Computers & Security 170 (2026) 105003

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

STAFF: Stateful Taint-Assisted Full-system Firmware Fuzzing

Alessio Izzillo [a,b], Riccardo Lazzeretti [a], Emilio Coppa [b,*]

[a] Sapienza University of Rome, Italy
[b] LUISS University, Italy

ARTICLE INFO

Keywords:
Firmware fuzzing
Embedded linux security
Stateful protocol fuzzing
Whole-system taint analysis
Coverage-guided fuzzing

ABSTRACT

Modern embedded Linux devices, such as routers, IP cameras, and IoT gateways, rely on complex software stacks where numerous daemons interact to provide services. Testing these devices is crucial from a security perspective since vendors often use custom closed- or open-source software without documenting releases and patches. Recent coverage-guided fuzzing solutions primarily test individual processes, ignoring deep dependencies between daemons and their persistent internal state.

This article presents STAFF, a firmware fuzzing framework for discovering bugs in Linux-based MIPS firmware with HTTP-managed workflowsuilt around three key ideas: (a) *user-driven multi-request recording*, which monitors HTTPuser interactions with emulated firmware to capture request sequences;(b) *intra- and inter-process dependency detection*, which uses whole-system taint analysis to track how input bytes influence user-space states, including CPU registers, memory accesses, IPC channels, and filesystem operations;(c) *protocol-aware taint-guided fuzzing*, which applies mutations to request sequences based on identified dependencies, exploiting multi-staged forkservers to efficiently checkpoint protocol states.

When evaluating STAFF on 15 Linux-based MIPSirmware targets, it identifies 42 crashes in 9 firmware imagesnvolving multiple network requests and different firmware daemons. Through systematic crash triage, these crashes consolidate into 20 previously unknownugs submitted for disclosure9 discovered exclusively by STAFF) and 4 previously disclosed CVEs (2 found only by STAFF), significantlyutperforming the considered full-system multi-processaselinefirmware fuzzing solutions in both the number and reproducibility of discovered crashes

1. Introduction

Internet of Things devices are increasingly integrated into daily life across essential domains including healthcare, smart cities, transportation, and agriculture (Organisation, 2022). This has opened the market to numerous manufacturers, each adopting custom embedded software that typically combines closed, proprietary code with open source components. Such embedded software is usually developed according to two primary architectural approaches (Holt and Huang, 2014). Monolithic implementations integrate all components into a single program running directly on hardware, offering efficient resource utilization but limited flexibility. Alternatively, systems can be built on general-purpose operating systems, such as Linux, which provide standardized interfaces and greater development flexibility, though with additional overhead. Several modern IoT devices often favor the latter architectural approach, increasingly adopting general-purpose operating systems (Research, 2024). However, these systems often include dozens or hundreds of interdependent binaries and configuration files from diverse sources, significantly expanding the attack surface and making devices particularly vulnerable to software flaws and security breaches (Redini et al., 2020; Binosi et al., 2024). Indeed, recent real-world exploitation campaigns demonstrate the ongoing severity of firmware vulnerabilities and the importance of understanding inter-binaries dependencies. In 2023, CVE-2023-1389, affecting TP-Link Archer routers, enabled unauthenticated attackers to execute arbitrary commands through inadequate validation logic distributed across the web server and CGI handler (Lin and Li, 2024). Similarly, CVE-2024-7261 in Zyxel access points and routers allowed command execution via manipulated cookies sent to CGI programs lacking proper parameter validation (Networks, 2024). These vulnerabilities exemplify how validation logic distributed across multiple cooperating daemons, typically involving web servers, CGI handlers, and backend configuration processes, may create attack surfaces when inter-binary dependencies are not adequately understood during security analysis.

Identifying and prioritizing vulnerabilities — via firmware analysis — is a necessary precondition to guarantee system security. Direct testing of physical devices provides the most accurate results; however,

* Corresponding author.
E-mail addresses: izzillo@diag.uniroma1.it (A. Izzillo), lazzeretti@diag.uniroma1.it (R. Lazzeretti), ecoppa@luiss.it (E. Coppa).

https://doi.org/10.1016/j.cose.2026.105003
Received 22 September 2025; Received in revised form 20 May 2026; Accepted 31 May 2026
Available online 6 June 2026
0167-4048/© 2026 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

it presents three main critical challenges (Broekman and Notenboom, 2003). First, devices must be acquired, making it difficult to test a large variety of devices. Second, runtime inspection capabilities during execution are limited since most manufacturers disable hardware debugging interfaces (e.g., JTAG), making it difficult to exploit non-black-box analysis techniques. Third, physical hardware makes it challenging to perform parallel experiments on the same devices without acquiring multiple units. For these reasons, the research community has explored approaches that emulate firmware images, which can be obtained directly from vendors (D-Link, 2025; TP-Link, 2025), retrieved during the update procedure, or manually extracted from device memory storage.

Emulation of firmware images enables scalable static and dynamic analysis but introduces its own challenges, as accurate rehosting[1] remains an open research problem (Wright et al., 2021). Nonetheless, on one hand, several works (Zhou et al., 2021) explore how to automatically model unknown peripherals and other vendor-specific hardware components. On the other hand, various rehosting frameworks (Maier et al., 2020; Hernandez et al., 2022; Produit et al., 2025) have focused their attention on specific devices and have shown how to effectively emulate their firmware images. In this article, we focus on Linux-based systems that can be adequately emulated with state-of-the-art rehosting frameworks (Liu et al., 2022; Tay et al., 2023). In particular, consistent with prior related work (Plc, 2017; Zheng et al., 2019, 2022), we primarily consider Linux-based routers and IP cameras.

One software testing technique exploited in several state-of-the-art firmware testing works (Yun et al., 2022; Feng et al., 2022; Zhang et al., 2024; Asmita et al., 2025) is software fuzzing (Zalewski, 2021; Zeller et al., 2019; Manès et al., 2019). This technique builds on the idea of testing the software a large number of times using different randomly mutated benign inputs. To evaluate the utility of each mutated input, fuzzing checks for crashes and, in their absence, exploits code coverage to understand whether a mutated input induced a different code execution path and thus could be a good candidate for further mutations. Projects such as OSS-Fuzz (Serebryany, 2017; Google, 2023) from Google have demonstrated the potential of software fuzzing by finding thousands of bugs and vulnerabilities in prominent and complex open-source projects.

A major limitation of state-of-the-art work in firmware fuzzing is their limited analysis scope, as they do not consider firmware as integrated multi-process ecosystems where vulnerabilities arise from interactions between cooperating services. Furthermore, they see the process as a stateless application. However, a large number of devices are built atop stripped-down Linux distributions and expose a rich and complex surface of network-accessible services (Tekeoglu and Tosun, 2016). These include standardized protocols such as HTTP, DHCP, Telnet, FTP, UPnP, and SNMP, as well as vendor-specific APIs for configuration, telemetry, and remote management. The daemons behind these protocols are stateful, keeping track of their internal state across the expected sequences of requests. Moreover, vendors build custom application protocols on top of these standardized protocols: for instance, as exemplified by Fig. 1, a device may exploit HTTP as a bridge service that controls and configures multiple backend daemons, where web-based management interfaces trigger cascading interactions across configuration managers, database services, and hardware control processes. Such custom application protocols are not documented, with the exception of loose descriptions within user manuals, and involve several processes with internal states kept in memory or on the disk.

The tight integration of stateful protocols, long-lived user sessions, and persistent inter-process state makes systematic security testing of firmware images a significant challenge. Traditional greybox fuzzers, such as AFL (Zalewski, 2021), which have achieved significant success on traditional platforms, operate on single, stateless applications

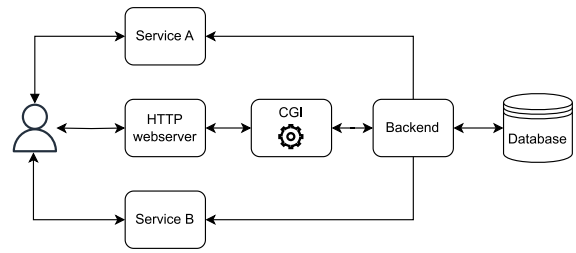


Fig. 1. Example of a firmware exposing an HTTP webserver that internally interacts with a backend in charge of controlling other network services (e.g., DHCP and UPnP).

and cannot model inter-request dependencies. Representative firmware fuzzing approaches, including TriforceAFL (Plc, 2017), FirmAFL (Zheng et al., 2019), and EQUAFL (Zheng et al., 2022) have primarily ported traditional fuzzers, focusing on the optimizations needed to amortize the substantial overhead introduced by emulation. Prior work has considered applications implementing standard network protocols by exploring protocol-aware fuzzers. In particular, proposals such as AFLNet (Pham et al., 2020) and StateAFL (Natella, 2022), and SG-Fuzz (Ba et al., 2022) extend AFL's state model to cover common network protocols, inferring protocol states from server responses, memory snapshots, or enumerated-type state variables respectively, but still focus on one daemon at a time and rely heavily on heuristics to infer valid transitions across internal states. Moreover, they target applications on traditional platforms and have not been ported to the context of firmware testing, where emulation overhead significantly impacts the execution cost during fuzzing. Similarly, other fuzzers, such as VUzzer (Rawat et al., 2017), Angora (Chen and Chen, 2018), WEIZZ (Fioraldi et al., 2020a), and RedQueen (Aschermann et al., 2019), investigate how to exploit taint analysis, or lighter approximations of this technique, to understand how input bytes affect application execution. However, they target single applications. Full-system taint analysis has been explored in works such as DECAF++ (Davanian et al., 2019) but has not been extensively used in the context of firmware fuzzing.

Our contribution. In this article, we introduce STAFF, *Stateful Taint-Assisted Full-system Fuzzing*, a semi-automated fuzzing framework aimed at Linux-based MIPS firmware images exposing HTTP-based services that control and interact with multiple cooperating daemons. STAFF encompasses three key stages: (a) *User-driven multi-request recording* (Section 4.2), which closely monitors user interactions with the emulated firmware to capture rich but minimal request sequences involving an HTTP-based application-layer service that controls and interacts with a potentially large number of daemons. Prior work often undermines the importance and availability of meaningful request sequences. While STAFF cannot make this phase fully automatic, it at least makes it a piece of the puzzle, providing support to users interested in fuzzing a Linux-based MIPS firmware image. (b) *Intra- and inter-service dependency detection* (Section 4.3), which exploits whole-system taint analysis to track how input bytes influence user-space states, including files, sockets, and memory regions, while keeping track of the request sequences possibly needed to reach interesting execution states. (c) *Protocol-aware taint-guided full-system fuzzing* (Section 4.4), which exploits the dependencies identified during whole-system taint analysis to effectively devise mutations to request sequences during fuzzing. To make the testing process more efficient, STAFF implements Multi-Staged Forkserver to efficiently checkpoint complex protocol states.

