This paper is accepted by Computers & Security, and is shared here for nc-nd use. cc-nc-nd end user license https://creativecommons.org/licenses/by-nc-nd/4.0/

DiffFuzz: Homology-Aware Function-Level Fuzzing of Wireless Router Firmware

Pulin Chen[a,b], Yingpei Zeng[a,b,*], Xiangpu Song[c], Yuhang Pan[a,b], Xiao Li[a,b] and Shanqing Guo[c]

[a]School of Cyberspace, Hangzhou Dianzi University, Hangzhou, 310018, China
[b]Zhejiang Provincial Key Laboratory of Sensitive Data Security Protection and Confidentiality Management, Hangzhou Dianzi University, Hangzhou, 310018, China
[c]School of Cyber Science and Technology, Shandong University, Qingdao, 266237, China

ARTICLE INFO

Keywords:
IoT security
firmware fuzzing
function-level fuzzing
vulnerability detection
differential analysis

ABSTRACT

The widespread deployment of wireless routers has made their security increasingly critical to daily life. Existing static and dynamic vulnerability detection techniques face notable limitations, including neglect of the similarity across firmware images, high false-positive rates, heavy dependence on firmware rehosting, and inefficient exploration. This paper presents DiffFuzz, a homology-aware function-level fuzzing approach that efficiently analyzes firmware batches by directly targeting service handler functions that handle user requests and eliminating redundant testing across homologous router firmware images. DiffFuzz integrates LLM-assisted static analysis, symbolic execution-based seed generation, and automated fuzzing harness construction to enable efficient function-level fuzzing. Our evaluation on 18 firmware images shows that DiffFuzz discovered 632 vulnerabilities, compared with 97 reported by Mango. Of the vulnerabilities reported by Mango, 11 were not identified by DiffFuzz. On the six firmware images supported by AFL++ (Greenhouse) after manual patching, DiffFuzz discovered 166 vulnerabilities, including all 11 vulnerabilities identified by this baseline. DiffFuzz further discovered 292 previously unknown vulnerabilities, including new vulnerabilities in firmware previously tested by prior approaches. To date, 61 of these vulnerabilities have been assigned CVE IDs. Among these disclosed vulnerabilities, 33 are rated Critical, 25 High, and 3 Medium under CVSS (Common Vulnerability Scoring System) v3.1.

1. Introduction

IoT (Internet of Things) devices are becoming increasingly prevalent in our daily lives; however, they may contain vulnerabilities that can be exploited by malicious attackers (Khan and Salah, 2018; Chen et al., 2018). Wireless routers are a representative example. On the one hand, an increasing number of wireless routers are being shipped and deployed in diverse environments (Research and Markets, 2026). The global wireless router market is projected to reach US$ 32.5 billion by 2034, up from US$ 15.44 billion in 2025 (Research and Markets, 2026), driven by the increasing penetration of Internet connectivity, the development of bandwidth-intensive applications, and the rapid expansion of smart homes and IoT devices (Research and Markets, 2026; Grand View Research, 2026). Meanwhile, wireless router vendors continue to invest heavily in research and development to release new products that integrate technologies such as artificial intelligence and support emerging standards including Wi-Fi 6E and Wi-Fi 7 (Grand View Research, 2026). On the other hand, wireless routers remain frequent targets of cyberattacks (Research and Markets, 2026; Chen et al., 2023). We analyzed recent CVEs affecting wireless routers from 11 vendors that either hold significant market shares (Research and Markets, 2026) or are positioned in the Leader Quadrant (Mordor Intelligence, 2026), and present the results in Fig. 1. As shown in the figure, a substantial number of CVEs have been reported each year from 2020 to 2026, with a slight upward trend over time. Overflow and command injection vulnerabilities account for the majority of reported cases, consistent with observations from previous studies (Chen et al., 2023; Qin et al., 2023; Li et al., 2024; Gibbs et al., 2024). In practice, routers account for approximately 75% of all IoT attacks (Chen et al., 2023). Based on both our observations and those reported in previous studies (Chen et al., 2023; Li et al., 2024; Gibbs et al., 2024), a major reason for the prevalence of vulnerabilities is that wireless routers typically run service programs, such as web servers, to process various user management requests transmitted via protocols such as HTTP (Hypertext Transfer Protocol). The corresponding request-handling logic is often complex, making it particularly susceptible to vulnerabilities.

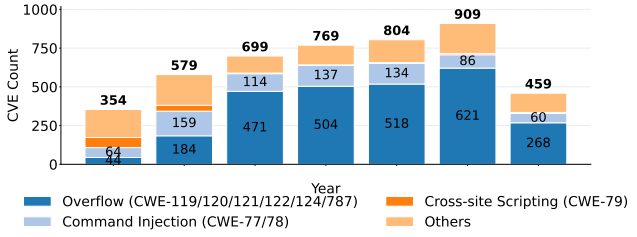


Fig. 1: CVE counts for wireless routers from 11 major vendors (TP-Link, Cisco, Huawei, ZTE, AVM, NETGEAR, ASUSTeK, D-Link, RUCKUS, Tenda, Xiaomi) from January 1st 2020 to June 30th 2026.

Many approaches have been proposed to detect vulnerabilities in IoT firmware, which can be broadly categorized

*Corresponding author
232270036@hdu.edu.cn (P. Chen); yzeng@hdu.edu.cn (Y. Zeng); songxiangpu@mail.sdu.edu.cn (X. Song); 242270054@hdu.edu.cn (Y. Pan); 252270051@hdu.edu.cn (X. Li); guoshanqing@sdu.edu.cn (S. Guo)

into static and dynamic methods. Firmware images can be obtained through various means (OWASP, 2026), such as downloading them from vendor websites (Chen et al., 2016) or, when security testing is conducted internally by the vendor, obtaining them from development teams within the company (OWASP, 2026). Static methods (Shoshitaishvili et al., 2015; Redini et al., 2020; Cheng et al., 2021; Chen et al., 2023; Gibbs et al., 2024; Liu et al., 2025) analyze IoT firmware using techniques such as taint analysis without executing the firmware, but they typically suffer from high false-positive rates. Dynamic approaches generally leverage fuzzing (Manès et al., 2019) to discover vulnerabilities. Among them, black-box fuzzers (Chen et al., 2018; Feng et al., 2021; Ma et al., 2024; Liu et al., 2024a; Wang et al., 2024) lack internal execution visibility, limiting their ability to thoroughly explore firmware servers. Grey-box fuzzers (Zheng et al., 2019, 2022; Tay et al., 2023; Li et al., 2024; Oliinyk et al., 2024; Xiao et al., 2025) provide more effective server exploration by retaining inputs that discover new execution paths as seeds for mutation. They typically emulate the server or the full firmware to improve fuzzing throughput and have successfully uncovered numerous vulnerabilities (Zheng et al., 2019, 2022; Tay et al., 2023; Li et al., 2024; Xiao et al., 2025).

However, existing grey-box fuzzing techniques for IoT firmware still suffer from several limitations. First, security testing may cover not only an individual firmware image but also firmware images from multiple device models or versions of the same vendor (Microsoft, 2025; Chen et al., 2016; Gibbs et al., 2024). Device vendors often reuse code across products that provide similar functionality to reduce development costs (TP-Link, 2026; NETGEAR, 2026), such as configuring router IP or DNS settings. Consequently, testing these firmware images independently may repeatedly fuzz the same code, yet existing fuzzing schedules do not exploit such cross-firmware similarities. Second, emulating the web server or the entire firmware for grey-box fuzzing requires substantial manual effort, and automated rehosting techniques remain unreliable, exhibiting a low success rate, for example, approximately 39.7% of firmware images for the state-of-the-art approach Greenhouse (Tay et al., 2023). Third, most existing approaches fuzz the entire server program, even though the functions responsible for handling user requests, referred to as *service handler functions* (Li et al., 2024) or simply *handler functions*, are deeply embedded and difficult to reach due to their reliance on structural inputs (Liu et al., 2023; Pham et al., 2019). T-Fuzz (Peng et al., 2018) improves general code reachability by reversing branch conditions through binary rewriting, but it does not specifically address the reachability of registered handler functions, which are difficult to reach directly through branch condition reversal alone. SHFuzz (Li et al., 2024) guides fuzzing toward service handlers and sensitive APIs. Nevertheless, both approaches still fuzz the entire program, and common code such as HTTP header processing must therefore be executed, which reduces overall fuzzing efficiency.

In this paper, we propose DiffFuzz, a homology-aware function-level fuzzing approach that accounts for **Diff**erences among homologous firmware to efficiently **Fuzz** batches of firmware while avoiding the aforementioned drawbacks. Function-level fuzzing (Murali et al., 2024; Liu et al., 2024b) is a recently introduced technique that directly fuzzes individual functions within a project, thereby improving coverage of the target functions. We apply a similar idea to fuzz service handler functions (Liu et al., 2024b) of the web servers in wireless router firmware, but without access to source code. This approach not only enhances fuzzing efficiency but also avoids the need for firmware rehosting, as these handler functions typically do not depend on special files or hardware. This means that we do not aim to solve the hardware-dependency issue, but rather to remove the need for it by fuzzing a specific set of functions in the firmware. Furthermore, we compare handler functions across different firmware to eliminate redundant fuzzing effort.

Specifically, DiffFuzz consists of a static analysis phase to extract required information and a dynamic fuzzing phase for vulnerability detection. With the assistance of Large Language Models (LLMs), DiffFuzz identifies the service handler functions as well as the input functions invoked by these handlers. In addition, DiffFuzz conducts LLM-assisted differential analysis on handler functions to generate a fuzzing test plan that removes redundant tests. During the fuzzing phase, for each handler function included in the test plan, DiffFuzz first generates initial seeds using symbolic execution and marks critical fields in seeds to accelerate fuzzing. It then automatically constructs fuzzing harnesses by hooking the handler functions and their input functions, and applies custom mutations for efficient fuzzing.

We compare DiffFuzz with the state-of-the-art dynamic fuzzing approach AFL++ (Greenhouse (Tay et al., 2023)) and the static analysis approach Mango (Gibbs et al., 2024) on 18 firmware images, including 10 from the SaTC dataset (Chen et al., 2023) and 8 newly collected firmware images. The results show that DiffFuzz discovers 632 vulnerabilities, whereas Mango identifies only 97 vulnerabilities, 11 of which are not identified by DiffFuzz. On the six firmware images that AFL++ (Greenhouse) supports after manual patching, AFL++ discovers 11 vulnerabilities, while DiffFuzz discovers 166 vulnerabilities. In addition, DiffFuzz finds 36 new vulnerabilities from the SaTC dataset, 19 of which have been assigned CVE IDs, even though these firmware images have already been extensively analyzed by existing solutions, demonstrating the unique advantages of our approach. In total, DiffFuzz discovers 292 new vulnerabilities across the 18 firmware images. To date, 61 of these vulnerabilities have been assigned CVE IDs. Even after excluding homology-based propagation, 142 unique zero-day vulnerabilities remain. DiffFuzz may require manual effort or produce false positives or miss vulnerabilities in some cases, as discussed later in Section 6. The prototype of DiffFuzz will be open-sourced on GitHub (`https://github.com/hdusoftsec/DiffFuzz`) for research purposes after paper publication.

```
1  void formDefineTendDa(){ ...
2    websFormDefine("setcfm", (void (*)(webs_t, char_t *,
         char_t *))formSetCfm); // register the handler
         function for "setcfm" URL path
3  }
4  void formSetCfm(webs_t wp, char_t *path, char_t *query){
       /* a handler function */ ...
5    FuncNp = websGetVar(wp, "funcname", byte_50ADD4);
6    if ( *FuncNp ){
7      if ( !strcmp(FuncNp, "save_list_data") ){
8            FuncP1 = websGetVar(wp, "funcpara1",
                   byte_50ADD4); // retrieve user data
9            FuncP2 = websGetVar(wp, "funcpara2",
                   byte_50ADD4);
10           // Pass in data for further processing
11           save_list_data(FuncP1, FuncP2, 126); }
12   } ...
13 }
14 void save_list_data(char *list_name, char *buf, char c){
       ...
15   // list_name could overflow the mib_name buffer
16   sprintf(mib_name, "%s.list%d", list_name, counta);
17 }
```

Listing 1: A code snippet of httpd in Tenda AC6, AC9, and AC1206.

