Effective On-Hardware Fuzzing of Embedded Operating Systems

Yuheng Shen
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China
syh1308@gmail.com

Jianzhong Liu
Shandong University
Qingdao, China
liujianzhong@sdu.edu.cn

Qiming Guo
Beihang University
Beijing, China
grub_49@163.com

Yifei Chu
Tsinghua University
Beijing, China
chuyf24@mails.tsinghua.edu.cn

Qiang Zhang
Hunan University
Changsha, China
zhangqiang9413@126.com

Heyuan Shi
Central South University
Changsha, China
hey.shi@foxmail.com

Yu Jiang
Tsinghua University
Beijing, China
jiangyu198964@126.com

Abstract

Fuzz testing embedded OSs is difficult because their implementations vary widely and rely on specialized hardware. These factors render many existing methods ineffective, since they prevent adapting established fuzzing routines, disrupt communication with the target OS, and impede observation of runtime behavior. This paper introduces EOF, a feedback-guided fuzzer designed to test embedded OSs running on actual hardware. Through the debug port, EOF communicates with the target embedded OS, executes test cases, and collect feedback data, with no dependence on OS services. Then, EOF deploys a cross-platform agent and executes API-aware input across diverse hardware. Last, EOF collects runtime coverage and critical execution events to identify interesting seeds and find potential bugs during fuzzing. We implemented EOF and evaluated its performance on four different embedded OSs, where EOF discovered 19 bugs and achieved a 50.84% coverage improvement on average compared with other comparable fuzzing methods.

CCS Concepts: • **Security and privacy → Operating systems security**; **Embedded systems security**.

Keywords: Fuzz Testing, Embedded Operating System

Heyuan Shi and Yu Jiang are the corresponding authors.

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769326

ACM Reference Format:
Yuheng Shen, Jianzhong Liu, Qiming Guo, Yifei Chu, Qiang Zhang, Heyuan Shi, and Yu Jiang. 2026. Effective On-Hardware Fuzzing of Embedded Operating Systems. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3767295.3769326

1 Introduction

Embedded Operating Systems (Embedded OSs) are essential in powering billions of IoT devices that are integrated into our daily lives. The diversity in implementation and runtime environment of embedded OSs allows them to suit different use scenarios. However, this diversity introduces complexity that makes these systems prone to errors. For example, eleven 0-day bugs in VxWorks [3] made 200 million devices worldwide vulnerable. Therefore, it is important to proactively identify these potential bugs to ensure the security and robustness of embedded OSs.

Background: Fuzz testing (fuzzing) [4, 15, 17, 18, 24] is a popular software testing technique. This approach involves randomly generating test cases for the system-under-test (SUT) and monitoring for unexpected behaviors. In order to test embedded OSs [8, 26, 27, 32], fuzzers typically generate API sequences and deploy the target OS on emulators or real devices to trigger and detect potential bugs. For instance, Tardis [28] utilizes the shared memory mechanism to test several embedded OS on top of the QEMU emulator, Gustave [9] employs a highly customized QEMU board to detect memory-related errors in POK OS, and GDBFuzz [10] proposes to use GDB as the communication channel to test embedded software on actual physical devices.

Motivation: Despite ongoing efforts, existing embedded OS fuzzing technologies fall short of conducting full-system

testing on hardware. Specifically, emulator-based testing is not a one-size-fits-all method; most embedded OSs need to run on diverse hardware to meet specific task requirements. For example, in industrial control and robotics, where embedded OSs run on STM32H745 [1], emulators are not available. This bars emulation-based tools such as Tardis from testing them. Another aspect is that application-level fuzzing falls short because it mutates random buffers at entry points without considering embedded OS API constraints or system state, resulting in the testing process being rejected early and leaving kernel subsystems under-exercised. This restricts the vulnerability discovery capabilities, as a large part of the embedded OS's functionalities and deeper kernel-level interactions remain under-explored. To this end, we need a method that is capable of fuzzing embedded OSs, which is capable of (1) testing embedded OSs running on actual hardware and (2) conducting full system testing. To achieve the above goal, we need to address three main challenges.

Challenges: First, to fuzz embedded OSs, we need to deploy our fuzzing harness across diverse embedded OSs and hardware platforms. Specifically, fuzzing needs an agent process on the target system to conduct data transmission and test case execution. However, different embedded OSs vary in implementation details, i.e., API definitions and error handling mechanisms, and embedded OSs tend to be tightly coupled to the specific hardware platforms. Each platform may have different processor architectures, memory layouts, and peripheral settings. Conventional OS fuzzing [25, 32] often targets uniform POSIX environments with standardized APIs. In contrast, embedded OSs often implement the same function differently. For task creation, FreeRTOS uses `xTaskCreate()` with optional static stacks and tick-driven scheduling, whereas Zephyr uses `k_thread_create()` under fully preemptive scheduling with work queues. Existing embedded-system fuzzers [10, 21] typically rely on random byte buffers, which ignore structured API constraints, and are thus rejected early, and fail to drive exploration of deeper kernel subsystems. This requires the fuzzer to navigate a broad spectrum of system implementations and tailor these approaches to accommodate different hardware architectures, significantly complicating the deployment and execution process.

Second, we need to monitor and maintain the integrity of the target OS. Since embedded OSs tend to be volatile, and previously executed test cases may result in the system entering certain degraded states that hinder further fuzzing payload executions. For example, due to some improper use of APIs, the execution may get stuck in an infinite polling loop. These degraded states, while not being actual bugs, need to be detected promptly and signal the fuzzer to restore the system to a benign state, where fuzzing can proceed. Acquiring such information is challenging due to hardware isolation, inherently limiting the fuzzer from observing and manipulating the target. General OS fuzzing approaches often rely on VM introspection and snapshots to monitor and reset the system state, while hardware-based fuzzing approaches [10] rarely address this issue, leading to the fuzzing process requiring certain manual interventions. Therefore, fuzzing on hardware requires watchdogs to check the system's liveness, facilitate a reliable state restoration, and accommodate different hardware platforms.

Third, we need to construct an effective fuzzing loop for the target system running on hardware. An effective fuzzing process must rely on various feedback mechanisms, such as code coverage and runtime bugs. However, embedded OSs are tightly integrated with their varied hardware architectures, which often lack standardized coverage tools and exhibit hardware-specific bug-handling behaviors, including unique interrupt signals and crash responses specific to each hardware environment. Existing mechanisms, such as KCOV [31] and Kernel Address Sanitizer (KASAN) [12], target Linux-style kernels with MMU-based processes. KCOV relies on a kcov device with per-thread buffers exposed via ioctls, and KASAN relies on compiler instrumentation with shadow memory and a runtime. However, typical embedded OS lack process isolation and user space, have minimal C library support, run much of the time in interrupt context, and impose tight RAM and allocator constraints, so it is hard to implement these mechanisms. Additionally, some fuzzing approaches [9, 21] adapt ASAN to embedded systems by heavily modifying the sanitizer, which is labor-intensive and challenging to port across different architectures.

Solutions: To effectively address these challenges, we propose EOF, a feedback-guided embedded OS fuzzer that can test embedded OSs running on actual physical devices. The core idea is to conduct guided fuzzing and runtime monitoring through the hardware debug interface without relying on the details of the OS implementation and architecture. EOF conducts fuzzing through the following procedures. First, via the hardware debug interface, EOF controls and synchronizes the fuzzing process, it establishes communication with the target, deploys a small cross-platform agent, and drives API-aware tests uniformly across different embedded OS implementations and hardware. Second, the debug interface enables EOF to establish a unified feedback-guided and liveness maintenance mechanism. It can collect runtime coverage and critical system events such as exceptions, error logs, and reset the system when the system reaches unrecoverable states or crashes are detected. By combining these procedures, EOF is capable of conducting effective embedded OS fuzzing on real physical devices.

Evaluation: We evaluated EOF on four widely-used embedded OSs. In detail, EOF found a total of 19 bugs among these embedded OSs. Also, compared to existing fuzzing approaches, including Tardis and GDBFuzz, EOF achieved an average of 34.84%, 35.51%, and 107.03% coverage improvement. Furthermore, we implemented EOF-nf, which is EOF without feedback guidance, and evaluated its effectiveness

compared to EOF, where EOF demonstrated an average of 35.16% coverage improvement. For the instrumentation overhead, EOF on average exerts a 6.44% and 23.39% runtime overhead in terms of the memory consumption and the execution latency.

