HouseFuzz: Service-Aware Grey-Box Fuzzing for Vulnerability Detection in Linux-Based Firmware

Haoyu Xiao*, Ziqi Wei*, Jiarun Dai, Bowen Li, Yuan Zhang, Min Yang
Fudan University, Shanghai, China

***Abstract*—To date, grey-box fuzzing has become an essential technique to detect vulnerabilities implied in Linux-based firmware. However, existing fuzzing approaches commonly encounter three overlooked obstacles stemming from firmware service characteristics, which largely hinder the effectiveness and efficiency of vulnerability identification. Firstly, the multi-process nature of firmware services is oversimplified during both the emulation and the fuzzing procedures, limiting the scope of firmware testing. Furthermore, firmware services usually incorporate customized service protocols, which feature rich and stringent semantic constraints, causing unique challenges for input generation. To address these obstacles, this paper proposes a service-aware grey-box fuzzing tool HOUSEFUZZ. During the firmware emulation, HOUSEFUZZ carefully traverses the system initialization procedure for identifying those network-facing and daemon processes overlooked by existing approaches. After that, during the fuzzing procedure, HOUSEFUZZ features a multi-process fuzzing framework, enabling the comprehensive inspection of firmware services activated via multiple processes. Furthermore, HOUSEFUZZ leverages both offline and online firmware service analysis to capture the token-level semantic constraints of customized service protocols, based on which HOUSEFUZZ can effectively generate high-quality test cases. In evaluation, compared to SoTA grey-box firmware fuzzing approaches, HOUSEFUZZ identified 76% more network services, achieved 33.4% more code coverage, and detected 175% more 0-day vulnerabilities on the same firmware dataset.**

1. Introduction

With the rapid growth of the Internet of Things (IoT), Linux-based firmware has become ubiquitous, powering 43% of IoT devices [1]. Despite the ubiquity of Linux-based firmware, it remains highly susceptible to network attacks due to firmware vulnerabilities, causing severe security risks (e.g., remote code execution [2], [3]) to vendors and end-users. Considering the fact that most vulnerabilities reside in network services [4], [5] (e.g., web services) of Linux firmware, it is imperative for security professionals and organizations to detect and mitigate vulnerabilities in Linux-based firmware network services.

To detect vulnerabilities in network services of Linux-based firmware, grey-box fuzzing [6], [7], [8], [9] has been embraced as a fundamental approach. Technically, firmware grey-box fuzzing first identifies and emulates the network services to create a controlled running environment, then generates mutated test cases for execution, and collects execution feedback (e.g., code coverage) from the emulated service as a guidance for the follow-up fuzzing procedure. Hence obviously, emulation, test case generation, and fuzzing feedback guidance are three keystones to achieving a comprehensive code exploration and effective vulnerability detection. However, in this work, we highlight that existing firmware grey-box fuzzing approaches commonly encounter overlooked obstacles in each of these keystones, mainly due to the lack of awareness of Linux service characteristics:

- ❶ **Limitations in Service Emulation.** Linux-based firmware services typically operate through multiple processes, making it imperative to *"comprehensively identify all network-service-related processes for holistic service emulation"*. These include not only network-facing processes but also those triggered by inter-process communications (IPCs). However, existing practices in service emulation for fuzzing, either through system-emulation-based solutions [6], [10], [11] or process-emulation-based solutions [7], [8], [9], would inevitably miss essential processes due to inherent design flaws. Specifically, system-emulation-based solutions emulate the entire Linux system including the OS kernel and all initialized processes, where the network-facing processes can be identified through public network channels (e.g., TCP ports). However, system emulation usually fails due to emulation roadblocks [10], [11], preventing the establishment of network channels. Process-emulation-based solutions commonly identify and emulate processes based on a process name whitelist. Obviously, the whitelist-based heuristics can hardly ensure a high recall of process identification. Consequently, given a limited number of emulated services, existing firmware grey-box fuzzing tools can hardly ensure a large scope of code exploration.
- ❷ **Limitations in Fuzzing Feedback Guidance.** Similarly, due to the multi-process runtime nature of Linux-based firmware services, one should *"build a multi-process fuzzing framework to monitor the execution feedback of all target-service-related processes, and accordingly guide the fuzzing procedure"*. However, existing grey-box firmware fuzzing tools commonly over-simplify the multi-process runtime nature as the single-process one. To sum up, they limit the feedback collection scope to a

*. Co-first author.

single process (i.e., usually the network-facing process). In such a manner, they fall short in detecting vulnerabilities that require multi-process cooperation to trigger [12]. Furthermore, they would miss necessary service execution feedback stemming from those unobserved processes, limiting the exploration space of firmware code.

- ❸ **Limitations in Test Case Generation.** Firmware services can be highly customized [13] to meet diverse product requirements. The customized service protocol would additionally introduce rich semantic constraints (e.g., specific key-value pairs with strict data constraints or pair-to-pair dependencies) to the standard service protocol (e.g., HTTP). Hence, one should *"generate semantically valid test cases to ensure the effectiveness of fuzzing"*. However, current grey-box firmware fuzzing tools lack awareness of customized service protocols. In general, they [7], [8], [9] usually leverage well-crafted seeds built upon standard protocol templates (e.g., HTTP), then merely apply random mutations on protocol contents. Hence, they cannot achieve satisfactory fuzzing performance.

Considering the above, we propose HOUSEFUZZ, a novel service-aware grey-box fuzzing tool, for effective vulnerability detection in Linux-based firmware. HOUSEFUZZ is characterized by three key aspects. First, to determine which processes require emulation, HOUSEFUZZ traverses the system's initialization procedure, which is responsible for setting up all network services. This analysis allows HOUSEFUZZ to identify the essential processes associated with each service, thereby enabling a more comprehensive service emulation compared to prior practices in firmware fuzzing [7], [8], [9]. Second, HOUSEFUZZ introduces an innovative multi-process fuzzing framework. The framework leverages code coverage collected from all service processes to guide comprehensive testing and detects vulnerabilities with multi-process vulnerability oracles. Third, HOUSEFUZZ formalizes the semantic constraints of customized service protocols with a Token Dependency Graph (TDG) . Specifically, TDG symbolizes customized semantic constraints as dependencies among tokens extracted from firmware. HOUSEFUZZ automatically infers TDG from firmware services with offline control-/data-flow analysis and online instrumentation-based analysis, and leverages TDG to ensure the semantic validity of generated test cases.

In evaluation, we designed extensive experiments to demonstrate the effectiveness of HOUSEFUZZ and our design choices. Among these experiments, HOUSEFUZZ detected 143 0-days with 45 CVE/CNVDs assigned, which demonstrates its overall effectiveness. Compared to SoTA firmware emulation approaches, HOUSEFUZZ identified 76% more network services on a dataset of 60 firmware images, which provide more fuzzing targets, where 12 more 0-days are discovered. This result demonstrates HOUSEFUZZ is effective in network service identification. Compared to SoTA firmware grey-box fuzzing approaches, HOUSEFUZZ discovered 33.4% more code coverage, and detected 175% more 0-day vulnerabilities on the same dataset consisting of 41 network services. The ablation study also demonstrates that the multi-process fuzzing framework and service-protocol-aware fuzzing technique greatly enhance the vulnerability detection capability of grey-box fuzzing in Linux-based firmware services.

Contributions. We position this paper with the following major contributions:

- We propose three key techniques to boost the effectiveness of existing grey-box fuzzing for Linux-based firmware, including holistic service identification and emulation, multi-process fuzzing framework, and service-protocol-guided test case generation.
- We implemented the three techniques into a prototype HOUSEFUZZ, and extensively evaluated it on real-world Linux-based firmware. Results show that HOUSEFUZZ significantly outperforms SoTA approaches in service discovery, code exploration, and vulnerability detection.
- We detected 177 vulnerabilities during experiments, where 156 are 0-days. We have responsibly disclosed these 0-days. Till now, we have received 45 CVEs/CNVDs.

2. Background

2.1. Services of Linux-Based Firmware

Concept of Linux-Based Firmware Service. Linux is an open-sourced system favored by firmware developers—43% firmware images are built upon Linux [1]. Linux strongly supports running multiple processes for complex firmware services. According to processes lifetime, there are long-running processes and utility processes. A long-running process provides a persistent channel accessible to service users or other processes; it can "invoke" utility processes to handle a short-term task (e.g., a single network request). We further classify long-running processes into network-facing processes and daemon processes based on whether the persistent channels can be accessed remotely or only locally. These three kinds of processes—network-facing, daemon, and utility processes can cooperate through inter-process communications (IPC) [12], which incur complexity as the running of one process might affect other running processes. For convenience, we leverage the term "service" to denote all cooperated processes that support the serving of one communication channel.

Research Scope of Firmware Service. A Linux-based firmware image may consist of multiple services that listen on either local or network channels. Local services serve only local channels (e.g., UNIX domain sockets) to support local functionalities (e.g., system monitoring), which can not be directly accessed from the network and thus require a stronger attack model (e.g., physical access). Network services serve network channels (e.g., a TCP port), and are more vulnerable to network attacks. Hence, in this work, we choose network services in firmware as our research targets.

2.2. Service Protocols of Linux-Based Firmware

The network services of Linux-based firmware are usually built upon standard network protocols and incorporate customized application-layer protocols (detailed as follows).

Standard Service Protocol. Standard service protocols are defined by official documentation, such as the HTTP protocol [14]. Firmware manufacturers generally adhere to these public protocol specifications when implementing service protocols. For instance, [15] indicates that the most widely used protocols are derived from recognized standard service protocols, including UPnP [16], HTTP [14], JetDirect [17], LPR [18], and mDNS [19]. Current firmware fuzzing practices [7], [8], [9] recommend crafting high-quality initial seeds using standard HTTP protocol templates to enhance the syntactic correctness of the mutated test cases.

Customized Service Protocol. To implement product-specific service features, firmware vendors typically apply application-layer customization upon standard service protocols [13], which creates customized service protocols. Specifically, a customized service protocol not only introduces thousands of new tokens (i.e. concrete data) for synthesizing protocol contents [4] but also constrains semantic relationships among these tokens [20]. Therefore, to synthesize a semantically valid test case, it is necessary to correctly position multiple interrelated tokens; otherwise, the test case might be rejected by superficial input validation. However, existing test case generation approaches [21], [22], [23], [24], [25], [26] totally disregard customized service protocols, restricting the exploration of potentially vulnerable paths that are guarded by semantic constraints.

3. Challenges and Insights

Here, we first illustrate a motivation example in §3.1 to demonstrate the necessity of considering the multi-process nature and customized service protocols when fuzzing Linux-based firmware. After that, we discuss the challenges to performing effective firmware service fuzzing, and outline our critical insights for overcoming these challenges in §3.2, §3.3, and §3.4, respectively.

3.1. Motivating Example

Figure 1 illustrates a buffer overflow vulnerability implied in Linux-based firmware. The vulnerability can be triggered by sequentially executing four specific steps:

- **Step-❶ Service Initialization.** The *ncc* process, which is automatically launched during system booting, initiates the *mini_httpd* process. This process then binds to a network channel and awaits incoming requests (L24-L26).
- **Step-❷ Request Sending.** An attacker sends a malicious packet to activate the *mini_httpd* process (L27,L28).
- **Step-❸ Request Handling.** The *handle_request()* function of *mini_httpd* then parses the HTTP request accordingly. It confirms that the request ends with *".ccp"* at L34 (i.e., indicating a special request for *ncc*), and thus dispatches it to *ncc* process through socket (L35,L36).
- **Step-❹ Vulnerability Triggering.** When *ncc* receives the IPC request at L3, it invokes *ipc_handler()* at L6 to process it. The function checks several data constraints at (L8, L13-L17). The execution eventually reaches the vulnerable code, resulting in a buffer overflow at L19—the long string read by *"pc_ip"* is used to format the *buf* string without any length validation.

