex threading and data transmission mechanisms. Moreover, significant implementation differences exist among various web servers (e.g., Nginx, Apache, Lighttpd). This diversity complicates the task of accurately capturing the launch timing of CGI programs and modifying their invocation methods. To address this, CGIFuzz constructs CGI Wrapper based on the web configuration file and LLM, which acts as an intermediary layer between the web server and the CGI program. The CGI Wrapper captures the critical moments when the CGI program is invoked and alters the launch method to the gdbserver-attached mode, enabling dynamic instrumentation. Specifically, based on gaining underlying command execution access to IoT devices, we have developed an automated method to deploy the CGI Wrapper. CGIFuzz locates the web process by identifying the web port (80 or 443) and then determines the web configuration file's name and location based on the process. It parses the web configuration file to specify settings related to the web server's invocation of CGI programs. Next, the tool modifies the configuration and deploys the CGI Wrapper to ensure that each CGI invocation is routed through the Wrapper. The CGI Wrapper, in turn, starts the respective CGI program in *gdbserver* mounting mode. Furthermore, due to significant differences in configuration file syntax among different web servers (e.g., Nginx, Apache, Lighttpd), LLM is used to automatically identify and modify key positions in the configuration file, thus supporting the wide range of complex web architectures. For instance, we provide the LLM with the server type (e.g., Nginx) and the goal (e.g., 'redirect all requests for login.cgi to a wrapper script'), and it generates the precise configuration syntax.

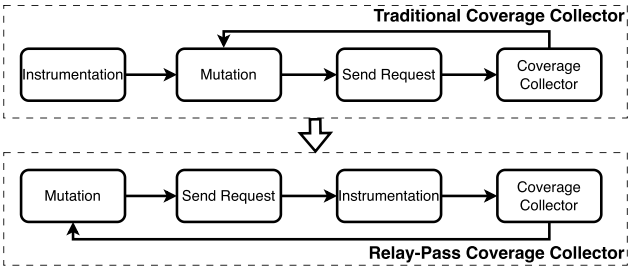


Fig. 3. The workflow of relay-pass coverage collector.

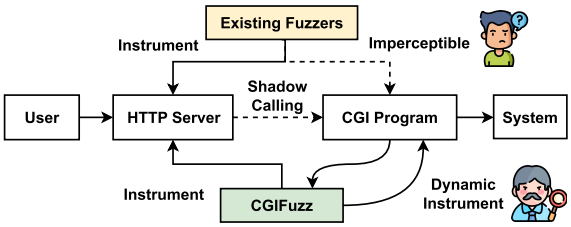


Fig. 4. The instrumentation of CGIFuzz and existing fuzzers.

*3) Relay-Pass Coverage Collector:* Upon receiving a request, the CGI program is launched in debug mode based on the CGI wrapper. The instrumentation breakpoints are deployed by the CGI Wrapper to get the basic block coverage of the request. After execution, the CGI program exits with the breakpoints automatically cleared. To support this dynamic behavior, in contrast to the workflow of traditional web fuzzing tools' pre-instrumentation, mutation, request sending, and coverage collection, the Relay-Pass Coverage Collector's workflow consists of mutation, request sending, dynamic instrumentation, and coverage collection, adapting to the CGI architecture, as is shown in Fig. 3. Our coverage collection follows a two-phase process. First, in a one-time offline analysis, we use the reverse engineering tool Ghidra [53] to disassemble the CGI programs, get the control flow graph (CFG), and extract all basic block addresses. Dominator relations are then used to optimize these addresses. Second, during the runtime fuzzing process, CGIFuzz dynamically deploys breakpoints at these pre-identified addresses for each request. The set of unique breakpoints triggered during an execution constitutes the code coverage data for that specific input. Unlike traditional fuzzing tools that require pre-instrumentation and program compilation, CGIFuzz offers an innovative solution for efficiently instrumenting CGI programs. As displayed in Fig. 4, while existing debugger-based approaches can instrument long-running processes like web servers, they cannot attach to the transient, server-spawned CGI programs.

## B. LLM-Assisted Test Input Generator

In the Shadow Calling architecture, the input needs to be syntactically valid to pass the web server protocol syntax validation. Meanwhile, the input needs to contain the correct combinations and values of required parameters to pass the pre-validation of CGI programs. Invalid packets are discarded and do not contribute to the increase of coverage. Thus, ensuring high-quality test inputs is important for CGI fuzzing. The LLM-Assisted Test Input Generator consists of two components to generate effective fuzz test input for CGI programs: ❶ the Semantic Packet Collector uses an

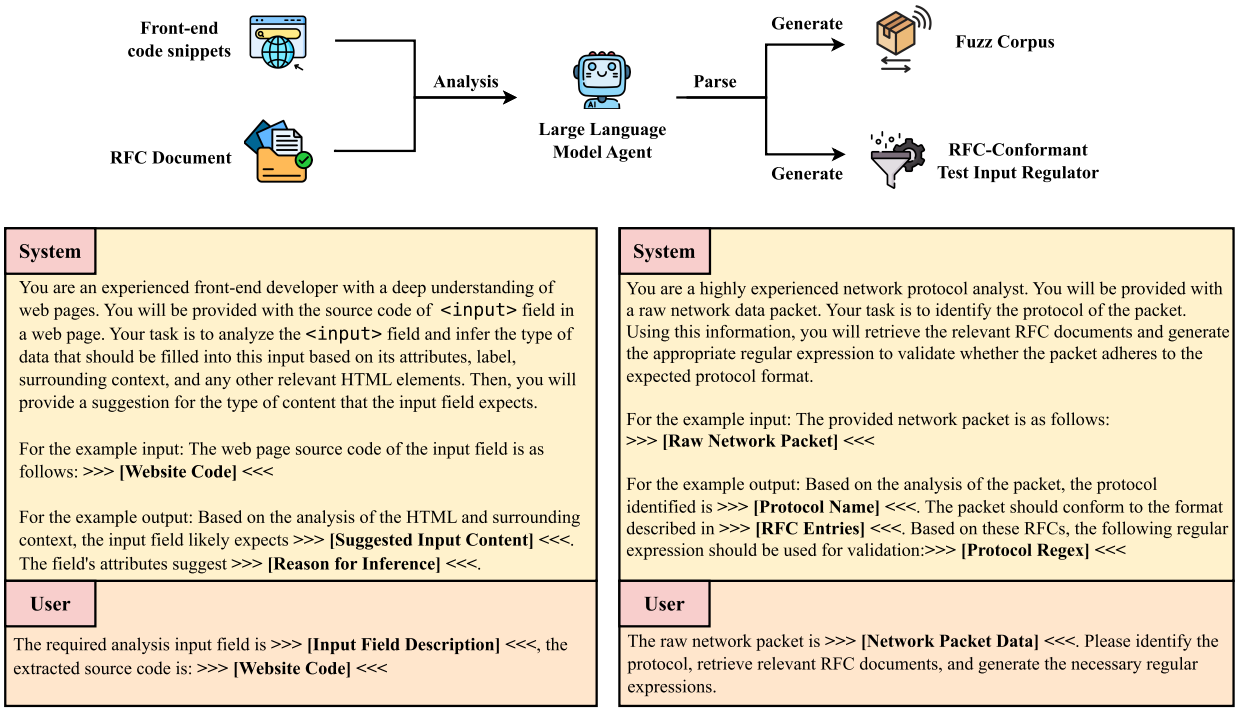


Fig. 5. Example prompt for LLM-Assisted fuzzing input generation.

```
POST /cgi-bin/cstecgi.cgi HTTP/1.1
Host: 192.168.0.1
Content-Length: 244
X-Requested-With: XMLHttpRequest
Accept-Language: zh-CN,zh;q=0.9
Accept: application/json, text/javascript, */*; q=0.01
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/130.0.6723.70 Safari/537.36
Origin: http://192.168.0.1
Referer: http://192.168.0.1/advance/l2tp.html?timestamp=1735909966333
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

Fig. 6. Example of the header of an IoT web packet.

```
{
    "enable": "1",
    "sip": "10.8.0.2",
    "eip": "10.8.0.51",
    "server": "10.8.0.1",
    "priDns": "8.8.8.8",
    "secDns": "10.20.0.1",
    "mtu": "1450",
    "mru": "1450",
    "ipsecL2tpEnable": "0",
    "ipsecPsk": "",
    "topicurl": "setL2tpServerCfg",
    "token": "ed0bc6a69d4a9564d26ee5649d3fc445"
}
```

Fig. 7. Example of extracted packet variables.

LLM-powered web crawler to collect all legitimate functional packets supported by the target system to be used as mutation seeds. The data mutation strategy from libfuzzer [28] is applied to perform data mutation. ❷ Since mutation operations may violate the original packet's syntax, the RFC-Comformant Test Input Filtration, based on retrieval-driven LLMs and RFC documents, is designed to filter the mutated packets, ensuring that each input sent to the target system is valid.