The semi-automated nature stems from stage (a), which requires a user analyst to record an initial seed corpus via manual interaction with the emulated firmware; while stages (b) and (c) are fully automated.

In more detail, our contributions can be summarized as follows:

[1] Rehosting refers to running software on a different platform than the one originally intended by the vendor.

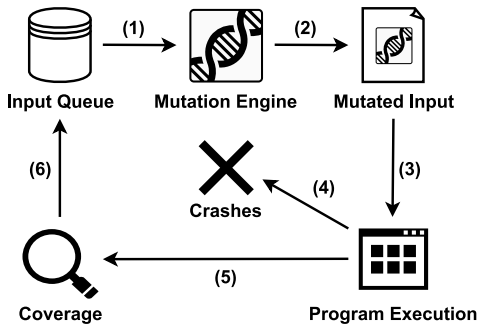


Fig. 2. Workflow of a coverage-guided fuzzer.

- We review foundational concepts in fuzzing and dynamic firmware analysis relevant to understanding STAFF's contributions, explaining the key ideas behind them (Section 2) while also pinpointing their limitations when put in perspective on a real-world case study (Section 3).
- We introduce our firmware fuzzing framework STAFF (Section 4), which encompasses the three key stages outlined above: (a) *User-driven multi-request recording*; (b) *Intra- and inter-service dependency detection*; (c) *Protocol-aware taint-guided full-system fuzzing.*
- We describe the essential implementation details behind our framework (Section 5). STAFF extends and improves DECAF++ for the whole-system taint analysis. We also present several optimizations required to make STAFF more effective when considering real-world Linux-based MIPS firmware images.
- We discuss the results of an experimental evaluation (Section 6) where STAFF is used on 15 Linux-based MIPS firmware images, comparing its effectiveness with system-mode TriforceAFL and system-mode AFLNet†.[2] STAFF identifies 42 crashes involving multiple network requests and different firmware daemons, which through systematic crash triage consolidate into 20 previously unknown bugs submitted for disclosure (9 discovered exclusively by STAFF) and 4 previously disclosed CVEs (2 found only by STAFF), significantly outperforming representative full-system multi-process firmware fuzzing solutions in both the number and reproducibility of discovered crashes.

To encourage further research, we make[3] our contributions available at https://github.com/alessioizzillo/STAFF.

2. Background

In this section, we provide the key ideas behind firmware fuzzing, presenting relevant work that we consider throughout the article.

2.1. Software fuzzing

Software fuzzing is a dynamic testing technique (Manès et al., 2019; Godefroid, 2020; Zeller et al., 2019) that generates randomly mutated inputs and fed them to an application with the ultimate goal of finding bugs and vulnerabilities. In this article, we consider coverage-guided fuzzing, a flavor that has gained widespread adoption through popular fuzzing frameworks such as AFL (American Fuzzy Lop) (Zalewski, 2021) and libFuzzer (LLVM-project, 2023), both recognized for their effectiveness in large-scale vulnerability discovery (Serebryany, 2017; Google, 2023). Over time, AFL has evolved through community-driven development: AFL++ (Fioraldi et al., 2020b), an actively maintained rewrite of AFL, extends the original design with numerous enhancements. While we primarily reference AFL throughout this paper, as it forms the foundational architecture inherited by our baseline tools (TriforceAFL and AFLNet), the core conceptual framework we discuss remains largely consistent across both AFL and AFL++.

The main steps of a coverage-guided fuzzer, as illustrated in Fig. 2, can be outlined as follows:

1. *Input Selection:* The fuzzer selects a test case from the *input queue*, a prioritized collection of candidate inputs. This queue is initially seeded with inputs, dubbed *seed corpus*, provided manually by the user or extracted from existing data samples. The selection process may incorporate various heuristics to prioritize inputs that are likely to exercise unexplored code paths or trigger new program behaviors.

2. *Input Mutation:* The selected input is subjected to one or more mutation strategies to produce a new test case. These mutations are typically lightweight and computationally inexpensive operations such as bit flipping, arithmetic adjustments, insertion of random byte sequences, injection of semantically meaningful constants (e.g., platform-specific magic numbers or dictionary tokens), or recombination with other inputs in the queue. The mutation scheme and intensity are often stochastically determined to maintain diversity in the generated inputs.

3. *Program Execution and Monitoring:* The target software is executed with the mutated input. The fuzzer actively monitors the execution for anomalous behaviors, including crashes, timeouts, or hangs. Meanwhile, code coverage information is collected to evaluate the input impact on the application behavior. This instrumentation can be achieved either statically or dynamically using different techniques. Static instrumentation includes both compile-time instrumentation, when source code is available, and static binary rewriting (Duck et al., 2020; Dinesh et al., 2020; Nagy et al., 2021; Di Bartolomeo et al., 2023), which directly transforms executable binaries to inject coverage or tracing logic without requiring source code. Dynamic instrumentation, in contrast, relies on runtime techniques such as dynamic binary translation (e.g., via QEMU (Bellard, 2005)) to insert monitoring logic on the fly, offering broad applicability at the cost of higher execution overhead.

4. *Crash Handling and Deduplication:* If the execution results in a crash, the corresponding input is preserved in a dedicated *crash queue*. To avoid redundant inputs, a crash deduplication mechanism is employed to identify and discard semantically equivalent failures.

5. *Code Coverage Feedback:* Inputs that do not cause crashes may still contribute if they increase the overall code coverage. A coverage-guided fuzzer leverages fine-grained execution metrics to determine whether an input has led to the exploration of previously unexecuted control flow paths or exercised rare branches. Such inputs are considered *interesting*.

6. *Queue Augmentation:* If an input is deemed interesting based on its contribution to program coverage or behavioral diversity, it is reinserted into the input queue for further mutation in future fuzzing iterations.

This iterative *mutate-and-execute* paradigm is repeated at scale, often millions or billions of times, enabling the fuzzer to autonomously discover deep vulnerabilities and edge-case behaviors even in complex applications.

[2] We ported the ideas behind AFLNet into a multi-process firmware fuzzing framework dubbed AFLNet†.

[3] We plan to release the code after acceptance.

2.2. Firmware fuzzing

Given the impressive results obtained by coverage-guided fuzzing on traditional platforms, researchers have naturally explored these techniques for firmware analysis (Gao et al., 2022; Shen et al., 2022; Coppa et al., 2023). Two primary approaches have emerged for fuzzing firmware images, each presenting distinct trade-offs between accuracy and effectiveness.

The first approach (Plc, 2017) employs full-system emulation, where the entire firmware image runs in an emulated environment while code coverage is monitored either system-wide or for specific processes when targeting individual applications. While this strategy provides high fidelity by preserving the complete execution context, it suffers from significant execution overhead. System emulation is slow, substantially limiting the fuzzing throughput and potentially compromising the overall effectiveness of the testing process.

The second approach, adopted by several fuzzing frameworks (Zheng et al., 2019, 2022), attempts to overcome these performance limitations through a hybrid emulation strategy. These tools initially use system emulation to bootstrap the firmware into a stable running state, then migrate target applications to faster user-space emulators. This migration can dramatically increase execution speed and fuzzing attempts. However, the approach requires switching back to system-mode emulation whenever the application interacts with the kernel through system calls, introducing both complexity and overhead.

This hybrid strategy faces several fundamental limitations. The performance benefits diminish as kernel interactions increase, since the costly synchronization and context switching between emulation modes becomes a bottleneck. More critically, this approach is fundamentally designed for isolated application testing, which can poorly match the reality of modern firmware architectures. Modern firmware images typically feature complex ecosystems of interdependent daemons that communicate through various inter-process communication mechanisms. User-space emulation struggles to accurately capture these inter-service dependencies, potentially missing critical bugs that arise from component interactions rather than isolated application logic.

In this paper, our goal is to simultaneously test multiple cooperating services within a firmware. To achieve this goal, we adopt fuzzing strategies based on system-mode emulation, which enables comprehensive analysis of inter-service interactions. Due to the huge overhead from the emulation, which can drastically reduce the number of fuzzing attempts, it is pivotal to generate inputs based on targeted mutations guided by identified intra- and inter-process dependencies, thus hopefully maximizing the chances of exploring interesting behaviors across different processes.

2.3. Protocol-aware fuzzing

Different works have considered fuzzing of applications implementing network protocols. A prominent example is AFLNet (Pham et al., 2020), which extends the popular AFL (Zalewski, 2021) in order to cope with sequences of requests instead of handling the application input as a single monolithic file. We now presents the main key ideas behind AFLNet since they are relevant for the remainder of this article.

Input as ordered requests. For applications implementing network protocols, a valid input interaction may depend not only on a single request but on ordered sequences of requests. While classic AFL treats inputs as a single, monolithic file, AFLNet interprets them as multi-step session governed by an implicit protocol state machine. For instance, consider the requests:

❶ *Initial page request*:

```
GET /index.html HTTP/1.1
Host: example.com
```

❷ *Login request*:

```
POST /login HTTP/1.1
Host: example.com
Content-Length: 30

username=admin&password=admin
```

❸ *Authenticated dashboard request*:

```
GET /admin/dashboard HTTP/1.1
Host: example.com
```

Each HTTP request (`GET`, `POST`, etc.) is treated as a distinct *request*. AFLNet extracts these requests from captured network traces (e.g., PCAP files) and mutates them individually while preserving their order. This notion of *protocol-aware mutation* allows AFLNet to maintain valid session structures: instead of corrupting random bytes, it mutates meaningful protocol elements, such as HTTP methods (`GET` → `PUT`), URIs, or header values. Through protocol-aware mutation, AFLNet ensures that mutated requests are still likely to pass initial protocol checks, increasing the chances of reaching deeper code paths such as login logic or administrative endpoints.

Implicit protocol state machine (IPSM). While mutating sequences, AFLNet observes the server responses, typically in the form of HTTP status codes (e.g., `200 OK` or `401 Unauthorized`). By correlating these codes with request sequences, AFLNet dynamically builds an *Intermediate Protocol State Machine*, where states correspond to unique server response codes or combinations of codes, while transitions represent HTTP requests that move the server from one state to another. For example:

$$Initial \xrightarrow{\texttt{GET/index}} Unauthenticated \xrightarrow{\texttt{POST/login}} Authenticated$$

This protocol state machine is inferred dynamically, without requiring a formal protocol specification. As AFLNet explores more sequences, the protocol state machine becomes richer, capturing both expected and unexpected states (e.g., unexpected error responses like `500 Internal Server Error`).

Target state selection. Not all protocol states are equally valuable for exploration. AFLNet introduces target state selection heuristics to focus on the most promising parts of the protocol state machine. These heuristics consider: rarity, states that are reached infrequently, states historically linked to new code paths or crashes. Once a target state is selected, AFLNet chooses sequences from the corpus that reliably reach it. This ensures that mutations start from a meaningful context, such as an authenticated session, rather than always from the initial unauthenticated handshake.