3.2. Challenges and Insights for Service Identification

Challenges. To trigger the vulnerability illustrated in Figure 1, both the network-facing process *mini_httpd* and the daemon process *ncc* should be identified as a full-featured service for a faithful emulation. However, existing approaches can hardly achieve this goal. To be specific, whitelist-based approaches [7], [8], [9] identify network-facing and daemon processes only when their names occur in given name whitelists. Unfortunately, the *ncc* process would be missed since its name is not a common entry in such whitelists. System-emulation-based approaches [10], [11] emulate all processes and identify network-facing processes when they establish network channels (e.g. L24). However, emulation issues may raise exceptions that interrupt the system emulation [11] before the establishment of network channels or IPC channels. The challenging point is that emulation exceptions might arise in various processes, but existing fuzzing works only monitor and fix exceptions in a single process [9], hindering holistic service emulation.

Key Insights. We find that exceptional processes can be identified by observing abnormal process events (e.g., crashes) and exception code can be pinpointed through a deep analysis of the process execution trace. We tend to resort to system emulation for service emulation while carefully identifying and patching the exception code in all service-related processes. This design choice helps the emulation traverse more system initialization procedures and identify more network services.

3.3. Challenges and Insights for Multi-Process Fuzzing

Challenges. Current grey-box firmware fuzzing techniques [6], [7], [8], [9] underestimate the complexity of Linux services that involve multiple processes, focusing solely on the network-facing process *mini_httpd* as their fuzzing target. Specifically, These fuzzing techniques are only guided with the code coverage of *mini_httpd*, while discarding the coverage of *ncc*, which is valuable for vulnerability triggering. This narrow focus results in a failure to uncover the vulnerability illustrated in Figure 1.

Key Insights. Here, our goal is to design a user-space multi-process fuzzing framework, which can not only monitor multi-process execution feedback but also offer a more favorable overhead compared with system-level fuzzing [11] [6]. Specifically, the proposed framework can monitor and merge the code coverage of all service-related processes to offer comprehensive guidance for service fuzzing. Besides, it features multi-process vulnerability oracles to notify the vulnerability triggering in different processes.

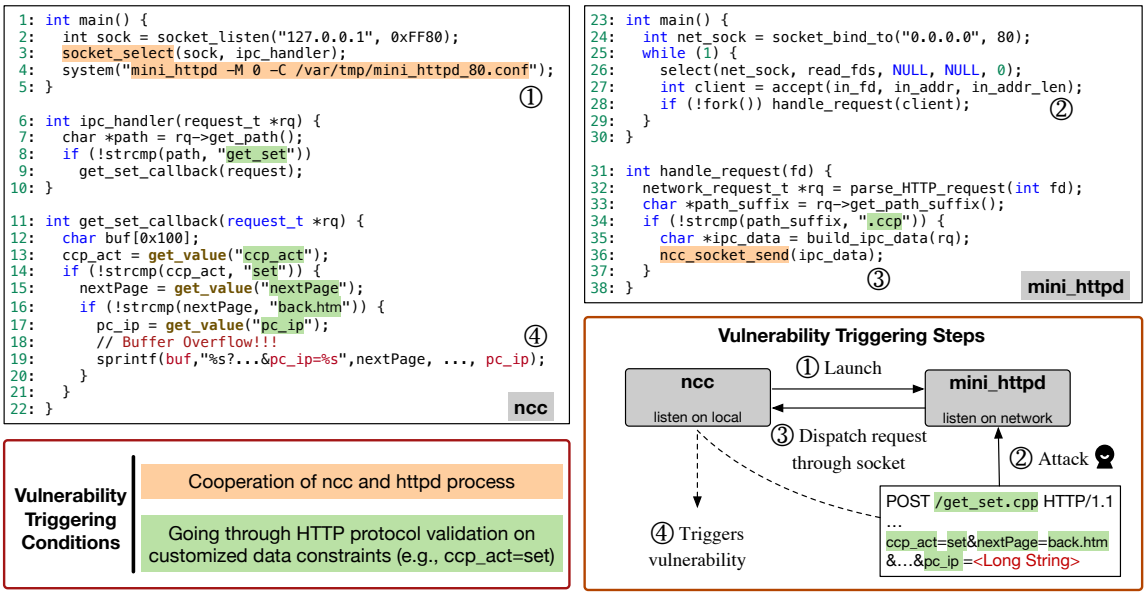

Figure 1: Motivating Example: A buffer overflow vulnerability in real-world Linux-based firmware. (We've restructured the decompiled code for ease of understanding)

3.4. Challenges and Insights for Handling Customized Service Protocols

Challenges. The vulnerability depicted in Figure 1 requires an attack input that adheres not only to the syntax of the standard HTTP protocol (validated at L32) but also meets the complex semantic constraints imposed by the customized service protocol. For example, to pass the verification at L14 in the *ncc* process, *"ccp_act"* and *"set"* must be combined to form a semantically-valid key-value pair. Unfortunately, existing test case generation [21], [22], [23], [24], [25], [26] approaches totally ignore such constraints and can hardly bypass the check.

Key Insights. As shown in Figure 1, the customized service protocol provides three valuable hints for generating high-quality test cases. First, *"set"* and *"ccp_act"* are interesting values to be inserted into a test case. Second, *"set"* and *"ccp_act"* correspond to a key and value respectively, which suggests they should be inserted as the left and right operands of an assignment. Third, when *"ccp_act"* is used as a field name, it is meaningful to use *"set"* as a corresponding value. Our idea is to formalize customized service protocols based on these hints, then infer the customized service protocols by analyzing firmware services, and finally leverage the inferred protocols to guide test case generation.

4. Overview of HouseFuzz

Figure 2 shows the overall workflow of HouseFuzz. HouseFuzz takes a firmware image and a set of standard service protocols as inputs, and automatically detects vulnerabilities with grey-box fuzzing. Initially, HouseFuzz emulates the services under test (§5). The core is to identify network services by carefully traversing the system initialization procedure. Then, HouseFuzz leverages multi-process fuzzing framework (§6) and service-protocol-guided fuzzing (§7), to improve existing single-process and protocol-unaware firmware fuzzing. The multi-process grey-box fuzzing framework is featured with two designs: it is guided by multi-process code coverage, and it detects vulnerabilities with multi-process vulnerability oracles. The service-protocol-guided fuzzing technique infers standard and customized service protocols and leverages the inferred protocols to generate high-quality test cases with syntax and semantic validity.

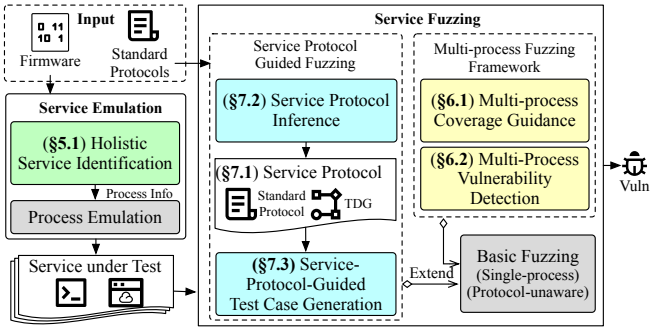

Figure 2: HouseFuzz Overview

5. Service Emulation

The service emulation consists of two steps. First, it identifies network services in the firmware (§5.1), specifically, the network-facing and daemon processes. Then, it emulates these processes with existing process emulation techniques [9] (§5.2).

5.1. Holistic Service Identification

The basic idea of service identification is to analyze the system initialization procedure. This is primarily because the

system initialization procedure is responsible for establishing both network and IPC channels for services. These channels then serve as the indicators for identifying network-facing and daemon processes. For identifying broader network services using this approach, it is crucial to traverse the system initialization procedure thoroughly. HOUSEFUZZ improves the traversing by carefully identifying and addressing emulation exceptions that interrupt or hang system initialization. We first introduce how HOUSEFUZZ emulates the system initialization (§5.1.1), then how HOUSEFUZZ identifies network services, including network-facing (§5.1.2 and daemon processes (§5.1.3).

5.1.1. Initialization Emulation. Linux system invokes a long-running user-space program—INIT [27] after the kernel is ready to perform system initialization. During system initialization, various emulation issues may cause exceptions that interrupt or hang up the initialization procedure [11]. (e.g., incorrect configurations of network devices and NVRAM storage devices). As a result, the initialization may fail to establish network and IPC channels, causing an overlook of corresponding service processes.

Inspired by GREENHOUSE [9], which patches service processes to bypass single-process service emulation roadblocks, HOUSEFUZZ fixes exceptions found during system initialization to identify more network services. Different from GREENHOUSE, which targets a single network process, HOUSEFUZZ targets the whole system initialization procedure involving multiple processes. Therefore, HOUSEFUZZ needs to identify which process raises an exception and then identify the exception code. A difficulty here is that exception processes exhibit quite similar behaviors as normal ones—both may programmatically abort or hang during service initialization. Therefore, simply using heuristic-based exception identification like GREENHOUSE inevitably yields false positives. To address this challenge, our idea is to use established IPC channel numbers to determine real exceptions.

Algorithm 1 depicts the overall INIT emulation algorithm of HOUSEFUZZ. Specifically, the algorithm first identifies the INIT program in the firmware image (Line 1) using name matching (the program is usually named as *"init"* or *"preinit"* [11]). Then it starts an emulation patching loop to fix all identified exceptions. The patching loop first launches the INIT process and collects execution traces of all processes (Line 4). When any emulation exception is detected from the traces (Line 6), HOUSEFUZZ patches the exception code (Line 13) and reruns the emulation. This loop continues until one of three conditions is satisfied: no exception is found (Line 7), the previous patch breaks the emulation (Line 5,10-12), or the loop reaches a maximum attempts limitation (Line 14-15).

❶ *Exception Process Identification.* In Line 6, the algorithm uses $IdentifyException()$ to identify which processes raise exceptions and what the exception code is. Specifically, HOUSEFUZZ uses 4 exception indicators in Table 1. When the INIT process aborts programmatically caused by validation (e.g., device checking) failures or

Algorithm 1 INIT Emulation

Input: Img - Firmware image,
N - Max attempt times
TO - Timeout of each emulation run
Output: Img - Patched firmware image,
C - Identified network or IPC channels
1: $P \leftarrow IdentifyInitProgram(Img)$
2: $C \leftarrow Nil$
3: **repeat**
4: $T \leftarrow TraceEmulation(Img, P, TO)$
5: $PrevC \leftarrow C; C \leftarrow IdentifyChannels(T)$
6: $Ex \leftarrow IdentifyException(T)$
7: **if** $Ex = Nil$ **then**
8: **return** Img, C
9: **end if**
10: **if** $PrevC.size > C.size$ **then**
11: **return** $PrevImg, PrevC$
12: **end if**
13: $PrevImg \leftarrow Img; Img \leftarrow PatchException(Img, Ex)$
14: $N \leftarrow N - 1$
15: **until** $N = 0$

unexpectedly due to memory faults (e.g., accessing an uninitialized and invalid pointer), the system initialization will be interrupted. HOUSEFUZZ identifies these situations when an *exit()* function call or a fatal signal is observed at the tailing of the INIT process execution trace. Similarly, abnormally interruptions of non-INIT processes may interrupt system initialization, where HOUSEFUZZ recognizes such exception only when a fatal exception signal (e.g., segment fault signal) is observed as non-INIT processes commonly exit.