The remainder of this paper is organized as follows. Section 2 presents the motivating example and outlines the key challenges. Section 3 describes the design of DiffFuzz, and Section 4 provides its implementation details. In Section 5, we report and analyze our experimental results. Section 6 further discusses our approach and its limitations. Section 7 surveys related work. Finally, Section 8 concludes the paper.

2. Motivation and Challenges

In this section, we first present a real-world vulnerability to illustrate our motivation and then discuss the limitations of existing research and the challenges we address.

2.1. Motivating Example

The vulnerability CVE-2025-55482 that we discovered appears in the web servers of Tenda AC6, AC9, and AC1206 devices. Vendors such as Tenda typically offer multiple device models, for example, supporting different Wi-Fi versions such as Wi-Fi 6 and Wi-Fi 7. To reduce development costs, firmware developed for different devices from the same vendor is often homologous, as evidenced by vendors' open-source router code and the lists of affected products in their security advisories (TP-Link, 2026; NETGEAR, 2026). As shown in Listing 1, the `formSetCfm` method is registered as the handler function for the URL path "setcfm" at line 2. When an attacker submits a request containing an excessively long value for the key "funcpara1", the data is retrieved through `websGetVar` at line 8, which ultimately triggers a buffer overflow in the `sprintf` call at line 16.

Existing approaches face difficulties in detecting this vulnerability. The dynamic fuzzing technique Greenhouse (Tay et al., 2023) fails to automatically rehost the firmware for fuzzing because correct execution requires completing special hardware initialization and network configuration. Even if the httpd binary is manually patched to enable fuzzing, Greenhouse does not trigger the vulnerability within 24 hours, as the server contains many handler functions, and inputs must traverse a long execution path from packet reception to the vulnerable line. The static analysis method Mango (Gibbs et al., 2024) also fails to detect this vulnerability because an implementation limitation in its `sprintf` analysis causes it to consider only the format string argument. This flaw can be corrected by analyzing the destination buffer, but doing so requires estimating both the buffer size and the formatted output length. In contrast, DiffFuzz detects this overflow through concrete execution. Even if some static methods handle such functions correctly, they are unlikely to achieve high vulnerability coverage while maintaining a low false positive rate (Murali et al., 2024; Redini et al., 2020; Chen et al., 2023; Gibbs et al., 2024). For example, CVE-2024-52275, discussed later in Section 5.3, requires specialized static rules to verify that pointers remain within buffer bounds (Redini et al., 2020; Chen et al., 2023; Gibbs et al., 2024). Finally, existing approaches do not consider similarities among handler functions across firmware from the same vendor, resulting in substantial redundant analysis.

2.2. Challenges and Our Solutions

The recently introduced function-level fuzzing technique (Murali et al., 2024; Liu et al., 2024b) can bypass irrelevant code and directly test targeted functions. It has also been used to reduce false positives in static analysis tools (Murali et al., 2024). Thus, it is a natural candidate for fuzzing handler functions in router servers. However, current function-level fuzzing techniques assume access to source code, and applying them to binary firmware introduces the following challenges.

(C1) How can we generate the fuzzing harnesses and efficiently perform function-level fuzzing on handler functions without access to source code? Handler functions often rely on additional contextual data to execute correctly, such as the `webs_t wp` structure in Listing 1, which is needed in the fuzzing harnesses as well (Liu et al., 2024b; Murali et al., 2024). These data structures are produced through a sequence of earlier processing functions starting from raw packets. Without source code and intercepted packets, producing valid inputs becomes challenging.

Solution: We observe that handler functions typically obtain concrete data from specific functions, which we refer to as *input functions*, such as `websGetVar` in Listing 1. Our manual analysis of 352 firmware CVEs, comprising all CVEs to date for seven devices from four different vendors, showed that, in 311 cases (88.35%), the vulnerabilities were located within handler functions and were triggered by input values retrieved through input functions. We therefore hook input functions and provide customized seed formats to supply test cases during function-level fuzzing. Because binaries may be stripped, we use an LLM to infer semantics

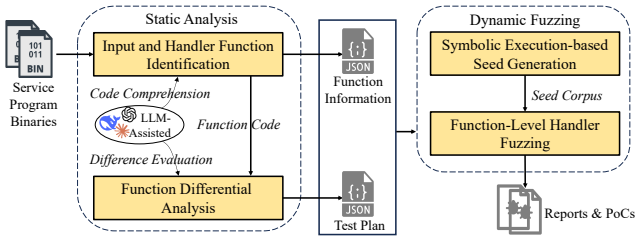


Fig. 2: Overall architecture of DiffFuzz.

and assist in identifying input functions. We also employ symbolic execution to generate initial seeds and annotate critical fields, further improving fuzzing efficiency.

(C2) When analyzing and testing firmware images from multiple device models or versions of the same vendor (Microsoft, 2025; Chen et al., 2016; Gibbs et al., 2024), how can we avoid redundant fuzzing caused by code reuse across their service programs? We must ensure that semantically identical handler functions are fuzzed only once, while still distinguishing small but security-relevant changes such as replacing `sprintf` with `snprintf`.

Solution: We specifically developed an algorithm that identifies all distinct handler functions within a firmware batch through four primary steps, which include grouping of service programs, extraction of function differences, LLM-assisted differential evaluation, and differential test plan generation.

3. Design

In this section, we present the design of DiffFuzz, a homology-aware function-level fuzzing approach to detect vulnerabilities in IoT firmware. Fig. 2 illustrates the overall architecture of DiffFuzz, which is composed of four core components: input and handler function identification, function differential analysis, symbolic execution-based seed generation, and function-level handler fuzzing.

3.1. Input and Handler Function Identification

To determine which functions should be fuzzed and to support effective fuzzing, DiffFuzz first identifies input functions and handler functions. Handler functions, also referred to as service handler functions (Qin et al., 2023; Li et al., 2024), are responsible for processing user requests, and vulnerabilities are highly likely to reside in these functions (Qin et al., 2023; Li et al., 2024). They therefore naturally become fuzzing targets. However, identifying handler functions alone is insufficient. As shown in Listing 1, the handler function `formSetCfm` requires parameters such as `webs_t wp` to execute; otherwise, it will crash immediately at line 5. The variable `wp` is constructed through a sequence of functions based on received packets (Liu et al., 2023) and is difficult to reproduce in a fuzzing harness without source code (Murali et al., 2024; Liu et al., 2024b). Fortunately, we observe that handler functions typically rely on what we refer to as input functions to retrieve input values, such as `websGetVar` in Listing 1. Therefore, by identifying these input functions and replacing them during fuzzing, we can supply fuzzing inputs effectively.

First, DiffFuzz identifies input functions by leveraging shared keywords (Chen et al., 2023) and LLM-assisted semantic analysis. We observe that the parameters of input functions often contain constant strings. Such strings have been identified as shared keywords between front-end and back-end in SaTC (Chen et al., 2023) and can be exploited in our identification process. Specifically, DiffFuzz first identifies all functions that reference shared keywords, ranks these functions by call frequency, and selects the top K as candidate functions. It then employs LLMs to identify the actual input functions through semantic comprehension. To ensure the reliability of the identification, the LLMs perform three independent queries for each candidate function, adopting a self-consistency strategy (Wang et al., 2022). The LLMs can distinguish target input functions, such as `websGetVar`, from functions like `cJSON_AddItemToObject` that do not read user input. The detailed prompt used for this identification process is illustrated in the left part of Fig. 3.

Second, DiffFuzz identifies handler functions by analyzing dispatching mechanisms and then filters out handler functions that do not invoke any input function. Service programs typically dispatch handler functions using two primary mechanisms (Li et al., 2024): either through registration functions that bind handler functions to corresponding URL path strings, or through function tables that aggregate handler pointers and are stored in the data segment of the ELF (Executable and Linkable Format) binary. By adopting a methodology similar to that of SHFuzz (Li et al., 2024), DiffFuzz extracts initial handler functions by detecting registration functions and scanning the data segment for structured function tables. Unlike SHFuzz, DiffFuzz further filters out handler functions that do not invoke input functions, as fuzzing handler functions that do not read input is useless.

3.2. Function Differential Analysis

DiffFuzz performs LLM-assisted differential analysis to eliminate redundant testing efforts across homologous firmware, which often exhibit significant code reuse. As shown in the motivation example, the same handler function may appear in multiple firmware images. DiffFuzz prunes these identical handler functions and concentrates its analysis on unique or semantically modified instances. The specific procedure for this differential analysis is summarized in Algorithm 1.

Step 1: Grouping of Service Programs. DiffFuzz preliminarily groups a batch of service programs from different firmware to ensure that subsequent differential analysis focuses on homologous firmware. First, DiffFuzz employs a binary differential comparison tool to perform pairwise comparisons between all service programs to construct a *SimMatrix* that records pairwise similarity (line 2). Second, based on *SimMatrix*, DiffFuzz utilizes a greedy grouping algorithm to partition the programs into homologous groups (line 3). The grouping is performed by iteratively selecting

Algorithm 1 Workflow of Function Differential Analysis

Input: P (Set of Service Program Binaries), H (Set of Handler Functions), θ (Similarity Threshold)
Output: T (Test Plan)

1: *// Step 1: Grouping of Service Programs.*
2: *SimMatrix* ← CalcPairwiseSimilarity(P)
3: *Groups* ← GroupByThreshold(*SimMatrix*, θ)
4: $T \leftarrow \emptyset$
5: **for all** $G \in$ *Groups* **do**
6: *// Step 2: Extraction of Function Differences*
7: *DiffData* ← PairwiseDiff(G)
8: *HandlerData* ← UnifyHandler(*DiffData*, H)
9: *// Step 3: LLM-assisted Differential Evaluation*
10: **for all** $h \in$ *HandlerData* **do**
11: *h.ModifiedPairs* ← LLMEvaluateDiffPair(h)
12: **if** *h.ModifiedPairs* $\neq \emptyset$ **then**
13: *h.Modified* ← True
14: **else**
15: *h.Modified* ← False
16: **end if**
17: **end for**
18: *// Step 4: Differential Test Plan Generation*
19: **for all** $h \in$ *HandlerData* **do**
20: **if not** *h.Modified* **then**
21: $T \leftarrow T \cup \{(\text{GetBinary}(h), h)\}$
22: **else**
23: *BinGroups* ← GroupBins(*h.ModifiedPairs*)
24: **for all** $g \in$ *BinGroups* **do**
25: $T \leftarrow T \cup \{(\text{GetBinary}(g), h)\}$
26: **end for**
27: **end if**
28: **end for**
29: **end for**
30: **return** T

#Instruction
Your task is to determine whether this function acts as an input function designed to read user-supplied data. This type of function is similar to websGetVar.....
Please make a comprehensive judgment based on these three factors:(1) function name, (2) code definition, and (3) call context.
Output Requirement:
The output must be a strict JSON object:
{"significant": true/false, "reason": "A concise explanation."}
#FuncName
{func_name}
#FuncCode
{func_code}
#CallContext
{call_context_code}

#Instruction
Analyze the provided diff code to determine if it contains a significant semantic difference. Please focus your analysis on the code between the + and - lines.
Significant changes directly impact functional logic, control flow, and security properties:
·Function Call Changes: Adding or removing custom or standard library functions.
·.....
Non-significant changes include:
·Debug/Logging: Modifications to printf, log libraries, or other diagnostic utilities.
·.....
Output Requirement:
The output must be a strict JSON object:
{"significant": true/false, "reason": "A concise explanation."}
#DiffCode
{diff_code}

Fig. 3: Prompts used in DiffFuzz. The left part shows the identification prompt for input functions, and the right part presents the evaluation prompt for identifying semantic differences in code.

an ungrouped program as a seed and transitively aggregating all candidates whose similarity to any existing group member meets the threshold θ.