In summary, this paper makes the following contributions:

- We propose EOF, a feedback guided embedded OS fuzzer that can test different embedded OSs across diverse environments, and we have open-sourced it [1].
- We present a hardware-based fuzzing technique. By leveraging the debug interface and cross-platform execution agent, EOF is capable of testing embedded OSs on hardware.
- For evaluation, EOF detected a total number of 19 bugs among four popular embedded OS, with 5 bugs confirmed, while achieving a higher code coverage compared to other fuzzing approaches.

2 Background and Related Works

2.1 Embedded Operating Systems

Embedded Operating Systems (a.k.a, embedded OSs) are specialized frameworks designed to manage hardware resources and provide essential services for embedded systems. They support a broad range of applications and are adaptable to various hardware architectures, serving devices from simple microcontrollers to complex controllers in automotive and industrial environments.

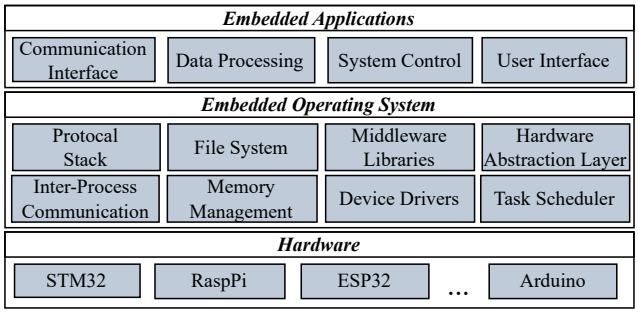


Figure 1. Architecture of embedded operating systems.

The overall architecture of an embedded OS is depicted in Figure 1. The embedded OS is deployed on various hardware platforms, such as STM32 on ARM and ESP32 on Xtensa or RISC-V. These boards enable the embedded OS to suit various application scenarios. Also, the embedded OS comprises several essential components. At its core, the task scheduler coordinates task execution, and the memory management optimizes resource allocation. Device drivers enable hardware communication, and inter-process communication (IPC) ensures process coordination. The hardware abstraction layer (HAL) standardizes interactions across different hardware, enhancing portability. Additionally, protocol stacks like TCP/IP manage networking, while file systems handle data storage. Middleware libraries provide essential services, such as security checks and support for higher-level functions. The top application layer integrates components for communication, data processing, system control, and user interface, facilitating tasks from external device interaction.

Due to the complex implementations and diverse deployment environments, embedded OS are inherently hard to run on emulated environments. In detail, running directly on hardware such as STM32 and ESP32, embedded OS often exhibits hardware-specific behaviors that are difficult to replicate in emulators, particularly peripheral simulation. Furthermore, communication with embedded OS on hardware is constrained by limited interfaces like serial ports or specific networking protocols, which complicates remote interaction compared to a general-purpose OS.

2.2 Embedded System Fuzzing

Fuzz testing [6, 14, 15, 20, 23, 26, 29, 30, 36, 38], a.k.a. fuzzing, is an automatic software testing technique. Fuzzing leverages program analysis, generating large test suites and monitoring targets for crashes or bugs. By leveraging different feedback information, such as code coverage or new system states [35], fuzzers can generate more high-quality test cases to explore more system behaviors. Due to its high efficiency, fuzzing has been widely adopted in various software testing scenarios. Currently, embedded system fuzzing can typically be classified into two types: embedded OS fuzzing and embedded application fuzzing.

For **embedded OSs**, the fuzzer usually runs the target OS in an emulation-based environment such as QEMU or VMware. Then, it uses predefined API specifications as input; each specification encompasses an API sequence with the corresponding arguments. Based on specifications, the fuzzer generates corresponding test cases and feeds them to the target embedded OS through the network stack or shared memory. Through some VM introspection techniques, the fuzzer collects runtime signals such as code coverage and potential crashes. For example, Tardis [28] is built upon Syzkaller; by extending Syzkaller to support data transfer using QEMU's shared memory mechanism, Tardis can test various embedded OSs. Gustave [9] is built upon AFL; it leverages the QEMU TCG mechanism to collect code coverage and detect memory corruption errors.

However, current fuzzing methods cannot cover the full spectrum of embedded OS that operate directly on hardware. In detail, the emulation-based fuzzing technique cannot test those embedded OSs solely running on hardware. These embedded OSs require direct interaction with hardware peripherals and sensors that are tailored for specific microcontrollers. This is particularly evident in industrial control and robotics, where embedded OSs run on customized hardware. For example, many STM32H7-based controllers lack peripheral-accurate emulators, making emulation-based

[1] the source code can be found at https://github.com/Rrooach/EOF-eurosys26

tools impractical. Consequently, the inability of emulation-based fuzzing to replicate and interact with these specific hardware setups significantly limits its testing coverage.

For fuzzing **embedded applications**, some works try to conduct fuzzing on emulators; for example, AFL's QEMU mode [16] runs embedded applications under QEMU and tests them like ordinary user-space programs. Some works attempted fuzzing on embedded applications running on real physical devices. For instance, GDBFuzz [10] leverages debug interfaces for effective fuzzing of embedded systems, enhancing feedback with hardware breakpoints across diverse microcontrollers. P2IM [11] employs automatic peripheral interface modeling to enable the fuzzing of embedded firmware across a broad range of devices. SHIFT [21] leverages semihosting instrumentation, enabling fuzzing on certain hardware platforms with sanitizer and coverage support.

However, current approaches remain insufficient for exercising the OS itself. Application-level fuzzing typically adopts AFL-style random buffers that rarely satisfy OS API preconditions, and focusing on high-level services such as network servers, seldom drives deep kernel behaviors such as scheduling and interrupt handling. Semihosting instrumentation can indeed uncover more bugs by enabling sanitizer and coverage signals on real hardware, but it is tied to specific architectures and OSs and often demands substantial manual, board-specific setup, which limits portability.

3 Motivation

Although current works such as Tardis, GDBFuzz, and SHIFT can test embedded systems effectively, adapting these fuzzers to test embedded OSs that are running on diverse hardware would require extensive redevelopment of their core functionalities and involve overcoming inherent limitations in how they interact with the target system. For example, such hypothetical modifications would require GDBFuzz and Tardis to integrate deeper-level system monitoring and control capabilities that can handle the complex interactions and dependencies of full operating systems. Moreover, SHIFT would need to extend its semihosting approach to support a wider range of architectures and embedded OSs beyond its current scope. Therefore, we need an approach that can effectively conduct full embedded OS fuzzing on hardware.

Despite embedded OS diversity, the basic fuzzing principles remain similar, as shown in Figure 2. The fuzzing process contains three steps: (1) fuzzing harness deployment, (2) liveness management, and (3) unified fuzzing loop construction, each of which poses unique challenges when fuzzing embedded OSs on actual physical devices.

3.1 Fuzzing Harness On-Target Deployment

The first step for embedded OS fuzzing loop is to establish communication with the target system and deploy the fuzzing harness on the target system. This includes establishing channels for data transfer and setting up a fuzzing agent on the target system to execute the test cases. Furthermore, we need to synchronize the execution status between the fuzzer and the target OS, i.e., tracking whether it is ready to receive the next test case and is prepared for execution.

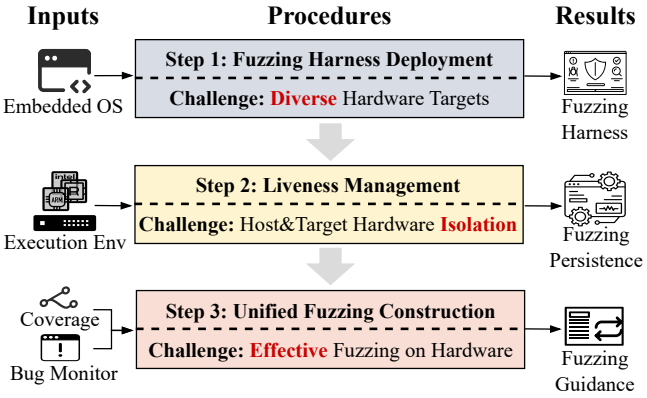


Figure 2. Embedded OS fuzzing steps and challenges.