Besides, the hangs of both INIT and non-INIT processes may prevent the execution of the initialization code. For instance, the INIT process may spawn an interactive debugging shell due to misconfigured NVRAM variables. For another instance, a non-INIT script may keep waiting for a network adapter before launching network-facing processes, which is endless because the network adapter never exists during emulation. Hanging is more covert than interruption due to a lack of strong indicators like signals. For a non-INIT process, HOUSEFUZZ regards it is stuck in busy waiting if it consumes lots of total CPU time (equal or more than $\frac{1}{3}$). For the INIT process, HOUSEFUZZ aggressively considers it hangs if no other exception is detected.

❷ *Exception Handling.* After an exception process is identified, HOUSEFUZZ analyzes the tailing execution trace of the process to identify the exception code. The rationale is that when a process aborts or hangs, the process stops its execution or keeps executing the same code within a loop, so the direct exception cause can be found at the tailing execution traces. Then, HOUSEFUZZ just patches the identified exception code at the tailing execution trace to prevent aborting or breaking the hanging loop. Specifically, HOUSEFUZZ selects a tailing function call that satisfies two conditions and rewrites the call instruction by replacing it with a *NOP* instruction. The two conditions are: (1) the called function size does not exceed a threshold, which minimizes the impact of patches; (2) the function call target is in the executable program of the process and is not a

stub call to external libraries, which avoids breaking shared libraries.

❸ *Robustness Enhancement.* As the expception identification involves heuristics and aggressive mechanisms, it may yield false positives. To address this problem, HOUSEFUZZ uses a reverting mechanism (Line 10-12). Specifically, when fewer network channels and IPC channels (i.e., C) were discovered during emulation, it means the initialization code logic is broken by patch. In this situation, HOUSEFUZZ regards the previous patch as error-prone and adopts prior emulation results to avoid errors and enhance robustness.

TABLE 1: Initialization emulation exceptions

Observed Situation	Explanation	Exception
INIT process terminates	It aborts programmatically or unexpectedly	Interrupts
Non-INIT process crashes	It aborts unexpectedly	
Non-INIT process takes lots of CPU time	It is busy waiting	Hangs
Other cases	INIT process may hang	

5.1.2. Network-Facing Process Identification. The network-facing process identification has two goals. First, it needs to identify network channels. Fuzzing must communicate with the network channels to execute test cases. Second, it requires inferring the correct command line to launch the process, including the program path, and arguments. Using actual arguments is essential for emulation fidelity; for example, missing a configuration path argument of a web server may cause it to fail to launch. HOUSEFUZZ first identifies processes that listen to network channels as network-facing processes. Specifically, HOUSEFUZZ collects system call traces during emulation, including PIDs, system call names and arguments, and inspects the concrete arguments of the network binding call (i.e., *bind()*) to extract the network channel. If the channel is exposed to the network (i.e., not "localhost"), it is considered as a network-facing process. Sometimes, a network-facing process (e.g., HTTP server) indirectly interacts with the network through an encrypted tunnel (e.g., HTTPS), so it may not directly listen to network channels. We denote them as proxied network-facing processes. HOUSEFUZZ identifies these processes based on known proxied ports (e.g., 80 for HTTP) because they can be fuzzed more efficiently than fuzzing them through the encrypted tunnel. After identifying a network-facing process, HOUSEFUZZ extracts its command line from the arguments of corresponding *execve()* system call.

5.1.3. Daemon Process Identification. Daemon process identification also has two goals. The first is to infer and reason the IPC dependencies between processes to find daemon processes for a given network service. Second, it also needs to identify the command lines of daemon processes.

According to Karonte [12], IPC channels of Linux processes are uniquely identified with specific "keys", such as socket file paths. IPCs through the same channel indicate dependencies between processes. HOUSEFUZZ first identifies "keys" of IPC channels with dynamic tracing, which is more accurate than static approach [12], then identifies daemon process if it establishes IPC channels dependent by other network service processes. Specifically, HOUSEFUZZ identifies IPC "keys" at system calls that create (e.g., *open()*) or bind (e.g., *bind()*) an IPC channel (e.g., file descriptors and socket addresses). When a process establishes IPC channels are read or written by other service processes, it is regarded as a daemon process. HOUSEFUZZ identifies the reading and writing event at specific system calls, such as *recv()* and *send()* and backtracks the system trace to identify the corresponding channel based on file descriptors.

5.2. Process Emulation

HOUSEFUZZ adopts the technique proposed by GREENHOUSE [9] for process emulation. To emulate a service, HOUSEFUZZ first launches its daemon processes, followed by the launch of the network-facing process. During this sequence, both the network-facing and daemon processes will autonomously initiate utility processes as required. During grey-box fuzzing, all service processes should be managed by the fuzzer. HOUSEFUZZ instruments the *execve()* syscall to trace all processes launched by the service under test with QEMU [28] for coverage collection and vulnerability detection.

6. Multi-Process Fuzzing Framework

Multi-process fuzzing framework has two core designs. First, it is guided with multi-process code coverage (§6.1), which is more comprehensive than a single-process one. Second, its vulnerability oracle detects vulnerabilities in all processes of the service under test (§6.2).

6.1. Multi-Process Coverage Guidance

Generally, code coverage guidance consists of three steps. First, the grey-box fuzzer records code coverage during the test case execution. Then, it detects whether the test case execution completes based on specific events (e.g., process termination). We denote this step as test completion event (TCE) detection. When the TCE is observed, the recorded coverage is regarded as the code coverage of the test case. Finally, fuzzing will use the collected coverage as guidance to identify whether the test case is valuable. In our multi-process fuzzing framework, these steps become more complex than in single-process fuzzing. We describe TCE detection in §6.1.1, coverage recording in §6.1.2, and coverage guidance in §6.1.3

6.1.1. Test Completion Event Detection. the main difficulty of multi-process code coverage guidance is identifying "when to collect code coverage". As depicted in Figure 3,

during multi-process fuzzing, test case executions will introduce a more complex process state transition than single-process fuzzing. Specifically, single-process code coverage collection only identifies whether one process stopped or the network resource (e.g., socket) is released. Instead, multi-process fuzzing requires detecting whether all service processes have finished test case handling. Otherwise, the fuzzing likely collects incomplete coverage. This is because some service processes are still handling the test case, where new coverage is ignored. Such incomplete coverage collection may overlook valuable test cases or cause instability that running the same test case likely yields different coverage. So HOUSEFUZZ regards the test case complete execution when TCEs of all processes have been observed. The TCEs of different processes require different detection methods. We describe how HOUSEFUZZ detects TCEs of utility, network-facing, and daemon processes in the following.

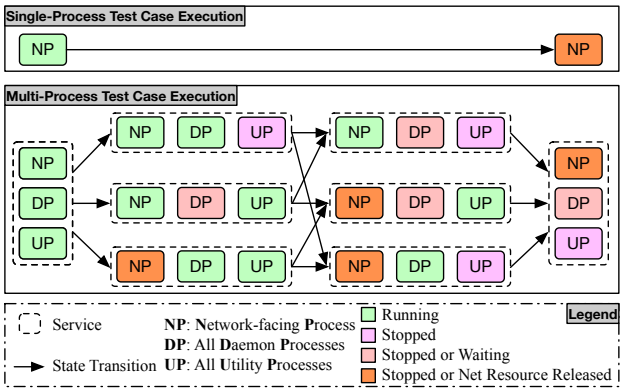

Figure 3: Different process state transition model during test case execution of single- and multiple-process fuzzing.

Utility processes have a short-term life cycle because they are launched specifically for handling single requests, which is similar to the PUT in single-process grey-box fuzzing. So HOUSEFUZZ just detects the terminating of a utility process as its TCE like traditional fuzzers.

Network-facing and daemon processes are long-running processes and typically keep running after test case handling. We find that network-facing processes usually release network resources after handling requests. Therefore, when HOUSEFUZZ detects the network resource (i.e., the network socket under fuzzing) is released (i.e., closed), it also deems the TCE of this process is satisfied.

Unlike network-facing processes that handle network requests, a *daemon process* mainly handles inter-process communications (IPCs). After handling IPC communication, the daemon process may reuse the IPC channels instead of closing them (e.g., maintaining an opening pipe for reading messages). In this situation, a daemon process repeatedly waits for IPC requests by invoking specific I/O listening system calls (e.g., select, poll), and starts handling the requests when it returns from these system calls. Based on this observation, HOUSEFUZZ instruments I/O listening system calls to detect completion of IPC request handling. When HOUSEFUZZ finds a daemon process re-invokes an I/O listening system call, HOUSEFUZZ assumes this process has finished IPC request handling. Completion of IPC request handling is not always the same as the completion of a test because network-facing and utility processes may issue multiple IPC requests during one test. To avoid this issue, HOUSEFUZZ detects TCEs of daemon processes after TCEs of all network-facing and utility processes have been detected. HOUSEFUZZ regards the test case complete execution when TCEs of all processes have been observed.

6.1.2. Coverage Recording. To record coverage of each process without data racing, HOUSEFUZZ allocates isolated shared memory as a coverage bitmap for each process when the process is launched; after being used by coverage guidance, all bitmaps will be cleared and placed into a shared memory pool for the next round of testing. To save memory space, the bitmaps of processes that load the same program will be merged into the same bitmap after the TCE of that process has been detected. To further reduce overhead, HOUSEFUZZ avoid collecting coverage from daemon processes that are not triggered during test case handling. This is achieved by maintaining an activation flag, which is set when they leave *accept()* system call.

6.1.3. Coverage Guidance. HOUSEFUZZ analyzes collected coverage bitmaps by comparing them with historical code coverage (i.e., virgin map in AFL++). If a fresh bit is found in the bitmap of any process, it means that the current test case has triggered a new path in the service. HOUSEFUZZ thus treats this test case as valuable and adds it to the seed queue for subsequent code exploration. A problem here is that the launched processes may vary across tests (e.g., launch different programs or with different sequences), and HOUSEFUZZ must ensure process coverage comparison is consistent—the compared coverage data corresponds to the same program. To address this issue, HOUSEFUZZ integrates coverage bitmaps of all processes that load the same executable ELF object (by summing up hit counters) and uses the output coverage bitmap for comparisons.

6.2. Multi-Process Vulnerability Detection

Different from single-process grey-box fuzzing [9], HOUSEFUZZ detects vulnerabilities in all service processes, including network-facing, daemon, and utility processes. Specifically, HOUSEFUZZ implements vulnerability oracles for memory corruption and command injection vulnerabilities, which are prevalent vulnerability types in firmware [2] and frequently attacked [4], [29]. For memory corruptions, HOUSEFUZZ detects crash signals (e.g., segmentation fault) in all service processes and excludes unexploitable crashes in utility processes by verifying whether the memory corruption impact can be controlled by user input. For command injection, HOUSEFUZZ adapts the web-application-based approach [30], [31] for firmware binaries, which instruments the *execve()* system call of all service processes. We present more technical details in Appendix-§B.

7. Service-Protocol-Guided Fuzzing

7.1. Formalization of Service Protocol

As introduced in §2.2, a firmware service protocol consists of two parts—the standard service protocol (e.g., HTTP, UPnP, JetDirect), and customized service protocol. To improve the quality of generated test cases, fuzzing should handle both of them to ensure the syntax and semantic validity of test cases. To unambiguously express this requirement, we formalize syntax constraints of standard service protocol with an existing method—context-free grammar (CFG), which has been widely adopted [22], [23], [24], [25], [26], and formalize the semantic constraints of customized service protocol with token dependency graph (TDG).