Step 2: Extraction of Function Differences. DiffFuzz extracts function differences and unifies handler functions across homologous groups. Initially, DiffFuzz performs exhaustive pairwise comparisons among service programs within the same homologous group to generate *DiffData*, which records function matching relationships and code differences (line 7). To address function naming variations across different service programs (e.g., caused by stripped binaries), DiffFuzz treats pairwise matches between handler functions as an equivalence relation and uses a disjoint-set data structure to unify the functions by exploiting the transitivity of this relation (Galler and Fisher, 1964) (line 8). Consequently, this process yields *HandlerData*, which unifies handler functions residing in different service programs and also stores their pairwise code differences.

Step 3: LLM-assisted Differential Evaluation. DiffFuzz utilizes LLMs to evaluate the semantics of code differences between handler functions by analyzing changes in functional logic, control flow, and security properties. While the previous step identifies code differences, many of these differences do not result in functional or security modifications because they often involve address offset changes, variable renaming, or logging operations. DiffFuzz invokes the LLM with the self-consistency strategy (as previously mentioned in Section 3.1) to ensure reliability, filtering out irrelevant changes and recording significant differential pairs into *ModifiedPairs* (line 11). For instance, in D-Link DIR-513 and DIR-605L, there is a critical change in `formLanguageChange` from `sprintf` to `snprintf`. Based on *ModifiedPairs*, DiffFuzz updates the *Modified* status to reflect whether any significant semantic difference is present within the handler function (lines 12–16). The detailed prompt used for differential evaluation is illustrated in the right part of Fig. 3. Despite the strong semantic analysis capabilities of current LLMs and the self-consistency strategy adopted by DiffFuzz, incorrect evaluations cannot be completely avoided (Wang et al., 2022). However, an incorrect evaluation affects only the corresponding handler function pair because DiffFuzz invokes the LLM only for pairs containing code differences. Specifically, misclassifying irrelevant changes as significant introduces redundant testing, whereas filtering out significant differential pairs may miss vulnerabilities introduced by the corresponding code differences. All other handler functions that do not require LLM evaluation are included in the test plan only for the first binary in their homologous group and are therefore tested once. The corresponding test plan generation procedure is detailed in the next step.

Step 4: Differential Test Plan Generation. DiffFuzz generates a test plan by categorizing handler functions based on their differential relationships within a homologous group. These relationships enable the classification of each handler function into four distinct categories: *Unique*, *Partial*, *Common*, and *Modified*. To formalize this classification process, let P be the set of service program binaries with $N = |P|$. For a handler function h, let $S(h)$ be the number of binaries containing h, and let *h.Modified* be a boolean indicator representing whether a significant semantic difference is identified by LLMs. The classification logic is defined by

the following formula:

$$\text{Category}(h) = \begin{cases} \textit{Unique}, & \text{if } S(h) == 1 \\ \textit{Partial}, & \text{if } (1 < S(h) < N) \\ & \quad \wedge \neg h.\textit{Modified} \\ \textit{Common}, & \text{if } S(h) == N \wedge \neg h.\textit{Modified} \\ \textit{Modified}, & \text{if } S(h) > 1 \wedge h.\textit{Modified} \end{cases} \tag{1}$$

Specifically, the *Unique* category indicates that the handler function is exclusive to a single firmware. The *Partial* category denotes that the handler function is present in a subset of the firmware, while *Common* signifies that the handler function is present across the entire group, with the handler functions in both cases not being *Modified*. Given the semantic identity in these categories, DiffFuzz eliminates redundancy by selecting the first binary as the test target and adding it to the test plan T (lines 20–21).

For handler functions categorized as *Modified*, DiffFuzz further partitions the service programs into equivalence groups based on *ModifiedPairs* by adopting a methodology analogous to the approach detailed in Step 2. As described in Step 3, *ModifiedPairs* records the significant differential pairs identified by the LLM. DiffFuzz regards function pairs without a recorded significant difference as semantically identical and groups their service programs into the same equivalent group (line 23). For instance, consider three service programs *(A, B, C)* in which a specific handler function is denoted as *Func1*, *Func2*, and *Func3*, respectively. If *Func1* and *Func2* are semantically identical, whereas *Func1* differs semantically from *Func3*, then *A* and *B* are assigned to the same equivalent group, while *C* forms a separate group. This results in the grouping *(A, B)* and *(C)*. Finally, DiffFuzz selects the first binary from each identified equivalent group and adds it to the test plan T (lines 24–26).

We further analyze the numbers of service program comparisons and LLM queries required by the procedure. Let $N = |P|$ denote the total number of service programs. For each homologous group indexed by i, let m_i and h_i denote the numbers of service programs and unified handler functions in *HandlerData*, respectively. Step 1 performs $\binom{N}{2}$ pairwise comparisons to calculate service program similarity, while Step 2 performs $\sum_i \binom{m_i}{2}$ pairwise comparisons to extract function differences within homologous groups. Thus, Steps 1 and 2 perform $\binom{N}{2} + \sum_i \binom{m_i}{2}$ service program comparisons in total. Under the self-consistency strategy, Step 3 performs three independent LLM queries for each handler function pair containing code differences. In the extreme case where every unified handler function is present in all service programs within its homologous group and every corresponding handler function pair contains code differences, Step 3 evaluates $h_i\binom{m_i}{2}$ pairs in group i, i.e., with a time complexity of $\mathcal{O}(h_i m_i^2)$. The total number of LLM queries therefore reaches $3\sum_i h_i\binom{m_i}{2}$, which represents the upper bound. In practice, the number of queries can be much lower because only pairs containing code differences require LLM evaluation. For example, among the 99 handler function pairs considered between AC6 and AC1206, only two contain code differences. Because the self-consistency strategy performs three queries per evaluated pair, these two pairs require $3\times2 = 6$ LLM queries, rather than the $3\times99 = 297$ queries that would be required if all 99 pairs contained code differences.

3.3. Symbolic Execution-based Seed Generation

DiffFuzz leverages symbolic execution to generate high-quality initial seeds to improve fuzzing efficiency (Li et al., 2025), and it also employs taint analysis to mark seed fields (i.e., key-value pairs) that are critical to vulnerability discovery (Rajpal et al., 2017). As shown in Listing 1, input values of handler functions may be compared against constant strings or integers, which are difficult to satisfy without well-constructed seeds (Stephens et al., 2016). In addition, input values that are passed to dangerous functions such as `sprintf` and `system` (Chen et al., 2023; Li et al., 2024; Redini et al., 2020; Gibbs et al., 2024) require increased attention during mutation in the fuzzing process.

We design a new seed format and use symbolic execution to generate a batch of seeds for each handler function. Each seed consists of multiple *key* = *value* lines, where the *key*s correspond to shared keywords used by input functions in the handler function, and the *value*s are their associated data. The symbolic execution strategy in DiffFuzz differs slightly from that of existing hybrid fuzzers, such as Driller (Stephens et al., 2016) and QSYM (Yun et al., 2018), as execution starts from individual functions rather than from the program entry point, and generally does not require the use of under-constrained symbolic execution (Ramos and Engler, 2015). This is because most handler functions are relatively self-contained units of business logic whose control flow and data flow are primarily driven by user-supplied input values retrieved through input functions, rather than by external program states or preceding execution contexts. T-Fuzz (Peng et al., 2018) also employs symbolic execution, but it applies this analysis after crashes are found in transformed binaries to filter false positives caused by program transformation and to generate inputs that reproduce true bugs in the original program. Moreover, DiffFuzz only generates initial seeds and does not produce new seeds during fuzzing (Stephens et al., 2016), since we find that initial seeds are sufficient to support function-level fuzzing.

Specifically, DiffFuzz first treats all values read from input functions as symbolic. When DiffFuzz detects that these input values are passed to dangerous functions such as `system`, it invokes the constraint solver of the symbolic execution engine to generate new seeds. It also generates seeds when input values are passed to custom functions, such as `save_list_data` in Listing 1, as this indicates that the values participate in more complex program logic. To limit analysis overhead, we impose a maximum execution step count and a timeout on symbolic execution. When execution terminates, DiffFuzz selects three representative

symbolic states, corresponding to the longest, median, and shortest execution paths, to generate seeds, even if none of the aforementioned kinds of functions are invoked within the handler function. In cases where symbolic execution fails to generate seeds (e.g., due to complex constraints), users may manually supply initial seeds; however, in our evaluation of DiffFuzz, we simply skip fuzzing such handler functions, as they represent a minority of cases.

DiffFuzz performs symbolic execution-based taint analysis following existing approaches (Chen et al., 2023; Redini et al., 2020), and it runs the taint analysis concurrently with the above symbolic execution process to discover critical fields. All input values are initially tainted, and when they are propagated into dangerous or custom functions during symbolic execution, the corresponding keys are marked as critical.

3.4. Function-level Handler Fuzzing

DiffFuzz performs function-level fuzzing over all handler functions specified in the test plan. It first automatically generates a fuzzing harness for each handler function, then utilizes the generated seeds as initial seeds, and finally applies custom mutation strategies that emphasize critical fields during fuzzing.

DiffFuzz generates fuzzing harnesses primarily by relying on hooking techniques. Existing function-level fuzzing solutions such as FuzzSlice (Murali et al., 2024) and AFGen (Liu et al., 2024b) are not applicable in this context, as they require access to source code, which is unavailable in our scenario. First, DiffFuzz employs ordinary function hooking (Roberts, 2019) to redirect the program entry function so that it invokes the target handler function. Second, DiffFuzz uses inline hooking (Brubacher, 1999) to intercept input functions. This step is necessary because input functions are often statically bound and cannot be redirected using ordinary function hooking. As no open-source inline hooking implementations were available for the MIPS and ARM architectures, we implemented this mechanism ourselves. The original input functions are replaced with a new function that parses the aforementioned seed format. Although handler functions receive complex request structures (e.g., `webs_t wp`) as parameters, they primarily access this structure through input functions (e.g., `websGetVar`) to retrieve user-supplied data, which are already hooked to supply fuzzing data. Therefore, DiffFuzz does not need to construct a fully valid request structure, while still preserving the complete data processing and vulnerability-triggering logic of the handler function. In rare cases where handler functions access uninitialized global variables, we also provide standardized hook templates that enable users to quickly add initialization code, as further described in Section 6. Finally, to detect command injection vulnerabilities, we hook functions such as `system`, following existing approaches (Li et al., 2024; Chen et al., 2016).

During fuzzing, DiffFuzz employs custom mutation strategies. First, it mutates seeds according to the custom seed format. The mutation operators include modifying, inserting, and deleting data in the *values* of key-value pairs, as well as havoc mutations, that is, stacked mutations (Fioraldi et al., 2020). Second, DiffFuzz allocates a fraction α of the total mutations to the previously marked critical key-value pairs, as these fields are more likely to trigger vulnerabilities. We set α to 80% based on our implementation experience. A larger α concentrates more mutations on critical fields but reduces the exploration of other fields, whereas a smaller α provides broader exploration but weakens the guidance from critical fields.

4. Implementation

We have developed a prototype of DiffFuzz comprising approximately 7.4k lines of Python and 200 lines of C code. The implementation of DiffFuzz is structured into two primary components: static analysis and dynamic fuzzing.