Challenge: Deployment of a fuzzing harness on diverse hardware targets. Different embedded OSs are tailored to specific hardware and use cases, so their APIs, resource abstractions, and memory models diverge, which dictate unique runtime logic and communication mechanisms. For example, automotive systems favor deterministic execution and often use QNX, managing I/O interfaces over CAN [19]. In contrast, IoT devices typically run FreeRTOS on single-tasking MCUs and prioritize low-power connectivity such as Bluetooth and USB. This heterogeneity breaks assumptions of general fuzzing that rely on IPC, a unified network stack, or glibc-based harnesses. However, on real hardware, such facilities are absent, and there is no standard control channel. Therefore, to enable embedded OS fuzzing across different environments, we need to establish a unified fuzzing harness that avoids specifics of each OS and platform and thus can accommodate diverse OS implementations and hardware architectures.

3.2 State and Liveness Management

To guarantee that fuzzing proceeds normally, we need to conduct liveness management. This requires monitoring the system's states to determine whether the system is running correctly. Then, in the case of reaching a degraded state due to improper API use or unexpected connection breakup, we need to restore it to a benign state effectively and correctly.

Challenge: Maintain system states under hardware-isolated scenarios. Embedded OSs are volatile due to the lack of proper isolation between user space and kernel space, so faults in application code can propagate into the kernel and leave the device hung or even with a corrupted image. Because on-board diagnostic methods are proprietary and

restricted, liveness monitoring on real hardware is difficult, as error signals and behavior vary by board, and UART logs may vanish after a fault, especially in hardware-isolated environments. Previous approaches detect degraded states by leveraging the program exit code or watchdog timers and resetting the system state with a simple reboot. However, in the embedded OS setting, a simple reboot is insufficient when the flash or filesystem is damaged, and reloading firmware requires board-specific bootloader steps. For example, on FreeRTOS with ESP32, liveness is typically inferred from UART logs, and recovery requires board-specific bootloader settings and memory layout tweaks to reflash the image. More broadly, there is no uniform, OS-independent way to read the fault status or memory health, nor a portable way to reset and reflash across platforms. Thus, any viable solution needs a single vendor-agnostic control and observation channel that works across boards and operating systems.

3.3 Unified Fuzzing Loop Construction

The final step for effective on-board fuzzing is to monitor the system's runtime feedback and use this information as generation guidance in the overall fuzzing process. In detail, this involves collecting coverage data, detecting potential bugs from different embedded OSs, and using this information to guide payload generation.

Challenge: Construct an effective fuzzing loop for the target system on hardware. Embedded OSs often lack comprehensive feedback mechanisms, like coverage or bug monitoring, due to limited hardware resources and diverse OS implementations. General fuzzing techniques leverage KCOV or other coverage tools to collect coverage and leverage KASAN to detect memory-related bugs. However, embedded OSs tend to have diverse error-handling mechanisms, such as different interrupt signals and error-handling functions. This inconsistency in feedback mechanisms prevents the fuzzer from effectively collecting coverage and monitoring vulnerabilities. Furthermore, compared to general-purpose OSs, embedded OSs have much smaller memory footprints and fewer facilities (e.g., no MMU or user space), making feedback tools such as KCOV or KASAN hard to adapt and deploy. Therefore, to conduct in-depth fuzzing on embedded OSs, a more comprehensive and unified feedback mechanism is needed to guide the fuzzing process and monitor the overall fuzzing status.

4 Design

In this section, we first introduce the threat model, then we propose our design choices that are tailored to address the identified challenges and threat model.

4.1 Threat Model

Throughout this paper, we use the following threat model. (1) Attackers' capabilities. We assume no OS–application isolation, and that an adversary can steer application code to invoke OS APIs with arbitrary arguments and sequences. This worst-case assumption facilitates systematic fuzzing but exceeds typical deployments. (2) Bug definition. A bug is a liveness or memory-safety failure (e.g., crash, out-of-bounds access) observable via explicit fault signals.

The justification is that memory-safety and liveness faults are common and can render systems unavailable. In practice, adversaries usually affect only a subset of APIs and arguments exposed by a specific application.

4.2 EOF Architecture

To this end, we propose EOF, a embedded OS fuzzer that runs on real hardware and uses the hardware debug interface as the single channel for control and observation. First, EOF runs a cross-platform agent on the device to execute API-aware tests and collect feedback, reducing porting effort and avoiding reliance on OS services. Second, it externalizes guidance and liveness over the debug interface, enabling abnormality detection and uniform coverage collection across different embedded OS implementations and boards. Third, it uses API specifications to generate inputs that satisfy preconditions and drive deeper kernel paths rather than mutating opaque buffers at application entry points. As shown in Figure 3, EOF instruments the target for sanitizer-style coverage, uses log and exception monitors for bug detection, extracts API specifications to produce API-aware test cases, and maintains liveness with stall and timeout watchdogs, as well as image restoration based on the target's memory layout. Hardware breakpoints keep the host fuzzer and target embedded OS synchronized during execution. Over the debug interface, EOF transfers test cases, GDB control, and reflash commands, and collects feedback such as coverage, exceptions, and logs. Inputs that trigger new coverage or a crash are marked as interesting and added to the corpus for further mutation.

4.3 Fuzzing Harness Deployment

The fuzzing harness is responsible for managing the fuzzing process. This involves connecting the host fuzzer to the hardware board, transferring and executing test cases, and synchronizing execution flow between the host fuzzer and the guest execution agent. However, embedded OSs are diverse in terms of implementation details and hardware environment and lack standard system abstraction. Therefore, to effectively conduct fuzzing on embedded OSs, we need to establish a unified interface that can accommodate the diversity of OS implementations and have a control mechanism that can synchronize the execution between host and guest.

4.3.1 Communication Establishment. Given the diversity of embedded OS implementations and runtime environments, with different system abstractions and communicating mechanisms, a standardized approach is essential

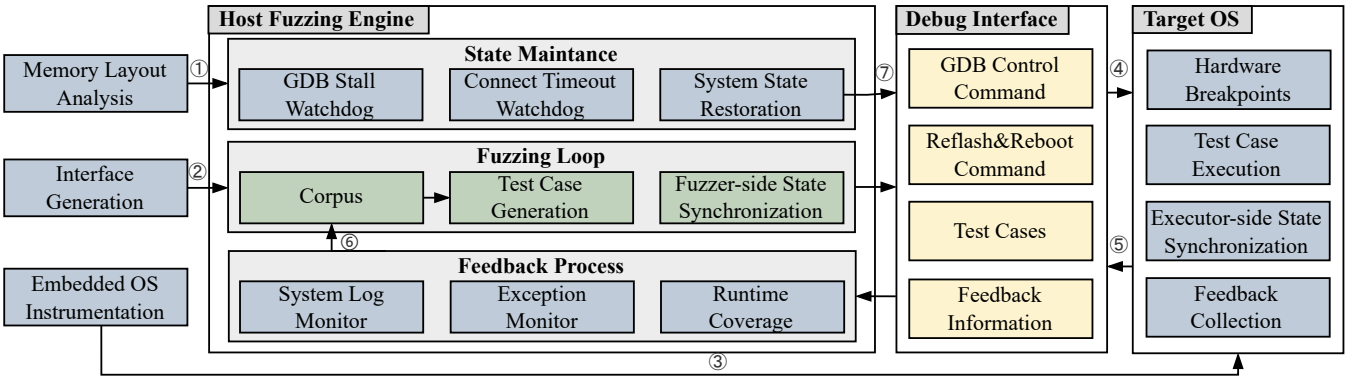


Figure 3. Overall workflow of EOF. Before fuzzing starts, EOF ① analyzes the target embedded OS's memory layout, ② extracts and generates the API specifications, and ③ instruments the target OS. Using the debug interface, EOF establishes the connection with target OSs, ④ sends test cases and fuzzing commands, and receives ⑤ feedback such as coverage and exceptions. Utilizing runtime coverage and detected bugs, EOF ⑥ guides the test case generation. The state maintenance module monitors the system's liveness and ⑦ resets it when the system reaches an unrecoverable state.

to ensure the interaction across different platforms. To address this, EOF utilizes debug interfaces on hardware, such as JTAG and SWD, to establish communication between the host fuzzer and the target embedded OS. These interfaces enable EOF to read/write memory, set breakpoints, and manage the fuzzing harness's execution.