State-aware mutations. Let $M = \langle M_1, M_2, M_3 \rangle$ denote a sequence of requests decomposed into a prefix M_1 (one or more requests that reach that chosen state s), the target M_2 (one or more intermediate requests targeted for mutation), and a suffix M_3 (remaining request(s) that may observe or materialize the effects of M_2). The state-aware strategy mutates only the target M_2, producing:

$$M' = \langle M_1, \text{mutate}(M_2), M_3 \rangle.$$

Preserving M_1 maintains the transition path to the target state so that the mutated target is exercised in the correct protocol context. Retaining M_3 ensures any downstream consumers, committers, or cleanup handlers that observe the mutated data still execute and can therefore expose faults that only manifest after the mutation is materialized. For example:

- **Prefix** M_1: `GET /index` followed by `POST /login`, which reliably establishes an authenticated state.

- **Target** M_2: Configuration change requests, e.g., `POST /lan_settings.cgi` with parameters for IP address, netmask, or DHCP range. These intermediate requests are mutated by the fuzzer to exercise parsing and validation logic.
- **Suffix** M_3: Commit or apply request, e.g., `POST /apply.cgi` or `POST /xmldb.php`, which finalizes the configuration changes by writing them to persistent storage and/or notifying backend daemons.

This *state-aware approach* significantly improves code coverage compared to stateless fuzzing or blindly mutating concatenated HTTP sessions.

Timeout-based delimitation. AFLNet determines response boundaries using two types of timeouts at the socket layer: a *polling timeout*, which specifies the maximum period to wait before concluding that no further responses will arrive, and a *socket timeout*, which defines the wait time for each individual response. These values act as heuristics to delimit request–response pairs when explicit protocol boundaries are not available.

2.4. Dynamic taint analysis

Dynamic Taint analysis (Newsome and Song, 2005; Clause et al., 2007) is a dynamic program analysis technique that tracks how sensitive input data propagates throughout program computation. Unlike symbolic execution (Baldoni et al., 2018; Borzacchiello et al., 2021; Coppa et al., 2022; Coppa and Izzillo, 2024), dynamic taint analysis tracks data flow dependencies but does not capture the specific transformations or constraints applied to the data. The main idea is that inputs are initially tainted when they are fetched from (untrusted) sources (e.g., read from a file) and then tracked during the execution to check whether they, or any derived data from them, reach sinks (e.g., security-sensitive operation). To propagate taints across variables, registers, and memory, the analysis applies propagation rules at each executed instruction, introducing significant execution overhead.

Taint-guided fuzzing. Given the success of taint analysis in different applications, such as the identification of information leakage (Yang and Yang, 2012), several fuzzing works have explored how to exploit this technique to devise targeted mutations (Bekrar et al., 2012; Liang et al., 2022). For instance, VUzzer (Rawat et al., 2017) exploits dynamic taint analysis to identify which input bytes may affect the decision points of a program, then exploits this knowledge during the mutation phase. However, a notable downside is the overhead introduced by dynamic taint analysis; hence, more recent works have investigated approximations of taint analysis that heuristically attempt to understand whether some input bytes have flowed into, e.g., a comparison operand. For instance, REDQUEEN (Aschermann et al., 2019) introduces the concept of *input-to-state correspondence* that leverages the fact that part of the input is often used as is, or after a few standard manipulations (e.g., addition by 1 or base64 encoding), in a comparison operation.

Whole system taint analysis. Dynamic taint analysis can be performed over a single application or on an entire system. The latter case is often called *whole-system taint analysis* and is particularly challenging to realize since *all* operations performed by the system must be tracked during execution, with sources and sinks that may be defined at software–hardware boundaries. In this article, we build our proposal on top of DECAF++ (Davanian et al., 2019), a dynamic taint analysis framework for QEMU that embeds propagation rules for TCG, QEMU's architecture-independent intermediate representation.

3. Motivation

Modern embedded devices may expose a variety of network protocols, such as HTTP, DHCP, ARP/ICMP, Telnet, FTP, UPnP, and SNMP, through different daemons that can often be configured and controlled via an HTTP interface. The device's internal architecture is far from simple. Manufacturers often combine custom, closed-source components with open-source components common to several Linux platforms, devising workflows that are undocumented and may contain bugs or vulnerabilities (Chen et al., 2016). A common entry point for these vendor-specific workflows is the HTTP interface. Indeed, HTTP requests often serve as entry points to complex multi-process workflows, where Common Gateway Interface (CGI) handlers written in PHP or other languages invoke internal backend daemons to read and write persistent configuration state, as shown in Fig. 1. This layered design means that bugs may arise not from a single malformed HTTP request alone but from a *sequence* of interdependent requests that transit through and affect multiple processes.

Motivating case study. To better illustrate the interplays that can emerge in real-world devices, we present a bug found in the access point D-Link DAP-2310 v1.00_o772 firmware. We discovered this bug thanks to the protocol-aware fuzzing capabilities of STAFF, which uses targeted mutations driven by the intra- and inter-service dependencies identified in the services running on the firmware.

The router runs the lightweight `httpd` HTTP server,[4] with CGI handlers implemented in PHP that use internal components such as `rgdb` (the client-mode interface, typically a symlink to the `xmldb` binary) and `xmldb` (a closed-source runtime configuration daemon). Exploiting the bug requires three coordinated HTTP requests, exemplified in Fig. 3, affecting three different processes. In more detail:

❶ *Authentication.* First, login is performed through the HTTP interface, establishing an authenticated session (internal session id) used by the server-side web stack:

```
POST /login.php HTTP/1.1
Host: 192.168.0.50
Content-Type: application/x-www-form-urlencoded

ACTION_POST=LOGIN&LOGIN_USER=admin&LOGIN_PASSWD=&
login=Login
```

❷ *Poisoned* `xmldb` *state via DHCP configuration.* Second, a crafted DHCP configuration is submitted via an HTTP POST request. This configuration is passed to a PHP CGI script that, in turn, forwards it to `xmldb` deamon through the `rgdb` interface. The `xmldb` deamon records the overlong string into its configuration tree (persisted to disk) without proper validation, inserting the overlong string highlighted in black:

```
POST /adv_dhcpd.php HTTP/1.1
Host: 192.168.0.50
Content-Type: application/x-www-form-urlencoded

ACTION_POST=adv_dhcpd&srv_enable=1
&ipaddr=192.168.0.30[Insert >1024 bytes]
&iprange=235&ipmask=255.255.255.0&...
```

❸ *Exploit buffer overflow in* `xmldb` *via configuration.* Finally, the configuration changes can be committed using an HTTP GET to the XGI handler (e.g., `cfg_valid.xgi`) with an

[4] `httpd` is *mathopd*, an open-source project available on GitHub, but it is not clear whether the vendor has customized its code and functionalities. The public development from the original author halted in 2024.

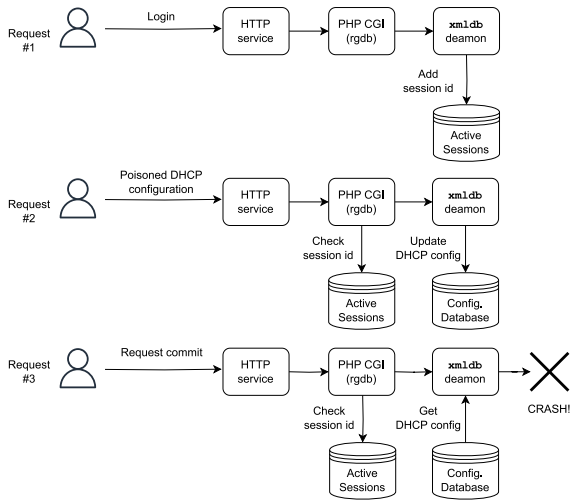


Fig. 3. Motivating case study: stack buffer overflow in router D-Link DAP-2310.

`exeshell=submit COMMIT` parameter. The handler writes a `submit COMMIT` command into the running `xmldb` daemon's control interface (pipe/socket/FIFO). Then, `xmldb` loads the poisoned value and copy it unsafely, resulting in a stack buffer overflow:

```
GET /cfg_valid.xgi?
  random=...&exeshell=submit%20COMMIT
  &exeshell=submit%20DHCPD&flag=1 HTTP/1.1
Host: 192.168.0.50
```

The vulnerable code performs an unsafe `strcpy`:

```
char acStack_410[1024];
...
strcpy(acStack_410, __nptr);
```

This real-world case study clearly shows how an HTTP service may act as the central control plane for device configuration. Yet triggering a serious memory corruption bug requires cross-process stateful interactions involving internal daemons accessed indirectly through HTTP CGI endpoints.

Challenges. As seen in the motivating case study, embedded firmware images may not be a single monolithic blob but a complex interplay of cooperating services, daemons, and kernel components sharing mutable state via IPC, filesystems, and runtime databases. Unfortunately, there are several challenges that limit the effectiveness of representative full-system multi-process firmware fuzzing solutions when considering these complex firmware systems:

C1: Stateful and multi-request interactions. Interactions with modern devices span several requests, such as login, token fetch, sending configuration parameters, and saving the overall configuration. Each of these requests builds on the previous ones, making them inherently stateful. Representative firmware fuzzers, such as TriforceAFL or FirmAFL, ignore such custom protocols, blindly destroying valid request sequences, thus failing to reach interesting states in most of their fuzzing attempts. Protocol-aware fuzzers, such as AFLNet, can perform significantly better on firmware systems such as the case study but have not been ported to the firmware analysis context.

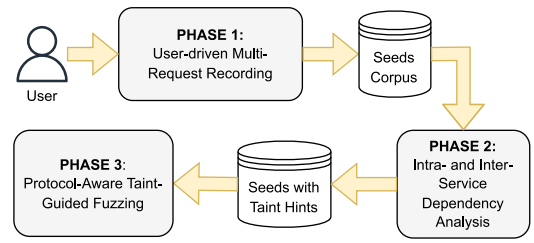


Fig. 4. Bird's-eye view of the conceptual phases in STAFF.

C2: Multiple interdependent services. While a user may interact with a single border service, e.g., the HTTP handler, modern devices are characterized by complex interdependent services. Hence, data sent to a network-facing service may traverse different processes, passing through IPC channels and persistent storage, and then finally reach a vulnerable service. State-of-the-art firmware fuzzers do not currently keep track of how input bytes, received from the network, may impact the overall execution of the entire firmware system. Protocol-aware fuzzers, being mostly designed for single-process applications, cannot cope with such multi-service complex dynamics.

C3: Execution overhead from system mode emulation. A major problem in firmware analysis is the execution overhead introduced by system mode emulation. While the previous challenges may apparently already find some solution ingredients in existing works, such as protocol-aware fuzzing, a significant problem is that these prior proposals cannot be blindly ported to the context of firmware analysis due to the huge cost of fuzzing attempts. Hence, a major challenge is how to be protocol-aware and track complex intra- and inter-service dependencies while minimizing the overhead. In particular, existing fuzzing strategies may require a number of attempts (i.e., requests) that is unsustainable when using system mode emulation, thus requiring optimizations and minimization strategies.