Static Analysis Component. The static analysis module encompasses LLM-assisted function identification and differential analysis. While DiffFuzz supports various LLMs (e.g., DeepSeek, GPT, and Claude series), we utilize Claude-Opus-4.6 (Anthropic, 2026) as it achieves the highest recall (see Table 3 in Section 5.2). DiffFuzz initially identifies service program binaries and shared keywords by adopting the methodology established in SaTC (Chen et al., 2023). For function identification, we implement the recognition and extraction of input and handler functions by interfacing Python-based scripts with the Ghidra (National Security Agency, 2024) API, supplemented by the reasoning capabilities of the Claude-Opus-4.6 model. Due to the high frequency of functions utilizing shared keywords, the target input function may not consistently rank at the top of the candidate list. Our empirical observations indicate that in G0 firmware, the actual input function ranks as low as 23rd. Therefore, to ensure a robust and conservative selection, we set the K value in Section 3.1 to 50, allowing the model to verify enough candidate input functions. Following function identification, a differential analysis is performed. To classify service programs into homologous groups, we attempted to utilize recent binary code similarity detection (BCSD) tools such as VexIR2Vec (VenkataKeerthy et al., 2025) and vSim (Wang and Lin, 2026). However, these tools demonstrated suboptimal performance on stripped IoT binaries, as VexIR2Vec encountered frequent errors during function extraction and vSim struggled with intermediate representation (IR) generation. Consequently, we employ BinDiff (Google, 2023) for its superior robustness in handling stripped executables, providing a reliable foundation for preliminary classification. Based on the empirical analysis of our dataset presented in Table 1, we observed that the lower bound of similarity for manually verified homologous firmware pairs is 86.05%, and thus we set the similarity threshold θ in Section 3.2 to 85%. Subsequently, Ghidriff (clearbluejar, 2024) is adopted to conduct pairwise comparisons of service programs belonging to the homologous group to retrieve the differential decompiled

code for each handler function. Finally, the Claude-Opus-4.6 model evaluates these differences, and its differential analysis results serve as the foundation for the generation of the differential test plan.

Dynamic Fuzzing Component. The dynamic fuzzing module is divided into two primary stages: symbolic execution-based seed generation and function-level handler fuzzing. DiffFuzz leverages angr (Wang and Shoshitaishvili, 2017) to perform symbolic execution and taint analysis on each handler function specified in the test plan to synthesize initial test seeds and mark critical fields. Based on the function information identified via static analysis, we utilize Python to construct a template that automatically generates fuzzing harnesses for each handler function. These harnesses are then linked to the target binary using patchelf (NixOS Community and Dolstra, 2004) or the *LD_PRELOAD* environment variable if patchelf fails. The fuzzing engine used is AFL++ (Fioraldi et al., 2020), and we modified it with custom mutation strategies. To ensure sufficient testing while avoiding the waste of computational resources, a fuzzing campaign for the corresponding function terminates when edge coverage remains stagnant for a continuous five-minute interval.

Table 1
Overview of target programs with device architectures and group categorization (Avg. Sim. denotes average similarity)

Program	Device	Brand	Arch	Group	Avg. Sim.(%)
httpd	AC6	Tenda	mipsel	1	94.47
httpd	AC9	Tenda	mipsel	1	
httpd	AC1206	Tenda	mipsel	1	
httpd	AC15	Tenda	armel	2	98.80
httpd	AC18	Tenda	armel	2	
httpd	G0	Tenda	mips	3	92.65
httpd	G0-POE	Tenda	mips	3	
httpd	G1	Tenda	armel	4	91.17
httpd	G3	Tenda	armel	4	
httpd	W20E	Tenda	armel	4	
webs	F9K1015	Belkin	mips	5	87.74
webs	F9K1122	Belkin	mips	5	
boa	DIR-513	D-Link	mips	6	86.05
boa	DIR-605L	D-Link	mips	6	
goahead	518G	UTT	mipsel	7	89.00
goahead	518GP	UTT	mipsel	7	
goahead	750W	UTT	mipsel	7	
goahead	840G	UTT	mipsel	7	

5. Experiment

We conducted a series of comparative experiments to address the following research questions.

- **RQ1:** Does the differential analysis in DiffFuzz improve testing efficiency?
- **RQ2:** Can function-level fuzzing effectively discover real-world vulnerabilities?
- **RQ3:** Do the seeds and critical field information generated by symbolic execution in DiffFuzz contribute positively to fuzzing performance?

5.1. Experimental Setup

Comparison Tool. Since recent firmware fuzzers such as HouseFuzz (Xiao et al., 2025) and SHFuzz (Li et al., 2024) were not available as open-source projects, we selected another open-source state-of-the-art solution, Greenhouse (Tay et al., 2023), which is built on AFL++ (Fioraldi et al., 2020), for our comparison. Because direct firmware emulation using the Greenhouse framework only succeeded in running a single firmware image within our dataset, we used a similar hooking mechanism (Xia et al., 2025) on top of AFL++ and reused the Greenhouse seeds to ensure a fair comparison. We referred to this baseline as AFL++ (Greenhouse). We also included Mango (Gibbs et al., 2024), a state-of-the-art static analysis tool specifically designed for IoT firmware, for comparison. For each firmware image, we used Mango to analyze only the corresponding service program binary rather than the entire firmware image. Additionally, we designed an ablation variant named DiffFuzz-D. This variant operated without the initial seeds and critical field information generated by symbolic execution, enabling a direct comparison to evaluate their combined contribution to the overall fuzzing performance under identical conditions.

Experiment Targets. We selected 18 firmware images from various vendors, including Tenda, Belkin, D-Link, and UTT, as detailed in Table 1. To ensure the representativeness of the test set, 10 Tenda firmware images were sourced from the SaTC (Chen et al., 2023) dataset, while the remaining 8 were newer firmware versions from other manufacturers to evaluate the generalizability of our methodology. All firmware images were publicly obtained from online sources and were used exclusively for academic research purposes. All service program binaries were identified as border binaries and extracted using the SaTC methodology (Chen et al., 2023). Due to the failure of Greenhouse rehosting, we manually patched 6 representative service programs to enable their execution in user mode. This ensured the AFL++ (Greenhouse) baseline could run successfully for comparative experiments with our approach.

Experiment Environment. All experiments were conducted on a machine equipped with a 16-core AMD Ryzen 9 7950X3D 4.2GHz CPU, 32GB RAM, and a 512GB SSD. The host ran Ubuntu 20.04 LTS. To account for the inherent randomness in fuzzing, the AFL++ (Greenhouse) baseline was tested with five independent trials for each target, with each trial lasting 24 hours (Klees et al., 2018). DiffFuzz likewise repeated the fuzzing of each tested handler function five times.

Table 2
Efficiency improvement through differential analysis (time denotes the total duration of static analysis, differential comparison, and differential evaluation)

Group	Device	N_{SH}	N_{SHT}	Common	Time	RRR(%)
1	AC6	99	99	96	6.23min	65.00
	AC9	102	4			
	AC1206	99	2			
2	AC15	103	103	103	1.73min	50.00
	AC18	103	0			
3	G0	8	8	8	1.74min	50.00
	G0-POE	8	0			
4	G1	121	121	77	20.45min	51.43
	G3	121	0			
	W20E	143	66			
5	F9K1015	80	80	45	9.95min	26.63
	F9K1122	89	44			
6	DIR-513	74	74	46	26.95min	27.22
	DIR-605L	95	49			
7	518G	299	299	223	81.62min	65.82
	518GP	301	12			
	750W	329	63			
	840G	285	41			
Average					8.26min	**56.69**

5.2. Efficiency Improvement through Differential Analysis (RQ1)

DiffFuzz obtained a test plan after performing function differential analysis, and we quantitatively evaluated the effectiveness of this analysis. First, we defined the Redundancy Reduction Ratio (RRR) to quantify the filtering efficacy of our differential analysis, i.e., the fraction of handler functions that do not require subsequent fuzz testing. For a test group containing n service programs, the RRR is calculated as follows:

$$\mathrm{RRR} = \frac{\sum_{i=1}^{n} N_{\mathrm{SH}}^{(i)} - \sum_{i=1}^{n} N_{\mathrm{SHT}}^{(i)}}{\sum_{i=1}^{n} N_{\mathrm{SH}}^{(i)}} \times 100\% \tag{2}$$

where $N_{\mathrm{SH}}^{(i)}$ denotes the total number of handler functions in service program i, and $N_{\mathrm{SHT}}^{(i)}$ denotes the number of handler functions in service program i that require testing after differential analysis.

Table 2 lists the RRR values and the time overhead incurred during the differential analysis process. Higher similarity among firmware images leads to higher RRR. For example, in Group 1, three firmware images shared 96 handler functions in common. After all handler functions in AC6 have been tested, only 4 handler functions in AC9 and 2 in AC1206 required fuzz testing, resulting in an RRR of 65.00%. In contrast, in Group 5, although the two firmware images each contained around 80 handler functions, they shared only 45 handler functions in common, leading to an RRR of only 26.63%. On average, the RRR reached 56.69%, indicating that more than half of the handler functions did not require fuzz testing, while vulnerabilities in these functions could still be effectively discovered. As analyzed in Section 3.2, the number of LLM queries is related to the number of handler function pairs containing code differences. This cost remained practical in our experiments because the differential analysis averaged only 8.26 minutes per group and each LLM-based inference required approximately 5–10 seconds. The resulting overhead was small compared with the hours or even days typically required for fuzzing (Klees et al., 2018; Metzman et al., 2021; Tay et al., 2023).

We further examined whether the LLM correctly recognized differences between handler function instances, and the results are shown in Table 3. To ensure an objective evaluation, we manually established the ground truth through reverse engineering of the *DiffData* instances. Specifically, our ground truth criteria designated any modification involving functional logic or security-relevant changes as a significant semantic difference. These instances consisted of pairs of differing function code, encompassing both genuine functional modifications and structural noise introduced by decompilation. We then analyzed four representative instances to identify true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). As illustrated in Table 3, the precision rates for DeepSeek-V3.2, GPT-5.4, and Claude-Opus-4.6 are 76.8%, 78.7%, and 73.1%, respectively, while their corresponding recall rates reach 81.8%, 96.1%, and 98.7%. Although the precision rates of these models are comparable, Claude-Opus-4.6 achieves the highest recall. To ensure the completeness of our testing, we ultimately selected Claude-Opus-4.6 for our work. The false positives observed across all evaluated models primarily stemmed from two common factors. First, LLMs may misunderstand certain program structures, such as complex loops. Second, the evaluation may lack sufficient context, for example, when the names of invoked functions differ. Nevertheless, we considered the minor redundancy introduced by false positives to be acceptable. Moreover, the sole false negative in Claude-Opus-4.6 occurred because functional modifications were obscured by dense structural noise (e.g., address offsets), though such infrequent omissions do not necessarily lead to security vulnerabilities.

Overall, the RRR results demonstrate that DiffFuzz significantly reduced redundant testing effort by leveraging firmware homology while ensuring completeness through difference identification.