Definition. TDG is a directed graph $G(N, E)$ (showcased in Figure 5). Each graph node $n \in N$ is a two-element tuple $n(v, t)$. The $n.v$ denotes the token value, which is a concrete string, and $n.t$ is the token type, which depicts token semantics. We define three basic token types that denote the semantic meanings of a wide range of tokens—*Path*, *Key*, and *Value*, where *Path* denotes a specific functionality routing, *Key* denotes a left operand of assignment, and *Value* represents an assigned value. For example, the token *"ccp_act"* and *"set"* in Figure 1 correspond to *Key* and *Value* types accordingly. For each graph edge $e(n_i, n_j) \in E$, it denotes a dependency (i.e., semantic constraint) from the source token n_i to the target token n_j; specifically, it means that when the n_j has been used by a test case, it requires to insert n_i to satisfy this constraint. As showcased in Figure 4, there are two kinds of dependency—control-flow and data-flow dependency. For control-flow dependency, n_i depends on n_j because n_j controls whether n_i will be accessed/required (showcased by scenarios 1 and 2). For data-flow dependency n_j decide how n_i is understood by protocol (showcased by scenario 3).

7.2. Service Protocol Inference

For a target service, HOUSEFUZZ identifies which standard protocol is used from known protocols, and infers customized service protocol with online dynamic instruction-based analysis and offline control-/data-flow analysis.

7.2.1. Standard Service Protocol Identification. For a network service, HOUSEFUZZ identifies known protocols with two protocol-specific features: network channels and magic constants. Standard protocols usually communicate through known network channels as a common practice [32] (e.g., HTTP protocol through TCP 80 port), so HOUSEFUZZ first identifies the channel with system tracing (§5) and then maps the channel to a known protocol. In some cases, a standard service communicates through unknown network channels, HOUSEFUZZ identifies them by magic constants. Specifically, HOUSEFUZZ extracts string constants from the identified network-facing binary and matches them with magic constants of known protocol (e.g., *"SUBSCRIBE"* of UPnP) to identify their corresponding standard protocols.

7.2.2. Customized Service Protocol Inference. For customized service protocols, HOUSEFUZZ infers token dependencies by analyzing the target service and uses these dependencies to construct TDG. To infer more token dependencies, we propose two orthogonal components—online and offline TDG inference. Each component infers token values, types, and dependencies with different methods. The online component leverages dynamic instrumentation during fuzzing to collect token values, and analyzes the fuzzing test cases to infer token types and dependencies. The offline component leverages static analysis to extract token values and infer the types and dependencies from firmware code. Both components are complementary to each other: the offline component is lightweight enough to quickly infer an initial TDG for early-stage code exploration, while the online component can gradually improve TDG using fuzzing feedback when the exploration goes deep.

❶ **Online TDG Inference.** Figure 5 showcases how HOUSEFUZZ infers TDG with the online component. First, the online component instruments string comparing functions like *strcmp()* during fuzzing to collect token values used by input parsing, which is already adopted by existing fuzzing to collect input corpus [31], [33]. Specifically, the online component infers tokens that are compared with an input token in the current fuzzing test case.

Then, the online component uses the type of compared input token as the type of the inferred token. The rationale is that service protocol parsing usually compares whether the input token equals to an expected token (e.g., *"set"*), where the expected token will be inferred and has the same type as the input token. To retrieve the type of input token, HOUSEFUZZ first parses the test case based on the standard service protocol to retrieve the field type of the input token, and then maps the field type of the input token to a token type. For example, the token *"set"* is parsed as an HTTP parameter value, which is mapped to the token type *Value*.

Finally, the online component infers the dependency of the inferred token based on its type. For example, the inferred token with a *Value* type likely depends on input tokens with *Key* type. As a test case may contain multiple tokens of the same type, HOUSEFUZZ refines the token types based on the standard service protocol to find its dependency more accurately. For example, an *Value* token that is parsed into an HTTP parameter value only depends on the corresponding parameter key. Such refinement is a one-time effort for each standard service protocol and is lightweight. For HTTP protocol, HOUSEFUZZ only refines the *Key* and *Value* types corresponding to headers and parameters.

❷ **Offline TDG Inference.** The offline component leverages static code analysis to extract tokens due to lacking test cases. The idea is to find tokens in code based on their semantic meanings denoted by token types. We find many existing static analysis tools have implemented the detector for identifying fields of HTTP service protocol [4], [12], [20], so HOUSEFUZZ directly uses these detectors. Specifically, HOUSEFUZZ leverages the approaches proposed by [20], [34] to identify URL paths and parameter keys of

Scenario 1: Direct control-flow dependency	Scenario 2: Indirect control-flow dependency	Scenario 3: Data-flow dependency

1 //B("action") is dependent on A("scandir.sgi")
2 if (!strcmp(uri, "scandir.sgi"))
3 return sub_1CFC0(packet, a2, a3);
4
5 sub_1CFC0(char* s1, int a2, int a3){
6 val = cgi_value(s1, "action");
7 sub_1EF60(val);
8 }
	1 // B ("username") is dependent on A("ddns.cgi")
	2 func_A = get_handler("ddns.cgi");
	3 func_A();
	4 sub_A8D0(char* input, int a2){
	5 name = websgetvar(input, "username")
	6 }
	
	.data: 0x1CDA0 DCD "ddns.cgi" DCD sub_A8D0
	.data: 0x1CDA8 DCD "ipv6.cgi" DCD sub_ACC0
		1 // B("::::::::") is dependent on A("ipv6_pri_dns")
		2 websgetvar(input, "ipv6_pri_dns", v10);
		3 if (strstr(v10, "::::::::")){
		4 if (!NvramConfig_match("ipv6_dns", "")){
		5 NvramConfig_set("ipv6_dns", v10);
		6 call_acos_service(v10);
		7 ...
		8 }
		9 }

Get function handler by name

Figure 4: Examples of static code patterns of token dependencies.

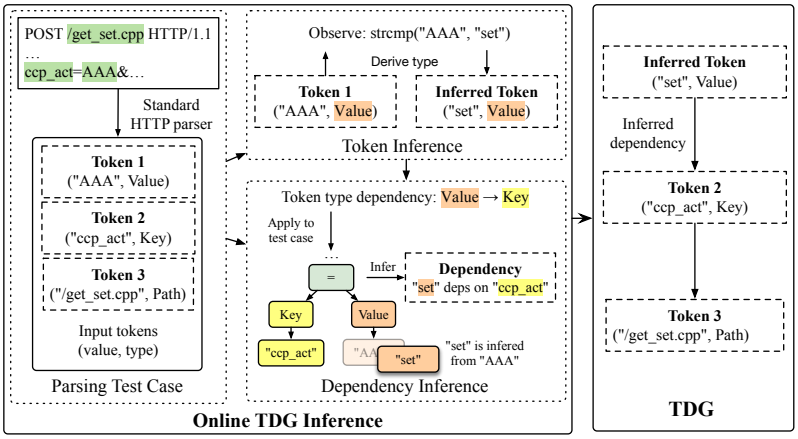

Figure 5: An example of online TDG inference

HTTP protocols, which extracts tokens with *Path* ans *Key* types. The *Value* tokens can be extracted when inferring token dependencies.

Then the offline component leverages three kinds of code patterns showcased by Figure 4 to infer token dependencies based on control-flow and data-flow analysis. Specifically, HOUSEFUZZ identifies direct control-flow-based token dependencies showcased in scenario 1 by analyzing the control-flow graph of the service binary; for indirect control-flow dependency, HOUSEFUZZ leverages the approach proposed by LARA [20] to identify handler functions of URL paths, and then analyze the control-flow graph of the handler function. For data-flow-based token dependencies, HOUSEFUZZ leverages the feature-based input approach proposed by FITS [34] to identify input reading location and accompanied *Key* tokens, then HOUSEFUZZ utilizes path-insensitive reaching definition analysis to track the read data, and detect *Value* tokens compared with the read data.

7.3. Service-Protocol-Guided Test Case Generation

HOUSEFUZZ combines both standard and customized service protocols to guide test case generation. HOUSEFUZZ formalizes standard service protocol with context-free grammar (CFG). CFG can represent most popular firmware service protocols such as HTTP and UPnP [15] and has been widely adopted for test case generation [22], [23], [24], [25], [26]. As shown in Figure 6, CFG consists of sets of production rules, where each indicates "a grammar symbol can be replaced by a list of grammar symbols and constant strings". By continuously applying production rules, a test case generator can finally create a derivation tree whose leaf nodes composite a syntactic valid test case.

For handling customized service protocol, HOUSEFUZZ mutates test cases while ensuring the token dependencies described by the inferred TDG. Given a test case generated with standard service protocol (e.g., CFG), HOUSEFUZZ carefully inserts the inferred tokens into the test case, and attempts to satisfy more semantic constraints of the inserted tokens. Specifically, HOUSEFUZZ first picks a token $n(v,t)$ from the TDG and inserts the token into the given test case. To insert the token at the proper position, HOUSEFUZZ picks a token in the test case that has the same token type $n.t$, and replaces it with n. For example, in Figure 5, the inferred token (*"set"*, *Value*) can be used to replace (*"AAA"*, *Value*). When the refined token type (§7.2.2) is available, it is also used to refine the insertion positions. After the token is inserted, HOUSEFUZZ attempts to insert more relevant tokens into the test case to satisfy more customized constraints. HOUSEFUZZ traverses the TDG backward starting from n to find all tokens that directly or indirectly depend on n. These tokens will also composite the mutated test to bypass semantic validation. Specifically, HOUSEFUZZ iterates found tokens by TDG traverse order and inserts each token to the test case, similar to inserting n.

Other than service-protocol-guided test case generation, HOUSEFUZZ also adopts random mutation [21] for generating malformed test cases for triggering protocol parsing vulnerabilities. HOUSEFUZZ schedules service-protocol-guided and random-mutation-based test case generation based on

code coverage discovery—when one approach fails to discover new code coverage within a time limit (i.e., 3 minutes in our experiment), HOUSEFUZZ switches to another approach. When the standard service protocol is unavailable, HOUSEFUZZ only uses random mutation for fuzzing. In this case, HOUSEFUZZ uses TDG for handling customized service protocol, but blindly inserts tokens into a test case due to a lack of the guidance of standard service protocol.

8. Evaluation

We conducted extensive experiments to demonstrate the effectiveness of HOUSEFUZZ. First, in §8.2 (**RQ1**), we compared the general fuzzing performance of HOUSEFUZZ with SoTA grey-box firmware fuzzing tools. Afer that, in §8.3 (**RQ2**), §8.4 (**RQ3**) and §8.5 (**RQ4**), we respectively demonstrate the advantages of the three key techniques of HOUSEFUZZ, i.e., holistic service identification, multi-process fuzzing framework, and service-protocol-guided fuzzing.

8.1. General Experiment Setup

Prototype. We implemented a prototype of HOUSEFUZZ with about 7k lines of C code and 5k lines of Python code. HOUSEFUZZ uses Binwalk [35] to extract file systems from Linux-based firmware. For system initialization, HOUSEFUZZ runs each firmware image with QEMU user mode, which also tracks the execution trace of system calls and basic blocks. The trace is further analyzed with Radare2 [36] to find and patch exception code to improve the robustness of system initialization emulation. When a network service is identified by analyzing the system initialization procedure, the corresponding service binaries are fed to an IDA Pro-based static analysis engine to extract offline TDG, which will be used in service-protocol-guided fuzzing. In the fuzzing stage, the prototype uses the technique proposed by GREENHOUSE [9] to emulate target services and modifies the QEMU mode of AFL++ [21] to add supports for multi-process feedback mechanism, vulnerability oracles, and online TDG inference. We will release our source code at https://github.com/seclab-fudan/HouseFuzz.

Firmware Image Dataset. To evaluate the effectiveness of grey-box firmware fuzzing techniques, we reused an existing real-world firmware image dataset that has been extensively used as a benchmark by recent SoTA works [8], [9]. Originally, this dataset consisted of 70 firmware images from 3 vendors. We updated 35 images in the dataset with the latest image versions of the same products, either because the original dataset did not annotate image versions, or the obsolete image versions are not available for downloading. Then, we used Binwalk [35] to extract these images. The extraction failed on 10 firmware images potentially due to the updated firmware images using new image packing methods. Finally, we used the 60 extractable firmware images (listed in Table 2) for most of experiments as described in §8.2.1, §8.3, §8.4, and §8.5.