Take testing FreeRTOS running on an ESP32 board, for example. The ESP32 development board uses JTAG as the debug interface, while the fuzzer employs OpenOCD [13] to connect to the JTAG interface. Once OpenOCD is connected to the JTAG, the fuzzer can interact with the target embedded OS, sending test cases via direct memory access, controlling execution flow by setting breakpoints, and monitoring the system's status. Additionally, EOF captures the target OS's UART output and redirects it to the stdout channel as the target OS's runtime log to facilitate further bug analysis. This setup allows for precise control over test execution and real-time monitoring of the target system. By leveraging such a communication framework, EOF can be deployed across different platforms with minimal customization.

4.3.2 Cross-platform Execution Agent. After establishing communication with the target embedded OS, EOF requires an execution and synchronization mechanism to manage the execution flow and data transfer between the host fuzzer and the target OS. However, embedded OSs are diverse in implementation details, EOF needs to ensure that the execution and synchronization mechanism can adapt to diverse embedded OS implementations. To this end, EOF employs a cross-platform agent, which is embedded within the target system to manage the overall fuzzing process.

This agent is responsible for deserializing and executing test cases. It utilizes only primitive operations such as integer/bitwise arithmetic and direct array reads/writes, and is independent of any specific architecture or OS services.

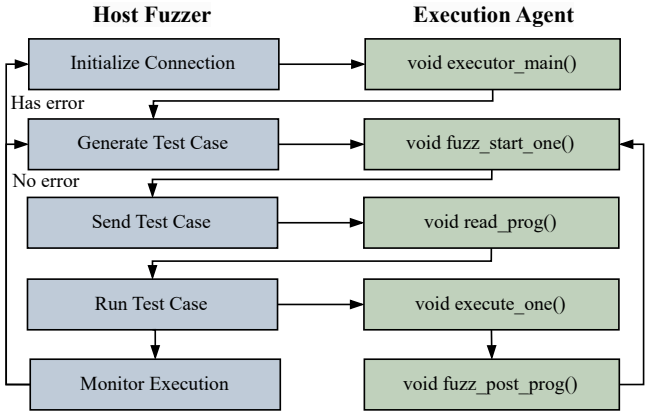


Figure 4. Execution and synchronization mechanism between the host fuzzer and target embedded OS.

For synchronization, once the target OS starts, EOF inserts breakpoints at key points within the agent's workflow, such as at the start and end of the execution. During fuzzing, the agent pauses at each breakpoint, waiting for the fuzzer to perform tasks such as test case transmission and feedback collection before continuing execution. This setup ensures that the fuzzer maintains precise control over the execution flow and data transfers, enabling efficient and correct input execution. It is worth noting that EOF focuses on testing the basic functions of embedded OS, and while features like interrupt handling are important, they are currently outside EOF's scope.

We use Figure 4 to show how EOF uses the debug interface to manage the fuzzing process after attaching it to the target OS. Initially, the fuzzer sets breakpoints at binding points within the agent's workflow, such as startup, test case reception, and execution start. When the target OS boots,

it pauses at `executor_main()`, where EOF generates and sends a new test case (target OS's API sequences). The agent receives this case and deserializes it in `read_prog()`. Once deserialization completes, the fuzzer commands the agent to execute the test case at `execute_one()`. After the execution, the agent loops back to the starting point of the fuzzing loop. During this loop, the fuzzer monitors for anomalies like system crashes or coverage buffer overflow. If a crash is detected at `handle_exception()`, EOF resets and reconnects to the system. If the coverage buffer is full, the agent program will pause while the fuzzer collects and clears the buffer accordingly. This method ensures precise control over the execution flow and data transfer between the host fuzzer and the execution agent, enabling efficient input handling and output monitoring.

4.4 System State Maintenance

To ensure the target embedded OS remains alive and operational during the fuzzing process, EOF needs to monitor the target embedded OS's runtime states continuously. This involves periodically checking if the system is still responsive and functioning as expected. These checks are vital because embedded OSs often operate under constrained and complex hardware conditions that can lead to unexpected failures. When a liveness check detects that the system has entered an unrecoverable state, such as when the embedded OS becomes unresponsive, EOF needs to recover the damaged OS; this action ensures that the fuzzing process can continue without manual intervention.

4.4.1 Liveness Watchdogs. Unlike general-purpose operating systems that run in virtualized environments, where system states can be probed using various VM introspection techniques, embedded OSs running on hardware often lack straightforward access points to assess their internal state. This makes it difficult to determine whether the system still functions correctly or has entered an unrecoverable state. To address this, EOF leverages two liveness watchdogs compatible across different hardware platforms that can reflect the system's operational status. These watchdogs do not detect actual bugs but are essential for determining whether the target embedded OS has become unresponsive or has reached some critical states that can impede further testing. Also, these watchdogs run on the host side with event-driven heartbeat checks and PC stall checks over the debug link, require no target instrumentation, and do not interfere with test execution. This is critical for maintaining system stability and preventing extended downtimes in scenarios where the system's functionality is compromised.

In detail, to make testing proceed effectively, the monitor detects three event types, namely boot failure, unresponsiveness to the host, and execution stall. The above events are then mapped to 2 watchdog checking mechanisms, (1) whether the debug interface encounters a connection timeout and (2) if the GDB's `exec-continue` command fails to change the program counter (PC) value, as shown in LivenessWatchDog() in Algorithm 1. For the first watchdog, when the debug interface encounters a timeout, it suggests that the system has either failed to boot correctly or has become entirely unresponsive (lines 4-5). This often indicates that the embedded OS image is compromised or the system has experienced a severe fault. The second watchdog triggers when the GDB continue command (`-exec-continue`) fails to change the PC value (lines 6-10). This means that the system cannot execute any instructions, likely due to a corrupted image. When either of these watchdogs is triggered, EOF determines that the system needs to be salvaged and initiates restoration procedures to restore the system to a known good state.

Algorithm 1: System State Monitor and Restore

```
   Input: KConfig: OS's Build Configuration File
          DebugPipe: Debug Interface
1  LastPC ← INT_MIN;
2  PartitionMap[file:offset] ← ∅;
3  Function LivenessWatchDog(DebugPipe):
4      if ConnectionTimeout(DebugPipe) then
5          return false;
6      if LastPC = INT_MIN then
7          LastPC ← GetCurrentPC(DebugPipe);
8          return true;
9      else if LastPC = GetCurrentPC(DebugPipe)
        then
10         return false;
11     return true;
12 Function StateRestoration():
13     PartitionMap ← GetPartitionTable(KConfig);
14     while true do
15         if ¬LivenessWatchDog(DebugPipe) then
16             for each Part ∈ PartitionMap do
17                 DebugPipe.flash(Part.file, Part.offset);
18             DebugPipe.reboot();
19         sleep(5s);
```

4.4.2 System State Restoration. After detecting that the system has entered an unrecoverable state, EOF needs to automatically restore the system to an initial state to resume the fuzzing process. However, unlike emulation-based fuzzing, where the system can be reset through snapshots or reboot, embedded OS running on hardware cannot be reset using such means, as there are no snapshot mechanisms in the embedded scenarios, and as such a failure is often due to

image damage, and a reboot cannot resolve it. Therefore, to restore the system state, EOF needs to reset the system by reflashing the entire image. However, the embedded OS image usually contains several components, such as the bootloader and kernel, and each component has its start address and offset; any misconfiguration in these addresses can lead to critical failures. To this end, EOF needs to analyze the target embedded OS's memory layout for different components under different hardware platforms and then leverage the debug interface to reflash the system.

The detailed procedure is depicted in StateRestoration(). In concrete, EOF first examines the embedded OS's build configuration, which extracts the memory partition table (a configuration file supplied by the developer) (line 13). Then, during fuzzing, EOF periodically checks the system's liveness based on the liveness watchdogs (lines 14-15). If EOF finds that the system is unresponsive or has entered erroneous states, it resets the system by reflashing the image and rebooting it using the debug interface (lines 16-18). This mechanism allows EOF to maintain the system's liveness across diverse hardware and embedded OSs, ensuring the fuzzing process can continue without manual intervention.

4.5 Feedback Guided Fuzzing

For feedback-guided fuzzing, EOF needs to generate test cases, collect the target OS's runtime feedback (i.e., coverage data and potential bugs), and leverage such information to guide future test case mutation. However, embedded OS's API are diverse, we need to ensure that the test case generation mechanism can adapt to different embedded OS implementations. Also, embedded OS works under limited memory space and functionalities; as a result, to collect such feedback information, we need a feedback collection mechanism that can be compatible with different embedded OSs and can effectively collect the feedback information.