*1) Semantic Packet Collector:* The Semantic Packet Collector is an automated crawler with the LLM and playwright [54], designed to comprehensively gather different types of web packets accepted by the web system, which subsequently serve as seeds for fuzzing. The HTTP header shown in Fig. 6 and the HTTP parameters shown in Fig. 7 depict a CGI packet captured when manually accessing a real device. When specific values are manually input into the corresponding web page input fields, followed by clicking the submit button, the browser will automatically construct this packet and send it to the device's web system. Our idea is to automate the process of simulating human access to the web system and use web crawlers to collect all types of packets. The key to achieving this is the proper construction of HTTP parameters. We find that all submitted data can be classified into explicit and implicit parameters based on whether they are user-entered. In the example data packet, the parameters such as *sip*, *eip*, *server*, *priDns*, *secDns*, *mtu*, *mru* are explicit because they are manually entered by users according to the page information. The explicit parameters usually impose stringent requirements on the input format and content. For example, the prompt information on the example web page clearly states that *eip*, *server* and *priDns* should be of the IP address data type, and the value of *mtu* and *mru* must be between 1,400 and 1,500. In addition, there may be other complex data requirements. Users need to enter valid values that satisfy the requirement according to the page prompt information, or it will fail the dynamic validation of the front-end JavaScript code, preventing the generation of the data packet. To address these issues, the LLM's ability is used to understand frontend code to address the generation of explicit parameters. During crawling, the system parses the web page code, extracts the relevant segments associated with the input, and sends them individually to the LLM (GPT-4 model) to understand the web prompt information and generate a valid explicit parameter. Meanwhile, the example packet's parameters *topiurl* and *token* are implicit because they are dynamically generated through JavaScript code on the webpage instead of directly controlled by the user. To solve this, our collector employs a

synergistic workflow between the LLM and the Playwright browser automation framework. For explicit parameters, the LLM first analyzes the relevant HTML/JS snippets to generate valid input values. Next, Playwright, acting as the controller, programmatically fills the web form with these LLM-generated values and then simulates the necessary user action, such as clicking the 'Submit' button. This crucial step triggers the browser's native engine to execute all client-side JavaScript, which computes any implicit parameters (e.g., session tokens) and constructs the final, legitimate HTTP request. Finally, Playwright's network interception feature captures this fully-formed data packet. This hybrid approach allows us to leverage the LLM's semantic understanding while relying on a high-fidelity browser environment to handle complex, dynamic client-side logic, ensuring the reliable collection of high-quality seed packets. Overall, the Semantic Packet Collector simulates and automates user behavior when accessing pages in a browser, ensuring the correctness of both explicit and implicit parameters. Thus, it guarantees the integrity and efficient collection of web packets.

*2) RFC-Conformant Test Input Filtration:* The web modules of IoT devices adhere to the standardized HTTP/1.1 protocol [41]. Meanwhile, different devices may utilize data formats such as XML [55] or SOAP [56] for their parameters. During the fuzzing, non-compliant test inputs are typically rejected outright by the web server, thus considerable time is wasted parsing such invalid inputs. The RFC-Conformant Test Input Filtration leverages the LLM at its core to address this inefficiency. Initially, the regulator deduces specific protocol and data field formats from the collected legitimate packets. To achieve this, the regulator employs a Retrieval-Augmented Generation (RAG) methodology that leverages both protocol standards and real-world examples. First, we construct a knowledge base from standard protocol specifications (e.g., RFC 2616 for HTTP/1.1, RFC 7303 for XML). When analyzing a set of similar legitimate packets, the regulator retrieves the most relevant technical specifications from this knowledge base. These retrieved documents, along with the collection of packet examples, are then provided to the LLM as a rich, combined context. This augmented prompt guides the LLM to understand both the official protocol rules and the specific implementation's structure, enabling it to generate a highly accurate regular expression for validation. The detailed prompt for this process is illustrated in Fig. 5. The regulator employs these rules to pre-screen candidate test inputs, ensuring that only those meeting compliance criteria are forwarded to IoT devices for fuzzing. Furthermore, according to the RFC standard, for data packets of *POST* and *PUT* type requests, the value of the *Content-Length* header must accurately reflect the size of the submitted data. Therefore, an additional calibration step for data packets is executed to ensure the relevant input strictly adheres to the RFC standard. By reducing the computational overhead of processing invalid inputs, this regulator significantly enhances the efficiency of the fuzz testing process.

### C. Command Injection Enhanced Vulnerability Detector

Memory corruption vulnerabilities and command injection vulnerabilities are two of the most common and critical security issues in IoT devices [57]. In IoT device fuzz testing, memory corruption vulnerabilities are typically detected by leveraging segmentation fault signals. However, detecting command injection vulnerabilities remains challenging due to the absence of distinctive signatures. Some existing methods [16], [49] attempt to detect command injection vulnerabilities by injecting specially crafted payloads during input mutation and monitoring their execution outcomes. However, these methods often operate with coarse granularity, and the triggering conditions for command injection vulnerabilities can be intricate (e.g., constraints on input length, character filtering, or encoding), resulting in potential false negatives and incomplete detection. Recent research [14] proposed a novel method for detecting command injection, but it necessitates replacing the system's native command execution program (e.g., dash). This approach is impractical for IoT devices due to their read-only file systems and limited computational resources.

**Algorithm 1** Command Injection Enhanced Vulnerability Detection

**Require:** Test input corpus $\mathcal{C}$, Target device $D$, Sensitive function list $F$, Threshold $K$
**Ensure:** Reported vulnerabilities $\mathcal{V}$
Initialize vulnerabilities $\mathcal{V} = [\ ]$
**for** each *input* in $\mathcal{C}$ **do**
  *trace* ← execute(*input*, $D$)
  **if** *trace*.hasCrash() **then**
    $Vul$ ← type: Memory Corruption, input: *input*, info: *trace*.crashInfo
    $\mathcal{V}$.append($Vul$)
    **continue** Proceed to the next input
  **end if**
  *triggered_calls* ← *trace*.getSensitiveFunctionCalls($F$)
  **for** each *call* in *triggered_calls* **do**
    **Stage 1: Command Execution Pattern Match**
    *arguments* ← *call*.getArguments()
    *score* ← calculateLCS(*arguments*, *input*)
    **if** *score* > $K$ **then**
      **Stage 2: Execution Legality Verification**
      *returnValue* ← *call*.getReturnValue() Abnormal return value indicates a potential issue
      **if** *returnValue* ≠ 0 **then**
        $Vul$ ← type: Command Injection, function: *call*.name, input: *input*, ret: *returnValue*
        $\mathcal{V}$.append($Vul$)
      **end if**
    **end if**
  **end for**
**end for**
**return** $\mathcal{V}$

To address these limitations, we propose the Command Injection Enhanced Vulnerability Detector, a two-stage oracle detailed in Algorithm 1. Notably, our detection logic is not based on static text or regular expression matching. Instead, it identifies a dynamic, two-condition behavioral pattern at runtime. The two stages of our oracle directly implement this pattern: First, the Command Execution Pattern Match mechanism fulfills the first condition by identifying a critical data flow between test inputs and arguments of functions susceptible to command injection. Next, the Execution Legality Verification mechanism fulfills the second condition

by confirming a potential injection through the analysis of the function's execution result. A potential vulnerability is reported only when both conditions are met simultaneously, which facilitates a generalized and efficient detection of command injection vulnerabilities during the fuzz testing process for IoT devices.

*1) Command Execution Pattern Match:* The underlying principle is that if any part of a user-controllable input packet is passed as the arguments $c^p$ of the command-injection-sensitive functions $f_i \in F$ such as *system*, *execve*, and *popen*, a potential vulnerability exists. Based on this concept, whenever the program executes a command-injection-sensitive function, we extract its arguments and compute their correlation with the input packet. According to the results obtained from the correlation, we identify potential command injection vulnerability test cases. To this end, we first identify all the command-injection-sensitive functions in CGI programs denoted as $F = \{f_1, f_2, \ldots, f_n\}$. The CGI instrumentation approach described in Section III-A is used to instrument each triggered function $f_i$ and capture all arguments $C^p = \{c_1^p, c_2^p, \ldots, c_n^p\}$ from memory during testing. CGIFuzz then calculates the correlation $\sum_{i=1}^{n} R(c_i^p, x^p), \forall c_i^p \in C^p$ between the arguments $C^p$ and the test input $x^p$ to determine whether the arguments are related with the input packet. We apply Longest Common Subsequence (LCS) [58] as the relativity function $R$ in CGIFuzz. To achieve a higher identification accuracy during the testing, we set the empirical hyperparameter $K$[1] as the threshold for filtering command injection vulnerability candidates $x^c$.

$$x_i^c = \left\{ x_i^p \,\middle|\, \sum_{i=1}^{n} \mathrm{LCS}(c_i^p, x_i^p) > K \right\} \cup \varnothing, \forall c_i^p \in C^p$$

*2) Execution Legality Verification:* After identifying potential vulnerabilities, we conduct an in-depth analysis of command execution outcomes to minimize false positives. In legitimate, non-vulnerable cases, even if arguments originate from user input, they have passed strict validation and filtering. Thus, the executed commands are valid, and the sensitive functions typically exit normally with a return value of zero. Conversely, when a command injection vulnerability is present, fuzzed inputs often lead to malformed command strings. The execution of such malformed commands causes the sensitive function to terminate with a non-zero error code. Therefore, we use the function's return value to verify each candidate.