4. Stateful Taint-Assisted Full-system Fuzzing

In this section, we present STAFF, a firmware fuzzing framework designed to discover bugs in Linux-based firmware software. We first present the overall architecture of STAFF, contextualizing it within the challenges presented in Section 3, and then provide detailed coverage of its most important conceptual components.

4.1. Overall approach

STAFF is structured into three main conceptual phases: *User-driven Multi-Request Recording* (Section 4.2), *Intra- and Inter-Service Dependency Analysis* (Section 4.3), and *Protocol-Aware Taint-Guided Fuzzing* (Section 4.4). Fig. 4 provides a bird's-eye view of these phases, which we now review at a high level.

In the first phase, *User-driven Multi-Request Recording*, the user interacts with the firmware under test by manually performing different interactions based on the intended functionalities of the device. As in prior works (Plc, 2017; Zheng et al., 2019; Srivastava et al., 2019), in this article, we focus our attention on interactions primarily carried out through the HTTP interface exposed by the firmware. However, different from prior works, users are not forced to restrict interactions to those affecting only the HTTP daemon but are free to explore the functionalities, since STAFF targets testing of *all binaries directly or indirectly impacted by HTTP inputs*. Hence, the ultimate goal of this phase is to exercise as many firmware functionalities as possible, such as configuration endpoints, status interfaces, or management operations,

allowing STAFF to capture a representative *seed corpus* of request sequences. We use the term *seed* to denote a sequence composed of multiple HTTP requests used by the fuzzer as initial input. Our framework automatically logs these requests at the packet level (e.g., in PCAP format), tags them with timing and response metadata, and stores them in a *seed corpus*. This corpus allows STAFF to replay these recorded sequences in later phases, faithfully reproducing the execution context and ensuring that subsequent fuzzing operates on exactly the same firmware state that the user analyst explored. Prior works have assumed the availability of such a corpus, without systematically providing support for building it. In contrast, STAFF makes this crucial preliminary phase an essential and well-structured component of its architecture.

In the second phase, *Intra- and Inter-Service Dependency Analysis*, STAFF replays each recorded sequence and instruments the firmware execution to extract detailed runtime information. During this replay, it tracks basic block coverage, memory accesses, and filesystem activities. In particular, a taint-based dataflow tracker logs how input bytes propagate through CPU registers, memory accesses, IPC channels, and filesystem operations. This analysis reveals which parts of the input requests may influence which parts of the firmware code, highlighting in detail the intra- and inter-service dependencies impacting the firmware execution. To the best of our knowledge, no prior work in firmware fuzzing has explored a similar analysis with the same fine granularity. The ultimate goal of this phase is to generate a seed corpus annotated with *taint hints*, i.e., valuable information about the firmware execution dependencies that could be exploited during fuzzing to reach interesting execution states.

In the final phase, *Protocol-Aware Taint-Guided Fuzzing*, STAFF sends mutated request sequences to the firmware, preserving their ordering and context as observed during the user interactions, while monitoring the firmware execution in search for crashes and other indicators of inconsistent running state. Mutations are protocol-aware and are performed exploiting the taint hints collected in the previous phase. Code coverage is tracked to evaluate whether non-crashing inputs should be added to the current corpus for further mutations. Due to the large overhead introduced by system-mode emulation, STAFF adopts different optimizations, including Multi-Staged Forkserver to efficiently checkpoint complex protocol states. The ultimate goal of this phase is to possibly generate input request sequences able to reach deep execution paths and surface subtle, inter-service, state-dependent vulnerabilities.

When considering the three challenges mentioned in Section 3, STAFF tackles them in the following way:

C1: Stateful and multi-request interactions. STAFF has been designed to be protocol-aware, capturing full request chains, including authentication, token exchanges, and configuration updates, preserving the exact inter-request context required to trigger deep logic paths.

C2: Multiple interdependent services. While prior systems have explored interdependent services in firmware analysis, STAFF addresses this challenge in conjunction with protocol-aware, multi-request scenarios by exploiting whole-system taint analysis to track user-controlled bytes across all user-space processes, CPU registers, memory accesses, IPC channels, and the filesystem operations. This unified view produces fine-grained byte-to-code mappings and uncovers both volatile and persistent dependencies across service boundaries.

C3: Execution overhead from system mode emulation. STAFF introduces different optimization and minimization strategies to make its analysis scalable and sustainable in the context of system-mode firmware analysis, including *approximated inter-request dependency tracking* and *Multi-Staged Forkserver*.

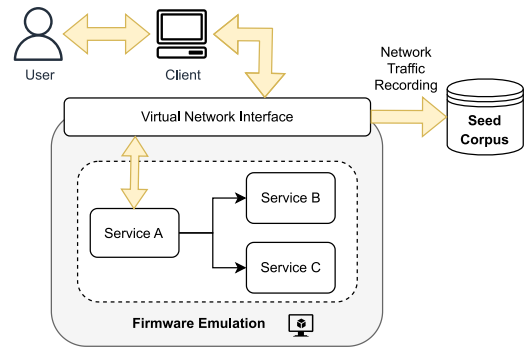


Fig. 5. *User-driven Multi-Request Recording* Phase: A client communicates with the firmware through a virtual network interface, exercising externally reachable services exposed by the firmware under emulation. STAFF records the network interactions (*seed corpus*) to allow replay in the next phases.

4.2. User-driven multi-request recording

A crucial requirement for performing effective fuzzing of software is the availability of a rich and diverse seed corpus. Unfortunately, in the context of firmware analysis, it is quite rare to obtain a set of benign input request interactions for testing from the manufacturer. For this reason, STAFF relies on input request sequences manually generated by a user analyst during an emulation session monitored by STAFF. This manual recording step is consistent with the seed acquisition strategy adopted by representative firmware and protocol fuzzers in the literature: AFLNet (Pham et al., 2020) requires recorded PCAP traces of real client–server exchanges; FIRM-AFL (Zheng et al., 2019) and TriforceAFL (Plc, 2017) rely on user-provided seed files; EQUAFL (Zheng et al., 2022) asks the analyst to provide one valid network request per target; and HouseFuzz (Xiao et al., 2025) reuses the same manually crafted HTTP template seeds as Greenhouse (Tay et al., 2023). Manual seed provision is therefore the norm rather than an exception in the field.

Fig. 5 graphically depicts the expected workflow during this phase. The emulation layer of STAFF follows the FirmAE methodology to prepare an executable virtual environment: it unpacks the firmware image, patches or rebuilds kernels when needed, and mounts the reconstructed root filesystem with valid device nodes and init scripts, configuring a system-mode emulation with virtualized CPU, memory, common peripherals, and injected dummy drivers or stubs for hardware components that cannot be emulated precisely (e.g., GPIOs, watchdogs, UARTs). Network configuration is adapted by modifying the kernel's network device setup to bridge emulated interfaces (e.g., `eth0`, `br0`) with the host network stack, enabling controlled external access to firmware services. To support whole-system dynamic taint analysis and virtual machine introspection (VMI), we replace FirmAE's default `qemu-system` binary with a DECAF++-instrumented QEMU and integrated DECAF++'s taint propagation logic into the emulation workflow. This integration enables byte-level taint tracking across user space and kernel space while preserving the execution semantics expected by the firmware. While this process yields execution environments that are functionally close to the target hardware, kernel version and ABI incompatibilities between FirmAE's reference kernels and DECAF++-instrumented kernels introduce additional compatibility constraints. As a result, a subset of firmware images required exclusion during dataset construction to ensure stable and reproducible instrumented execution (see Section 6.1).

When the emulated environment is fully ready, the user analyst is expected to perform meaningful interactions using a client, e.g., a browser, towards the virtual network interface where STAFF has made available the different network services spawned by the firmware. Our current implementation of STAFF is tailored towards interactions

targeting the HTTP service. These interactions are recorded as network packet traces, where each trace reflects a client session captured to preserve protocol structure, ordering, authentication flows, and state transitions. Given the large number of HTTP requests possibly recorded during the interactions, STAFF performs a preliminary filtering of likely uninteresting requests, discarding endpoints related to static files, such as Javascript code, images, and style sheets.

The recorded sessions form the initial *seed corpus* that are replayed in the next phase to identify interesting service interactions.

Example. During an emulation session of the firmware D-Link DAP-2310 v1.00_o772, the analyst used a web browser to interact with the emulated HTTP service exposed by the virtual network interface. The session began with a `GET` request to `/login.php` to retrieve the login page, followed by a `POST` request submitting the login form with the default `admin` username and an empty password to authenticate the user. Upon successful login, the analyst navigated through several configuration pages, including `/index.php` (dashboard overview), `/home_sys.php` (system status), and `/adv_dhcpd.php` (DHCP server settings), the latter receiving a `POST` request to modify DHCP server parameters such as IP range, subnet mask, and lease time. The interaction concluded with multiple requests to firmware-specific endpoints such as `/cfg_valid.xgi`, `/cfg_valid.php`, and `/__action.php`, which are used to apply and commit configuration changes. All HTTP exchanges were recorded as part of the initial seed corpus, preserving the protocol order, authentication flow, and session state transitions for subsequent fuzzing stages.

4.3. Intra- and inter-service dependency analysis

After collecting a rich corpus of network interactions, STAFF initiates the intra- and inter-service dependency analysis, whose main workflow is depicted in Fig. 6. The ultimate goal of this analysis is to generate actionable and valuable *taint hints* that can help the fuzzing engine to perform effective mutations based on the intra- and inter-service dependencies identified during the analysis. In the remainder of this section, we provide details about each step executed during the analysis.

4.3.1. Session-aware HTTP request modeling

After extracting a seed, i.e., a network interaction, from the corpus, STAFF splits it into distinct network requests, and then processes them with the *session-aware HTTP request modeling* component to reconstruct a structured view of the HTTP requests. Such processing is performed through a HTTP parser that manages a correct *request reassembly* by inferring its termination using HTTP protocol-defined rules:

- For Content-Length-based requests, byte-precise parsing is used.
- Chunked requests are reconstructed by parsing each chunk until termination.
- In the absence of explicit length, connection close semantics are respected.
- For markup-heavy responses (HTML/XML), logical end-tags (e.g., `</html>`) are additionally used to infer completeness.

Differently from timeout-based strategies used in prior protocol-aware works (see Section 2.3), such as AFLNet, our approach handles partial, malformed, or delayed responses gracefully by relying on protocol-aware parsing rather than timing heuristics, improving robustness under real-world conditions. This improved robustness makes it possible to significantly improve the success rate of network interaction reproduction during later stages. Moreover, by obtaining a structural view of the HTTP requests, STAFF can focus its dependency analysis and fuzzing mutations on the significant parts of the network interactions.