TABLE 2: Firmware service dataset for comparison experiments.

Vendor	# Images	# Extracted Images	# Target Services [1]
D-Link	30	22	17
Netgear	29	29	20
Trendnet	11	9	4

[1] Services could be emulated and fuzzed by both GREENHOUSE and HOUSEFUZZ.

Environment. All experiments were conducted on a Ubuntu 22.04 machine with an Intel Xeon(R) Gold 5218 CPU (2.30 GHz $\times$ 64 cores) and 245 GB memory.

Vulnerability Disclosure. We have responsibly reported all 156 0-day vulnerabilities found during experiments to vendors and then the vulnerability database, including 58 buffer overflow, 15 OS command injection, and 83 denial-of-service vulnerabilities. Currently, we have received 45 CVE/CNVDs as shown in Table 9.

8.2. General Fuzzing Performance (RQ1)

Experiment Overview. In this section, we conducted two experiments, i.e., comparison experiment and vulnerability hunting experiment, to demonstrate the superior code exploration and vulnerability detection capabilities of HOUSEFUZZ.

8.2.1. Comparison Experiment. This experiment compares the general fuzzing performance of HOUSEFUZZ with the SoTA grey-box firmware fuzzing technique under the same fuzzing settings on the dataset shown in Table 2.

Baseline Choice. Here, we mainly considered SoTA grey-box firmware fuzzing approaches that are based on process emulation, which greatly outperforms those built upon system emulation in the aspect of fuzzing throughput [7]. We identified three baselines that met our criteria: GREENHOUSE [9], EQUAFL [8], and Firm-AFL [7]. Given that these approaches have already been thoroughly evaluated for their fuzzing effectiveness in prior works (i.e., existing evaluations [8], [9] have shown that GREENHOUSE is 2x faster than EQUAFL, and EQUAFL is 14x faster than Firm-AFL), we finally chose the technique that achieves the best performance as our baseline—GREENHOUSE.

Fuzzing Targets (Firmware Service Dataset). To fairly evaluate the effectiveness of firmware service fuzzing between GREENHOUSE and HOUSEFUZZ, we need to build a firmware service dataset that can be fuzzed by both GREENHOUSE and HOUSEFUZZ. Hence, given the 60 extracted firmware images in the existing firmware image dataset, we first run GREENHOUSE to collect all emulatable and fuzzable services among them, which contain a total of 41 web services. Since we confirmed that HOUSEFUZZ could also successfully fuzz these services, we used them as the firmware service dataset (listed in Table 2).

Fuzzing Settings. We then ran GREENHOUSE and HOUSEFUZZ to fuzz the firmware service dataset under the same emulation environments (i.e., running the same

TABLE 3: General fuzzing performance comparison (**RQ1**).

Vendor	Series	# Services	Avg. Edge Code Coverage[1]		Number of Detected Vulnerabilities	
			GREENHOUSE	HOUSEFUZZ	GREENHOUSE	HOUSEFUZZ
D-Link	DAP	12	20704	28924	1	14
	DIR	3	4480	5678	0	0
	DSP	1	2449	2969	4	5
	GO	1	1442	1841	0	0
Netgear[2]	WN/WNCE	2	1522	2625	6	19
	WN*AP/WAC	7	15715	16408	0	0
	WN(D/DR/R)	8	6886	11209	20	57
	WPN	1	766	994	2	2
	X(AV/W)N	2	3620	4745	5	22
Trendnet	TEW	4	3012	4068	8	9
Summary		41	11072	14767	46	128

[1] Summing image edge numbers of the same series and dividing with the round number. [2] Netgear product series are merged based on device type.

processes on the same file system). For each tool, we ran the fuzzing on each service for 3 rounds, and each round used 1 CPU core for 24 hours. Both tools used the same initial seeds from GREENHOUSE. For test case generation, GREENHOUSE used fuzzing dictionaries extracted from target services, while HOUSEFUZZ only used the token dependency graph it automatically extracted. Note that, when evaluating HOUSEFUZZ, we counted the time consumed by TDG inference (including both offline and online components) into the fuzzing time.

Comparison Metrics. We mainly compared GREENHOUSE and HOUSEFUZZ in the aspect of vulnerability detection and code coverage. To evaluate vulnerability detection, we first grouped all crashes detected by crash sites and further leveraged QEMU-based sanitizer and manual root cause analysis to pick up unique vulnerabilities. In particular, command injections detected by HOUSEFUZZ were excluded from the comparison for fairness because GREENHOUSE cannot identify this kind of vulnerability. For evaluating the code coverage, we only considered the edge coverage of the network-facing program because GREENHOUSE does not have the capability of collecting multi-process coverage like HOUSEFUZZ. Following the representative fuzzing evaluation practices [37], we calculated the p-value and $\hat{A}_{12}$ score for measuring the significance and effect size, and regarded $p < 0.05$ as an indicator of significant difference between two tools, while $\hat{A}_{12} \geq 0.71$ as an indicator that HOUSEFUZZ likely outperforms GREENHOUSE. Besides, we also analyzed the execution overhead introduced by HOUSEFUZZ based on the total test case execution times.

Results on Code Coverage. Table 3 shows the code coverage of both tools on services of different product series. Overall, HOUSEFUZZ outperformed GREENHOUSE on edge code coverage by 33.4%. Among 41 tested services, HOUSEFUZZ achieved $p < 0.01$ and $\hat{A}_{12} \geq 0.71$ on 36 services, which showed that HOUSEFUZZ significantly outperformed GREENHOUSE in code coverage. Meanwhile, HOUSEFUZZ also achieved higher average code coverage than GREENHOUSE in the remaining 5 services. These results show that HOUSEFUZZ was more effective in code exploration than GREENHOUSE.

Results on Vulnerability Detection. In total, we analyzed 705 and 2,519 crashes detected by GREENHOUSE and HOUSEFUZZ, and summarized the results (i.e., unique vulnerabilities) in Table 3. In summary, GREENHOUSE only detected 46 vulnerabilities on 18 services, including 40 0-days, while HOUSEFUZZ detected 128 vulnerabilities on 25 services, including 110 0-days. This means HOUSEFUZZ detected 175% more 0-days than GREENHOUSE. Statistically, HOUSEFUZZ achieved significantly better results (i.e., $p < 0.01$ and $\hat{A}_{12} \geq 0.71$) than GREENHOUSE on 5 services, and found more vulnerabilities than GREENHOUSE on 18 services. Here, HOUSEFUZZ identified slightly fewer vulnerabilities (i.e., one less vulnerability) than GREENHOUSE on 2 services, mainly due to the high overhead of multi-process code coverage collection.

Notably, we tried our best to confirm that HOUSEFUZZ and GREENHOUSE correspondingly detected 110 and 40 previously unknown vulnerabilities (i.e., unrecorded by NVD [38] database), with 23 and 10 CVE/CNVD assigned, where 9 were detected by both tools.

Results on Overhead. Due to the multi-process feedback collection and the TDG inference, HOUSEFUZZ averagely introduced 6.8x more time overhead than GREENHOUSE on the 41 services. For memory overhead, HOUSEFUZZ allocated a (128 KB) coverage bitmap for each active process, and the memory overhead is linear to the number of active processes. Here, we argue that the overhead is quite acceptable, considering the fact that HOUSEFUZZ significantly outperformed GREENHOUSE in aspects of vulnerability detection and code coverage.

8.2.2. Vulnerability Hunting Experiment. Given that the firmware image dataset used by the prior comparison experiment only involves legacy firmware released before 2019, this experiment aims to demonstrate that HOUSEFUZZ can also detect vulnerabilities in latest firmware images of actively maintained products. Therefore, we conducted this experiment on a new firmware image dataset, where the firmware images are released after 2020.

Fuzzing Targets (Firmware Service Dataset). We constructed a new firmware service dataset following the below steps. First, we collected the latest firmware images of 45 actively maintained products, where these products are sampled from recent security advisory web pages of 9 vendors and have firmware images released after 2020. Then, we extracted these images and found 12 ones can be successfully

extracted. Finally, we follow the same workflow described in §8.2.1 to collect emulatable and fuzzable services from the new firmware image dataset as fuzzing targets. As a result, we identified 12 web services from the 12 extracted firmware images as shown in Table 4.

TABLE 4: Firmware service dataset for vulnerability hunting.

Vendor	# Images	# Extracted Images	# Target Services [1]
ASUS	2	2	2
D-Link	10	1	1
Draytek	2	1	1
Netgear	11	3	3
Ruijie	2	0	0
TOTOLINK	5	3	3
TP-Link	4	0	0
Trendnet	5	2	2
Xiaomi	4	0	0
Total	45	12	12

[1] Services could be emulated and fuzzed by HOUSEFUZZ.

Results on Vulnerability Hunting. We used HOUSEFUZZ to fuzz each identified web service for 24 hours, and successfully detected 21 0-day vulnerabilities in 5 services, with 21 assigned CVE/CNVDs. This result demonstrates that HOUSEFUZZ is also effective in vulnerability detection when applied to the new firmware image dataset.

8.3. Experiments on Service Identification (RQ2)

Experiment Overview. In the previous section, we mainly compared HOUSEFUZZ and GREENHOUSE on the effectiveness of firmware fuzzing, considering only firmware services that can be emulated by both tools. Here, we also conducted experiments to demonstrate the effectiveness of our service identification approach, which actually enables HOUSEFUZZ to identify those overlooked services with implied vulnerabilities.

Baseline Choices. We compared the service identification technique of HOUSEFUZZ with two SoTAs approaches—GREENHOUSE [9] and FirmAE [11]. GREENHOUSE uses name whitelists to identify network-facing programs and daemon programs; then it launches the identified programs with pre-defined arguments or arguments extracted from FirmAE emulation results to obtain network-facing and daemon processes. FirmAE emulates the entire firmware system without service identification. To compare with FirmAE, we ran the netstat [39] tool to identify network-facing processes during emulation, which is similar to running HOUSEFUZZ without patching emulation exception code.

Results on Network-Facing Process Identification and Network Service Identification. For network-facing process identification, we ran HOUSEFUZZ, GREENHOUSE, and FirmAE on the same firmware image dataset introduced in §8.1. A network-facing process was considered identified when it opened a non-local network channel (e.g., TCP/UDP ports on public IP addresses). Finally, results showed that HOUSEFUZZ, FirmAE, and GREENHOUSE successfully identified 311, 128, and 44 network-facing processes, respectively from the 60 extracted firmware images. We manually confirmed that these processes were actual network-facing, so the identification precision is 100%. This indicates HOUSEFUZZ identified at least 143% more network-facing processes than existing work. Utilizing the total 387 unique network-facing processes as a benchmark, HOUSEFUZZ achieved a recall rate of 80.4%, significantly higher than the 33.1% and 18.1% of FirmAE and GREENHOUSE, respectively. Table 5 illustrates the distribution of identified network services by three tools. GREENHOUSE identified the least network services due to the limitation of the name whitelist, limiting the discovery of unlisted network-facing processes. Besides, we found GREENHOUSE attempted to emulate 3 web servers with wrong arguments, causing the emulated process to abort before establishing network channels. By comparing HOUSEFUZZ and FirmAE, we found FirmAE only successfully identified network services in 25 firmware images, while HOUSEFUZZ succeeded in 59 ones. This is because HOUSEFUZZ automatically identified and handled exceptions to analyze more system initialization procedures.