As the return value of a function is stored in a specific CPU register immediately after its execution, we use the CGI instrumentation approach in Section III-A to place a breakpoint on the instruction following the call to the sensitive function. This allows us to capture the return value from the designated register. Consequently, CGIFuzz examines the execution output status $f_i(c_i^p)$ for each $c_i^p \in C^p$. The return values of command executions from our candidates are parsed, and those candidates with significantly deviated return values from the successful exit value (0) will be reported as command injection vulnerabilities.

[1]We set the parameter $K$ as 6 in evaluations based on the empirical analysis of the vulnerability dataset presented in Table I.

## IV. Evaluation

In this section, we aim to answer the following questions through the evaluation of CGIFuzz:

- **RQ1**: How effective is CGIFuzz and what are the contributions of the three core components?
- **RQ2**: How effective is CGIFuzz when compared with state-of-the-art?
- **RQ3**: Can CGIFuzz discover real-world vulnerabilities?
- **RQ4**: How effective and practical is the integration of LLMs in the CGIFuzz framework?

### A. Evaluation Setup

*1) Fuzz Target Selection:* To evaluate CGIFuzz, we selected ten diverse IoT devices, as detailed in Table II. These devices include models from Cisco, D-Link, TOTOLink, Trendnet, Vivotek, and ipTime. Our selection process was guided by four principles to ensure both the suitability of the targets and the generalizability of our results.

- **Suitability:** All chosen devices are Linux-based, provide web services, and utilize CGI programs in their web architecture, aligning with CGIFuzz's design objectives.
- **Representativeness:** We focused on devices from well-known manufacturers with a significant market share [59], [60], which are frequently cited in related IoT security research [12], [16], [29], [49], [61], [62].
- **Methodology Diversity:** Our testing setup incorporated both physical hardware (six devices) and full-system emulation via FirmAE [63] (four devices) to demonstrate broad applicability.
- **Category Coverage:** We aimed for comprehensive coverage across common device types, including wireless routers, wireless access points (APs), Network-Attached Storage (NAS) devices, and network cameras.

It is noteworthy that several devices in our test set, particularly the D-Link DNS-320 and ipTIME C200, feature a modular architecture with numerous individual CGI programs. To focus our evaluation on the most significant components, we adopted a systematic target selection methodology for these cases. We prioritized CGI binaries based on their functional criticality (e.g., those handling authentication, system configuration, or file uploads) and their code complexity, for which we used the binary's file size as a practical proxy. This principled approach ensures that our fuzzing efforts are directed at the components most likely to contain impactful vulnerabilities.

*2) Environment Setup:* Experiments were conducted on an Ubuntu 22.04 machine (Intel i7-7700, 8GB RAM). Physical devices were connected via a wired network. A prerequisite for our testing on physical hardware was gaining initial command execution access. To achieve this, we systematically applied the tiered access strategy detailed in Section III-A. This approach proved successful across all ten devices in our evaluation. For transparency, Table III outlines the specific method used to gain a root shell for each device, empirically validating the practicality of our prerequisite access methodology. With root access established on each target, we then deployed the necessary `gdbserver` and CGI Wrapper, configured CGIFuzz, and initiated the tests.

*3) Vulnerability Confirmation:* We manually verified all generated alerts, confirming exploitability and cross-referencing against public CVEs. CGIFuzz identified 69 distinct vulnerabilities: 56 previously known and **13 zero-day**.

TABLE II
SUMMARY OF DEVICE TESTING INFORMATION

| Vendor | Model | CPU Arch | Version | Device Type | Tested Status |
|---|---|---|---|---|---|
| TOTOLink | X5000 | MIPS LSB | V9.1.0cu.2350 | Wireless Router | Physical Device |
| TOTOLink | A7000R | MIPS LSB | V9.1.0u.6268_B20220504 | Wireless Router | Physical Device |
| D-Link | DIR-505 | MIPS MSB | 1.06 | Wireless AP | FirmAE Emulation |
| D-Link | DNS320 | ARM LSB | 1.0 | Network Attached Storage | Physical Device |
| D-Link | DIR850L | MIPS MSB | 2.23 | Wireless Router | FirmAE Emulation |
| Cisco | RV110W | MIPS LSB | 1.2.2.5 | VPN Device | Physical Device |
| Cisco | RV340 | ARM LSB | 1.0.03.19 | VPN Device | Physical Device |
| Trendnet | TEW-827DRU | MIPS LSB | 2.06B04 | Wireless Router | FirmAE Emulation |
| Vivotek | C8160 | ARM LSB | 1.0 | Network Camera | FirmAE Emulation |
| ipTIME | C200 | MIPS LSB | 1.058 | Network Camera | FirmAE Emulation |

TABLE III
METHOD USED TO GAIN ROOT SHELL ACCESS FOR EACH DEVICE

| Vendor | Model | Method Used to Gain Access |
|---|---|---|
| D-Link | DIR-505 | Software-Based Exploitation (via a public exploit [64]) |
| D-Link | DNS-320 | Software-Based Exploitation (via a public exploit [65]) |
| TOTOLink | X5000 | Hardware-Based Access (via UART/TTL interface) |
| Cisco | RV110W | Hardware-Based Access (via UART/TTL interface) |
| Trendnet | TEW-827DRU | Hardware-Based Access (via UART/TTL interface) |
| ipTIME | C200 | Hardware-Based Access (via UART/TTL interface) |
| TOTOLink | A7000R | Firmware Modification (via flash memory re-flashing) |
| Cisco | RV340 | Firmware Modification (via flash memory re-flashing) |
| Vivotek | C8160 | Firmware Modification (via flash memory re-flashing) |
| D-Link | DIR-850L | Firmware Modification (via flash memory re-flashing) |

We responsibly disclosed the new findings, resulting in **9 assigned CVEs** with 4 pending.

### B. Evaluation of CGIFuzz Components (RQ1)

To evaluate the contribution of each core component to CGIFuzz's effectiveness, we conducted an ablation study using various configurations of our tool. The three core components refer to the Relay-Pass Dynamic CGI Instrumentation (RDCI), the LLM-Assisted Test Input Generator (LTIG), and the Command Injection Enhanced Vulnerability Detector (CIEVD). We design the following configuration options:

- **CGIFuzz**: The full framework including RDCI, LTIG, and CIEVD.
- **CGIFuzz-LC**: The configuration without RDCI (i.e., "Less Coverage")..
- **CGIFuzz-RC**: The configuration without LTIG (i.e., "Random Corpus").
- **CGIFuzz-RL**: The configuration without CIEVD (i.e., "Reduced Logic" for detection).

To comprehensively address RQ1, we evaluated these configuration options based on two key metrics: real-world vulnerability detection and basic block coverage. The experimental procedures are as follows: First, We deploy the four configuration options across ten different IoT devices and conduct fuzzing over a ten-day period. During this time, we recorded the number of vulnerabilities detected under each configuration option to compare their real-world vulnerability detection capabilities. The results are summarized in Table V, where the columns C, LC, RC, and RL represent CGIFuzz, CGIFuzz-LC, CGIFuzz-RC, and CGIFuzz-RL, Second, since vulnerability discovery can be influenced by random factors, we conducted additional experiments on some devices to measure the code coverage achieved by each component. These results, which more directly reflect a component's ability to explore new program states, are presented in Table IV. As the CIEVD component is a detection oracle and does not influence path exploration, this coverage experiment included only three configurations: CGIFuzz, CGIFuzz-RC, and CGIFuzz-LC. In coverage experiments, we ran each configuration three times and reported the best result.

*1) Evaluation of RDCI:* The RDCI component is designed to collect basic block coverage for CGI programs running on the target device. To assess the impact and effectiveness of this component, we compare the performance of the full CGIFuzz framework against CGIFuzz-LC, a variant that lacks the RDCI component and thus operates without coverage feedback. As shown in the Table V, the inclusion of RDCI leads to substantially better vulnerability detection. The full CGIFuzz framework successfully identified 69 vulnerabilities, while CGIFuzz-LC detected only 36, representing a 91.67% improvement. This trend is mirrored in code coverage, as shown in Table IV. CGIFuzz covered a total of 4795 basic blocks, whereas CGIFuzz-LC reached only 3798, demonstrating that RDCI improved basic block coverage by 26.25%. A deeper analysis of the results reveals the mechanism behind this improvement. We found that several zero-day vulnerabilities discovered by CGIFuzz, namely CVE-2024-32349, CVE-2024-32350, and CVE-2024-32351, could all be triggered by mutations of the same initial seed. Equipped with RDCI, CGIFuzz could recognize that this seed was exploring valuable new paths and therefore prioritized it, ultimately leading to the comprehensive discovery of these related vulnerabilities and higher overall coverage. In contrast, CGIFuzz-LC, lacking coverage feedback, treated the seed randomly and failed to explore its potential sufficiently. As a result, it only identified one of the vulnerabilities and achieved significantly lower coverage. These results clearly demonstrate the critical role of the RDCI component in enabling effective and deep fuzzing of CGI programs.