LLM-based Input Generation. EOF leverages the system call specification, also known as the Syzlang [34], which is adapted from Syzkaller as the initial corpus. Each API specification defines the API signature (name, typed arguments), as well as its constraints. Behaviors that Syzlang does not model well, such as event setting, callbacks, and priorities, are expressed as pseudo syscalls [33]. Each pseudo function can encompass a target OS API sequence that is designed to conduct certain tasks, such as creating and processing a JSON object or sending a network packet, as we can see in Figure 6 lines 3-8. Because manual authoring is tedious and error-prone, we used GPT-4o to generate these specifications, as shown in the left part of Figure 5. The model is prompted with the target embedded OS's headers, unit test examples, and API reference text. We then asked to extract the API signature, typed arguments, and constraints such as value bounds and flags, and to emit pseudo functions that satisfy dependencies. Generated specifications are then post-validated by parsing and type checking, and only validated specifications are admitted to the corpus.

When fuzzing starts, EOF converts Syzlang into an internal abstract syntax tree (AST) that encodes API name, typed arguments, and constraints to facilitate input generation. During each iteration, it constructs a test input by selecting and mutating API specification sequences, scoring call adjacency by resource dependencies and recent coverage, and then sends it to the target embedded OS for execution under the debug interface. After execution, EOF collects coverage and bug information and determines whether the last test case triggered new coverage or revealed faults such as system exceptions or error logs. If so, EOF saves the case to the corpus for further mutation by altering API parameters or adjusting the order of the sequence; otherwise, it discards the case and generates a new one according to the specifications.

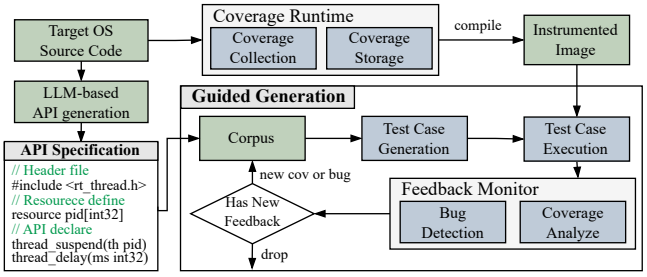


Figure 5. Diagram of the guided fuzzing process.

4.5.1 Coverage Instrumentation. Embedded OSs lack many facilities that current fuzzers rely on, they can hardly provide valid feedback information during fuzzing. Therefore, to measure the runtime coverage and conduct feedback-guided fuzzing, EOF needs to have a coverage collection mechanism that can be deployed across different embedded OSs, with limited overhead.

To this end, EOF instruments the target embedded OS with Sanitizer Coverage (SanCov) [7]. In detail, SanCov provides low-overhead, architecture-agnostic edge coverage that guides fuzzing across diverse embedded OSs, improving portability and efficiency. As shown in the upper part of Figure 5, during compilation, EOF inserts a set of callback functions `__sanitizer_cov_trace_cmp()` at each branch condition to track the program counter and return address at each basic branch. Then, these callback functions call the function `write_comp_data()` to store this data into a local coverage buffer, facilitating EOF to collect and analyze such data at the host machine. Additionally, when the coverage buffer is full, it traps at the function `_kcmp_buf_full()` to inform EOF to reset the coverage buffer and ensure continuous data collection.

4.5.2 Bug Monitors. Unlike general-purpose OSs, which have unified error-handling mechanisms, different embedded OSs have different error-handling behaviors in terms of error signals and exception-handling functions. To effectively monitor the system's abnormal behaviors and detect bugs, EOF needs to implement a unified monitoring mechanism that can adaptively capture and analyze error behaviors across various embedded OS implementations.

To detect potential bugs in embedded OSs, EOF leverages two types of bug monitors. (1) Log Monitor. EOF tracks the system output from both the debug interface and the target OS to identify potential system errors during testing. In particular, EOF redirects all kernel and user logs to the stdout channel and monitors it for any output that matches predefined patterns using regular expressions. Any output matching the defined patterns is considered indicative of a crash. (2) Exception Monitor. During fuzzing initialization, EOF also inserts breakpoints at various embedded OS-specific exception functions like `panic_handler()` in FreeRTOS and `common_exception()` in RT-Thread. Once the agent reaches these functions, the fuzzer captures the relevant crash information. These two monitors can capture the system's explicit errors, such as crashes and assertions, enabling it to detect and analyze potential system exception behaviors.

4.6 Implementation

Overall, we implemented EOF using Golang, Rust, and C. EOF now supports four widely used open-source embedded OSs, namely FreeRTOS [5], RT-Thread [37], NuttX [2], and Zephyr [22]. The execution agent is implemented in C and is used to initialize the target OS, deserialize and execute test cases, and collect feedback information. The host fuzzer engine is implemented in Golang; it controls the overall fuzzing process, including fuzzing loop control, test case generation, mutation, feedback analysis, corpus management, and log output. For host-guest connectivity, we leverage OpenOCD [13], a widely recognized tool for hardware debugging known for its broad support across various platforms. The interaction between the host fuzzer and OpenOCD is implemented using Rust.

Embedded OS Adaptation. To adapt EOF to different embedded OSs, take FreeRTOS as an example. First, we prepare the kernel image. We adapt the execution agent by adding FreeRTOS's system initialization and boot-check logic to the start of the agent, which requires around 50 lines of code. Second, we add instrumentation options during compilation and analyze the kernel build configuration to obtain the memory layout, which requires around 10 lines of modifications to the compilation scripts and an inspection of the build configuration file. Third, we prepare the API specifications for the target OS. This requires writing the corresponding API specifications based on FreeRTOS's source code and documentation; a detailed example can be found in Figure 6, lines 1–8. To ensure a comprehensive specification, we generate 203 lines of API specification code. Fourth, we register the target OS in EOF, which involves adding the target's specifications, such as architecture type and endianness information, and requires around 100 lines of code. Lastly, we provide a configuration file to explicitly set QEMU arguments to boot the kernel and OpenOCD arguments to establish the connection, which requires around 20 lines of code.

5 Evaluation

We list the following research questions to help us understand EOF's performance and effectiveness.

- **RQ1:** What is the adaptability of EOF?
- **RQ2:** Is EOF able to uncover new bugs in embedded OSs running on actual hardware?
- **RQ3:** Can EOF achieve comparable or better code coverage than existing methods?
- **RQ4:** Does EOF present an acceptable instrumentation overhead during fuzzing?

RQ1 verifies whether the cross-platform harness can be deployed on various boards and operating systems, addressing the challenge of harness deployment. **RQ2** checks if EOF's liveness management mechanisms and bug detectors enable it to uncover previously unknown bugs on real hardware, addressing the liveness-management and hardware-isolation challenge. **RQ3** verifies whether EOF's fuzzing methods explore deeper code paths than state-of-the-art baselines, thereby resolving the effective fuzzing challenge. **RQ4** measures the memory and runtime cost added by these features to confirm that they remain acceptable for typical microcontroller budgets.

5.1 Experiment Setup

We evaluate EOF on FreeRTOS(v5.4), RT-Thread(2f55990), NuttX(fc99353), Zephyr(143b14b), and PoKOS(b2e1cc3), we chose these embedded OSs because they are widely used in various embedded scenarios and have different system architectures and implementations, which can effectively demonstrate EOF's adaptability. To adapt fuzzing to the above target, we predefined API specifications based on their user manuals, API specifications, and source code. For coverage comparison, we compared EOF with Gustave [9], SHIFT [21], GDBFuzz [10], Tardis [28], and EOF-nf (EOF without the feedback guidance). Firstly, to demonstrate the full-system fuzzing performance, we compared EOF with Tardis, EOF-nf, and Gustave on target operating systems using the same fuzzing payload. Since Tardis does not support hardware fuzzing, the evaluations are conducted on QEMU. Further, to demonstrate the effectiveness of hardware fuzzing, we compared EOF with GDBFuzz and SHIFT. Since GDBFuzz does not support full-system testing, we chose to test the HTTP server and JSON component in FreeRTOS running on STM32. EOF is limited to testing the HTTP server and JSON API, only with JSON and HTTP server instrumented, same

as GDBFuzz. Furthermore, to demonstrate the effectiveness of guided fuzzing, we implemented EOF-nf, which is EOF without the coverage guidance mechanism.

For overhead analysis, we compared the OS images with and without instrumentation to evaluate the memory overhead by comparing the image sizes. Also, we record the number of executed inputs and the execution time to evaluate the execution overhead. To minimize statistical bias, each experiment is repeated 5 times with a duration of 24 hours.