To understand the effectiveness of exception identification and handling described in §5.1.1, we analyzed each identified emulation exception. We observed that HOUSEFUZZ identified *85* emulation exceptions of 30 firmware images, where 8 were automatically identified as false positives using reverting mechanism. As a result, this mechanism avoids overlooking 77 services in 8 firmware images. After handling identified exceptions, HOUSEFUZZ successfully identified 119 (37%) more network services on 23 firmware images. These results demonstrate that HOUSEFUZZ's exception identification and handling techniques are effective for network service identification.

TABLE 5: Network service identification comparison (**RQ2**).

Protocol	# Service	GREENHOUSE * Recall	FirmAE* Recall	HOUSEFUZZ* Recall
HTTP	98	44.9%	43.9%	95.9%
Telnet	40	0	30.0%	100.0%
UPNP	28	0	39.3%	100.0%
NetBIOS	24	0	33.3%	91.7%
DHCP	23	0	34.8%	91.3%
(DNS)	22	0	18.2%	90.9%
NCI	18	0	11.1%	100.0%
mDNS	13	0	61.5%	76.9%
LLMNR	11	0	63.6%	63.6%
SSH	7	0	28.6%	100.0%
RIP	4	0	0	100.0%
AFP	4	0	0	100.0%
TFTP	3	0	33.3%	100.0%
STP	1	0	0	100.0%
Unknown	142	0	64.1%	47.2%
Summary	438	10.0%	45.0%	79.0%

* The recall is calculated considering all unique network services identified by three tools as ground truth.

Results on Daemon Process Identification. We confirmed that GREENHOUSE and HOUSEFUZZ achieved a precision of 100% in daemon process identification with manual analysis. However, it was hard to construct a ground truth of daemon processes for evaluating recall because analyzing IPC dependencies requires significant reverse engineering efforts. Alternatively, we present an example to demonstrate the drawback of the whitelist-based approach. We

found GREENHOUSE failed to identify an essential daemon process named "zebra". According to the document [40], running zebra is mandatory to run ripd services, so zebra is vital to identify for testing ripd services. HOUSEFUZZ successfully found all these daemon processes without using name whitelists.

Identified Vulnerabilities in Overlooked Services. Without identifying the network-facing processes, grey-box fuzzing can not test the corresponding network services properly and will fail to discover vulnerabilities in these services. As mentioned above, HOUSEFUZZ successfully identified more overlooked network services than the baseline. Here, we further selected 3 web services not identified by both GREENHOUSE and FirmAE, fuzzed each one with HOUSEFUZZ for 72 hours, and successfully detected 12 0-day vulnerabilities.

8.4. Experiments on Multi-Process Fuzzing Framework (RQ3)

Experiment Overview. We conducted three experiments in this section to demonstrate the effectiveness of HOUSEFUZZ's multi-process fuzzing framework. First, we conducted an ablation experiment to evaluate the overall effectiveness of the multi-process fuzzing framework. Second, we performed a manual vulnerability analysis to answer whether the multi-process fuzzing framework indeed contributed to multi-process vulnerability detection. Third, we evaluated whether the TCE detection used by multi-process fuzzing framework was robust enough for consistent code coverage collection.

8.4.1. Ablation Experiment. We compared two ablation setups—GREENHOUSE and HOUSEMP to evaluate the overall effectiveness of multi-process fuzzing framework. Specifically, GREENHOUSE serves as the foundational baseline, which only performed single-process fuzzing, and HOUSEMP extends GREENHOUSE by integrating the multi-process fuzzing framework. We then compared different baselines using the same dataset and settings in §8.2.1. Table 6 and Table 7 show the results on code coverage and vulnerability detection.

According to Table 7, HOUSEMP found 7 fewer vulnerabilities than GREENHOUSE in total, which is mainly due to the overhead of multi-process feedback collection. Nevertheless, HOUSEMP achieved significantly better code coverage discovery than GREENHOUSE in 16 target services and detected 14 vulnerabilities not detected by GREENHOUSE. We also observed that 53.7% services under test exhibit IPCs among network-facing, daemon, and utility processes and 100% services include at least two types of these processes. These observations also demonstrate the necessity of considering non-network-facing processes during grey-box fuzzing. In summary, even though the multi-process fuzzing framework brings extra overhead to grey-box fuzzing, it significantly contributes to extra code coverage and vulnerability detection.

8.4.2. Multi-process Vulnerability Analysis. To understand whether the multi-process fuzzing framework indeed contributed to multi-process vulnerability detection. We manually analyzed vulnerabilities detected by HOUSEFUZZ in RQ1 (§8.2) to identify multi-process vulnerabilities. To detect such vulnerabilities, fuzzing must trigger both the IPC in one process and the vulnerable code in another process, which requires feedback from both processes. Therefore, single-process fuzzing can hardly detect such vulnerabilities due to the limited feedback collection scope.

We identified that 3 out of 128 and 15 out of 21 vulnerabilities were multi-process vulnerabilities in the comparison experiment (§8.2.1) and vulnerability hunting experiment (§8.2.2) accordingly. As shown in Table 9, these vulnerabilities have been assigned with 17 CVE/CNVDs, occupying 38% of all 45 assigned CVE/CNVDs. This result demonstrates that multi-process fuzzing framework is effective in detecting real multi-process vulnerabilities.

8.4.3. Experiment on TCE Detection Stability. The multi-process fuzzing framework leverage TCE detection to robustly determine when to collect code coverage, which ensures a stable code coverage collection. Therefore, this experiment aims to demonstrate the stability (robustness) of TCE detection from the perspective of stability of coverage collection. Specifically, we leveraged queued seed generated by HOUSEFUZZ from one fuzzing campain in §8.2.1 as test case dataset, and repeatedly executed each seed for N (N=20) times. If all executions of the same test case yield a consistent multi-process code coverage, it demonstrates that the TCE detection is stable for this seed.

One problem is that unstable code coverage could be caused by not only the instability of the TCE detection but also nondeterministic code logic (NCL). The NCL can introduce nondeterministic behavior due to time variance, random numbers, uncertainty of event occurrences, etc. To mitigate such impacts, we adopt three countermeasures. First, all seed executions ran under a consistent running environment with a constant random number device (e.g., */dev/random*). Second, we excluded test cases that are already unstable in single-process fuzzing mode. Third, we manually examined traces of unstable executions to inspect the root causes.

In summary, we evaluated 13,453 seeds, where the execution of 8,746 (65%) seeds result in consistent coverage. These seeds were considered stable in the *single-process fuzzing mode* and thus were used to evaluate the stability of TCE detection in *multi-process fuzzing mode*. Among these 8,746 seeds, 358 seeds yielded inconsistent code coverage across execution in multi-process fuzzing mode. We found this was caused by NCL of *select()* system calls when handling multiple IPC I/O events. For example, when handling 2 packets, *select()* system call will return one or two times to handle two packets together or separately, resulting in different code coverage. After excluding seeds affected by such an NCL, we found the TCE detection technique achieved 100% consistent code coverage on the remaining 8,388 seeds. This

TABLE 6: Ablation study results on edge code coverage (**RQ3 & RQ4**).

Vendor	Series	GREENHOUSE	HOUSEMP			HOUSECFG			HOUSETDGOL			HOUSEFUZZ		
		Avg.[1]	Avg.[1]	p-val	$\hat{A}_{12}$	Avg.[1]	p-val	$\hat{A}_{12}$	Avg.[1]	p-val	$\hat{A}_{12}$	Avg.[1]	p-val	$\hat{A}_{12}$
D-Link	DAP	20704	22315	**<0.01**	**1.00**	26439	**<0.01**	**1.00**	26966	**0.02**	**1.00**	28924	**<0.01**	**1.00**
	DIR	4480	4669	**0.02**	**1.00**	5041	**<0.01**	**1.00**	5220	0.28	0.67	5678	**<0.01**	**1.00**
	DSP	2449	3114	**<0.01**	**1.00**	2285	0.03	0	2969	0.09	**1.00**	3207	**0.01**	**1.00**
	GO	1442	1523	0.08	**0.89**	1775	**<0.01**	**1.00**	1614	0.46	0.67	1841	**<0.01**	**1.00**
Netgear[2]	WN/WNCE	1522	1255	0.08	0	1191	0.03	0	2303	**0.02**	**1.00**	2625	**<0.01**	**1.00**
	WN*AP/WAC	15715	15701	0.96	0.44	16946	**0.03**	**1.00**	12216	0.12	0	16408	0.05	**1.00**
	WN(D/DR/R)	6886	6446	0.05	0	7589	**0.02**	**1.00**	9250	**<0.01**	**1.00**	11209	**<0.01**	**1.00**
	WPN	766	775	0.70	**0.78**	786	0.41	**0.89**	951	**<0.01**	**1.00**	994	**<0.01**	**1.00**
	X(AV/W)N	3620	2226	0.01	0	2100	0.02	0	4018	0.19	**0.78**	4745	**<0.01**	**1.00**
Trendnet	TEW	3012	2788	0.17	0	2545	0.03	0	3921	**<0.01**	**1.00**	4068	**<0.01**	**1.00**

[1] Summing image edge numbers of the same series and dividing with the round number. [2] Netgear product series are merged based on device type.

TABLE 7: Ablation study results on the number of detected vulnerabilities. (**RQ3 & RQ4**).

Vendor	Series	GREENHOUSE		HOUSEMP				HOUSECFG				HOUSETDGOL				HOUSEFUZZ			
		Avg.	Tot.	Avg.	Tot.	p-val	$\hat{A}_{12}$	Avg.	Tot.	p-val	$\hat{A}_{12}$	Avg.	Tot.	p-val	$\hat{A}_{12}$	Avg.	Tot.	p-val	$\hat{A}_{12}$
D-Link	DAP	0.7	1	0.3	1	0.52	0.33	0.3	1	0.52	0.33	1.0	3	0.37	0.67	8.0	14	**<0.01**	**1.00**
	DIR	0	0	0.7	1	0.12	**0.83**	0	0	n.a.	0.50	0	0	n.a.	0.50	0	0	n.a.	0.50
	DSP	2.3	4	3.0	3	0.37	0.67	1.3	2	0.35	0.22	2.7	4	0.68	0.56	2.7	5	0.78	0.61
	GO	0	0	0.7	1	0.12	**0.83**	0	0	n.a.	0.50	0	0	n.a.	0.50	0	0	n.a.	0.50
Netgear	WN/WNCE[2]	4.0	6	2.3	5	0.19	0.17	2.7	4	0.12	0.11	5.0	8	0.16	**0.83**	10.0	19	**<0.01**	**1.00**
	WN*AP/WAC[2]	0	0	0	0	n.a.	0.50	0	0	n.a.	0.50	0	0	n.a.	0.50	0	0	n.a.	0.50
	WN(D/DR/R)[2]	13.3	20	8.0	17	0.03	0	9.0	20	0.29	0.22	21.3	28	**<0.01**	**1.00**	30.0	57	**0.03**	**1.00**
	WPN[2]	1.7	2	1.0	2	0.12	0.17	1.0	2	0.12	0.17	2.0	3	0.37	0.67	1.0	2	0.37	0.28
	X(AV/W)N[2]	3.3	5	1.3	3	0.10	0.05	2.7	4	0.52	0.39	9.0	13	0.07	**0.94**	13.6	22	**<0.01**	**1.00**
Trendnet	TEW	6.3	8	2.7	6	0.07	0	1.3	3	**<0.01**	0	4.7	10	0.15	0.11	6.3	9	1.00	0.44
Summary		31.6	46	20.0	39	**0.03**	0	18.3	36	**0.01**	0	45.7	69	**<0.01**	**1.00**	71.6	128	**0.01**	**1.00**

TABLE 8: Ablation experiment setups (**RQ3 & RQ4**).