*2) Evaluation of LTIG:* The LTIG component is designed to generate fuzzing test input for web CGI programs. To isolate and measure its effectiveness, we established the CGIFuzz-RC configuration as a baseline. For this configuration, we initiated the fuzzing process with a single, legitimate packet captured from normal device interaction as the sole initial seed. The mutation was then handled by the mutator from libFuzzer. By comparing CGIFuzz (with LTIG) against CGIFuzz-RC, we can directly assess LTIG's contribution. The core hypothesis is that without LTIG's semantic guidance and RFC-compliant filtering, a generic mutator will quickly corrupt the packet's structure, leading to a high volume of invalid inputs rejected by the web server or CGI program, thus limiting deep path exploration. As shown in the Table V, CGIFuzz successfully detected 69 vulnerabilities, whereas CGIFuzz-RC identified only 6, demonstrating a substantial improvement in vulnerability detection capability. Regarding basic block coverage data, CGIFuzz achieved a cumulative coverage of 4795 basic

TABLE IV
CGI COVERAGE COMPARISON UNDER DIFFERENT CONFIGURATION OPTIONS

| Program | Device | CGIFuzz | CGIFuzz-RC | CGIFuzz-LC | GDBFuzz-SPC | GDBFuzz | Boofuzz |
|---|---|---|---|---|---|---|---|
| my_cgi.cgi | D-Link DIR505 | 1276 | 150 (+750.66%) | 491 (+159.88%) | 445 (+186.74%) | 137 (+831.89%) | 128 (+896.88) |
| cestcgi.cgi | TOTOLink A7000R | 145 | 72 (+101.39%) | 115 (+26.09%) | 57 (+154.39%) | 49 (+195.92%) | 123 (+17.89%) |
| cestcgi.cgi | TOTOLink X5000 | 1082 | 607 (+78.25%) | 1004 (+7.77%) | 157 (+589.17%) | 69 (+1468.12%) | 72 (+743.06%) |
| account_mgr.cgi | D-Link DNS320 | 908 | 263 (+245.25%) | 907 (+0.11%) | 442 (+105.43%) | 244 (+272.13%) | 594 (+52.86%) |
| app_mgr.cgi | D-Link DNS320 | 376 | 173 (+117.34%) | 372 (+1.08%) | 352 (+6.82%) | 348 (+8.05%) | 365 (+3.01%) |
| system_mgr.cgi | D-Link DNS320 | 1008 | 221 (+356.11%) | 909 (+10.89%) | 230 (+338.26%) | 112 (+800%) | 905 (+11.38%) |
| **Total** | - | 4795 | 1486 (+222.68%) | 3798 (+26.25%) | 1683 (+184.91%) | 959 (+400%) | 2187 (+119.25%) |

TABLE V
FOUND VULNERABILITY AMOUNT IN DIFFERENT CONFIGURATIONS. 'C' REFERS TO 'COMPLETE', 'B' REFERS TO 'BASE'

| **Device** | **CGIFuzz (Ours)** | | | | **GDBFuzz** | | **BooFuzz** |
|---|---|---|---|---|---|---|---|
| | **C** | **LC** | **RC** | **RL** | **SPC** | **B** | |
| TOTOLink X5000 | 19 | 10 | 2 | 0 | 0 | 0 | 1 |
| TOTOLink A7000R | 2 | 1 | 0 | 2 | 1 | 0 | 0 |
| D-Link DIR-505 | 3 | 2 | 0 | 2 | 1 | 0 | 1 |
| D-Link DNS320 | 23 | 10 | 1 | 2 | 1 | 0 | 1 |
| D-Link DIR850L | 2 | 1 | 1 | 2 | 0 | 0 | 0 |
| Cisco RV110W | 4 | 4 | 1 | 4 | 1 | 0 | 1 |
| Cisco RV340 | 4 | 2 | 1 | 1 | 2 | 0 | 1 |
| Trendnet TEW-827DRU | 8 | 4 | 0 | 3 | 1 | 0 | 0 |
| Vivotek C8160 | 2 | 1 | 0 | 1 | 1 | 0 | 0 |
| ipTIME C200 | 2 | 1 | 0 | 0 | 1 | 0 | 0 |
| **Total** | 69 | 36 | 6 | 17 | 9 | 0 | 5 |

blocks, compared to only 1486 by CGIFuzz-RC, representing a significant 222.68% improvement due to the LTIG component.

Further analysis of the experimental data revealed that CGIFuzz successfully detected the CVE-2024-50917 vulnerability, which CGIFuzz-RC failed to detect. CVE-2024-50917 is a 0-day vulnerability discovered in the D-Link DIR-850L device, caused by a buffer overflow in the *cgibin* program when processing UPnP requests. The vulnerability's root issue lies in the absence of validating the length of the *HTTP_SOAPACTION* parameter. Triggering this vulnerability requires the test packets to conform to HTTP syntax, the data field to satisfy SOAP formatting, and the values of parameters such as *Service* and *SOAPAction* to be semantically valid. However, during mutation, CGIFuzz-RC often generates overly random packets that disrupt the required syntax and fail to produce semantically valid parameter values. As a result, most test packets are discarded by the web server or CGI pre-check logic, preventing detection of this vulnerability. These experimental results demonstrate the critical role and effectiveness of the LTIG component.

*3) Evaluation of CIEVD:* The CIEVD is designed as an enhanced oracle for fuzzing. Unlike traditional vulnerability detectors in fuzzers, which typically identify only memory corruption vulnerabilities by monitoring for system error signals (e.g., segmentation faults) and timeouts, CIEVD extends this capability by incorporating a specialized analysis module for detecting command injection vulnerabilities. To evaluate the effectiveness of this enhancement, we compared the full CGIFuzz framework against CGIFuzz-RL, a configuration that includes all components except for CIEVD's command injection analysis. As shown in Table V, it shows that the full CGIFuzz discovered a total of 69 unique vulnerabilities, comprising 52 command injection and 17 memory corruption vulnerabilities. In contrast, CGIFuzz-RL was only able to identify the 17 memory corruption vulnerabilities, demonstrating the critical contribution and effectiveness of our enhanced detection design.

To further evaluate the precision of CIEVD, we analyzed all alerts generated during testing. Across all runs, CIEVD produced 113 unique alerts for potential vulnerabilities. Manual verification confirmed that 69 of these were true positives, corresponding to an overall false positive rate of 38.9%. We investigated the root causes of these false positives, which primarily originated from the command injection detection logic. A portion of them stemmed from transient environmental issues, such as network jitter, which we believe can be mitigated through improved engineering solutions. The more significant category of false positives arises from the misinterpretation of benign "injections" with legitimate non-zero return codes. For example, many devices feature a "ping test" utility where user-provided IP addresses are passed to a command like *system("ping -c 4 user_ip")*. If a fuzzer provides an unreachable IP address, the system function correctly returns a non-zero value to indicate failure. Our detector flags this scenario because it correctly identifies both the data flow from user input to a sensitive function and the subsequent abnormal return. While our detector correctly identifies the data flow and abnormal return, this specific scenario is not an exploitable vulnerability. Such cases, where legitimate user-input errors mimic command injection failures, were found to be relatively infrequent. We contend that the current false positive rate is acceptable for a practical vulnerability discovery tool, though future work could still explore deeper semantic analysis to further enhance precision. These results show the important role of the CIEVD component in enabling CGIFuzz to detect command injection vulnerabilities. A detailed comparison of CIEVD with other related approaches is presented in Section IV-C.3.

All three core components of CGIFuzz contribute significantly to its overall performance, each playing a distinct role. The RDCI is fundamental for achieving effective path coverage for CGI programs, thereby facilitating the comprehensive exploration of potential execution paths and promising seeds. The LTIG is crucial for synthesizing semantically and syntactically valid test inputs, which is paramount for CGI programs given their stringent data format and semantic content requirements. The CIEVD, unlike traditional fuzzing tools primarily focused on memory corruption, provides specialized detection capabilities for command injection vulnerabilities, a particularly pervasive and critical class of flaws in IoT devices. Our ablation study reveals varying impacts from these components. Quantitatively, the LTIG generally contributes more significantly to path coverage improvements than the RDCI. This can be attributed to the fact that without LTIG, the fuzzer (e.g., CGIFuzz-RC) may generate a high volume of

```
// Get the value of the 'overwrite' with a length
    not exceeding 8
11 cgiFormString("overwrite", v3, 8);
// Concatenate the obtained value to a command
13 sprintf(s, "account_mgr -f %s -t %s -o %s", "/tmp
    /import_users", v2, v3);
// Directly execute the command
14 system(s)
```

Fig. 8. A command injection case of D-Link DNS320.

invalid inputs, which are swiftly discarded by the web server or CGI pre-checks, thus impeding efficient coverage exploration. Conversely, the CIEVD component operates as a functional enhancement rather than a performance optimization. Consequently, it does not directly influence path coverage. Its contribution to the total number of detected vulnerabilities is directly correlated with the prevalence of command injection vulnerabilities within the target devices.