5.2 System adaptability

To answer **RQ1** and evaluate EOF's adaptability, we compared the targets that EOF supports against those supported by GDBFuzz, Tardis, and SHIFT. As we can see from Table 1, EOF supports a broader range of target systems and boards than other tools.

Table 1. Comparsion on Supported Targets Between EOF, GDBFuzz, SHIFT and Tardis.

Target Systems	Arch	EOF	GDBFuzz	Tardis	SHIFT
FreeRTOS	ARM	✓	–	✓	✓
	RISC-V	✓	–	✓	✓
	Power PC	–	–	–	✓
	MIPS	–	–	–	✓
RTThread	ARM	✓	–	✓	–
Nuttx	ARM	✓	–	✓	–
Zephyr	ARM	✓	–	✓	–
Applications	ARM	✓	✓	–	✓
	RISC-V	✓	–	–	✓
	Power PC	–	–	–	✓
	MIPS	–	–	–	✓
	MSP430	–	✓	–	–

In detail, EOF supports both embedded OS and application-level fuzzing, and can support multiple hardware platforms, including ARM and RISC-V. While GDBFuzz supports embedded applications testing on various hardware platforms, it lacks support for direct embedded OS testing. Furthermore, Tardis can support fuzzing on diverse embedded OSs and on different platforms comparable to EOF, but its inherently design prevents it from testing embedded OSs on hardware, and is limited to QEMU-based fuzzing. SHIFT, on the other hand, shows good compatibility, where it is spans many architectures and both levels. However, SHIFT only supports FreeRTOS, where the rest are not adapted yet.

Although EOF does not yet support architectures such as MIPS and PowerPC, this reflects missing per-platform adaptations rather than inherent capability limits. These platforms provide mature toolchains and boards with standard hardware-debug interfaces (e.g., JTAG/SWD). In practice, any board exposing such interfaces can be ported to EOF, further extending EOF's applicability.

5.3 Bug Detection Capability

To answer **RQ2** and evaluate EOF's bug detection capabilities in embedded OSs, we collected and analyzed the crashes reported by EOF. In detail, EOF found 19 previously unknown bugs, with 5 confirmed, as listed in Table 2.

Table 2. Previously unknown bugs detected by EOF.

#	Target OSs	Scope	Bug Types	Operations	Status
1	Zephyr	Heap	Kernel Panic	sys_heap_stress()	
2	Zephyr	Kernel	Kernel Panic	z_impl_k_msgq_get()	✓
3	Zephyr	JSON	Kernel Panic	json_obj_encode()	✓
4	Zephyr	KHeap	Kernel Panic	k_heap_init()	✓
5	Rt-Thread	Kernel	Kernel Assertion	rt_object_get_type()	
6	Rt-Thread	RTService	Kernel Panic	rt_list_isempty()	
7	Rt-Thread	Memory	Kernel Panic	rt_mp_alloc()	
8	Rt-Thread	Kernel	Kernel Assertion	rt_object_init()	
9	Rt-Thread	Heap	Kernel Panic	_heap_lock()	
10	Rt-Thread	IPC	Kernel Panic	rt_event_send()	
11	Rt-Thread	Memory	Kernel Panic	rt_smem_setname()	✓
12	Rt-Thread	Serial	Kernel Panic	rt_serial_write()	
13	FreeRTOS	Kernel	Kernel Panic	load_partitions()	
14	NuttX	Kernel	Kernel Panic	setenv()	✓
15	NuttX	Libc	Kernel Panic	gettimeofday()	
16	NuttX	MQueue	Kernel Panic	nxmq_timedsend()	
17	NuttX	Semaphore	Kernel Assertion	nxsem_trywait()	
18	NuttX	Timer	Kernel Panic	timer_create()	
19	NuttX	Libc	Kernel Panic	clock_getres()	

As we can see, EOF is capable of detecting bugs in all four target OSs, ranging from user space to kernel space. Concretely speaking, EOF detected 4 bugs in Zephyr, 1 in FreeRTOS, 6 in NuttX, and 8 in RT-Thread, respectively. Among the detected bugs, 3 were found within the user API, mainly the JSON library and libc library (bugs #3, 15, 19), and 16 were located within the kernel, indicating that EOF is capable of conducting full-system level testing of the entire embedded OS. The location of detected bugs is mainly attributed to the fact that EOF's initial specifications target those modules. Also, we find that, with the help of different bug monitors, EOF is capable of detecting a wide range of bugs. In detail, the log monitor helps detect 3 bugs (bugs #5, 8, 17), and the exception function helps detect 16 bugs (bugs #1-4, 6-7, 9-16, 18-19). Among this bug, 5bugs (bugs #2-4, 11, 14) have been confirmed by corresponding maintainers.

Bugs in embedded OSs can cause severe consequences. Specifically, the bugs detected by EOF can lead to potential data loss and system crashes, causing the target system to transition into erroneous states. For example, kernel panic-like bugs #4, 12 are usually raised from illegal memory accesses within the system, which can crash the system and cause data loss. While assertion bugs like #5, 17 indicate that the program encountered an unexpected condition, i.e., an infinite loop in most cases, which leaves the system hung and results in denial-of-service.

5.3.1 Case Study. We use bug #12 in Table 2 to briefly describe a previously unknown kernel panic bug found by EOF in Rt-Thread as the case study to demonstrate the bug detection capability.

```
1  // The predefined syscall definition and implementation
2  syz_create_bind_socket (0xbc78, 0x0, 0x101, 0x0)
3  long syz_create_bind_socket(long domain, long type,
       long protocol, long sockaddr_ptr) {
4    int sock = socket((int)domain, (int)type,
         (int)protocol);
5    if (sock < 0)
6        return -1;
7   ...
8  }
9  // The corresponding stack trace
10 Stack frames at BUG: unexpected stop:
11 Level: 1: /path/to/serial.c : rt_serial_write : 917
12 Level: 2: /path/to/device.c : rt_device_write : 396
13 Level: 3: /path/to/kservice.c : _kputs : 298
14 Level: 4: /path/to/kservice.c : rt_kprintf : 349
15 Level: 5: /path/tosal_socket.c : sal_socket : 1059
16 Level: 6: /path/to/net_sockets.c : socket : 244
17 Level: 7: /path/to/agent : syz_create_bind_socket : 896
18 // the bug is triggered in rt_serial_write, line 917,
       where finally calls _serial_poll_tx
19 rt_inline int _serial_poll_tx(struct rt_serial_device
       *serial, ...) {
20   RT_ASSERT(serial != RT_NULL);
21   ...
22   if (*data == '\n' && (serial->parent.open_flag &
        RT_DEVICE_FLAG_STREAM)) {
23       serial->ops->putc(serial, '\r');
24   }
25 }
```

Figure 6. A previously unknown bug in Rt-Thread.

In this case, EOF tries to test Rt-Thread's socket-related API (lines 3-8), where it tries to create a socket instance. The `socket()` call uses valid arguments (line 2). According to the backtrace provided by EOF (lines 9-17), the error is located in function `rt_serial_write()` (line 11). In detail, during socket creation, the system attempts to print a log message over the serial port (lines 11–14), which eventually invokes `rt_serial_write()`. Inside `rt_serial_write()`, the routine forwards the `serial` pointer to `_serial_poll_tx()`. Although `serial` is non-NULL, it is stale (dangling after an unregister or incomplete init), so the `RT_ASSERT()` does not trigger (line 20). Subsequent indirect accesses dereference corrupted fields (lines 22-23), causing a bus/usage fault. The exception leaves the system unresponsive; EOF detects and records the crash when the fault propagates to the OS's exception handler. EOF exposes this bug because (1) its specification allows full-system testing of embedded OS, including the networking stack; (2) its coverage-guided generator reaches the serial-logging path during socket creation; and (3) its crash monitor recognizes the fault signature and attributes it via the captured backtrace.

5.4 Comparison with Existing Fuzzers

To answer **RQ3** and evaluate the effectiveness of EOF, we compare the code coverage achieved by EOF with other methods, including Tardis, GDBFuzz, and EOF-nf. By comparing code coverage metrics, we demonstrate EOF's performance across diverse hardware platforms and systems, highlighting its ability to conduct comprehensive full-system testing, including both application and system-level operations on both hardware and emulation environments.