Example. During the replay of the previously recorded interaction with the D-Link DAP-2310 v1.00_o772 firmware, the request `GET /index.php` was issued to the emulated HTTP web server. The server responded with a long, chunked HTTP response containing the HTML structure and embedded JavaScript code of the device's administration dashboard. The payload was split across multiple TCP packets, starting with the `Transfer-Encoding: chunked` header and spanning over several kilobytes until the closing `</html>` tag. In timeout-based request reassembly strategies, as adopted by AFLNet, chunked responses may be truncated prematurely. This results in incomplete reassembly of the intended response. The remaining chunks can then be mistakenly associated with the following request, as the first data received after issuing it may still belong to the unfinished previous response. In contrast, STAFF's session-aware HTTP request modeling correctly reassembles the response by parsing the HTTP headers, recognizing the chunked transfer encoding, and reading each chunk until completion, with an additional safeguard of detecting logical HTML end-tags.

4.3.2. Whole system taint analysis

While parsing the network interactions through the session-aware HTTP request modeling component, STAFF replays the identified structured requests into an emulated environment running the firmware under test. During the emulation, STAFF performs whole-system dynamic taint analysis aimed at tracking how request bytes propagate through CPU registers, memory accesses, IPC channels, and filesystem operations.

One major design choice in STAFF is how many distinct taint sources should be tracked. Ideally, we would like to track how each input (request) byte propagates throughout the execution; however, this approach does not scale: binary-level dynamic taint analysis typically supports only a limited number of taint sources to limit the time and space overhead, e.g., up to 8 distinct taint sources. With a limited number of taint sources, a strategy could be to perform several executions, selectively tainting only a few bytes per execution. However, such an approach does not scale since it requires repeating the same execution a large number of times (i.e., N/n executions, where N is the total number of bytes across all requests and n is the number of distinct taint sources) while considering different subsets of tainted bytes from the input. For these reasons, STAFF adopts a more scalable approach where a single taint source is used, exploiting a byte annotation heuristic (Section 4.3.3), inspired by the input-to-state relationships from REDQUEEN (Section 2), to refine the taint dependencies.

During taint analysis, STAFF generates two types of events: *tainted memory accesses* and *filesystem-based dependencies across requests*. For both types of events, STAFF exploits the concept of a *processing time window* of a request, which starts at the time when the HTTP request related to the request is received by the firmware and ends at the time of receipt of the first subsequent HTTP response.

Memory accesses. During execution, STAFF emits byte-granular tainted memory access events by instrumenting memory operations at the guest instruction level. The taint propagation mechanism operates through DECAF++'s Tiny Code Generator (TCG) instrumentation layer, which intercepts all memory load and store operations before they execute. When a tainted memory location is accessed, the taint engine propagates taint labels across the entire guest system, including both user-space and kernel-space memory regions, according to DECAF++ propagation rules: loads from tainted memory assign the corresponding taint label to the destination registersq, stores from tainted registers propagate the taint label to the destination memory location via shadow memory, and arithmetic or logical operations combine the taint labels of their operands. For instance, when firmware processes network data through the kernel's TCP/IP stack, tainted bytes move from socket buffers (`struct sk_buff`) through kernel page cache

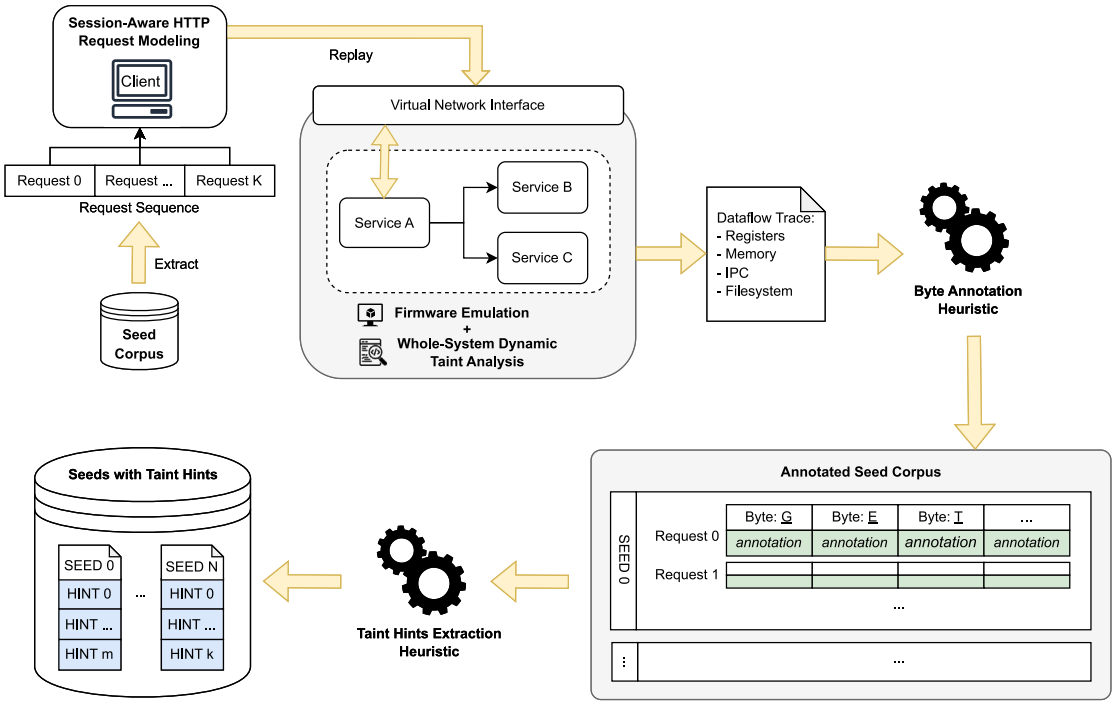


Fig. 6. Intra- and Inter-Service Dependency Analysis: (a) network interactions are extracted from the seed corpus; (b) split into requests; (c) replayed using a *session-aware HTTP request modeling* component within an emulated environment where *whole-system dynamic taint analysis* is performed to track how request bytes flow into CPU registers, memory accesses, IPC channels, filesystem operations, and more generally impact firmware service execution; (d) the *byte annotation heuristic* enriches each byte of each request with annotations summarizing its impact on the firmware execution; (e) the *taint hints extraction heuristic* transforms the annotations into actionable *taint hints* to guide the mutations during fuzzing.

into application-accessible memory regions, with each transition captured by the taint engine. Each emitted event captures the following information:

- `request_id`: identifier of the request whose processing time window contains the memory access.
- `pc`: identifier combining the inode[5] and the program counter (PC) of the basic block performing the memory access.
- `gpa`: the physical address of the memory access.
- `value`: the tainted (loaded or stored) byte value.

Each event is stored in a per-seed data structure `tainted_memory_events`: `tainted_memory_events[load]` collects the *load* events, while `tainted_memory_events[store]` collects the *store* events. The chronological ordering of these events preserves the temporal relationship between memory operations, enabling subsequent analysis to reconstruct data flow paths through the execution.

IPC channels. STAFF can naturally track tainted bytes traversing inter-process communication (IPC) channels thanks to its taint analysis being able to observe memory regions (even the ones used by the kernel to store temporary data) or other virtual files involved in IPC channels. For IPC mechanisms implemented via the virtual filesystem, including Unix domain sockets, named pipes (FIFOs), regular files used for message passing, and pseudo-filesystems like `/proc`, the taint engine observes data flow as ordinary memory writes and subsequent reads across processes. When one process writes to such a channel, the kernel copies data into kernel buffers or page cache structures. When another process subsequently reads from the same channel, the kernel delivers this data into the reader's memory space. For IPC mechanisms that bypass filesystem semantics entirely, such as anonymous pipes created via `pipe()`, POSIX shared memory segments created through `shm_open()`, and memory-mapped regions with `MAP_SHARED` flags, STAFF relies exclusively on DECAF++'s memory-level taint propagation. By tracking taint at the level of memory reads and writes, STAFF provides uniform, mechanism-agnostic coverage of IPC without requiring channel-specific instrumentation, ensuring complete inter-process data-flow tracking across the firmware.

Filesystem operations. Beyond byte-level taint propagation, STAFF maintains semantic-level tracking of filesystem operations to establish control-flow dependencies between requests. This tracking operates by intercepting system calls at the emulation layer and maintaining per-process tables that map file descriptors to filesystem nodes (identified by device and inode numbers). When a process invokes a writer syscall, STAFF records the association between the currently processing request and the affected file node. When a subsequent reader syscall accesses the same file node during a later request's processing window, STAFF establishes a producer–consumer dependency between the writer and reader requests.

The tracking mechanism handles various filesystem scenarios that appear in firmware. For regular file operations, the inode-based identification ensures that dependencies are preserved even when processes use different paths to access the same underlying file (e.g., through symbolic links or different mount points). For operations on virtual filesystems like `/proc` or `/sys`, where file nodes represent dynamic kernel or process state rather than persistent storage, the tracking captures how one process's modification of system state through writes to these pseudo-files affects subsequent reads by other processes. For `tmpfs`-based files commonly used in embedded systems for temporary storage, the tracking operates identically to regular files despite the in-memory nature of the filesystem. In more detail, syscalls are split into two semantic classes:

[5] The PC of the basic block may not be unique across different processes, hence, STAFF uses the filesystem inode number of the application binary to discriminate the same PC across different applications.

- *Writer syscalls:* operations that create or modify file content or metadata (`write`, `pwrite`), truncates data (`truncate`, `ftruncate`), changes file structure (`rename`, `unlink`), or creates new entries (`open` with `O_CREAT`, `mkdir`, `mknod`).
- *Reader syscalls:* operations that observe existing file state without modification (`read`, `pread`), access metadata (`stat`, `lstat`, `fstat`, `access`), or open files in read-only mode (`open` with `O_RDONLY`). This classification enables STAFF to capture not only direct content dependencies but also metadata dependencies where one request's structural changes (e.g., file creation or deletion) affect another request's filesystem probes.

Given these two semantic classes, STAFF identifies a dependency between two requests R_i and R_j within the same seed when during the processing of a request R_i a writer syscall is executed to a file node f_k, and, later, during the processing time window of request R_j a reader syscall is executed on the same file node f_k. More formally, the inference rule is:

$$R_i \xrightarrow{\texttt{writer}} f_k \xrightarrow{\texttt{reader}} R_j \quad \Rightarrow \quad R_i \xrightarrow{f_k} R_j$$

STAFF stores these types of request dependencies within a per-seed map `request_deps`, where each reader–writer pair (R_j, R_i) is mapped to the list of involved files:

$$\texttt{request_deps}[R_j][R_i] = [\ldots, f_k, \ldots]$$

Scalability. While dynamic taint analysis can introduce a significant runtime overhead, it is crucial for the scalability of the intra- and inter-service dependency analysis devised by STAFF. Indeed, the scalability of the Byte Annotation Heuristic (Section 4.3.3) critically depends on the volume of memory events. Without taint filtering, the heuristic would need to examine all memory operations executed during request processing, which is infeasible at firmware scale. During the development, we quantified the impact of taint filtering when considering the sessions used in our experimental evaluation (see Section 6) under two configurations: (i) logging all memory operations, and (ii) logging only operations involving tainted memory. Without taint analysis, executions generated between 1.3×10^9 and 5.7×10^{11} memory operations per replay (median: 3.7×10^{10}). With taint filtering enabled, this dropped to 6.2×10^5–3.8×10^8 events (median: 3.5×10^7). Overall, taint filtering yields a median reduction of 99.93% in analyzed events (mean: 99.83%), with reduction factors ranging from 1/85 to 1/29,554 (median: 1/1,532). This dramatic reduction reflects the sparsity of tainted data propagation: although firmware execution involves billions of memory accesses, only a small fraction originate from network inputs. By restricting analysis to taint-dependent operations, STAFF transforms an otherwise intractable problem into a scalable procedure for whole-system firmware analysis.