### C. Comparison With Baselines (RQ2)

To evaluate the advancement and effectiveness of CGIFuzz, we performed a comparative analysis against state-of-the-art fuzzers. Our selection of baselines was guided by a core principle: to ensure a fair and relevant comparison, the chosen tools must align with CGIFuzz's primary objective of testing web CGI programs on IoT devices, particularly on physical hardware without depending on full-system emulation. Consequently, we selected the following baselines:

- **GDBFuzz** [13]: As the state-of-the-art gray-box fuzzer that uses a debugger-based approach compatible with physical devices, GDBFuzz is the most suitable counterpart to CGIFuzz. As discussed in Section II, most other gray-box fuzzers are emulation-based and cannot run on our target hardware.
- **BooFuzz** [10]: As our black-box baseline, we selected BooFuzz, a mature and widely-adopted network protocol fuzzing framework. It can be readily deployed against diverse hardware, and its template-based nature allows for crafting inputs appropriate for CGI testing.
- **SRFuzzer** [49]: To provide a direct and focused comparison for our command injection detection capabilities, we implemented the core principles of SRFuzzer. As a well-known fuzzer specialized in finding command injections in IoT routers, it serves as an ideal specialized baseline for evaluating our CIEVD component.

The evaluation was conducted based on two key metrics: code coverage and the number of vulnerabilities detected. The results are detailed in Table V and Table IV, and the coverage trend is illustrated in Figure 9.

*1) Comparison With Gray-Box Fuzzers:* As established, we chose GDBFuzz as our gray-box baseline because its debugger-based approach supports physical devices, aligning with our objectives. However, since GDBFuzz does not support the "shadow calling" characteristic of CGI programs, it cannot directly instrument them. To accommodate this, we adjusted its instrumentation target to the web server in our experiments. Furthermore, GDBFuzz lacks a component for generating semantically-aware inputs, like our LLM-Assisted Test Input Generator (LTIG). As demonstrated in our RQ1 analysis (Section IV-B.2), the absence of semantically valid inputs significantly reduces fuzzing effectiveness. Therefore, a direct comparison with the original GDBFuzz would be unfair. To create a stronger baseline, we enhanced GDBFuzz with our Semantic Packet Collector (SPC) functionality, creating a new configuration named GDBFuzz-SPC.

The experimental results in Table V show that CGIFuzz significantly outperforms this enhanced baseline. In terms of vulnerability detection, CGIFuzz identified 69 vulnerabilities, nearly eight times more than the 9 vulnerabilities found by GDBFuzz-SPC. Regarding code coverage, the results in Table IV show that CGIFuzz achieves 184.91% higher CGI code coverage compared to GDBFuzz-SPC, and a 400% improvement over the original GDBFuzz.

Further analysis attributes the poor performance of the GDBFuzz-based approaches to their lack of adaptability to CGI programs. Testing CGI programs requires inputs that are not only syntactically valid but also semantically coherent to effectively trigger and explore deep code paths. The test data generated by the original GDBFuzz often fails these constraints. Although GDBFuzz-SPC starts with valid seeds, its generic, unconstrained mutation process quickly corrupts the packet structure. Crucially, neither GDBFuzz nor GDBFuzz-SPC can collect feedback from the internal paths of the CGI programs themselves, making their exploration relatively blind. Moreover, both are unable to detect the command injection vulnerabilities prevalent in our target devices.

Notably, a comparison between GDBFuzz-SPC and CGIFuzz-LC offers a key insight. The primary difference is that the former instruments the web server while the latter uses no instrumentation at all (but benefits from LTIG). As shown in Table IV, CGIFuzz-LC still outperforms GDBFuzz-SPC in achieving CGI code coverage. This finding strongly suggests that feedback-guided fuzzing based on the coverage of the web server is largely ineffective for enhancing the exploration of back-end CGI programs.

*2) Comparison With Black-Box Fuzzers:* We selected BooFuzz as the black-box baseline due to its wide adoption and its flexibility for deployment on both emulated and physical devices. BooFuzz's fuzzing efficiency heavily relies on the quality of manually crafted templates. To ensure a fair comparison, we invested effort in creating high-quality templates that reflect a solid understanding of the target CGI program's expected inputs.

The experimental results in Table V show a stark difference in performance. CGIFuzz identified 69 vulnerabilities, more than 13 times the 5 vulnerabilities detected by BooFuzz. Regarding code coverage, as detailed in Table IV, CGIFuzz achieved 119.25% higher CGI code coverage compared to BooFuzz.

Further analysis reveals that BooFuzz's limitation stems from its black-box nature: it lacks any understanding of the target's internal state and logic, preventing it from generating semantically valid test packets for complex cases. For instance, successfully triggering the CVE-2024-32351 vulnerability requires the test input to contain specific values for fields like *topicurl* and *enable*. As a black-box tool, BooFuzz has no way to infer these semantic requirements. Consequently, its randomly generated inputs consistently fail the program's validation checks, making it incapable of detecting such vulnerabilities and leading to its overall weaker performance.

*3) Comparison Regarding Command Injection Vulnerability Identification Capabilities:* To specifically evaluate our *Command Injection Enhanced Vulnerability Detector* (CIEVD), we

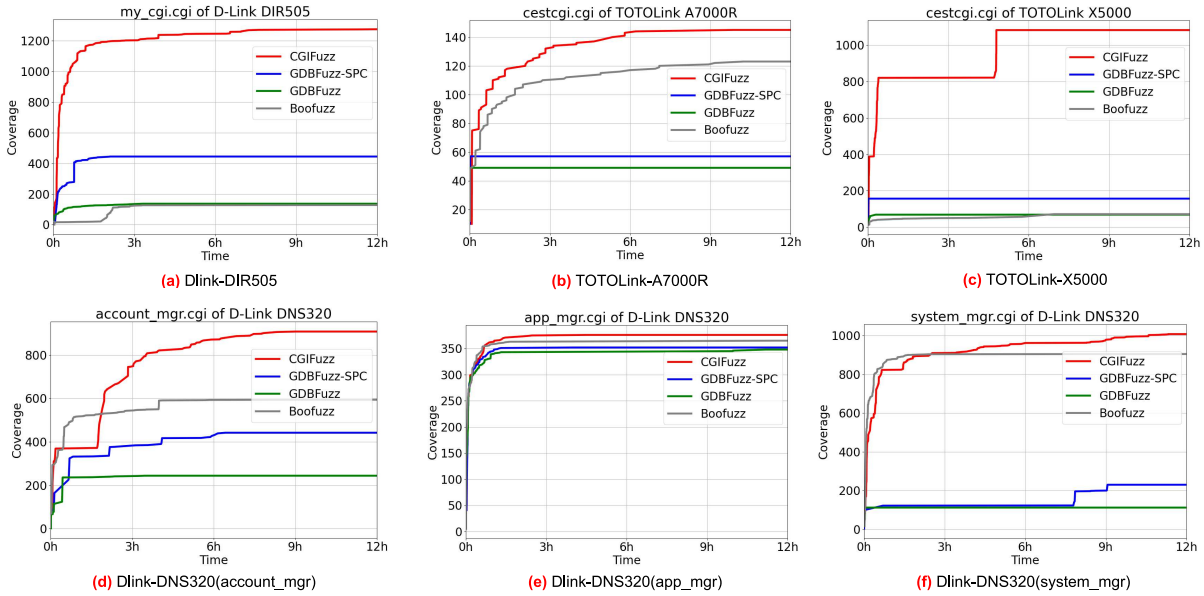


Fig. 9. Coverage of CGIFuzz compared to baselines.

selected SRFuzzer [49] as a specialized baseline. SRFuzzer is a well-regarded fuzzing tool for IoT devices, known for its strong performance in command injection vulnerability detection. Its core methodology involves injecting out-of-band interaction payloads (e.g., using *wget*) and monitoring for network callbacks to confirm a successful injection.

Since SRFuzzer's source code is not publicly available, we re-implemented its vulnerability detection module based on its published principles. Our implementation constructs a payload string such as *;wget http://192.168.0.10:8000/index.html;* and randomly inserts it into various input fields during mutation. A listening server on our local machine (192.168.0.10) monitors for incoming network traffic to identify successful injections.

To ensure a fair, "apples-to-apples" comparison between the two detection oracles, it is crucial to eliminate differences in input generation capabilities. We therefore created a hybrid baseline, CGIFuzz-SR, which combines CGIFuzz's advanced input generation and coverage guidance engine with our implementation of SRFuzzer's detection logic. This allows us to compare the effectiveness of our CIEVD against SRFuzzer's principles under an identical, high-quality stream of test cases.

The experimental results reveal a significant performance gap between the two detection oracles. CGIFuzz's CIEVD detected a total of 52 command injection vulnerabilities. In contrast, the SRFuzzer-based oracle in CGIFuzz-SR detected only 30 vulnerabilities, all of which were also found by CGIFuzz. This indicates that CGIFuzz's detection mechanism is approximately 1.73 times more effective.