5.4.1 Full-system Fuzzing Analysis. To demonstrate EOF's full-system testing capability, we compared EOF with Tardis and EOF-nf (without feedback guidance) with the same input payloads.

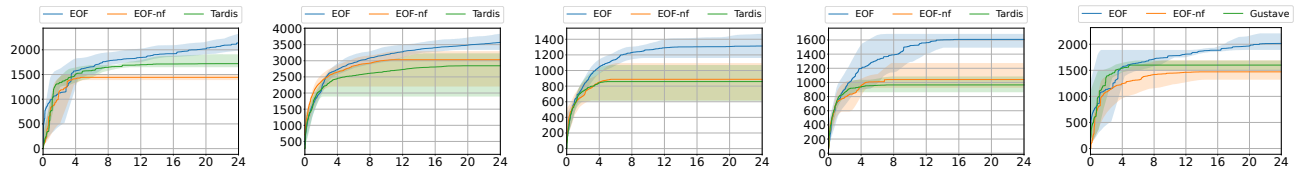
Coverage Statistics. We first plot the coverage growth curve for 24 hours, as shown in Figure 7. Comparing with EOF-nf, the improvement achieved by EOF shows the effectiveness of the coverage guidance, where EOF can identify which payloads trigger new code coverage and provide them with a higher chance of mutation, thereby further explore the embedded OS's code space, as we can refer from 7(e) and 7(b) that both EOF and EOF-nf can reach a high coverage in the first 12 hours, while EOF-nf reach a certain saturation status, and EOF can keep growing slowly.

Table 3. Coverage comparison between EOF, EOF-nf, Tardis, and Gustave on five embedded OSs. The base unit is the average number of branches found by each fuzzer, and parentheses indicate EOF's improvement.

Target OSs	EOF	EOF-nf	Tardis	Gustave
NuttX	2139.0	1719.4 (+24.40%)	1442.6 (+48.27%)	–
Rt-Thread	3572.2	2844.6 (+25.58%)	3031.8 (+17.82%)	–
Zephyr	1313.4	858.2 (+53.04%)	887.4 (+48.00%)	–
FreeRTOS	1608.4	965.0 (+66.67%)	1040.4 (+54.59%)	–
PoKOS	2015.8	1470.6 (+37.07%)	–	1600.2 (+25.97%)

The detailed coverage statistics are given in Table 3. Compared with Tardis, EOF improves coverage by 48.27% on NuttX, 17.82% on RT Thread, 48.00% on Zephyr, and 54.59% on FreeRTOS. Compared with the no feedback variant, improvements are 24.40% on NuttX, 25.58% on RT Thread, 53.04% on Zephyr, 66.67% on FreeRTOS, and 37.07% on PoKOS. On PoKOS, where Tardis is unavailable, EOF exceeds Gustave by 25.97%. These gains come from two aspects. First, EOF leverages LLM to help generate API specifications, which can cover more system functionalities. Second, unified feedback from coverage, exceptions, and logs guides exploration toward new paths.

Bug Detection. The EOF-nf variant detects 11 bugs (bugs #1-5, 8-9, 13, 15, 18-19) as listed in Table 2, while Tardis can detect 6 bugs (bugs #3-5, 8, 18, 15). This improvement can be attributed to EOF's liveness watchdog and bug monitors, which help it to identify system crashes and hangs. Unlike Tardis, which solely relies on the timeout mechanism as the



(a) Coverage on NuttX (b) Coverage on Rt-Thread (c) Coverage on Zephyr (d) Coverage on FreeRTOS (e) Coverage on PokOS

Figure 7. Coverage comparison on four embedded OS. The x-axis shows elapsed testing time (hours), and the y-axis shows branch coverage count, where the shaded area indicates the maximum and minimum coverage achieved by each tool.

bug monitor and liveness checker, the diverse watchdog and bug monitors mechanism helps EOF not only maintain the embedded OS's runtime state and ensure fuzzing efficiency but also effectively identify potential bugs. In other words, even if Tardis can generate a test case that triggers such an error, it cannot identify the bug.

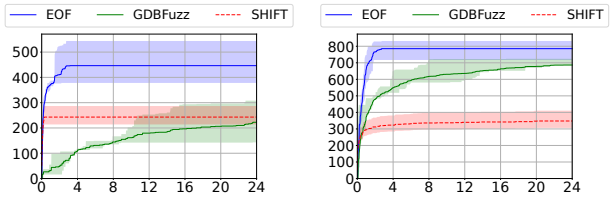
Table 4. Coverage comparison of EOF and GDBFuzz on HTTP Server and JSON running on hardware. The base unit is the average number of branches found by each fuzzer, and parentheses indicate EOF's improvement.

Fuzzers	HTTP Server	JSON	Average
EOF	446.4	785.4	615.9
GDBFuzz	222.4 (+100.01%)	686.6 (+14.39%)	454.5 (+35.51%)
SHIFT	246.2 (+81.13%)	348.8 (+125.17%)	297.5 (+107.03%)

5.4.2 Application-Level Fuzzing Analysis. To further demonstrate the fuzzing performance of EOF, we compared the coverage achieved by EOF and GDBFuzz, running on the ESP32 development board. Also, since GDBFuzz only supports application-level testing, here we chose http_server and JSON interface as the testing target, and our instrumentation is strictly confined to these two modules.

The overall coverage statistics are demonstrated in Table 4. In detail, EOF finds an average of 446.4 and 785.4 branches on the http_server and JSON modules, and gains 35.51% and 107.03% branches compared with GDBFuzz and SHIFT on average. The overall coverage growth curve for 24 hours is depicted in Figure 8.

As we can see from these two figures, EOF can achieve higher code coverage with faster speed. Such improvement is primarily attributed to two factors. First, EOF's bug monitors turn OS-level signals, such as exceptions and error logs into actionable feedback. In detail, when a payload triggers a panic or invariant violation, EOF gives such a payload higher weights for mutations, which yields a more interesting payload that progresses deeper. Second, unlike GDBFuzz and SHIFT, which follow the AFL-style design that sends randomly generated data buffers, EOF performs API-aware, dependency-guided generation. From specifications with typed arguments and explicit constraints, EOF builds API



(a) Coverage on http_server (b) Coverage on json

Figure 8. Coverage comparison on EOF and GDBFuzz. The x-axis shows elapsed testing time (hours), and the y-axis shows branch coverage count.

call sequences and orders them by resource production/-consumption, ensuring preconditions are met before deeper handlers execute. This allows EOF to conduct in-depth testing on the target embedded OS. Also, we notice that both EOF and EOF-nf stop growing after the first four hours, which we attribute to the limited exposed and API specifications by these modules.

5.5 Instrumentation Overhead

Since EOF needs to run the target OS directly on hardware, it is important to ensure the memory and execution efficiency. Therefore, to address **RQ4** and assess the impact of instrumentation on the system's overall performance, we collect the extra memory overhead and the execution overhead before and after the instrumentation.

5.5.1 Memory Overhead. We evaluated the binary sizes of the target operating systems, observing an increase in size due to instrumentation. For NuttX, the binary size increased by 4.76%, showing a change from 3.36 MB to 3.52 MB. Rt-Thread's size increased from 2.53 MB to 2.71 MB (7.11%), Zephyr from 0.803 MB to 0.88 MB (9.58%), and FreeRTOS from 2.825 MB to 2.947 MB (4.32%). On average, the increase was around 6.44%. This inflation is primarily due to coverage instrumentation, which adds space for callback functions and the necessary memory for coverage collection, although OS codebase sizes vary, all figures are reported with the same number of significant digits.

5.5.2 Execution Overhead. In terms of execution speed, we measured the total payloads (API sequences) executed within 10 minutes for each target OS. NuttX showed a decrease from 1070 to 740.2 payloads, marking an overhead of 30.82%. Rt-Thread executed 721.0 payloads after instrumentation, down from 858.2, resulting in a 15.99% overhead. Zephyr's payload count decreased from 1308 to 989.8 (24.32%), and FreeRTOS saw a reduction from 1580.4 to 1193.6 (24.44%). On average, the execution overhead was about 23.39%. This overhead remains within acceptable limits, considering KCOV can introduce around 10% overhead, and AFL, which may slow down applications by 2x to 5x. Also, consider the enhanced coverage and bug-detection capabilities provided by the instrumentation. This trade-off between performance and improved testing efficacy highlights the value of instrumentation in fuzzing environments.