	Multi-process Fuzzing Framework	Standard Protocol Guidance	TDG Guidance	
			Online	Offline
GREENHOUSE	×	×	×	×
HOUSEMP	✓	×	×	×
HOUSECFG	✓	✓	×	×
HOUSETDGOL	✓	✓	✓	×
HOUSEFUZZ	✓	✓	✓	✓

result demonstrates TCE detection approach is stable in multi-process fuzzing.

8.5. Experiments on Service-Protocol-Guided Fuzzing (RQ4)

Experiment Overview. We conducted two experiments in this section to demonstrate the effectiveness of HOUSEFUZZ's service-protocol-guided fuzzing: an ablation experiment to evaluate the overall effectiveness of service-protocol-guided fuzzing, and a manual analysis-based study to evaluate the effectiveness of TDG inference.

8.5.1. Ablation Experiment. We compared HOUSECFG, HOUSETDGOL, and HOUSEFUZZ to demonstrate the effective ness of service-protocol-guided fuzzing. These three baselines, as shown in Table 8, are developed based on HOUSEMP introduced in §8.4.1. Specifically, HOUSECFG introduces standard service-protocol-guided test case generation based on Control Flow Graph (CFG), utilizing only the extracted tokens for test case generation without incorporating customized service protocols. HOUSETDGOL is a variant of HOUSEFUZZ that excludes the offline Token Dependency Graph (TDG) inference component. We compared baselines using the same dataset and settings in §8.2.1. Table 6 and Table 7 show the results on code coverage and vulnerability detection.

The results show that HOUSECFG only found 36 vulnerabilities. By considering customized service protocol, HOUSETDGOL detected 49% more and 94% more vulnerabilities than GREENHOUSE and HOUSECFG. By further applying offline token dependency inference, HOUSEFUZZ found 55 more vulnerabilities than HOUSETDGOL. These improvements are brought by two reasons. First, by handling customized service protocols formalized with TDG, HOUSEFUZZ effectively generates high-quality test cases. Second, the offline TDG inference component complements the online one by providing an initial TDG; the offline component also identified more tokens and their dependencies because the online component only instrumented standard string comparison functions (e.g., *strcmp()*) to infer tokens, and thus overlooked customized string comparison functions. In summary, service-protocol-guided fuzzing is effective for vulnerability detection in Linux firmware services.

8.5.2. Study on TDG Inference Effectiveness. This experiment aims to demonstrate the capability of TDG inference by analyzing TDGs generated by HOUSEFUZZ in the prior experiment. We first calculated the size of the generated TDG and found that, on average, each TDG consisted of 734 nodes (i.e., tokens) and 2,610 edges (i.e., dependencies). To demonstrate the prevalence of customized service protocols, we empirically selected and analyzed token dependencies from each generated TDG, along with their corresponding code reference sites. Our findings indicate that all TDGs

feature unique yet meaningful token dependencies that describe actual input structures. This suggests that customized service protocols are indeed common in Linux firmware services and modeling them using TDGs will contribute to high-quality test case generation.

Next, we aimed to quantitatively access the quality, i.e., precision and recall, of inferred TDGs. However, such an evaluation is challenging due to the lack of ground truth, which makes it impossible to evaluate the recall. Therefore, we conducted a manual analysis-based study on a randomly sampled firmware image to evaluate the precision of inferred TDGs. The study aims to answer the following two questions: (1) how precise is the token inference, and (2) how precise is the token dependency inference?

To answer the first question, we randomly sampled 100 tokens from the sampled TDG and manually verified whether they were valid true positives; that is, they would actually build valid path constraints. We also sampled and analyzed 100 tokens from the fuzzing dictionary generated by GREENHOUSE as a baseline. The result shows that HOUSEFUZZ's token inference technique achieved a precision of 62% and greatly outperforms the precision of GREENHOUSE, which is only 15% due to the coarse-grained string scanning. The improvement benefited from fine-grained static analysis and dynamic analysis.

To answer the second question, we randomly sampled 100 inferred TDG edges and inspected whether they described actual data dependencies. To separately evaluate the dependency inference precision, we excluded 91 edges that involve incorrect tokens. The result shows that the token dependency inference achieved a precision of 44%. As fuzzing can quickly retry different combinations of tokens and dependencies, it will automatically filter correct tokens and dependencies. Therefore, such a TDG inference precision is quite acceptable. During manual analysis, we found several limitations of the current TDG inference prototype, which is summarized in §10.

9. Related Work

Grey-Box Fuzzing of Linux-based Firmware. Emulation is typically the prerequisite of grey-box firmware fuzzing, which bypasses the challenge to instrument real devices [41]. Existing grey-box firmware fuzzing works focus on improving emulation techniques. Firmadyne [10] and FirmAE [11] emulate firmware in a full-system manner by emulating user-space programs together with customized OS kernels and subject the emulated system to dynamic analysis. Recent grey-box firmware fuzzing approaches have shifted toward "process emulation" techniques [7], [8], [9], which reduce or avoid the overhead for emulating a guest OS kernel to improve fuzzing throughput. Specifically, FirmAFL [7], hybrid system emulation and user-mode emulation to reduce system emulation overhead while preserving fidelity; EQUAFL [8] and Greenhouse [9] solely rely on user-mode emulation to emulate the target processes. This paper focuses on addressing obstacles stemming from the multi-process nature and protocol customization of Linux-based firmware services.

Protocol-Guided Fuzzing. Protocol-guided fuzzing leverages known protocol knowledge to craft high-quality test cases for uncovering deep bugs. A notable example of this approach is Peach fuzzer [42], which relies on manually curated specification files for generating test cases based on protocol structures. To encompass a broader array of protocols, including complex formats like XML and JavaScript, grammar-aware fuzzing techniques [22], [23], [24], [26], [43] have been introduced. These approaches employ grammars (e.g., context-free grammars) to accurately model the syntax of various protocols. Building upon syntax modeling, recent advancements have focused on incorporating protocol semantics into test case generation. Program language-oriented approaches model language features to improve fuzzing of interpreters and compilers [24], [44], [45], [46]. Skyfire trains probabilistic models from large-scale public datasets to understand protocols' general usage patterns [25]. These approaches focus on standard protocols but overlook application-specific protocol constraints, which are particularly prevalent within domains like Linux-based firmware services. As another line of research, protocol reverse engineering [47], [48], [49], [50] and structure-aware fuzzing [33], [51], [52], [53], [54] focus on handling unknown protocols, which lack the guidance of knowledge of standard protocols.

10. Discussion

Limitations of TCE Detection. The HOUSEFUZZ's implementation of TCE detection assumes that the target service uses specific system calls (i.e., *select()* and *poll()* to wait for IPC requests, which is typical in multi-process programming—followed by all services in our experiments. However, a Linux firmware service could use other system calls such as inotify [55] for IPC purposes. Since such system calls are limited and have finite programming paradigms [12], HOUSEFUZZ can easily support them with engineering efforts.

Limitations of TDG Inference. HOUSEFUZZ leverages existing program analysis techniques [20], [33], [34] for TDG inference, which suffer two major limitations that lead to false positives/negatives (FPs/FNs). First, the offline TDG inference is limited by the unsoundness and incompleteness of static analysis, leading to FPs/FNs. Second, the online TDG inference only analyzes string comparisons instead of their data dependencies, so it relies on predefined token types to deduce dependencies, which is unsound and leads to FPs; besides, it fails to identify tokens compared using unknown (thus not instrumented) APIs, leading to FNs.

Limitations of Process Emulation. Like existing fuzzing techniques that rely on process emulation [7], [8], [9], HOUSEFUZZ encounters emulation limitations that undermine its effectiveness. In particular, HOUSEFUZZ depends on GREENHOUSE for process emulation, but GREENHOUSE has intrinsic limitations. For example, it failed to test 19

extracted firmware images. Further, the mitigation of system initialization exceptions employed by HOUSEFUZZ is still imperfect, as evidenced by Table 5. These results underscore the urgent need for continuous advancements to enhance the fidelity and reliability of process emulation techniques.

Authentication Handling. Firmware service may require proper authentication to reach privileged code logic, which will prevent HOUSEFUZZ from achieving higher code coverage or detecting post-authentication vulnerabilities. This issue could be addressed by maintaining an authenticated fuzzing session. We leave it for future work.

Unknown Service Protocols. HOUSEFUZZ relies on standard service protocols such as context-free grammar to improve test case generation effectiveness. According to Table 5, 32.4% services use protocols that are not well-known, so their standard service protocols can be hard to retrieve, which hinders the effectiveness of service-protocol-guided test case generation. To generate high-quality test cases for such protocols, HOUSEFUZZ could adopt protocol reverse engineering [47], [48], [49], [50] or structure-aware fuzzing [33], [51], [52], [53], [54] approaches. This direction presents an avenue for future work.

11. Conclusion

This paper introduces HOUSEFUZZ, which incorporates three novel techniques to improve grey-box fuzzing for vulnerability detection in Linux firmware services: (1) holistic service identification exposes more service processes essential to emulate for detecting service vulnerabilities; (2) multi-process grey-box fuzzing facilitates the detection of multi-process vulnerabilities; (3) service-protocol-guided fuzzing models data constraints introduced by firmware customization as token dependencies, and greatly improves the effectiveness of grammar-aware fuzzing. Our evaluations demonstrate the three techniques help detect vulnerabilities in Linux firmware services, and HOUSEFUZZ superiorly outperforms the SoTA approach by identifying 76% more network services, achieving detecting 175% more 0-day vulnerabilities with 33.4% more code coverage on the same dataset.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U2436207, 62172105, 62402116). Yuan Zhang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] E. Foundation, "2023 IoT & Edge Developer Survey Report," https://outreach.eclipse.foundation/iot-edge-developer-survey-2023, 2023.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the Mirai Botnet," in USENIX Security'17, 2017.

[3] H. J. Griffioen and C. Doerr, "Examining Mirai's Battle over the Internet of Things," in CCS'20, 2020.

[4] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems," in USENIX Security'21, 2021.

[5] "Attacks escalating against linux-based iot devices," https://www.esecurityplanet.com/threats/attacks-escalating-against-linux-based-iot-devices/, 2024.

[6] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. E. Shrobe, and M. Payer, "Firmfuzz: Automated iot firmware introspection and analysis," in IoT S&P'19, 2019.

[7] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in USENIX Security'19, 2019.

[8] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation," in ISSTA'22, 2022.

[9] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, F. Dong, Z. Smith, A. Doupé, T. Bao, Y. Shoshitaishvili, and R. Wang, "Greenhouse: Single-service rehosting of linux-based firmware binaries in user-space emulation," in USENIX Security'23, 2023.

[10] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in NDSS'16, 2016.

[11] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in ACSAC'20, 2020.

[12] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware," in Oakland'20, 2020.

[13] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in NDSS'18, 2018.

[14] "Http resources and specifications," https://developer.mozilla.org/en-US/docs/Web/HTTP/Resources_and_specifications, 2024.

[15] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, "All Things Considered: An Analysis of IoT Devices on Home Networks," in USENIX Security'19, 2019.

[16] "Universal plug and play protocol (upnp)," https://en.wikipedia.org/wiki/Universal_Plug_and_Play, 2024.

[17] "Hp jetdirect protocol (jetdirect)," https://en.wikipedia.org/wiki/JetDirect, 2024.

[18] "Line printer daemon protocol (lpr)," https://en.wikipedia.org/wiki/Line_Printer_Daemon_protocol, 2024.

[19] "Multicast dns protocol (mdns)," https://en.wikipedia.org/wiki/Multicast_DNS, 2024.

[20] J. Zhao, Y. Li, Y. Zou, Z. Liang, Y. Xiao, Y. Li, B. Peng, N. Zhong, X. Wang, W. Wang et al., "Leveraging semantic relations in code and data to enhance taint analysis of embedded systems," in USENIX Security'24, 2024.