Further analysis highlights an inherent limitation in SRFuzzer's out-of-band detection methodology: its difficulty in identifying command injection vulnerabilities that are subject to length restrictions. For example, a zero-day vulnerability we discovered in the D-Link DNS320 device, shown in Fig. 8, illustrates this weakness. The vulnerable code uses the function *cgiFormString("overwrite", v3, 8)*, which extracts the value of the *overwrite* parameter into the variable *v3* but truncates it to a maximum of 8 characters. Subsequently, *v3* is concatenated into a command string and executed by the *system()* function, creating a classic command injection scenario.

When testing this, the SRFuzzer-based approach injects its payload, *;wget http://...;*. However, due to the length restriction, this payload is truncated to just the first 8 characters, *;wget ht*. This truncated string is syntactically invalid as a command and thus fails to trigger the expected network request, causing the SRFuzzer oracle to miss the vulnerability entirely. This case highlights a fundamental limitation of detection methods that rely on the successful execution of a lengthy, externally-visible payload. In contrast, CGIFuzz's CIEVD, by analyzing internal memory data and CPU register values (i.e., the arguments to and return value from *system()*), is independent of the payload's successful execution. This capability underscores its superiority in handling complex, real-world command injection scenarios.

### D. Reveal of Real-World Vulnerabilities (RQ3)

During the evaluation, CGIFuzz identifies a total of 69 vulnerabilities across ten different IoT devices, as detailed in Table VI. These include 52 command injection vulnerabilities and 17 memory corruption vulnerabilities, showcasing CGIFuzz's ability to uncover diverse vulnerability types. After cross-referencing our findings with publicly available CVE information, we confirmed that 56 of these were previously known vulnerabilities. More importantly, we discovered **13 previously unknown (zero-day) vulnerabilities**. Following a responsible disclosure process with the device manufacturers and CVE organizations, we have successfully been assigned **9 new CVE identifiers**, with 4 more currently under review. The distribution of confirmed vulnerabilities by CVSS base score reveals 10 low-severity vulnerabilities (CVSS $<$ 4.0), 9 medium-severity vulnerabilities (CVSS 4.0–6.9), and 33 high-severity vulnerabilities (CVSS $\geq$ 7.0). The average CVSS base score for all confirmed vulnerabilities is 7.97, indicating a high proportion of critical findings.

TABLE VI
THE VULNERABILITIES DISCOVERED BY CGIFUZZ CATEGORIZED BY TYPE AND CVSS SCORE, WITH UNIQUE VULNERABILITIES HIGHLIGHTED

| Device | Vulnerabilities Discovered | |
|---|---|---|
| | **Command Injection** | **Memory Corruption** |
| TOTOLink X5000 | **CVE-2024-32349 (6.0)**, **CVE-2024-32350 (8.8)**, **CVE-2024-32351 (8.8)**, **CVE-2024-32352 (8.8)**, **CVE-2024-32353 (9.8)**, **CVE-2024-32354 (6.0)**, **CVE-2024-32355 (8.0)**, CVE-2024-42736 (7.8), CVE-2024-42737 (8.8), CVE-2024-42738 (8.8), CVE-2024-42739 (8.8), CVE-2024-42740 (6.8), CVE-2024-42741 (8.8), CVE-2024-42742 (8.8), CVE-2024-42743 (8.8), CVE-2024-42744 (8.8), CVE-2024-42745 (8.8), CVE-2024-42747 (8.8), CVE-2024-42748 (8.8) | |
| TOTOLink A7000R | | CVE-2024-7213 (8.8), CVE-2024-7212 (8.8) |
| D-Link DIR-505 | **2 additional CVEs pending** | CVE-2014-3936 (10.0) |
| D-Link DNS320 | CVE-2019-16057 (9.8), CVE-2020-25506 (9.8), CVE-2014-7857 (9.8), CVE-2014-7859 (9.8), CVE-2024-7828 (8.8), CVE-2024-7829 (8.8), CVE-2024-7830 (8.8), CVE-2024-7832 (8.8), CVE-2024-8210 (6.3), CVE-2024-8211 (6.3), CVE-2024-8213 (6.3), CVE-2024-8214 (6.3), CVE-2024-8127 (6.3), CVE-2024-8128 (6.3), CVE-2024-8129 (6.3), CVE-2024-8130 (6.3), CVE-2024-8131 (6.3), CVE-2024-8132 (6.3), CVE-2024-8133 (6.3), CVE-2024-8134 (6.3), **2 CVEs pending** | CVE-2024-7831 (8.8) |
| D-Link DIR850L | | **CVE-2024-50917 (reserved)**, **CVE-2024-50918 (reserved)** |
| Cisco RV110 | | CVE-2020-3331 (9.8), CVE-2020-3145 (8.8), CVE-2020-3146 (8.8), CVE-2020-3323 (9.8) |
| Cisco RV340 | CVE-2022-20707 (10.0), CVE-2020-3451 (4.7), CVE-2022-20801 (4.7) | CVE-2022-20753 (4.7) |
| TRENDnet TEW-827DRU | CVE-2020-14081 (8.8), CVE-2019-21270 (reserved), CVE-2019-21271 (reserved) | CVE-2020-14076 (8.8), CVE-2020-14077 (8.8), CVE-2020-14078 (8.8), CVE-2020-14079 (8.8), CVE-2020-14080 (9.8) |
| VIVOTEK C8160 01.00 | CVE-2024-7440 (6.3) | CVE-2024-7439 (8.8) |
| ipTIME C200 | CVE-2021-26614 (7.5), CVE-2020-7848 (8.0) | |

The results demonstrate the effectiveness and significance of CGIFuzz in detecting both known and unknown, high-impact vulnerabilities in real-world CGI-based IoT devices.

*1) Case Study:* To illustrate CGIFuzz's capabilities, we examine **CVE-2024-32351** [66], a zero-day vulnerability we discovered. It is a critical-level (CVSS 8.8) command injection vulnerability (CWE-77 [67]) in the TOTOLINK X5000R router, which could allow a remote attacker to execute arbitrary commands with root privileges, potentially leading to a complete device takeover.

As shown in Fig. 10, the vulnerability's root cause is the direct and unsanitized concatenation of the user-controlled *mru* parameter into a command string that is later executed by the *system()* function. Discovering this vulnerability presented a dual challenge that CGIFuzz is uniquely designed to overcome. First, reaching the vulnerable code path requires satisfying specific contextual preconditions (e.g., setting the *topicurl* parameter to *setL2tpServerCfg* and *enable* to *1*). Second, because this type of command injection does not cause a service crash, its detection requires a specialized, non-crash-based oracle.

In our evaluation, only CGIFuzz successfully identified this vulnerability. Its *LLM-Assisted Test Input Generator* was able to produce the semantically valid inputs required to satisfy the preconditions and reach the vulnerable code. Concurrently, its *Command Injection Enhanced Vulnerability Detector* correctly identified the data flow from the user-controlled input to the sensitive function and flagged the resulting abnormal return value, thereby confirming the vulnerability.

In contrast, the baseline tools failed due to their inherent limitations. GDBFuzz-SPC, with its structure-unaware mutation strategy, could not consistently generate inputs satisfying the strict syntactic and semantic requirements. The black-box fuzzer, BooFuzz, lacking insight into the program's internal

```
// The main function of cestcgi.cgi
1   int main()
2   {
338     v6 = cJSON_Parse(input);
340     if( v6 ){
342         topicurl = websGetVar(v6, "topicurl", ""
    );
354         if !strncmp(topicurl, "setL2tpServerCfg"
     ,64){ // Must satisfied context requirement (1)
355             vuln_func(request)
359         }
408     }
419 }

// The vulnerable function of cestcgi.cgi
1   int vuln_func(request)
2   {
19      enable = websGetVar(request, "enable", "0")
    // Must satisfied context requirement (2)
25      mru = websGetVar(request, "mru", "")
29      if ( atoi(enable) == 1 ){
40          Uci_Set_Str(24, "xl2tpd",  "
    ipsec_l2tp_xauth_psk", mru); // Concat variables
54      }
72  }

// the function Uci_Set_Str is defined in
    libcscommon.so
1   int Uci_Set_Str(flag, name1, name2, value){
371     vsnprintf(buf, length, "uci -c %s set %s.%s
    .%s=\"%s\"", v6, v7, name1, name2, value)
380     system(buf) // Execute injected command
392 }
```

Fig. 10. Detail of CVE-2024-32351.

state, was unable to fulfill the specific contextual conditions needed to navigate to the vulnerable logic. Moreover, even if these tools were to accidentally trigger the vulnerability, their crash-oriented oracles would likely have missed it.

### E. Discussion on the Practicality of LLM Integration (RQ4)

In our design of CGIFuzz, the LLM serves as a key

discussion on its effectiveness, operational costs, and reliability. Instead of being used as a general-purpose reasoning engine, the LLM is applied to three specific, independent, and well-defined tasks.