Example. Fig. 7 depicts an example of memory and file system tracking performed by STAFF during whole system taint analysis. In particular, the example considers an interaction constituted by three requests:

R_0 When processing the request `POST login.php`, the program first loads the bytes `LOGIN` from the request body of R_0 and then stores them into a different memory area. STAFF collects several memory events to capture these loads and stores of tainted bytes from the request. Meanwhile, the program also creates the file `/var/proc/web/session:1/user/ac_auth`, thus acting as a writer, an operation tracked by STAFF. Note that the window processing time of a request starts with its receipt and ends with the first response.

R_1 When processing the request `POST adv_dhcp.php?ipaddr=192.168.0.30`, STAFF tracks how the tainted bytes `192.168.` from request R_1 are moved around in memory. Also, STAFF marks the second request R_1 as a reader of the file f with path `/var/proc/web/session:1/user/ac_auth` previously written by request R_0, i.e., $R_0 \xrightarrow{f} R_1$.

R_2 When processing the request `GET /cfg_valid.xgi&exeshell=submit`, STAFF identifies how some bytes (`192.168.`) from request R_1 are loaded during the processing of request R_2. Finally, it also marks the third request R_2 as a reader of the file f with path `/var/proc/web/session:1/user/ac_auth` previously written by request R_0, i.e., $R_0 \xrightarrow{f} R_2$.

4.3.3. Byte annotation heuristic

Algorithm 1: Byte Annotation Heuristic

Input : Seeds S, where $\forall s \in S$:
- s.`requests` (seed bytes for each request)
- s.`tainted_memory_events` (list)
- s.`request_deps` (per request map)

Output: Annotated seeds S, where $\forall s \in S$:
- s.`requests` (seed bytes for each request)
- s.`tainted_memory_events` (list)
- s.`request_deps` (per request map)
- s.`annotated_bytes` (annotated bytes)

```
1  PCs ← {}                                   // mapping: pc ↦ seed
2  for s ∈ S do
3      FilterAndFlattenRequestDeps(s)
4      T ← IndexRequestByteSequencesInTrie(s)
5      for r ← 0 to len(s.requests) − 1 do
6          s.annotated_bytes[r] ←
7              [{deps: {}, PCs: {}} : b ∈ s.requests[r]]
8      for t ∈ {load, store} do
9          E ← []
10         foreach e ∈ s.tainted_memory_events[t] do
11             ProcessMemEvent(s, e, E, T, PCs)
12 return S
```

After collecting events during whole-system taint analysis, STAFF exploits the *Byte Annotation Heuristic* (Algorithm 1) to enrich each region's bytes with metadata describing how bytes have propagated through CPU registers, memory accesses, IPC channels, and filesystem operations, which basic blocks accessed them, and which inter-region dependencies related to filesystem operations exist. As mentioned in Section 4.3.2, since STAFF adopts a dynamic taint analysis with a single taint label to make the approach more scalable, the heuristic has to infer how specific byte sequences impact specific basic blocks using a value-matching strategy, inspired by the input-to-state relationships from REDQUEEN (Section 2), to refine the taint dependencies.

The heuristic takes as input a set of seeds S that must be annotated. Each seed s is composed of:

- `requests`: arrays of bytes, one for each input request;
- `tainted_memory_events`: a chronologically ordered list of tainted memory access events (Section 4.3.2);
- `request_deps`: a per-seed data structure storing filesystem-related request dependencies (Section 4.3.2).

The output is the same set of seeds S, where each seed has been enriched with:

- `annotated_bytes`: for each byte in each request, a structure containing the set of identifiers (`deps`) for the requests that access the byte during processing and the set of program counters (`PCs`) related to the basic blocks that access it.

Initially, the heuristic initializes the map `PCs` (line 1). This map tracks, for each basic block (identified by its program counter), which seed first[6] executed that block and generated a byte annotation. The

[6] The seed ordering does not matter.

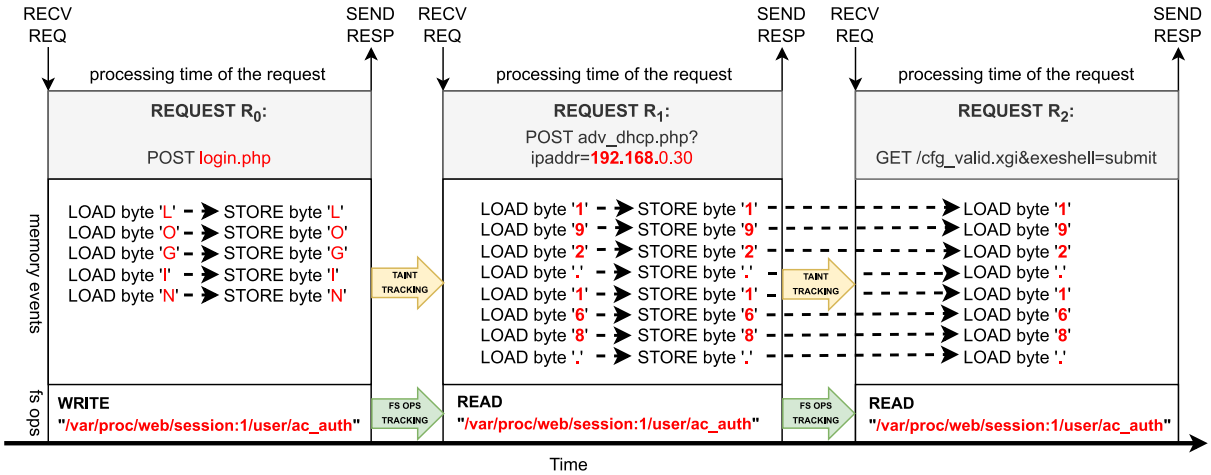


Fig. 7. Example of inter-request dependencies within a single seed. The figure depicts one seed composed of three requests (three HTTP requests). The file `/var/proc/web/session:1/user/ac_auth` is created (writer) by the first request R_0 and then read (readers) by the two subsequent requests R_1 and R_2; the bytes `192.168.` from the request R_1 are propagated to the request R_2..

Algorithm 2: Filter and Flatten Request Deps

1 **Procedure** FilterAndFlattenRequestDeps(s)
2 $\quad deps \leftarrow s$.`request_deps`
3 $\quad MWF[f] \leftarrow \min\{w \mid \exists r : f \in deps[r][w]\}$
4 $\quad FW[w] \leftarrow \{f \mid MWF[f] = w\}$
5 $\quad W[r] \leftarrow \{w \mid w \in deps[r]\}$
6 $\quad deps[r] \leftarrow \{\text{BuildDeps}(r, W, FW, \{\}) \mid r \in deps\}$

Algorithm 3: Build Filtered Dependencies

1 **Procedure** BuildDeps(r, W, FW, V)
2 $\quad$ **if** $r \in V$ **then return** $\{\}$ **else** V.insert(r)
3 $\quad deps \leftarrow \{\}$ // `mapping: writer ↦ set(files)`
4 $\quad$ **foreach** $w \in \text{sorted}(W[r])$ **do**
5 $\quad\quad$ **if** $w \in FW$ **then**
6 $\quad\quad\quad deps[w]$.add(f) $\quad \forall f \in FW[w]$
7 $\quad\quad wdeps \leftarrow \text{BuildDeps}(w, W, FW, V)$
8 $\quad\quad$ **foreach** $ww \in wdeps$ **do**
9 $\quad\quad\quad deps[ww] \leftarrow deps[ww] \cup wdeps[ww]$
10 $\quad$ **return** $deps$

algorithm then considers each seed s from S (line 2) and performs three main steps:

1. *Filtering and Flattening Inter-Request Dependencies.* The algorithm filters the raw request dependencies s.`request_deps` by invoking the procedure *FilterAndFlattenRequestDeps()* at line 3 (Algorithm 2). The goal of this filtering procedure is to retain only the most relevant request dependencies by eliminating redundant or less significant associations.

2. *Multi-Request Subsequence Indexing.* The algorithm builds a per-seed trie[7] T at line 4 to track the positions of all byte subsequences from the seed's requests using the procedure *IndexRequestByteSequencesInTrie*. The trie T is used in the next step during value-matching to determine whether a given byte sequence appears in any request.

3. *Request Byte Annotation.* The algorithm initializes the annotations s.`annotated_bytes` for each byte of each request at lines 5–7. It then processes the memory events in s.`tainted_memory_events` at lines 8–11, distinguishing between *load* and *store* events, to generate the annotations using the procedure *ProcessMemEvent* (Algorithm 4).

We now review in more detail the procedures used by these three steps, explaining their goals and motivation.

The procedure *FilterAndFlattenRequestDeps()* (Algorithm 2) filters and flattens the inter-request dependencies `request_deps` from each seed. Fig. 8 provides a high-level intuition of the work performed by the procedure[8]: first, the algorithm identifies the first time files are written during the interaction (green files and edges); then, edges and files related to non-first writers (red files and edges) are removed and replaced with edges connecting the affected readers to the first writers (blue edges). In more detail, *FilterAndFlattenRequestDeps()* operates in four distinct phases. First, each file f is assigned to the first (*minimum*) request writer $MWF[f]$, i.e., the smallest request identifier[9] that writes f (line 3). This assignment ensures that each file contributes exactly once to the filtered dependency structure. Second, files are grouped by their minimum writer into the FW mapping (line 4). Third, a writer mapping W is constructed by inverting the original `request_deps` edges (line 5). Finally, for every reader request r, the algorithm invokes `BuildDeps` to compute a filtered, flat mapping from writers to files reachable from that reader (line 6); this result replaces the original $deps[r]$ entry. The recursive procedure *BuildDeps()* (Algorithm 3) returns, for a given reader r, a mapping *deps* between its writers and their filtered written files. For each writer w associated with the current reader, the procedure (i) adds the files for which w is the minimum writer by incorporating $FW[w]$ into $deps[w]$ (lines 5–6), (ii) recursively computes the writer's dependencies *wdeps* (line 7), and (iii) merges (via set union) the writer's mapping into the reader's $deps$ (lines 8–9). Through this process, the procedure eliminates redundant contributions from files referenced by multiple writers. The computed $deps$ mapping is returned to the caller (line 10).