[21] A. Fioraldi, D. C. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in WOOT @ USENIX Security'20, 2020.

[22] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," NDSS'19, 2019.

[23] P. Srivastava and M. Payer, "Gramatron: effective grammar-aware fuzzing," ISSTA'21, 2021.

[24] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in ESORICS'16, 2016.

[25] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," Oakland'17, 2017.

[26] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in PLDI'11, 2011.

[27] "Linux init process," https://en.wikipedia.org/wiki/Init, 2024.

[28] "Qemu: A generic and open source machine emulator and virtualizer," https://www.qemu.org, 2024.

[29] H. Xiao, Y. Zhang, M. Shen, C. Lin, C. Zhang, S. Liu, and M. Yang, "Accurate and efficient recurring vulnerability detection for iot firmware," in CCS'24, 2024.

[30] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities," in Oakland'23, 2023.

[31] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, "Atropos: Effective fuzzing of web applications for server-side vulnerabilities," in USENIX Security'24, 2024.

[32] "List of tcp and udp port numbers," https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers, 2024.

[33] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," NDSS'19, 2019.

[34] P. Liu, Y. Zheng, C. Sun, C. Qin, D. Fang, M. Liu, and L. Sun, "FITS: Inferring Intermediate Taint Sources for Effective Vulnerability Analysis of IoT Device Firmware," in ASPLOS'23, 2023.

[35] "Binwalk:firmware analysis tool," https://github.com/ReFirmLabs/binwalk, 2024.

[36] "Radare2: Unix-like reverse engineering framework and command-line toolset," https://github.com/radareorg/radare2, 2024.

[37] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. A. Beyah, P. Cheng, K. Lu, and T. Wang, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in USENIX Security'20, 2020.

[38] "Nvd database," http://nvd.nist.gov, 2024.

[39] "netstat: Linux network information display utility," https://en.wikipedia.org/wiki/Netstat, 2024.

[40] "ripd: Routing information protocol," http://isp.vsi.ru/library/Other/Zebra/ripd.html, 2024.

[41] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in CCS'21, 2021.

[42] "Peach fuzzer," https://peachtech.gitlab.io/peach-fuzzer-community, 2024.

[43] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in USENIX Security'12, 2012.

[44] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines," NDSS'19, 2019.

[45] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," Oakland'20, 2020.

[46] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases," NDSS'21, 2021.

[47] J. Shi, Z. Wang, Z. Feng, Y. Lan, S. Qin, W. You, W. Zou, M. Payer, C. Zhang, and CAS-KLONAT, "Aifore: Smart fuzzing based on automatic input format reverse engineering," in USENIX Security'23, 2023.

[48] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces." in USENIX Security'07, 2007.

[49] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, "Netplier: Probabilistic network protocol reverse engineering from message traces." in NDSS'21, 2021.

[50] Z. Luo, K. Liang, Y. Zhao, F. Wu, J. Yu, H. Shi, and Y. Jiang, "Dynpre: Protocol reverse engineering via dynamic inference," in NDSS'24, 2024.

[51] P. Deng, Z. Yang, L. Zhang, G. Yang, W. Hong, Y. Zhang, and M. Yang, "Nestfuzz: Enhancing fuzzing with comprehensive understanding of input processing logic," in CCS'23, 2023.

[52] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in Oakland'19, 2019.

[53] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "Tiff: Using input type inference to improve fuzzing," in ACSAC'18, 2018.

[54] N. Bars, M. Schloegel, T. Scharnowski, N. Schiller, and T. Holz, "Fuzztruction: Using fault injection-based fuzzing to leverage implicit domain knowledge," in USENIX Security'23, 2023.

[55] "inotify(7) — linux manual page," https://man7.org/linux/man-pages/man7/inotify.7.html, 2025.

[56] X. Yin, R. Cai, Y. Zhang, L. Li, Q. Yang, and S. Liu, "Accelerating Command Injection Vulnerability Discovery in Embedded Firmware with Static Backtracking Analysis," in IOT'22, 2022.

[57] "Cwe-476: Null pointer dereference," https://cwe.mitre.org/data/definitions/476.html, 2024.

Appendix A. Example of Context-Free Grammar

Figure 6 showcases a context-free grammar of HTTP protocol. HOUSEFUZZ leverages such a protocol to describe standard HTTP protocol for guiding test case generation.

$$
\begin{aligned}
\langle \text{HTTP} \rangle &\rightarrow \langle \text{SLine} \rangle \langle \text{Headers} \rangle \langle \text{Body} \rangle \langle \text{LineEnd} \rangle \\
\langle \text{SLine} \rangle &\rightarrow \langle \text{Method} \rangle \langle \text{Space} \rangle \langle \text{URI} \rangle \langle \text{Space} \rangle \langle \text{Version} \rangle \\
\langle \text{URI} \rangle &\rightarrow \langle \text{Scheme} \rangle \langle \text{Authority} \rangle \langle \text{Path} \rangle \langle \text{Query} \rangle \\
\langle \text{Query} \rangle &\rightarrow ?\langle \text{QS} \rangle | \langle \text{Empty} \rangle \\
\langle \text{QS} \rangle &\rightarrow \langle \text{KV} \rangle | \langle \text{KV} \rangle \& \langle \text{QS} \rangle \\
\langle \text{Headers} \rangle &\rightarrow \langle \text{Header} \rangle \langle \text{LineEnd} \rangle \langle \text{Headers} \rangle \\
\langle \text{Header} \rangle &\rightarrow \langle \text{Key} \rangle : \langle \text{Value} \rangle \\
\langle \text{KV} \rangle &\rightarrow \langle \text{Key} \rangle = \langle \text{Value} \rangle \\
&\cdots \\
\langle \text{Body} \rangle &\rightarrow \langle \text{JSON} \rangle | \langle \text{XML} \rangle | \langle \text{RAW} \rangle
\end{aligned}
$$

Figure 6: Part of a context-free grammar example of HTTP protocol ($\langle \text{HTTP} \rangle$ is the starting symbol)

.

Appendix B.
Vulnerability Oracle Implementation

HOUSEFUZZ aims to detect two kinds of firmware vulnerabilities—memory corruption and command injection vulnerabilities because they are prevalent in firmware [2] and frequently attacked [4], [20], [29], [56]. For memory corruptions, HOUSEFUZZ adopts the common approach to detect crash signals (e.g., segmentation fault) in all service processes. However, some memory corruptions in utility processes are unexploitable. For example, not checking nullable query results may lead to NULL pointer dereference bug [57] in a utility process, which simply causes the failure of request handling fails, but has no security impacts to the service. So we regard such memory violations as bugs instead of vulnerabilities. HOUSEFUZZ filters out unexploitable crashes in utility processes by further testing whether attacker-controlled inputs may control the memory corruption impacts. Specifically, HOUSEFUZZ flips each bit of the crash-causing test case (i.e., PoC) to create new test cases and feed these test cases to the service under test. If any new test case triggers the same crashing site with different corruption impacts (e.g., accessing a different broken pointer instead of a NULL pointer), we consider the original PoC to correspond to a real vulnerability, otherwise, it is just regarded as a bug. This filtering process is efficient to apply because the crashing of utility processes is typically a rare event during fuzzing.

Command injection vulnerabilities are typically exploited with "trigger strings". Trigger strings involve characters that have special meaning for a command interpreter, to allow commands embedded in a trigger string to be executed. Former work [30], [31] thus detect command injections by inserting trigger strings into test cases and then detecting the execution of the embedded command at high-level command execution APIs (e.g., *php_exec_ex()* and *eval()* in PHP). HOUSEFUZZ introduces two optimizations to this approach for firmware vulnerability detection. First, HOUSEFUZZ instrument *execve()* system call instead of high-level command execution APIs because vendors may implement customized APIs. Directly finding these APIs in firmware binary is challenging without symbols and API documentation. Second, HOUSEFUZZ does not expect fuzzing to directly generate a PoC by blindly injecting trigger strings, because trigger strings can be rejected by even weak input sanitization. For example, the sanitization may reject semicolon-based trigger string (e.g., *";CMD"*), but allow back quote-based trigger string (e.g., *" `CMD` "*) to reach *system*-like APIs that interprets *"CMD"* as a command. Instead, HOUSEFUZZ first uses literals consisting of common letters to bypass input sanitizations. When these literals are detected at arguments of *execve()*, HOUSEFUZZ tries to exploit the *execve()* call by replacing the literals with 22 summarized trigger strings and detecting real command injections.

TABLE 9: CVE/CNVDs discovered using HOUSEFUZZ

Vendor	Binary	Security Impact	Assigned ID
TRENDnet	ssi	BOF[1]	CVE-2024-36728[*‡]
TRENDnet	ssi	BOF[1]	CVE-2024-36729[*‡]
TOTOLINK	cstecgi.cgi	BOF[1]	CVE-2024-41632[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41636[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41637[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41638[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41639[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41640[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41641[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41642[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41643[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41644[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41645[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41647[*‡]
TOTOLINK	cstecgi.cgi	CLI[2]	CVE-2024-41648[*‡]
ASUS	httpd	BOF[1]	CNVD-2024-34840[*]
ASUS	httpd	DoS[3]	CNVD-2024-34836[*]
ASUS	httpd	DoS[3]	CNVD-2024-34837[*]
ASUS	httpd	DoS[3]	CNVD-2024-34838[*]
ASUS	httpd	DoS[3]	CNVD-2024-34839[*]
ASUS	httpd	DoS[3]	CNVD-2024-34841[*]
D-Link	atp	BOF[1]	CVE-2024-55163[*‡]
D-Link	atp	BOF[1]	CVE-2024-55167[*‡]
D-Link	lighttpd	BOF[1]	CVE-2024-55164[†]
D-Link	lighttpd	BOF[1]	CVE-2024-55165[†]
D-Link	lighttpd	BOF[1]	CVE-2024-55166[†]
D-Link	lighttpd	DoS[3]	CVE-2024-55168[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55169[†]
NETGEAR	uhttpd	BOF[1]	CVE-2024-55170[†]
NETGEAR	uhttpd	BOF[1]	CVE-2024-55171[†]
NETGEAR	uhttpd	BOF[1]	CVE-2024-55172[†]
NETGEAR	uhttpd	BOF[1]	CVE-2024-55173[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55174[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55175[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55176[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55177[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55178[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55179[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55181[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55183[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55184[†]
NETGEAR	uhttpd	DoS[3]	CVE-2024-55185[†]
TRENDnet	httpd	DoS[3]	CVE-2025-26139[†]
TRENDnet	httpd	BOF[1]	CVE-2025-26140[†]
TRENDnet	httpd	BOF[1]	CVE-2025-26141[†]

[1] BOF: Buffer overflow.
[2] CLI: OS Command Injection.
[3] DoS: Denial of Service.
[*] Vulnerability detected by HOUSEFUZZ in §8.2.2.
[†] Vulnerability detected by HOUSEFUZZ in §8.2.1.
[‡] Multi-process vulnerability.

Appendix C.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

This paper presents a grey-box fuzzing method for multi-process interaction of Linux-based firmware services. HouseFuzz considers the interacting process as a single unit and fuzzes them as a whole. The evaluation shows that HouseFuzz was able to perform significantly better than the state-of-the-art approaches that ignore process interactions.

C.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field

C.3. Reasons for Acceptance

1) This paper addresses an important and comprehensive approach to solving the problem of fuzzing multi-process interactions of Linux-based firmware services.
2) Experimental results demonstrate that the techniques for handling multi-process interactions were effective at improving coverage.
3) Significant impact of the system, with 156 0-day vulnerabilities found.

C.4. Noteworthy Concerns

1) The reviewers noted that while the paper demonstrates customized service protocols and complex multi-process communications are common in the services under test, it is unclear how prevalent they are in real-world settings more broadly.