5.3. Vulnerability Discovery Capability of Function-Level Fuzzing (RQ2)

The evaluation compared DiffFuzz with the dynamic fuzzing baseline, AFL++ (Greenhouse), and the static analysis baseline, Mango. We primarily compared the number of discovered vulnerabilities rather than their severity scores because the vulnerabilities found by the evaluated approaches had similar severity levels. All the discovered

Table 3
Comparative accuracy of LLM-assisted differential evaluation (P and R denote precision and recall, respectively)

DiffData	DeepSeek-V3.2			GPT-5.4			Claude-Opus-4.6		
	TP/TN/FP/FN	P(%)	R(%)	TP/TN/FP/FN	P(%)	R(%)	TP/TN/FP/FN	P(%)	R(%)
(AC6, AC1206)	**1/0/0/1**	100.0	50.0	**2/0/0/0**	100.0	100.0	**2/0/0/0**	100.0	100.0
(G1, W20E)	**31/4/1/6**	96.9	83.8	**36/5/0/1**	100.0	97.3	**36/2/3/1**	92.3	97.3
(F9K1015, F9K1122)	**12/23/18/0**	40.0	100.0	**10/24/17/2**	37.0	83.3	**12/18/23/0**	34.3	100.0
(DIR-513, DIR-605L)	**19/5/0/7**	100.0	73.1	**26/2/3/0**	89.7	100.0	**26/3/2/0**	92.9	100.0
Total	**63/32/19/14**	**76.8**	**81.8**	**74/31/20/3**	**78.7**	**96.1**	**76/23/28/1**	**73.1**	**98.7**

Table 4
Vulnerability detection comparison (AFL++ (G), Vul., and time denote AFL++ (Greenhouse), the number of discovered vulnerabilities, and total fuzzing hours, respectively)

		AFL++ (G)		DiffFuzz		DiffFuzz-D	
Prog.	Device	Vul.	Time	Vul.	Time	Vul.	Time
httpd	AC6	3	24h	39.6	7.2h	27.4	5.0h
httpd	AC15	4	24h	41.8	8.1h	33.8	6.2h
httpd	G0	0	24h	2	0.5h	2	0.4h
httpd	G1	0	24h	16	10.1h	14.2	8.2h
webs	F9K1015	2	24h	15.8	5.8h	7.8	4.6h
boa	DIR-513	2	24h	50.2	5.0h	42.4	3.5h
Total		**11**	**144h**	**165.4**	**36.7h**	**127.6**	**27.9h**

vulnerabilities were manually verified. We defined our vulnerability counting standard based on key-based identification. If multiple keys within the seed key-value pairs independently triggered a vulnerability through their respective values, each key was recorded as a separate vulnerability instance. We have reported all discovered vulnerabilities to the respective vendors following the principles of responsible disclosure.

Comparison with the Dynamic Fuzzing Baseline. As shown in Table 4, across six firmware images from multiple brands and architectures, DiffFuzz demonstrated significant advantages in all cases. Not only did it discover a substantially higher total number of vulnerabilities than AFL++ (Greenhouse), but it also achieved this result in significantly less total fuzzing time. DiffFuzz also discovered all vulnerabilities identified by AFL++ (Greenhouse). For instance, in the httpd program of the AC6 device, DiffFuzz found about 40 vulnerabilities in less than 8 hours, whereas AFL++ found only 3 vulnerabilities in the full 24 hours. This discrepancy primarily arose because AFL++ (Greenhouse) lacked sufficient guidance regarding internal constraints, especially when high-quality initial seeds that satisfy complex input validation were unavailable. This limitation often prevented its mutated test cases from passing the initial protocol parsing logic. Consequently, AFL++ (Greenhouse) failed to reach deep execution logic or trigger vulnerabilities within handler functions, resulting in the unproductive depletion of mutation resources during early-stage packet processing. In contrast, our function-level testing approach directly concentrated resources on critical handler functions while avoiding other code paths such as protocol parsing. Furthermore, the symbolic execution-based seed generation provided initial seeds that could quickly satisfy some path constraints.

Table 5
Vulnerability discovery capabilities of Mango and DiffFuzz across service programs (Cmdi denotes command injection, all values denote the number of discovered vulnerabilities, and the numbers in parentheses "()" indicate the total number of unique vulnerabilities discovered across all five runs)

Program	Device	Mango		DiffFuzz	
		Overflow	Cmdi	Overflow	Cmdi
httpd	AC6	1	0	36.6	3
httpd	AC9	24	1	36.6	5
httpd	AC1206	23	0	37	2.2
httpd	AC15	11	3	35.8	6
httpd	AC18	13	3	35.8	6
httpd	G0	0	0	1	1
httpd	G0-POE	1	0	1	1
httpd	G1	1	0	13	3
httpd	G3	1	0	13	3
httpd	W20E	2	0	20.6	3.8
webs	F9K1015	0	0	11.8	4
webs	F9K1122	0	0	11	4
boa	DIR-513	0	0	49.2	1
boa	DIR-605L	0	0	52	1
goahead	518G	3	1	49	11.4
goahead	518GP	3	1	46.2	11
goahead	750W	5	0	48	18
goahead	840G	0	0	40	10
Total		**88**	**9**	**537.6(542)**	**94.4(96)**

Comparison with the Static Analysis Baseline. We compared DiffFuzz and Mango on 18 firmware images. Table 5 shows that DiffFuzz and Mango discovered a total of 638 (632 on average) and 97 vulnerabilities, respectively. As described in Section 5.1, we configured Mango to analyze only the corresponding service program binary for each firmware image and did not enable analysis across program binaries. Under this configuration, all vulnerabilities reported by Mango in our evaluation were located in handler functions. DiffFuzz missed 11 of these vulnerabilities due to incomplete emulation of certain handler functions, such

Table 6
Summary of disclosed vulnerabilities and their CVSS v3.1 scores reported by NVD and CISA-ADP (Cmdi denotes command injection, N/A indicates that no score was available from the corresponding source, and severity is determined using the higher available score)

Device	CVE ID	Type	NVD	CISA	Severity	Device	CVE ID	Type	NVD	CISA	Severity
AC6	CVE-2025-55482	Overflow	N/A	7.5	High	DIR-513	CVE-2025-70231	Overflow	N/A	9.8	Critical
	CVE-2025-55483	Overflow	N/A	7.5	High		CVE-2025-70232	Overflow	N/A	9.8	Critical
	CVE-2025-55495	Overflow	N/A	6.5	Medium		CVE-2025-70233	Overflow	N/A	9.8	Critical
	CVE-2025-55498	Overflow	N/A	7.5	High		CVE-2025-70234	Overflow	9.8	7.5	Critical
	CVE-2025-55499	Overflow	N/A	6.5	Medium		CVE-2025-70236	Overflow	9.8	5.3	Critical
	CVE-2025-55503	Overflow	N/A	7.3	High		CVE-2025-70237	Overflow	9.8	7.5	Critical
	CVE-2025-70252	Overflow	7.5	7.5	High		CVE-2025-70238	Overflow	N/A	7.5	High
AC15	CVE-2026-24101	Cmdi	N/A	9.8	Critical		CVE-2025-70239	Overflow	9.8	7.5	Critical
	CVE-2026-24103	Overflow	N/A	9.8	Critical		CVE-2025-70240	Overflow	9.8	7.5	Critical
	CVE-2026-24105	Cmdi	N/A	9.8	Critical		CVE-2025-70241	Overflow	9.8	7.5	Critical
W20E	CVE-2026-24107	Cmdi	N/A	9.8	Critical		CVE-2025-70242	Overflow	N/A	7.5	High
	CVE-2026-24108	Overflow	N/A	9.8	Critical		CVE-2025-70243	Overflow	N/A	7.5	High
	CVE-2026-24109	Overflow	9.8	9.8	Critical		CVE-2025-70244	Overflow	N/A	7.5	High
	CVE-2026-24110	Overflow	N/A	9.8	Critical		CVE-2025-70245	Overflow	9.8	7.5	Critical
	CVE-2026-24111	Overflow	9.8	9.8	Critical		CVE-2025-70246	Overflow	N/A	7.5	High
	CVE-2026-24112	Overflow	9.8	7.5	Critical		CVE-2025-70247	Overflow	N/A	7.5	High
	CVE-2026-24113	Overflow	9.8	9.8	Critical		CVE-2025-70249	Overflow	N/A	7.5	High
	CVE-2026-24114	Overflow	9.8	7.5	Critical		CVE-2025-70250	Overflow	N/A	7.5	High
	CVE-2026-24115	Overflow	9.8	9.8	Critical		CVE-2025-70251	Overflow	N/A	7.5	High
DIR-513	CVE-2025-46108	Overflow	N/A	9.8	Critical	518G	CVE-2026-52187	Overflow	N/A	7.5	High
	CVE-2025-70218	Overflow	N/A	9.8	Critical		CVE-2026-52188	Overflow	N/A	6.5	Medium
	CVE-2025-70219	Overflow	N/A	9.8	Critical		CVE-2026-52189	Overflow	N/A	7.5	High
	CVE-2025-70220	Overflow	N/A	9.8	Critical		CVE-2026-52190	Overflow	N/A	7.5	High
	CVE-2025-70221	Overflow	N/A	9.8	Critical		CVE-2026-52191	Overflow	N/A	7.5	High
	CVE-2025-70222	Overflow	N/A	9.8	Critical		CVE-2026-52192	Cmdi	N/A	7.5	High
	CVE-2025-70223	Overflow	N/A	9.8	Critical		CVE-2026-52193	Overflow	N/A	7.5	High
	CVE-2025-70225	Overflow	N/A	9.8	Critical		CVE-2026-52195	Overflow	N/A	7.5	High
	CVE-2025-70226	Overflow	N/A	9.8	Critical		CVE-2026-52196	Overflow	N/A	7.5	High
	CVE-2025-70227	Overflow	N/A	7.5	High		CVE-2026-52197	Cmdi	N/A	7.5	High
	CVE-2025-70229	Overflow	N/A	9.8	Critical		CVE-2026-52198	Overflow	N/A	7.5	High
	CVE-2025-70230	Overflow	N/A	9.8	Critical						

as uninitialized dependencies. If analysis across program binaries were enabled, Mango might identify additional vulnerabilities outside handler functions, which are beyond the current scope of DiffFuzz. These limitations are discussed in Section 6. In our evaluation, Mango reported a total of 920 vulnerabilities with a rank greater than 0, and manual verification showed that Mango had a false-positive rate of 89.46%. Under the evaluated configuration, all Mango reports had ranks below 10. We therefore included all reports with ranks greater than 0 rather than applying a rank threshold greater than 10. In contrast, DiffFuzz detects vulnerabilities through concrete execution. Note that approximately 10.1% of handler functions cannot be automatically fuzzed with DiffFuzz due to incomplete emulation, which may also cause crashes (explained later in Section 6); we excluded such crashes (false positives) from the analysis whenever we did not manually fix the fuzzing of these handler functions. One factor contributing to the difference in vulnerability counts was that DiffFuzz explored the execution paths of all fuzzed handler functions without encountering path explosion. Another factor was Mango's reliance on predefined input and sink functions. In our evaluation, Mango did not recognize custom input sources such as `websGetVar`, so the corresponding paths did not receive scores. Mango also did not detect vulnerabilities located in functions outside its predefined sink points because its analysis was limited to those points. The comparison was also affected by limitations of the evaluated Mango implementation. Specifically, this implementation did not process stripped binaries, so four firmware images from the D-Link and Belkin series were not analyzed by Mango. Its analysis also terminated when anomalies occurred at certain sink points, leaving subsequent vulnerability candidates unanalyzed.