[7] A trie (Fredkin, 1960) is a specialized search tree where each node represents a character and stores indices indicating where the string from root to node appears.

[8] We remark that `request_deps` maps readers to writers, hence representing writer–reader edges across requests in the opposite direction.

[9] Request identifiers reflect the request ordering from an interaction.

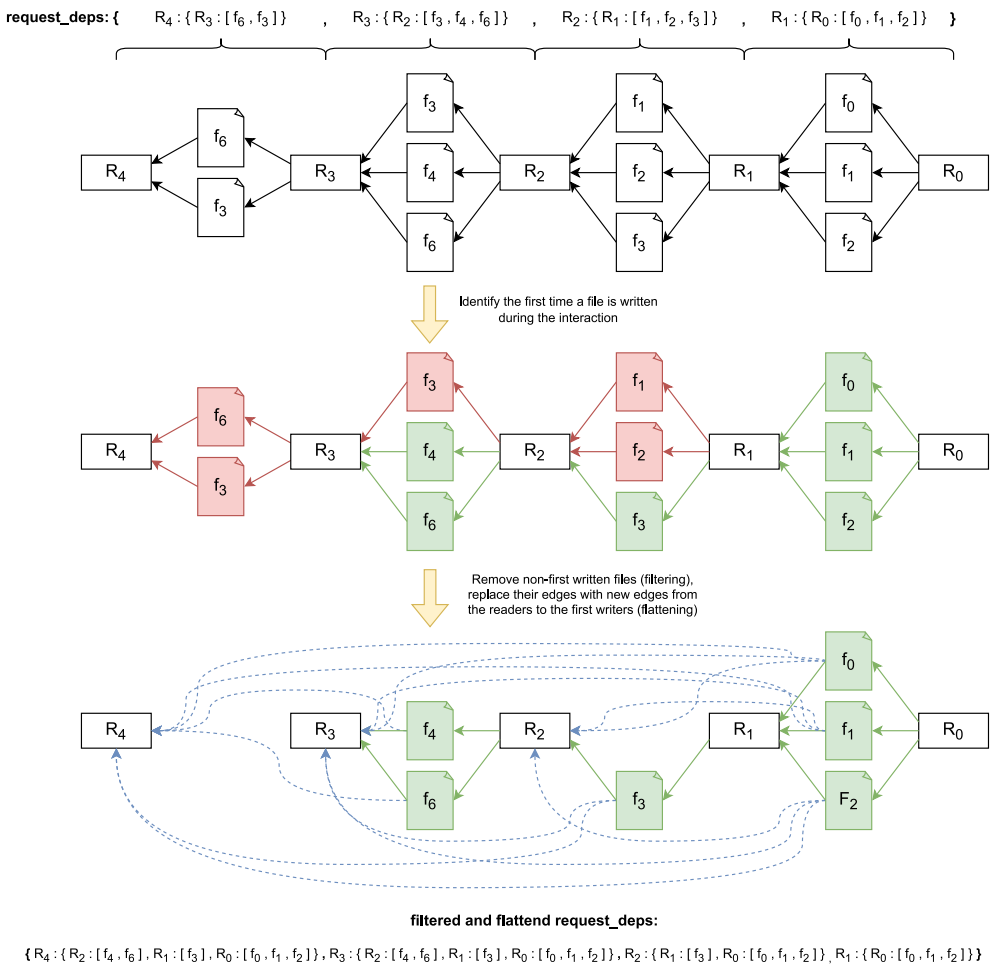


Fig. 8. Example of the filtering and flattening of inter-request dependencies (`request_deps` from each seed) performed by the procedure *FilterAndFlattenRequestDeps()* (Algorithm 2).

Algorithm 4: Process Memory Access Events

1 **Procedure** ProcessMemEvent(s, e, E, T, PCs)
2 $\quad$ $e_prev \leftarrow E[e.\texttt{type}]$.last()
3 $\quad$ **if** $e_prev = \texttt{None}$ **or** ($e.\texttt{request_id} = e_prev.\texttt{request_id}$ **and** $e.\texttt{gpa} = e_prev.\texttt{gpa} + 1$) **then**
4 $\quad\quad$ $E[t]$.append(e)
5 $\quad$ **else**
6 $\quad\quad$ $r \leftarrow e_prev.\texttt{request_id}$
7 $\quad\quad$ AnnotateSubrequest(s, E, r, T, PCs)
8 $\quad\quad$ $E \leftarrow [e]$

Algorithm 5: Annotation of a Subrequest Sequence

1 **Procedure** AnnotateSubrequest(s, E, r, T, PCs)
2 $\quad$ **for** $\ell \leftarrow \text{len}(E)$ **to** 1 **do**
3 $\quad\quad$ **for** $k \leftarrow 0$ **to** $len(E) - \ell$ **do**
4 $\quad\quad\quad$ $seq \leftarrow \left[E[i].\texttt{value} \mid i \in [k : k + \ell]\right]$
5 $\quad\quad\quad$ $matches \leftarrow$ FindByteSeqInRequests(T, seq)
6 $\quad\quad\quad$ **if** len($matches$) $\neq 1$ **then continue**
7 $\quad\quad\quad$ $r' \leftarrow matches[0].\texttt{request_id}$
8 $\quad\quad\quad$ $p \leftarrow matches[0].\texttt{position}$
9 $\quad\quad\quad$ **if** $r' > r$ **then continue**
10 $\quad\quad\quad$ **for** $j \leftarrow 0$ **to** $\ell - 1$ **do**
11 $\quad\quad\quad\quad$ $b \leftarrow s.\texttt{annotated_bytes}[r'][p + j]$
12 $\quad\quad\quad\quad$ b.deps.insert(r)
13 $\quad\quad\quad\quad$ $pc \leftarrow E[j].\texttt{pc}$
14 $\quad\quad\quad\quad$ **if** $pc \notin PCs$ **then** $PCs[pc] \leftarrow s$
15 $\quad\quad\quad\quad$ **if** $PCs[pc] = s$ **then** b.PCs.insert(pc)

The procedure *IndexRequestByteSequencesInTrie()* implements a standard trie construction algorithm that tracks the positions of all byte subsequences from the seed's requests. We omit the implementation details for brevity, but discuss how to avoid memory explosion during construction in Section 5.

The procedure *ProcessMemEvent()* (Algorithm 4) analyzes tainted memory access events by grouping contiguous memory accesses into sequences and triggering annotation when sequences are complete. When a new event e arrives, the procedure checks (lines 2–3) if it continues a contiguous sequence from a previous event of the same type (load or store) by comparing request identifiers and guest physical addresses. If this is the case, it adds the event to the current sequence (line 4). Otherwise, if the event breaks contiguity, the procedure calls *AnnotateSubrequest()* (line 7) to process the completed sequence,[10] then clears the event buffer E (line 8) to start a new sequence. The

[10] The procedure is also called when processing the last event from s.tainted_memory_events[t]. The code is omitted for clarity.

Algorithm 6: Taint Hints Extraction Heuristic

```
   Input  : Annotated seeds S
   Output: Hints H[s] for each seed s ∈ S
1  H ← {s ↦ [] : s ∈ S}
2  for s ∈ S do
3      W ← {}                                  // request writers
4      for r ← 0 to |s.annotated_bytes| − 1 do
5          for j ← 0 to |B| − 1 do
6              for w ∈ B[j].deps do
7                  W[w] ← r
8      for r ← 0 to |s.annotated_bytes| − 1 do
9          D ← s.request_deps[r] ∪ W[r]
10         P ← {}        // pending data to process for PCs
11         B ← s.annotated_bytes[r]
12         for j ← 0 to |B| − 1 do
13             for pc ∈ B[j].PCs do
14                 if pc ∉ P then
15                     P[pc] ← { offsets: [], deps: D}
16                 P[pc].offsets.append(j)
17                 P[pc].deps ← P[pc].deps ∪B[j].deps
18                 if j = |B| − 1 or pc ∉ B[j + 1].PCs then
19                     H[s].append({
20                       request : r
21                       deps : P[pc].deps
22                       offset : P[pc].offsets[0]
23                       len : |P[pc].offsets|
24                     })
25                     P.remove(pc)
26         H[s] ← SortFilterHints(H[s])
```

AnnotateSubrequest() procedure (Algorithm 5) performs the core annotation logic. It extracts byte sequences of decreasing lengths from the completed event sequence E (line 4) and uses the trie T to find matching patterns in the seed requests (line 5). When a unique match is found in an earlier request (ensuring temporal correctness) via lines 6–9, the procedure annotates the corresponding bytes with dependency information (line 12). The requirement for unique matches is designed to reduce ambiguity and false positives, but it may introduce false negatives by discarding some true dependencies by removing ambiguous byte-pattern matches. This design choice intentionally trades recall for precision and scalability. Additionally, it conditionally records the program counters of basic blocks containing memory accesses that contributed to the match (line 15), but only when the current seed is the first to execute those basic blocks and generate annotations for them (line 14). This conditional check helps determine which seed should serve as the reference when targeting specific basic blocks.

4.3.4. Taint hints extraction heuristic

After generating the byte annotations, STAFF computes the *taint hints* that will guide the mutations during the fuzzing phase. Each taint hint is associated with a specific seed s and is characterized by the following fields:

- `request`: the identifier of the request in s to be mutated.
- `offset`: the byte offset within the request where the mutation should begin.
- `len`: the number of consecutive bytes starting from `offset` for which the mutation should be applied.
- `deps`: the set of requests that must be preserved during the replay of s since they either influence the mutated request or are influenced by it.

Algorithm 7: Sort and Filter Hints

```
1  Procedure SortFilterHints(H)
2      I ← ∅
3      H̄ ← []
4      for h ∈ SortAscByLen(H) do
5          start ← h.offset
6          end ← h.offset +h.len −1
7          if NoIntervalOverlap(start, end, I) then
8              H̄.append(h)
9              I.append([start, end])
10     return H̄
```

Each taint hint represents a sequence of bytes from a request that can impact the execution of a specific basic block and thus could be worth mutating.

The algorithm for taint hints extraction (Algorithm 6) starts by initializing the hints H for all seeds at line 1 and then considers each seed s from the annotated seeds S (Section 4.3.3) at lines 2–26. First, it groups the readers by writer in W at lines 3–7. Then, for each request of the seed, the algorithm tracks some pending data about the influence of the request on a specific basic block using a map P (line 10), which accumulates data until taint hints are finalized during the processing of all byte annotations (lines 12–26).