*1) LLM Task Specification:* Our framework leverages the LLM for the following tasks:

- *Automating Web Server Configuration Modification:* The LLM automates the modification of diverse web server configuration files to intercept CGI program calls. For instance, when provided with a device's `lighttpd.conf` file, the LLM can be prompted to redirect all CGI requests to our wrapper script, returning the correctly modified configuration file. This automates a process that would otherwise require security experts to manually study specific documentation for various servers (e.g., Apache, Nginx).
- *Generating Semantically Valid Inputs:* The LLM analyzes webpage source code to infer the expected format of user inputs for generating high-quality initial seeds. For example, by analyzing the HTML snippet for a "MAC Address" input field, including its `< label>` and `maxlength=''17''` attribute, the LLM correctly infers the expected format (e.g., `00:1A:2B:3C:4D:5E`). This ensures the seeds can pass the CGI program's initial semantic validation.
- *Generating Protocol Validation Filters:* The LLM creates protocol-specific validation filters to ensure that mutated test cases remain syntactically correct. Given a sample HTTP request, the LLM identifies the protocol, References the relevant RFCs (e.g., RFC 2616, RFC 8259), and generates a precise regular expression to validate all subsequent mutated packets.

*2) Effectiveness and Empirical Validation:* The effectiveness of this integration is validated both qualitatively and quantitatively. Qualitatively, as mentioned, it automates complex, expert-level tasks. Quantitatively, the contribution of the LLM-powered components forming our LLM-Assisted Test Input Generator (LTIG) is empirically validated by our ablation study. As shown in Table IV and Table V, the full CGIFuzz framework achieved 222.68% higher path coverage compared to CGIFuzz-RC, which lacks the LLM components. Furthermore, it discovered 69 unique vulnerabilities, a significant increase over the 6 found by CGIFuzz-RC. This strongly validates our LLM integration.

*3) Operational Cost:* A critical aspect of our design is the management of operational costs, encompassing both API token consumption and time latency. All LLM-driven tasks are intentionally confined to the one-off or low-frequency pre-processing stage, completely avoiding the high-frequency fuzzing loop. To quantify this, we estimated the token consumption based on our experiments. For initial seed generation, analyzing a webpage with approximately 20 input fields required about 4000-8000 tokens; for a device with 50 distinct pages, this one-time cost is manageable. The modification of a typical server configuration file (e.g., `lighttpd.conf`) consumed around 2000-6000 tokens in a single call. Finally, generating a protocol filter via our RAG approach required less than 100,000 tokens per protocol. These minimal, low-frequency costs are well-justified by the substantial improvements in automation and subsequent fuzzing efficiency.

*4) Reliability and Inaccuracy Mitigation:* We proactively address the reliability of LLM outputs with a two-fold strategy. First, we minimize the risk of "hallucinations" by decomposing complex problems into minimal, well-defined sub-tasks with bounded inputs. Second, and more importantly, we mitigate potential inaccuracies using a closed-loop validation mechanism for each task. For server configurations, any LLM-generated output is first validated using the server's native syntax-checking tool (e.g., `lighttpd -t -f ...`) before deployment. For generated inputs, we leverage the target application's own client-side (JavaScript) and server-side validation logic as an implicit and effective check. For protocol filters, the generated regular expressions are verified against a pre-defined set of both valid and invalid packet samples before being deployed in the fuzzer.

## V. Discussion

A limitation of our approach is the performance overhead inherent to our debugger-based instrumentation, which represents a deliberate trade-off for broader applicability. Our expriments shows that this method introduces a latency increase compared to uninstrumented execution. However, this overhead is the necessary price for enabling gray-box fuzzing on source-unavailable, physical IoT devices where faster, traditional instrumentation is not an option. The resulting vulnerability discoveries validate the effectiveness of this trade-off. Future work will focus on mitigating this overhead by exploring more efficient communication channels to replace network sockets and by leveraging GDB's tracepoint feature to reduce communication frequency. Furthermore, CGIFuzz's applicability is predicated on gaining initial shell access to the device. While our multi-pronged strategy proved effective for the consumer-grade devices in our evaluation, this prerequisite might not be met on highly-hardened systems with advanced physical security countermeasures.

This research employs GDB for the dynamic instrumentation of binary CGI programs, establishing a general-purpose fuzzing framework. Similar debugging tools exist for software systems written in other languages, such as Java [68], Python [69], and PHP [70], with corresponding debuggers like JDB [71], PDB [72], and Xdebug [73]. We intend to expand our framework to support testing the web systems of these diverse ecosystems.

More broadly, the central concept of CGIFuzz is to identify and target the weakest link in a complex system's security chain. This methodology, which involves analyzing a system to find discrepancies in security posture among its components and then focusing testing efforts on these weak points, is highly generalizable. We believe this approach can be extended beyond IoT to large-scale software, operating systems, and cloud services.

## VI. Conclusion

In this paper, we propose CGIFuzz, a new gray-box fuzzing framework designed for testing CGI programs of Linux-based physical IoT devices. CGIFuzz automatically captures CGI programs for dynamic instrumentation. To improve the efficiency of IoT web fuzzing, CGIFuzz leverages LLM to provide semantic-aware seed packets and the RFC-conformant test input filters to filter out invalid inputs. Moreover, CGIFuzz enables the efficient detection oracle of the command

injection vulnerability in IoT devices. Our experiment demonstrates the effectiveness of CGIFuzz, which outperforms state-of-the-art fuzzers in both basic block coverage and vulnerability-detecting ability. In total, CGIFuzz discovered 69 vulnerabilities across ten popular IoT devices, including 13 previously unknown vulnerabilities (9 of which were assigned CVEs), whereas GDBFuzz found only 9 vulnerabilities in total.

## REFERENCES


[1] K. Rose et al., "The Internet of Things: An overview," in *Proc. ISOC*, 2015, pp. 1–54.

[2] S. Panjwani, S. Tan, K. M. Jarrin, and M. Cukier, "An experimental evaluation to determine if port scans are precursors to an attack," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, 2005, pp. 602–611.

[3] R. Brinsley, *Common Gateway Interface (CGI)*. Boca Raton, FL, USA: CRC Press, 2002.

[4] C. Kou and F. Springsteel, "The security mechanism in the world wide web (WWW) and the common gateway interface (CGI). Example of central police university entrance examination system," in *Proc. IEEE 31st Annu. Int. Carnahan Conf. Secur. Technol.*, Sep. 1997, pp. 114–119.

[5] R. T. Fielding and G. Kaiser, "The apache HTTP server project," *IEEE Internet Comput.*, vol. 1, no. 4, pp. 88–90, Apr. 1997.

[6] Embedthis. (2025). *Embedded Web Server-GoAhead*. [Online]. Available: https://www.embedthis.com/goahead/

[7] W. Reese, "Nginx: The high-performance web server and reverse proxy," *Linux J.*, vol. 2008, no. 173, p. 2, Sep. 2008.

[8] loginsoft. (2024). *Introduction To Common Gateway Interface and CGI Vulnerabilities*. [Online]. Available: https://www.loginsoft.com/post/introduction-to-common-gateway-interface%

[9] N. N. Sulthan. (2014). *Common Gateway Interface Vulnerability*. [Online]. Available: https:// beaglesecurity.com/blog/vulnerability/common-gateway-interface.html

[10] J. Pereyda. *Boofuzz*. Accessed: Jan. 2, 2025. [Online]. Available: https://boofuzz.readthedocs.io/en/stable/user/quickstart.html

[11] J. Chen et al., "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[12] Y. Zheng et al., "FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation," in *Proc. USENIX Secur.*, 2019, pp. 1099–1114.

[13] M. Eisele, D. Ebert, C. Huth, and A. Zeller, "Fuzzing embedded systems using debug interfaces," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2023, pp. 1031–1042.

[14] E. Trickel et al., "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 2658–2675.

[15] A. A. Clements et al., "HALucinator: Firmware re-hosting through abstraction layer emulation," in *Proc. USENIX Security*, 2020, pp. 1201–1218.

[16] H. Liu et al., "Labrador: Response guided directed fuzzing for black-box IoT devices," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2024, pp. 1920–1938.

[17] S. Luo. (2024). *Trapfuzzer—Coverage-Guided Binary Fuzzing With Breakpoints*. [Online]. Available: https://github.com/hac425xxx/trapfuzzer

[18] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," in *Proc. Annu. Int. Conf. Commun. ACM*, vol. 33, 1990, pp. 32–44.

[19] Y. Pan, W. Lin, L. Jiao, and Y. Zhu, "Model-based grey-box fuzzing of network protocols," *Security Commun. Netw.*, vol. 2022, no. 1, 2022, Art. no. 6880677.

[20] Y. Yu, Z. Chen, S. Gan, and X. Wang, "SGPFuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations," *IEEE Access*, vol. 8, pp. 198668–198678, 2020.

[21] Y. Qin, X. Li, J. Tian, T. Gu, and X. Kuang, "Gradient-oriented gray-box protocol fuzzing," in *Proc. IEEE 6th Int. Conf. Data Sci. Cyberspace (DSC)*, Oct. 2021, pp. 353–360.