6 Threats to Validity

Failure Handling on Hardware Levels. EOF's current failure detection and restoration mechanisms, primarily designed to address software-level failures, may not be sufficient for addressing hardware faults, such as memory failures. Hardware faults can occur unpredictably and may propagate errors that software mechanisms are not equipped to handle, potentially leading to data corruption or system instability. In the future, EOF can integrate more hardware debug mechanisms and implement additional redundancy strategies, such as error-correcting memory and hardware watchdogs, which can detect and mitigate hardware-related issues. Additionally, enhancing EOF's restoration can include hardware fault tolerance measures that will ensure better system reliability.

Limitations in More Comprehensive Bug Detectors. Currently, EOF primarily detects explicit errors, such as kernel panics and assertion errors, which are straightforward to identify because they have clear system signals and definable system states. However, for more subtle errors that do not result in immediate system crashes but may nonetheless significantly impact system performance and security, we cannot currently uncover them effectively. These include timing issues that can lead to race conditions or silent memory corruption, such as use after free and double free. In the future, we can adapt more diverse bug detectors, such as address sanitizer and thread sanitizer, to the embedded OS domain. These enhanced detectors can analyze detailed memory and thread operations to identify subtle system faults and can extend EOF's detection capabilities. Also, we can leverage hardware signals, such as power consumption, to spot spikes/plateaus that indicate liveness issues, and JTAG sampling to catch tight loops or repeated memory accesses. These signals can inform EOF to stop unproductive runs and reset quickly.

Limitations in Quality of Input Payload. To adapt EOF to different embedded OSs with acceptable overhead, we currently generate input payloads with LLM assistance from embedded OS documentation and API usage; while flexible, this can yield suboptimal cases such as API misuse and meaningless arguments, which reduce effectiveness. Also, currently EOF does not exercise interrupt handlers or low-level I/O, which would require hardware event injection such as GPIO toggles or serial input. In future work, we can augment the corpus with real firmware and test traces that exhibit correct API usage and well-formed arguments. We can also apply lightweight validation, such as type and range checks, and handle lifecycle checks to prune low-quality payloads. Additionally, we can introduce lightweight peripheral models to drive interrupt paths and I/O error handling.

Lessons learned. We summarize two lessons learned during the design of EOF. First, defining detectable events is important. Embedded systems running on isolated hardware can fail silently or enter degraded states, making it hard to detect failures. Practically define detectable events, such as exceptions and log messages, that can be monitored and acted upon to maintain system liveness and identify bugs. Second, corpus quality affects testing performance. Low-quality inputs with invalid API usage or meaningless arguments waste cycles and rarely reach deep kernel paths; in our evaluation, we found AFL-style inputs struggled to satisfy API preconditions and make progress. To address this, we built a curated corpus from LLM-derived API specifications and pseudo functions, thereby improving testing validity and efficiency.

7 Conclusion

In this paper, we introduce EOF, an embedded OS fuzzer that performs feedback-guided testing directly on hardware platforms. Traditional methods either rely on emulation or cannot exercise full system paths on real boards. EOF uses the hardware debug interface as the control and observation channel between the host and target board, without depending on embedded OS services. Over this link, EOF deploys an agent to execute API-sequences-based test cases, collects runtime coverage, and monitors system crashes. System liveness is maintained with watchdogs and fast restoration so that the fuzzing process can continue after crashes or stalls. We adapted EOF to four embedded OSs, discovering 19 bugs with 5 confirmed. Compared with existing fuzzers, EOF increased code coverage by 50.84%, with an average of 6.44% and 23.39% memory and execution overhead.

8 Acknowledgments

We thank the shepherd and reviewers for their valuable comments. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62525207,62472448,62202500).

References

[1] Peter BENSCH Anuj. STM32H745. https://community.st.com/t5/stm32-mcus-products/emulator-for-stm32h745/td-p/649287, 2024.

[2] Apache Software Foundation. Apache NuttX: A POSIX-Compliant Real-Time Operating System. https://github.com/apache/nuttx, 2024. Written in C, C++, assembly; Real-time microkernel; Initial release: 2007; Latest release: 12.5.1 (April 15, 2024).

[3] Armis. URGENT/11: Exposing Security Flaws in the TCP/IP Stack. https://www.armis.com/research/urgent-11/, 2019. Accessed: 2024-09-04.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[5] Richard Barry et al. FreeRTOS. *Internet, Oct*, 4:18, 2008.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Clang Project. Sanitizer Coverage, 2024. Accessed: 2024-09-13.

[8] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Stéphane Duverger and Anaïs Gantet. GUSTAVE: Fuzz it like it's app. *DMU Cyber Week*, 2021.

[10] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing Embedded Systems using Debug Interfaces. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1031–1042, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1237–1254. USENIX Association, August 2020.

[12] Google. Kernel Address Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[13] Hubert Högl and Dominic Rath. Open On-Chip Debugger. *Fakultät für Informatik, Tech. Rep*, 2006.

[14] Hsin-Wei Hung and Ardalan Amiri Sani. BRF: Fuzzing the eBPF Runtime. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024.

[15] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding Bugs in File Systems with an Extensible Fuzzing Framework. *ACM Trans. Storage*, 16(2), may 2020.

[16] lcamtuf. American Fuzzy Lop, 2013. https://lcamtuf.coredump.cx/afl/.

[17] Jianzhong Liu, Yuheng Shen, Yiru Xu, and Yu Jiang. Leveraging binary coverage for effective generation guidance in kernel fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3763–3777, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. Horus: Accelerating Kernel Fuzzing through Efficient Host-VM Memory Access Procedures. *ACM Trans. Softw. Eng. Methodol.*, 33(1), nov 2023.

[19] Siti-Farhana Lokman, Abu Talib Othman, and Muhammad-Husaini Abu-Bakar. Intrusion Detection System for Automotive Controller Area Network (CAN) Bus System: A Review. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):1–17, 2019.

[20] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1949–1966, USA, 2019. USENIX Association.

[21] Alejandro Mera, Changming Liu, Ruimin Sun, Engin Kirda, and Long Lu. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5323–5340, Philadelphia, PA, August 2024. USENIX Association.

[22] Anas Nashif. Zephyr is a new generation, scalable, optimized, secure RTOS, 2016. https://github.com/zephyrproject-rtos/zephyr.

[23] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.

[24] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2197–2213, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Sergej Schumilo, Cornelius Aschermann, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

[26] Yuheng Shen, Shijun Chen, Jianzhong Liu, Yiru Xu, Qiang Zhang, Runzhe Wang, Heyuan Shi, and Yu Jiang. Brief Industry Paper: Directed Kernel Fuzz Testing on Real-time Linux. In *2023 IEEE Real-Time Systems Symposium (RTSS)*, pages 495–499. IEEE, 2023.

[27] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. Rtkaller: State-Aware Task Generation for RTOS Fuzzing. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[28] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. Tardis: Coverage-Guided Embedded Operating System Fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4563–4574, 2022.

[29] Hao Sun and Zhendong Su. Validating the eBPF Verifier via State Embedding. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 615–628, Santa Clara, CA, July 2024. USENIX Association.

[30] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding correctness bugs in eBPF verifier with structured and sanitized program. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 689–703, New York, NY, USA, 2024. Association for Computing Machinery.

[31] The Linux Kernel Organization. KCOV: Kernel Code Coverage. https://docs.kernel.org/dev-tools/kcov.html.

[32] Dmitry Vyukov. Syzkaller: an unsupervised coverage-guided kernel fuzzer, 2015. https://github.com/google/syzkaller.

[33] Dmitry Vyukov and Andrey Konovalov. Pseudo-syscalls, 2015. https://github.com/google/syzkaller/blob/master/docs/pseudo_syscalls.md.

[34] Dmitry Vyukov and Andrey Konovalov. Syzlang: System Call Description Language, 2015. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[35] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Jingzhou Fu, Zhuo Su, Qing Liao, Bin Gu, Bodong Wu, and Yu Jiang. Data Coverage for Guided Fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2511–2526, Philadelphia, PA, August 2024. USENIX Association.

[36] Trey Woodlief, Sebastian Elbaum, and Kevin Sullivan. Fuzzing Mobile Robot Environments for Fast Automated Crash Detection. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5417–5423, 2021.

[37] Bernard Xiong and Man Jianting. RT-Thread is an open source IoT operating system., 2007. https://github.com/RT-Thread/rt-thread.

[38] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, Baltimore, MD, August 2018. USENIX Association.