Case Study: CVE-2024-52275. DiffFuzz successfully identified a vulnerability while fuzzing the Tenda AC6 firmware. The vulnerability occurs in a common handler function, `fromWizardHandle`, as shown in Listing 2, and therefore also exists in AC9 and AC1206. It was previously reported by others as CVE-2024-52275 on AC6 and as CVE-2022-37805 on AC1206, and remains a zero-day vulnerability on AC9. This vulnerability involved a buffer overflow in the `decodePwd` function, where the `pppoepwd` variable is copied into the `decode_pwd` buffer. Before reaching `decodePwd`, the program checks whether the `WANS` field value is less than `wanflag` and whether the `WANT` field value equals 2. Therefore, triggering this vulnerability required constructing a POST request that satisfies these specific constraints. Although AFL++ (Greenhouse) successfully reached the interior of the `fromWizardHandle` handler function, the lack of constraint information prevented its mutation engine from generating the specific test cases required to trigger the vulnerability. Mango also failed to detect this vulnerability. Because its analysis was limited to predefined sink points,

```
1  void fromWizardHandle(webs_t wp, char_t *path, char_t *
       query){
2    char decode_pwd[72];
3    wantstr = websGetVar(wp, "WANT", "0");
4    wanpstr = websGetVar(wp, "WANS", "0");
5    wanport = atoi(wanpstr);
6    ...
7    if ( atoi(wanflag) >= wanport ){
8      switch ( want ){
9        case 2:
10        // retrieve user data below
11        pppoepwd = websGetVar(wp, "PPW", byte_50C8C0);
12        decodePwd(pppoepwd, decode_pwd); ...}}
13 }
14 void decodePwd(char *srcStr, char *dstStr){
15   char *srcStra;char *dstStra;
16   srcStra = srcStr;dstStra = dstStr;
17   if ( srcStr && dstStr ){
18      while ( *srcStra ){
19        if ( *srcStra == '\\' ) { ++srcStra; }
20        *dstStra++ = *srcStra++;
21      } //  ^overflow the passed-in decode_pwd buffer
22      *dstStra = 0;
23   }
24 }
```

Listing 2: Code snippet of the fromWizardHandle function in Tenda AC6 firmware illustrating CVE-2024-52275, which also corresponds to CVE-2022-37805 in AC1206 and represents a zero-day vulnerability in AC9.

it did not recognize the `decodePwd` function. In contrast, DiffFuzz, through function-level fuzzing, directly entered `fromWizardHandle`, bypassing the complex POST request construction. It obtained constraint information from seeds generated by symbolic execution and allocated more mutation attempts to the critical field PPW, thereby quickly triggering the overflow.

In summary, through comparative analysis with both dynamic and static baseline tools, we demonstrated that DiffFuzz exhibited significant advantages in the quantity of vulnerability discovery. Our approach successfully uncovered a total of 292 zero-day vulnerabilities across the evaluated service programs, including 36 zero-day vulnerabilities from the SaTC dataset, and we are currently in the process of applying for official CVE IDs for these findings. Even when excluding homology-based propagation and considering only the uniqueness of each vulnerability, 142 zero-day vulnerabilities remained after redundancy reduction. To date, 61 vulnerabilities identified by our approach have been assigned official CVE IDs and are publicly disclosed (as detailed in Table 6), including 19 vulnerabilities from the SaTC dataset. Of these disclosed vulnerabilities, 33 are rated Critical, 25 High, and 3 Medium based on the higher available CVSS v3.1 score reported by NVD (National Vulnerability Database) and CISA-ADP (Cybersecurity and Infrastructure Security Agency-Authorized Data Publisher). In accordance with responsible disclosure practices, we will release details of the remaining findings in DiffFuzz's GitHub repository at a later time.

5.4. Enhancement through Symbolic Execution (RQ3)

We compared DiffFuzz with its ablation variant, DiffFuzz-D, evaluating the specific contributions of symbolic execution-based seed generation and custom mutation guided by critical field information. Our analysis focused on vulnerability triggering capability and code exploration efficiency, measured by the number of identified vulnerabilities and covered edges, respectively.

Vulnerability Triggering Capability Comparison. DiffFuzz outperformed DiffFuzz-D on five of the six tested firmware images and found the same number of vulnerabilities as DiffFuzz-D on G0 (Table 4). Overall, DiffFuzz found 37 more vulnerabilities than DiffFuzz-D, demonstrating better vulnerability triggering capability in this evaluation.

Code Exploration Efficiency Comparison. We first compared the initial and final edge coverage across the fuzzing data from the six firmware images listed in Table 4, as shown in Fig. 4. The initial edge coverage was obtained by executing the initial seeds before any mutation, whereas the final edge coverage was recorded when the fuzzing campaign terminated after edge coverage remained stagnant for a continuous five-minute interval. The average initial edge coverage was 97.7 for DiffFuzz and 78.0 for DiffFuzz-D, while the corresponding final values were 112.6 and 90.6, respectively. In most cases, DiffFuzz covered at least as many edges as DiffFuzz-D at both stages. The higher average initial coverage indicates that the high-quality initial seeds generated by symbolic execution enabled DiffFuzz to cover more edges before mutation began. To provide a more intuitive view of edge coverage growth and further support the overall comparison in Fig. 4, we randomly selected one handler function from each of the six firmware images and plotted the corresponding growth curves in Fig. 5. These curves show how the edge coverage of DiffFuzz and DiffFuzz-D evolved during fuzzing. We also observed that edge coverage for the selected handler functions from the six firmware images generally reached saturation at approximately 300 seconds. DiffFuzz continued fuzzing for an additional five minutes to explore deeper coverage, such as state coverage (Song et al., 2025).

Case Study: G1 `formSetPortMirror`. DiffFuzz successfully identified a buffer overflow while fuzzing the Tenda G1 firmware. The vulnerability occurs in the `formSetPortMirror` handler function, as shown in Listing 3. The function retrieves the values of the `portMirrorEn` and `portMirrorMirroredPorts` fields through `websGetVar` at lines 5–6. Before reaching the vulnerable `sprintf` call, the program checks whether the `portMirrorEn` field value equals `true` at line 12. When this condition is satisfied, the value of `portMirrorMirroredPorts` is copied into the fixed buffer `mibValue` through `sprintf` at line 14. An excessively long value therefore triggers a buffer overflow. Inspection of the initial seeds showed that a seed generated by symbolic execution contained `portMirrorEn=true`. This seed already satisfied the corresponding path constraint and enabled DiffFuzz to reach

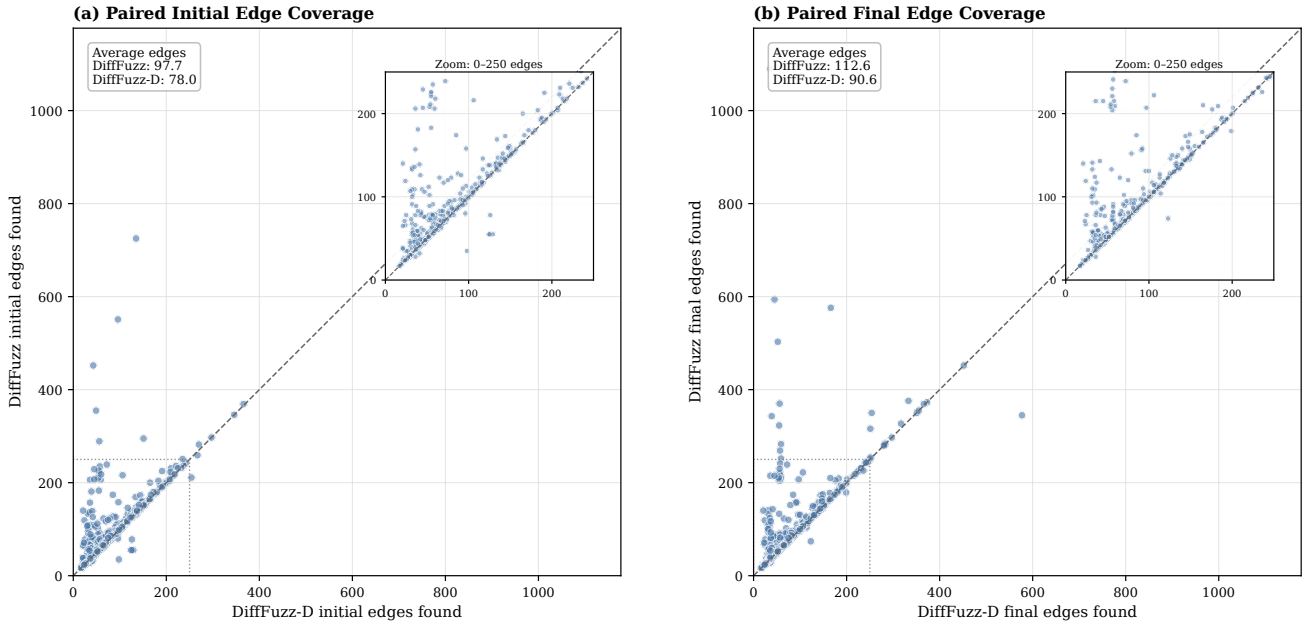


Fig. 4: Comparison of initial and final edge coverage for DiffFuzz and DiffFuzz-D. The diagonal indicates equal numbers of covered edges; points above the diagonal indicate that DiffFuzz covers more edges, whereas points below indicate that DiffFuzz-D covers more edges.

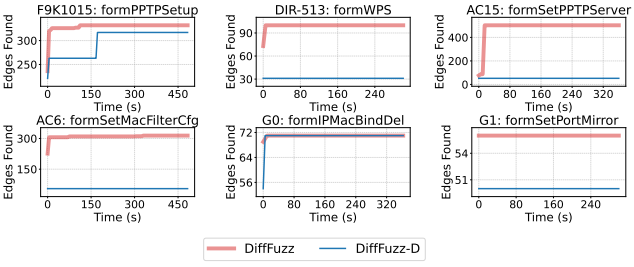


Fig. 5: Edge coverage growth of DiffFuzz and DiffFuzz-D for six randomly selected handler functions, one from each evaluated firmware image.

```
1  void formSetPortMirror(Request *req)
2  {
3      char mibValue[256] = {0};
4      // retrieve user data
5      enable = websGetVar(req, "portMirrorEn", "");
6      mirroredPorts = websGetVar(req, "
           portMirrorMirroredPorts", "");
7
8      if (!enable || !mirroredPorts) {
9          outputToWebs(req, "-1");
10         return;
11     }
12     if (!strcmp(enable, "true")) {
13         // overlong mirroredPorts can overflow mibValue
14         sprintf(mibValue, "1;%d;%s", 0, mirroredPorts);
15         prod_cfm_set_val();
16     }
17     ...
18     outputToWebs(req, "1");
19 }
```

Listing 3: Code snippet of the formSetPortMirror function in Tenda G1 firmware.

the vulnerable line at the beginning of fuzzing. The symbolic execution-based taint analysis also marked `portMirrorMirroredPorts` as a critical field because its value was propagated into `sprintf`. DiffFuzz therefore allocated more mutation attempts to this critical field and quickly triggered the overflow. In contrast, inspection of the test cases generated by DiffFuzz-D showed that none contained a `portMirrorEn` value equal to `true` during the entire fuzzing campaign. DiffFuzz-D therefore could not reach the vulnerable `sprintf` call or trigger the overflow, even if its mutations generated a `portMirrorMirroredPorts` value long enough to overflow `mibValue`. Inspection of other handler functions identified multiple similar cases in which the initial seeds generated by symbolic execution satisfied path constraints that were not satisfied by any test case generated by DiffFuzz-D during fuzzing. The inability to satisfy this path constraint also explains why DiffFuzz-D covered fewer edges for this handler in Fig. 5, illustrating how the initial seeds generated by symbolic execution improved code exploration in this case.

In summary, the ablation results show that the seeds and critical field information generated by symbolic execution contributed positively to fuzzing performance by improving vulnerability triggering capability and code exploration efficiency in our evaluation.

6. Discussions and Limitations

While DiffFuzz effectively enhances vulnerability discovery by integrating function-level fuzzing with differential analysis, it also has several limitations.