[22] K. Lu, M.-T. Walter, D. Pfaff, S. Nuernberger, W. Lee, and M. Backes, "Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[23] S. Schumilo et al., "KAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. USENIX Secur.*, 2017, pp. 167–182.

[24] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh, "SyzVegas: Beating kernel fuzzing odds with reinforcement learning," in *Proc. USENIX Security*, 2021, pp. 2741–2758.

[25] X. Xie et al., "DeepHunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proc. ISSTA*, 2019, pp. 146–157.

[26] A. Odena and I. Goodfellow, "TensorFuzz: Debugging neural networks with coverage-guided fuzzing," in *Proc. ICML*, 2018, pp. 4901–4911.

[27] M. Zalewski. (2024). *American Fuzzy Lop*. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[28] K. Serebryany. (2025). *LibFuzzer—A Library for Coverage-Guided Fuzz Testing*. Accessed: 2025-01-02. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[29] J. Wang, L. Yu, and X. Luo, "LLMIF: Augmented large language model for fuzzing IoT devices," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2024, pp. 881–896.

[30] M. Eddington. (2024). *Peach Fuzzer*. [Online]. Available: https://peachtech.gitlab.io/peach-fuzzer-community/

[31] Z. Shu and G. Yan, "IoTInfer: Automated blackbox fuzz testing of IoT network protocols guided by finite state machine inference," *IEEE Internet Things J.*, vol. 9, no. 22, pp. 22737–22751, Nov. 2022.

[32] F. Bellard. (2024). *QEMU: A Generic and Open Source Machine Emulator and Virtualizer*. [Online]. Available: https://www.qemu.org/

[33] T. Newsham. (2024). *Nccgroup/TriforceAFL: AFL/QEMU Fuzzing With Full-System Emulation*. [Online]. Available: https://github.com/nccgroup/TriforceAFL

[34] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: Peripheral proxying supported embedded code testing," in *Proc. 9th ACM Symp. Inf., Comput. Commun. Secur.*, Jun. 2014, pp. 329–340.

[35] L. H. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "PARTEMU: Enabling dynamic analysis of real-world TrustZone software using emulation," in *Proc. USENIX Security*, 2020, pp. 789–806.

[36] P. Alves. (2024). *GDB Documentation*. [Online]. Available: https://www.sourceware.org/gdb/documentation/

[37] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proc. ISSTA*, 2022, pp. 423–435.

[38] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt fuzzing for fuzz driver generation," 2023, *arXiv:2312.17677*.

[39] D. Lohiya, M. R. Golla, S. Godboley, and P. R. Krishna, "Poster: Gpt-CombFuzz: Combinatorial oriented LLM seed generation for effective fuzzing," in *Proc. IEEE Conf. Softw. Test., Verification Validation (ICST)*, May 2024, pp. 438–441.

[40] J. Eom, S. Jeong, and T. Kwon, "Fuzzing Javascript interpreters with coverage-guided reinforcement learning for LLM-based mutation," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Sep. 2024, pp. 1656–1668.

[41] (2024). *RFC 2616-HTTP/1.1*. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2616

[42] (2024). *Configuration Files-Apache HTTP Server*. [Online]. Available: https://httpd.apache.org/docs/2.4/configuring.html

[43] CISA. (2024). *CVE Mitre*. [Online]. Available: https://cve.mitre.org/

[44] (2024). *CVEProject: CVE Cache of the Official CVE List*. [Online]. Available: https://github.com/CVEProject/cvelistV5

[45] U.S. -CERT. (2024). *National Vulnerability Database*. [Online]. Available: https://nvd.nist.gov/

[46] P. Mell, K. Scarfone, and S. Romanosky, "Common vulnerability scoring system," *IEEE Security Privacy*, vol. 4, no. 6, pp. 85–89, 2006.

[47] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Hoboken, NJ, USA: Wiley, 2011.

[48] OWASP. (2024). *Command Injection*. [Online]. Available: https://owasp.org/www-community/attacks/CommandInjection

[49] Y. Zhang et al., "SRFuzzer: An automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, Dec. 2019, pp. 544–556.

[50] S. Deep, X. Zheng, A. Jolfaei, D. Yu, P. Ostovari, and A. K. Bashir, "A survey of security and privacy issues in the Internet of Things from the layered context," *Trans. Emerg. Telecommun. Technol.*, vol. 33, no. 6, p. 3935, Jun. 2022.

[51] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, "Demystifying IoT security: An exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale IoT exploitations," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 3, pp. 2702–2733, 3rd Quart., 2019.

[52] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 95–110.

[53] NationalSecurityAgency. (2024). *Ghidra*. [Online]. Available: https://ghidra-sre.org/

[54] Microsoft. (2024). *Playwright: A Framework for Web Testing and Automation*. [Online]. Available: https://github.com/microsoft/playwright

[55] A. G. Miguel, *RFC 7303*. Fremont, CA, USA: InternetEngineering Task Force (IETF), 2008.

[56] D. Box. (2024). *Simple Object Access Protocol (SOAP) 1.1*. [Online]. Available: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[57] O. Ali, M. K. Ishak, M. K. L. Bhatti, I. Khan, and K.-I. Kim, "A comprehensive review of Internet of Things: Technology stack, middlewares, and fog/edge computing interface," *Sensors*, vol. 22, no. 3, p. 995, Jan. 2022.

[58] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Proc. IEEE 16th Int. Conf. Data Mining (ICDM)*, Dec. 2016, pp. 859–864.

[59] A. Das. (2024). *Worldwide Service Provider Router Market Shares, 2024: Huawei Remains Market Leader; Nokia Gains Market Share*. [Online]. Available: https://my.idc.com/getdoc.jsp?containerId=US52475725

[60] *Home Wireless Router-Global Market Share and Ranking*. Beijing, China: QYResearch, 2024.

[61] J. Baek et al., "A study on vulnerability analysis and memory forensics of ESP32," *J. Internet Comput. Services*, vol. 25, no. 3, pp. 1–8, 2024.

[62] X. Ma, Q. Zeng, H. Chi, and L. Luo, "No more companion apps hacking but one dongle: Hub-based blackbox fuzzing of IoT firmware," in *Proc. 21st Annu. Int. Conf. Mobile Syst., Appl. Services*, Jun. 2023, pp. 205–218.

[63] 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 *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2020, pp. 733–745.

[64] A. D. Pinto. (2013). *D-Link Dir-505 1.06-Multiple Vulnerabilities-Hardware Webapps Exploit*. [Online]. Available: https://www.exploit-db.com/exploits/28184

[65] (2025). *Netsecfish/dlink*. [Online]. Available: https://github.com/netsecfish/dlink

[66] NVD.(2024). *CVE-2024-32351*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-32351

[67] MITRE.(2024). *CWE-77: Improper Neutralization of Special Elements Used in a Command ('Command Injection')*. [Online]. Available: http://cwe.mitre.org/data/definitions/77.html

[68] Oracle.(2024). *Java Software—Oracle*. [Online]. Available: https://www.oracle.com/java/

[69] (2024). *Python Software*. [Online]. Available: https://www.python.org/

[70] The PHP Group.(2024). *PHP: Hypertext Preprocessor*. Accessed: 10. [Online]. Available: https://www.php.net/

[71] Oracle.(2024). *Jdb*. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/jdb.html

[72] Python Software Foundation.(2024). *Pdb—The Python Debugger*. [Online]. Available: https://docs.python.org/3/library/pdb.html

[73] D. Rethans. (2024). *Xdebug-Debugger and Profiler Tool for PHP*. [Online]. Available: https://xdebug.org/


**Cheng Shi** received the B.Eng. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2013, where he is currently pursuing the master's degree with the College of Computer Science and Technology. His research interests include the Internet of Things security, fuzz testing, and software security.

**Jiongchi Yu** received the bachelor's degree from Zhejiang University, China. He is currently pursuing the Ph.D. degree with the School of Computing and Information Systems, Singapore Management University (SMU), Singapore. His research interests include traditional software testing and security issues of cloud native infrastructures.

**Ziming Zhao** (Member, IEEE) is currently a ZJU 100 Young Professor with the School of Software Technology, Zhejiang University, China. He has published more than 40 papers in international journals and conference proceedings, including IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, MobiCom, SIGCOMM, WWW, INFOCOM, KDD, CCS, RTSS, IJCAI, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, COSE, DATE, ESE, and AAAI. His research interests include machine learning, traffic identification, network security, and mobile computing.

**Jiongyi Chen** received the B.Eng. and Ph.D. degrees from The Chinese University of Hong Kong in 2015 and 2019, respectively. He is currently a Lecturer with the National University of Defense Technology, Changsha, China. His research interests include software security.

**Fan Zhang** (Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Engineering, University of Connecticut, CT, USA, in 2011. He is currently a Full Professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China, and also with the Alibaba–Zhejiang University Joint Institute of Frontier Technologies, Hangzhou. His research interests include system security, hardware security, network security, cryptography, and computer architecture.