First, in our evaluation, approximately 10.1% of handler functions could not be automatically fuzzed by DiffFuzz. To enable stable function-level fuzzing, DiffFuzz's hook library includes pre-built inline hooks that bypass commonly encountered utility functions (e.g., `websWrite`, `websDone`) that access the request structure only for response generation or connection management, as they are irrelevant to the core data processing logic. Despite this, some handler functions still fail due to incomplete symbolic execution that fails to generate valid initial seeds for certain handler functions; and handler functions that internally access uninitialized memory regions or pointers (e.g., global variables or structures produced during the program's initialization phase), leading to crashes during function-level fuzzing. For the former, users can manually supply seeds based on the key-value seed format defined by DiffFuzz; for the latter, DiffFuzz provides standardized hook templates that enable users to quickly add initialization code for specific variables or shield problematic memory accesses. Both types of failures can thus be resolved with minimal effort.

Second, handler functions may retrieve configuration data from NVRAM, which may cause the emulation to deviate from the behavior of the real device. Because DiffFuzz does not have access to stored NVRAM values, it currently returns blank strings. As the values read from NVRAM typically do not significantly affect execution paths, this approach generally works well. However, for some firmware, returning correct values may be necessary, which could be addressed through symbolic execution or by collecting default values from online sources (Chen et al., 2016). We acknowledge that in such cases, as well as in other cases where DiffFuzz fails to faithfully simulate the real environment, DiffFuzz may produce false positives in vulnerability discovery, since its simulated execution may differ from execution on real devices.

Third, the current implementation of DiffFuzz is limited to function-level fuzzing of handler functions in firmware running web servers. It therefore cannot discover vulnerabilities located outside handler functions. In addition, it assumes that the handler functions retrieve user inputs through identifiable input functions and contain most of the request processing logic. Its applicability is limited when the target firmware does not run a web server or when the web server does not exhibit this structure. Consequently, the number of firmware images to which DiffFuzz can be applied is limited, and DiffFuzz cannot be applied as broadly as Mango, which analyzed a large dataset containing 1,698 firmware samples from nine vendors (Gibbs et al., 2024).

Last but not least, DiffFuzz uses LLMs in several steps, which may cause vulnerabilities to be missed due to the stochastic nature of current LLMs (Radford et al., 2019). We use LLMs to help recognize input functions and evaluate differences between handler functions; however, since current LLMs are probability-based (Radford et al., 2019), even though we employ a self-consistency strategy (Wang et al., 2022) to improve reliability, it remains possible for LLMs to produce incorrect results. For example, the correct input function may not be recognized, or a handler function that differs from another may be incorrectly marked as identical. In such cases, DiffFuzz may skip the fuzzing of certain functions that could contain vulnerabilities. The situation is further exacerbated if the LLMs used are malicious and intentionally produce incorrect outputs (Mo et al., 2024), or if the LLMs used are misled by malicious inputs (Bernstein et al., 2026).

7. Related Work

IoT firmware fuzzing. Existing research on IoT firmware fuzzing can be broadly classified into black-box fuzzing (Chen et al., 2018; Feng et al., 2021; Ma et al., 2024; Liu et al., 2024a; Wang et al., 2024) and grey-box fuzzing (Zheng et al., 2019, 2022; Tay et al., 2023; Li et al., 2024; Oliinyk et al., 2024; Xiao et al., 2025). Black-box IoT fuzzing lacks access to internal runtime information (Manès et al., 2019; Zhu et al., 2022) and usually achieves lower fuzzing throughput because it relies on real IoT devices (Zaddach et al., 2014) or full-system emulation (Chen et al., 2016). In contrast, grey-box fuzzing can obtain execution paths and progressively explore the program under test in a stepwise manner (Manès et al., 2019; Zhu et al., 2022). Firm-AFL (Zheng et al., 2019) employs augmented process emulation to execute the program primarily in user-mode emulation, switching to full-system emulation only when necessary to improve throughput. EQUAFL (Zheng et al., 2022) and Greenhouse (Tay et al., 2023) automatically configure the execution environment to run in user-mode emulation by recording and migrating the environment from full-system emulation or by patching the binary. However, Greenhouse still succeeds on only 39.7% of firmware images because it fuzzes the entire embedded program in user mode. Some studies focus on specific programs, such as BusyBox (Oliinyk et al., 2024), or particular multi-process services, such as HouseFuzz (Xiao et al., 2025), which limit their general applicability to firmware fuzzing. SHFuzz (Li et al., 2024) directs fuzzing to handlers in service programs that use sensitive APIs, yet it still targets the entire program, making it difficult to efficiently explore all execution paths.

IoT firmware static vulnerability analysis. Static vulnerability analysis of firmware does not require executing the firmware and is generally faster than fuzzing (Shoshitaishvili et al., 2015; Cheng et al., 2018; Redini et al., 2020; Cheng et al., 2021; Chen et al., 2023; Gibbs et al., 2024; Liu et al., 2025). Firmalice (Shoshitaishvili et al., 2015) provides a framework based on symbolic execution and program slicing to detect authentication bypass vulnerabilities in binary firmware. KARONTE (Redini et al., 2020) leverages static analysis techniques to perform multi-binary taint tracking. SaTC (Chen et al., 2023) exploits

shared keywords between IoT device front-end and back-end components to identify potential vulnerabilities. Mango (Gibbs et al., 2024) enhances the precision and scalability of taint analysis through backward sink-to-source analysis. LATTE (Liu et al., 2025) further improves taint propagation using large language models. However, static vulnerability analysis often results in false positives. For example, Mango reports a false positive rate of approximately 43% for command injection TruPoCs (Gibbs et al., 2024).

Function-level fuzzing. Recently, function-level fuzzing has been proposed to target arbitrary functions (Murali et al., 2024; Liu et al., 2024b). FuzzSlice (Murali et al., 2024) uses function-level fuzzing to reduce the false positives produced by static analysis warnings and automatically generates small programs containing the functions flagged by static analysis for fuzzing within a predefined time limit. AFGen (Liu et al., 2024b), which follows a similar idea, further extracts similar calling contexts of target functions from the program. However, both approaches require access to source code, which is typically unavailable in firmware. SFuzz (Chen et al., 2022) proposes a different slice-based fuzzing approach, targeting code slices that connect functions such as `recvfrom` to sink functions like `memcpy`. Slice-based fuzzing operates across multiple functions and thus works at a higher level than function-level fuzzing. SFuzz is generally applicable to RTOS environments but not to Linux-based IoT firmware. Another related concept is in-memory fuzzing (Takanen et al., 2018; Kim et al., 2024), which repeatedly fuzzes a function or an instruction block after capturing a runtime snapshot of the program. In contrast, function-level fuzzing does not require such runtime snapshot capture before fuzzing.

Binary code similarity detection (BCSD). Numerous BCSD approaches have been proposed (Haq and Caballero, 2021), and this research area remains highly active (Wang and Lin, 2026). BinDiff (Google, 2023) is a widely used BCSD tool that relies on structural graph matching to identify similarities. Deep learning-based methods have also been introduced (VenkataKeerthy et al., 2025; Dong et al., 2025). For example, VexIR2Vec (VenkataKeerthy et al., 2025) produces function embeddings using a Siamese network over VEX-IR. However, deep learning-based approaches inevitably suffer from Out-of-Distribution (OOD) degradation (Wang and Lin, 2026). vSim (Wang and Lin, 2026) instead focuses on precise semantic value matching by conducting under-constrained symbolic execution on lifted IR; however, it requires lifting to IR and inherits the limitations of symbolic execution, such as potential state explosion. Recent BCSD approaches have also explored the use of LLMs to validate uncertain function pairs (Dong et al., 2025). Furthermore, when BCSD is directly applied to vulnerability detection (Haq and Caballero, 2021; He et al., 2022), it can only identify vulnerabilities that have already been observed in similar binaries; moreover, unlike fuzzing-based approaches, it may produce false positives.

8. Conclusion

We present DiffFuzz, a homology-aware function-level fuzzing approach that leverages implementation similarities across firmware. DiffFuzz employs LLM-assisted static analysis to identify the required function information and to generate a nonredundant test plan. It further incorporates symbolic execution and taint analysis to enable more efficient function-level fuzzing. We evaluated DiffFuzz on 18 firmware images and showed that it achieved an average reduction of 56.69% in testing redundancy. DiffFuzz discovered 292 zero-day vulnerabilities, 61 of which have been assigned CVE IDs. Our evaluation also demonstrated that DiffFuzz outperformed existing state-of-the-art static and dynamic approaches.

CRediT authorship contribution statement

Pulin Chen: Investigation, Methodology, Software, Validation, Visualization, Writing – original draft. **Yingpei Zeng:** Conceptualization, Formal analysis, Funding acquisition, Project administration, Supervision, Writing – review & editing. **Xiangpu Song:** Writing – review & editing. **Yuhang Pan:** Data curation. **Xiao Li:** Data curation. **Shanqing Guo:** Resources, Supervision, Funding acquisition.

Acknowledgements

The work of Y. Zeng was supported in part by the Zhejiang Provincial Natural Science Foundation of China (No. LY22F020022), in part by the National Natural Science Foundation of China (No. 61902098), and in part by Zhejiang Provincial Key Laboratory for Sensitive Data Security Protection and Confidentiality Management (No. 2024E10048). The work of S. Guo was supported by Major Scientific and Technological Innovation Projects of Shandong Province, China (No. 2024CXGC010114), National Natural Science Foundation of China (No. 62372268), Shandong Provincial Natural Science Foundation, China (No. ZR2022LZH013, No. ZR2021LZH007), and Jinan City "20 New Universities" Funding Project (2021GXRC084).

Declaration of Generative AI and AI-assisted Technologies in the Writing Process

During the preparation of this work the authors used Claude and ChatGPT in order to polish and refine the language of the manuscript. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

Data Availability

We will release our tool after our paper has been published.

Declaration of Competing Interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Anthropic, 2026. Claude-opus-4.6. Retrieved from https://www.anthropic.com/. accessed: March 27, 2026.

Bernstein, S., Beste, D., Ayzenshteyn, D., Schonherr, L., Mirsky, Y., 2026. Trust me, i know this function: Hijacking llm static analysis using bias, in: Proceedings of the Network and Distributed System Security Symposium (NDSS).

Brubacher, D., 1999. Detours: Binary interception of win32 functions, in: Proceedings of the 3rd USENIX Windows NT Symposium.

Chen, D.D., Woo, M., Brumley, D., Egele, M., 2016. Towards automated dynamic analysis for Linux-based embedded firmware, in: Proceedings of the Network and Distributed System Security Symposium (NDSS), pp. 1–1. https://doi.org/10.14722/ndss.2016.23415.

Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W.C., Sun, M., Yang, R., Zhang, K., 2018. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing, in: Proceedings of the Network and Distributed System Security Symposium (NDSS), pp. 1–15. https://doi.org/10.14722/ndss.2018.23159.

Chen, L., Cai, Q., Ma, Z., Wang, Y., Hu, H., Shen, M., Liu, Y., Guo, S., Duan, H., Jiang, K., et al., 2022. Sfuzz: Slice-based fuzzing for real-time operating systems, in: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 485–498. https://doi.org/10.1145/3548606.3559367.

Chen, L., Wang, Y., Linghu, J., Hou, Q., Cai, Q., Guo, S., Xue, Z., 2023. Satc: Shared-keyword aware taint checking for detecting bugs in embedded systems. IEEE Transactions on Dependable and Secure Computing 21, 2421–2433. https://doi.org/10.1109/tdsc.2023.3307430/mm1.

Cheng, K., Li, Q., Wang, L., Chen, Q., Zheng, Y., Sun, L., Liang, Z., 2018. Dtaint: detecting the taint-style vulnerability in embedded device firmware, in: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE. pp. 430–441. https://doi.org/10.1109/dsn.2018.00052.

Cheng, K., Liu, T., Guan, L., Liu, P., Li, H., Zhu, H., Sun, L., 2021. Finding taint-style vulnerabilities in Linux-based embedded firmware with SSE-based alias analysis. arXiv preprint arXiv:2109.12209 https://doi.org/10.48550/arXiv.2109.12209.

clearbluejar, 2024. ghidriff: Python Command-Line Ghidra Binary Diffing Engine. GitHub. Retrieved from https://github.com/clearbluejar/ghidriff. accessed: February 13, 2025.

Dong, C., Guo, J., Yang, S., Li, Y., Fang, D., Xiao, Y., Chen, Y., Sun, L., 2025. Advancing binary code similarity detection via context-content fusion and llm verification, in: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 304–316. https://doi.org/10.1109/ASE63991.2025.00033.

Feng, X., Sun, R., Zhu, X., Xue, M., Wen, S., Liu, D., Nepal, S., Xiang, Y., 2021. Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 337–350. https://doi.org/10.1145/3460120.3484543.

Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M., 2020. AFL++: Combining incremental steps of fuzzing research, in: 14th USENIX Workshop on Offensive Technologies (WOOT 20).

Galler, B.A., Fisher, M.J., 1964. An improved equivalence algorithm. Communications of the ACM 7, 301–303. https://doi.org/10.1145/364099.364331.

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

Google, 2023. BinDiff. GitHub. Retrieved from https://github.com/google/bindiff. accessed: February 5, 2025.

Grand View Research, 2026. Wireless router market (2025 - 2033). Retrieved from https://www.grandviewresearch.com/industry-analysis/wireless-router-market-report. accessed: July 28, 2026.

Haq, I.U., Caballero, J., 2021. A survey of binary code similarity. ACM Computing Surveys (CSUR) 54, 1–38. https://doi.org/10.1145/3446371.

He, D., Yu, X., Li, T., Chan, S., Guizani, M., 2022. Firmware vulnerabilities homology detection based on clonal selection algorithm for IoT devices. IEEE Internet of Things Journal 9, 16438–16445. https://doi.org/10.1109/jiot.2022.3152364.

Khan, M.A., Salah, K., 2018. Iot security: Review, blockchain solutions, and open challenges. Future generation computer systems 82, 395–411.

Kim, T., Hong, S., Cho, Y., 2024. AIMFuzz: Automated function-level in-memory fuzzing on binaries, in: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, pp. 1510–1522. https://doi.org/10.1145/3634737.3644996.

Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M., 2018. Evaluating fuzz testing, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138. https://doi.org/10.1145/3243734.3243804.

Li, X., Zhao, L., Wei, Q., Wu, Z., Shi, W., Wang, Y., 2024. SH-Fuzz: Service handler-aware fuzzing for detecting multi-type vulnerabilities in embedded devices. Computers & Security 138, 103618. https://doi.org/10.1016/j.cose.2023.103618.

Li, Y., Zeng, Y., Song, X., Guo, S., 2025. Improving seed quality with historical fuzzing results. Information and Software Technology 179, 107651. https://doi.org/10.1016/j.infsof.2024.107651.

Liu, H., Gan, S., Zhang, C., Gao, Z., Zhang, H., Wang, X., Gao, G., 2024a. Labrador: Response guided directed fuzzing for black-box IoT devices, in: 2024 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 1920–1938. https://doi.org/10.1109/sp54263.2024.00127.

Liu, P., Sun, C., Zheng, Y., Feng, X., Qin, C., Wang, Y., Xu, Z., Li, Z., Di, P., Jiang, Y., et al., 2025. LLM-powered static binary taint analysis. ACM Transactions on Software Engineering and Methodology 34, 1–36. https://doi.org/10.1145/3711816.

Liu, P., Zheng, Y., Sun, C., Qin, C., Fang, D., Liu, M., Sun, L., 2023. Fits: Inferring intermediate taint sources for effective vulnerability analysis of IoT device firmware, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, pp. 138–152. https://doi.org/10.1145/3623278.3624759.

Liu, Y., Wang, Y., Jia, X., Zhang, Z., Su, P., 2024b. Afgen: Whole-function fuzzing for applications and libraries, in: 2024 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 1901–1919. https://doi.org/10.1109/sp54263.2024.00011.

Ma, X., Luo, L., Zeng, Q., 2024. From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter IoT devices, in: 33rd USENIX Security Symposium (USENIX Security 24), pp. 4783–4800.

Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M., 2019. The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering 47, 2312–2331. https://doi.org/10.1109/tse.2019.2946563.

Metzman, J., Szekeres, L., Simon, L., Sprabery, R., Arya, A., 2021. Fuzzbench: an open fuzzer benchmarking platform and service, in: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp. 1393–1403.

Microsoft, 2025. Announcing the firmware analysis public preview. Retrieved from https://techcommunity.microsoft.com/blog/iotblog/announcing-the-firmware-analysis-public-preview/4420116. accessed: July 28, 2026.

Mo, L., Wang, B., Chen, M., Sun, H., 2024. How trustworthy are open-source llms? an assessment under malicious demonstrations shows their vulnerabilities, in: Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics:

Human Language Technologies (Volume 1: Long Papers), pp. 2775–2792.

Mordor Intelligence, 2026. Wireless router companies: Leaders, top & emerging players and strategic moves. Retrieved from https://www.mordorintelligence.com/industry-reports/wireless-router-market/companies. accessed: July 28, 2026.

Murali, A., Mathews, N., Alfadel, M., Nagappan, M., Xu, M., 2024. Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pp. 1–13. https://doi.org/10.1145/3597503.3623321.

National Security Agency, 2024. Ghidra. GitHub. Retrieved from https://github.com/NationalSecurityAgency/ghidra. accessed: February 9, 2025.

NETGEAR, 2026. June 2026 netgear security advisory. Retrieved from https://kb.netgear.com/000070811/June-2026-NETGEAR-Security-Advisory. accessed: July 28, 2026.

NixOS Community, Dolstra, E., 2004. Patchelf: A small utility to modify the dynamic linker and rpath of elf executables. GitHub. Retrieved from https://github.com/NixOS/patchelf. accessed: February 21, 2025.

Oliinyk, Y., Scott, M., Tsang, R., Fang, C., Homayoun, H., et al., 2024. Fuzzing BusyBox: Leveraging LLM and crash reuse for embedded bug unearthing, in: 33rd USENIX Security Symposium (USENIX Security 24), pp. 883–900.

OWASP, 2026. Owasp firmware security testing methodology. Retrieved from https://github.com/scriptingxss/owasp-fstm#stage-2-obtaining-firmware. accessed: July 28, 2026.

Peng, H., Shoshitaishvili, Y., Payer, M., 2018. T-Fuzz: Fuzzing by program transformation, in: 2018 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 697–710. https://doi.org/10.1109/SP.2018.00056.

Pham, V.T., Böhme, M., Santosa, A.E., Căciulescu, A.R., Roychoudhury, A., 2019. Smart greybox fuzzing. IEEE Transactions on Software Engineering 47, 1980–1997. https://doi.org/10.1109/tse.2019.2941681.

Qin, C., Peng, J., Liu, P., Zheng, Y., Cheng, K., Zhang, W., Sun, L., 2023. Ucrf: Static analyzing firmware to generate under-constrained seed for fuzzing SOHO router. Computers & Security 128, 103157. https://doi.org/10.1016/j.cose.2023.103157.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al., 2019. Language models are unsupervised multitask learners. OpenAI blog 1, 9.

Rajpal, M., Blum, W., Singh, R., 2017. Not all bytes are equal: Neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596 https://doi.org/10.48550/arXiv.1711.04596.

Ramos, D.A., Engler, D., 2015. Under-Constrained symbolic execution: Correctness checking for real code, in: 24th USENIX Security Symposium (USENIX Security 15), pp. 49–64.

Redini, N., Machiry, A., Wang, R., Spensky, C., Continella, A., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware, in: 2020 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 1544–1561. https://doi.org/10.1109/sp40000.2020.00036.

Research and Markets, 2026. Wireless router market report by component type, end user, country and company analysis 2026-2034. Retrieved from https://www.researchandmarkets.com/report/wireless-router. accessed: July 28, 2026.

Roberts, C., 2019. Modern vulnerability research techniques on embedded systems. Breaking Bits. Retrieved from https://breaking-bits.gitbook.io/breaking-bits/vulnerability-discovery/reverse-engineering/modern-approaches-toward-embedded-research. accessed: February 25, 2025.

Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G., 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware, in: Proceedings of the Network and Distributed System Security Symposium (NDSS), pp. 1–1. https://doi.org/10.14722/ndss.2015.23294.

Song, X., Zeng, Y., Wu, J., Li, H., Zuo, C., Zhao, Q., Guo, S., 2025. CSFuzzer: A grey-box fuzzer for network protocol using context-aware state feedback. Computers & Security , 104581https://doi.org/10.1016/j.cose.2025.104581.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution, in: Proceedings of the Network and Distributed System Security Symposium (NDSS), pp. 1–16. https://doi.org/10.14722/ndss.2016.23368.

Takanen, A., Demott, J.D., Miller, C., Kettunen, A., 2018. Fuzzing for software security testing and quality assurance. Artech House.

Tay, H.J., Zeng, K., Vadayath, J.M., Raj, A.S., Dutcher, A., Reddy, T., Gibbs, W., Basque, Z.L., Dong, F., Smith, Z., et al., 2023. Greenhouse:single-service rehosting of linux-based firmware binaries in user-space emulation, in: 32nd USENIX Security Symposium (USENIX Security 23), pp. 5791–5808.

TP-Link, 2026. Gpl code center. Retrieved from https://www.tp-link.com/sg/support/gpl-code/. accessed: July 28, 2026.

VenkataKeerthy, S., Banerjee, S., Dey, S., Andaluri, Y., PS, R., Kalyanasundaram, S., Quintão Pereira, F.M., Upadrasta, R., 2025. Vexir2vec: an architecture-neutral embedding framework for binary similarity. ACM Transactions on Software Engineering and Methodology 34, 1–54. https://doi.org/10.1145/3721481.

Wang, F., Shoshitaishvili, Y., 2017. Angr-the next generation of binary analysis, in: 2017 IEEE Cybersecurity Development (SecDev), IEEE. pp. 8–9. https://doi.org/10.1109/secdev.2017.14.

Wang, H., Lin, Z., 2026. vsim: Semantics-aware value extraction for efficient binary code similarity analysis, in: Proceedings of the Network and Distributed System Security Symposium (NDSS).

Wang, J., Yu, L., Luo, X., 2024. Llmif: Augmented large language model for fuzzing IoT devices, in: 2024 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 881–896. https://doi.org/10.1109/sp54263.2024.00211.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., Zhou, D., 2022. Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171 https://doi.org/10.48550/arXiv.2203.11171.

Xia, Z., Zeng, Y., Song, X., Guo, S., Wu, T., 2025. HSPFuzzer: High-speed network protocol fuzzing with connection reuse. IEEE Internet of Things Journal https://doi.org/10.1109/jiot.2025.3590178.

Xiao, H., Wei, Z., Dai, J., Li, B., Zhang, Y., Yang, M., 2025. Housefuzz: Service-aware grey-box fuzzing for vulnerability detection in Linux-based firmware, in: 2025 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 3801–3819. https://doi.org/10.1109/sp61157.2025.00213.

Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T., 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing, in: 27th USENIX Security Symposium (USENIX Security 18), pp. 745–761.

Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al., 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares, in: Proceedings of the Network and Distributed System Security Symposium (NDSS), pp. 1–16. https://doi.org/10.14722/ndss.2014.23229.

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

Zheng, Y., Li, Y., Zhang, C., Zhu, H., Liu, Y., Sun, L., 2022. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 417–428. https://doi.org/10.1145/3533767.3534414.

Zhu, X., Wen, S., Camtepe, S., Xiang, Y., 2022. Fuzzing: a survey for roadmap. ACM Computing Surveys (CSUR) 54, 1–36. https://doi.org/10.1145/3512345.