During the processing of each byte annotation, the algorithm iterates over the basic blocks PCs influenced by the byte (line 13). If a basic block pc is not in P (line 14), i.e., the current byte is the start offset of a new taint hint for a basic block, then $P[pc]$ is initialized (line 15), considering the request dependencies D as its initial dependencies $P[pc]$.`deps`. Then, $P[pc]$ is updated by adding the current byte index to the list of `offsets` (line 16) and merging the byte dependencies with $P[pc]$.`deps` (line 17). Afterward, the algorithm evaluates whether the pending data for the current pc can be turned into a taint hint at line 18 by checking whether the next annotation byte does not influence pc, making this byte the last in the pending sequence influencing pc, or whether this is the last byte from the request. The new finalized taint hint picks the first offset from $P[pc]$.`offsets` as the starting `offset` (line 22) and a `len` that is equal to the number of consecutive bytes influencing pc (line 23). After adding the taint hint to $H[s]$, the pending data $P[pc]$ for the related pc is cleared.

The last step of the algorithm when processing a request is to sort and filter the hints $H[s]$, which is done through the procedure *SortFilterHints()* (Algorithm 7) that sorts the hints by their length (line 4), thus favoring shorter byte sequences, and then keeps only non-overlapping hints (lines 5–9) when considering their interval (s.`offset`, s.`offset` +s.`len`-1).

4.4. Protocol-aware taint-guided fuzzing

After generating taint hints, STAFF can start the fuzzing phase, whose workflow is depicted in Fig. 9. First, the seeds and their related taint hints are imported into the input queue (step a), then an input is picked (step b), determining the fuzzing strategy to consider for the input: *state-aware* or *taint-guided*. The first one is inspired by protocol-aware fuzzing, while the latter is driven by taint hints (Section 4.3.4). Regardless of the strategy, during *Mutation Target Selection*, the input is split (step c) into a prefix, one or more target requests T, and a suffix. In the taint-guided strategy, T consists of a single request with a specific mutation window derived from taint hints. In the state-aware strategy, T may comprise multiple requests selected for mutation. Then, the input is moved to *Stateful Sequence Execution* (step d), where the target request T is mutated according to the strategy and the prefix is replayed (step f), adopting a Multi-Staged Forkserver to amortize the cost of future fuzzing attempts from the same prefix (step e). At this

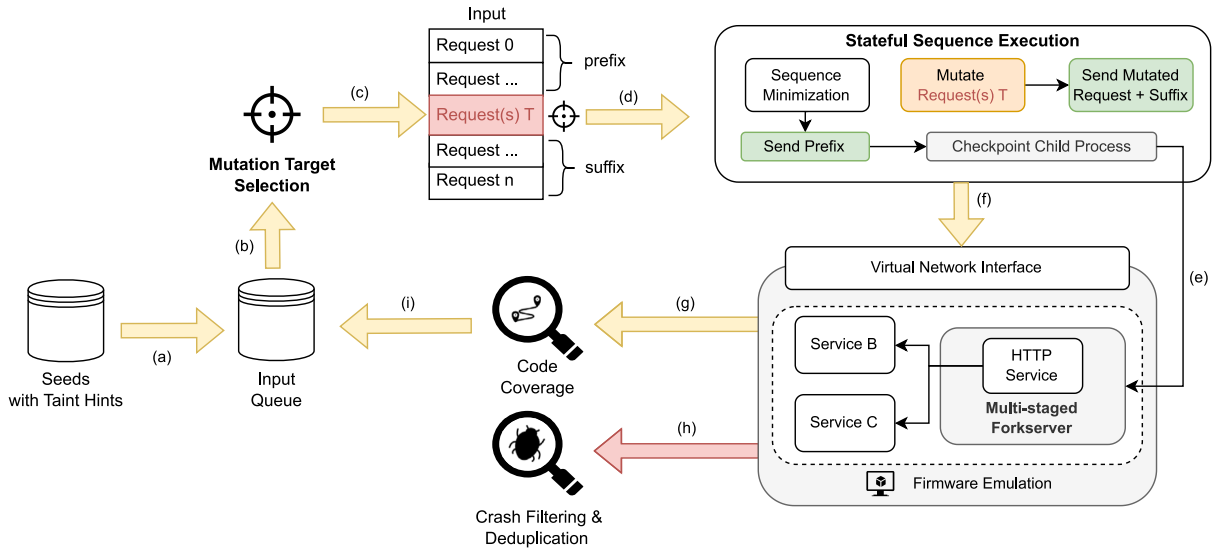


Fig. 9. Protocol-Aware Taint-Guided Fuzzing.

point, the mutated request T and the suffix are replayed (step f). During firmware execution, STAFF saves mutated inputs that lead to crashes for later analysis (step h) and inserts non-crashing inputs into the queue when they achieve improved code coverage (step g). Additional fuzzing attempts with different mutations are then considered over the current input, repeating steps d–i several times. When sufficient effort has been spent mutating an input, the cycle (steps b–i) is repeated by considering other inputs and switching between the two fuzzing strategy. This cycle continues until the time budget for the fuzzing session is fully consumed. The remainder of this section provides detailed descriptions of these steps.

4.4.1. Input queue scheduling

The scheduler alternates the two fuzzing strategies to balance *state-aware* mutations with *taint-guided* mutations, ensuring neither mode starves the other and that both receive comparable fuzzing budget during a campaign.

State-aware strategy. In AFLNet's model, a *state* corresponds to a unique server response code (e.g., HTTP status code 200 OK, 401 Unauthorized) or combination of response codes observed after executing request sequences. The state identifier is typically the response code itself or a hash value derived from the response pattern. By dynamically observing these responses during fuzzing, AFLNet constructs an implicit protocol state machine where transitions represent requests that move the server between different response states. When the state-aware strategy is active, STAFF follows the traditional AFLNet scheduling in protocol-aware fuzzing (Section 2): STAFF maintains a queue of inputs (both seeds and newly generated mutants), grouping them by the server state reached and choosing randomly within each group, following a round-robin policy between groups. Once a seed is selected from a state group, the scheduler must identify which portion of the request sequence to mutate using AFLNet's subsequence decomposition model. This model partitions each request sequence into three components (see Section 2.3): a prefix subsequence containing requests that drive the server to the target state, a subsequence T containing requests that can be executed while remaining in the target state, and a suffix subsequence containing the remaining requests. The scheduler mutates only T to ensure that mutations occur at or immediately after reaching the targeted state, while preserving the prefix to guarantee the target state is reached and maintaining the suffix to observe potential delayed effects of the mutations. The target state identifier corresponds to the state associated with the selected group and determines how these subsequences are identified. When the target state identifier is zero, indicating focus on the initial protocol interaction, no prefix subsequence is required since the server begins in its initial state. In this case, prefix is empty and T starts from the first request. The scheduler identifies T by iterating through the sequence and counting consecutive requests that share the same state count as the first request. This grouping captures the initial protocol handshake or authentication phase where multiple requests produce equivalent state characteristics. The first request whose execution produces a different state count marks the boundary where T ends and the suffix begins. For non-zero target states, the scheduler must first locate which request achieves the desired state. It iterates through the sequence examining each request's state annotations until finding the first request whose execution reaches the target state. This request marks the beginning of T, while all preceding requests constitute the prefix. The scheduler then counts subsequent consecutive requests maintaining the same state count as this target request, establishing the extent of T. This subsequence represents the request of protocol behavior where the server remains in the targeted state, allowing mutations to explore variations in how the protocol handles interactions at that specific point. The identified candidate subsequence becomes the focus of mutation, enabling systematic exploration of protocol behavior at particular server states while maintaining the structural integrity necessary to reach those states.

Taint-guided strategy. When the taint-guided strategy is active, the scheduler operates on a separate input queue containing exclusively the initial seed sequences from the recording phase, excluding any mutants generated during the fuzzing campaign. This separation reflects that taint hints derive from whole-system taint analysis performed on the original recorded traces, and thus apply only to the seed sequences for which dependency information was captured. The scheduler first examines whether a selected seed has unexploited taint hints. If such hints exist, the scheduler processes the seed in taint-guided mode; otherwise, it defers the seed to state-aware scheduling. Seeds with available taint hints are processed using a round-robin scheme across all initial seeds that maintain such hints. For each seed, the scheduler retrieves the next unused *taint hint* and schedules a complete round of fuzzing attempts targeting the specific request and byte range identified by that hint. Each scheduling cycle processes only one hint per seed, and once all hints associated with a seed are exhausted, the scheduler naturally transitions that seed to state-aware exploration. This round-robin approach ensures each seed receives an equal share of fuzzing energy and guarantees that all *taint hints* are eventually explored. When the scheduler reaches the last enqueued seed, it wraps around to the first seed and continues the cycle. For seeds containing multiple hints,

a per-seed pointer tracks progress through the hints and only one hint is processed per scheduling cycle, preventing any single seed or hint from dominating the fuzzing effort.

4.4.2. Mutation target selection

The *Mutation Target Selection* phase decides which part of the input should be mutated.

With the *state-aware* strategy, STAFF adopts state-aware target selection from protocol-aware fuzzing (Section 2.3), which relies on the *Session-Aware HTTP Request Modeling* (see Section 4.3.1) to select the subsequence of requests that corresponds to a particular server state.

With the *taint-guided* strategy, the mutation target is chosen according to the taint hint suggested by the input scheduler. Each taint hint specifies a mutation window as a triple (`target`, `offset`, `len`), where `target` is the target request to mutate in the input, `offset` is the starting offset for the mutation, and `len` is the maximum number of bytes to mutate.

4.4.3. Stateful sequence execution

The *Stateful Sequence Execution* phase performs mutations according to the fuzzing strategy against a chosen input and replays the interaction against the target HTTP service of the firmware, while ensuring each mutated input runs in the correct protocol state and minimizing the replay execution cost. These goals are achieved through a few key ideas that we now review.

Sequence Minimization. When the target selection is driven by taint hints, STAFF performs a *Sequence Minimization* procedure, aimed at removing any requests before the target request T that do not influence T, keeping only the minimal set of requests in the prefix that are required to preserve data- and filesystem-driven dependencies.

Input mutation. STAFF adopts two complementary mutation strategies: deterministic bit-flips and havoc stage. Bit-flips are confined to the mutation window selected by the *Mutation Target Selection* and do not alter request length or overall message layout. Both single- and multi-bit flips are used to corrupt token contents, numeric fields, or header bytes while preserving offsets. The havoc stage composes several stacked mutations into each single trial, including value overwrites, small arithmetic operations, block copy/clone, byte-range insertions/deletions, and request overwrites. This allows STAFF to cover both fine-grained corruptions and broader structural distortions of the input. The same bit-flip and havoc mutations are applied under both taint-guided and state-aware fuzzing strategies. However, one subset of havoc edits, i.e., the request-level operations that change the number of requests in the sequence (request replacement, insertion, duplication, and deletion), are disabled during taint-guided fuzzing. These edits alter the prefix/suffix structure of the request sequence, thereby shifting offsets and breaking the alignment required for taint hints, which assume the original number of requests. By contrast, state-aware mutations are resilient to these shifts, since they do not rely on precise request-preservation semantics.

Multi-Staged Forkserver. STAFF performs multiple fuzzing attempts on each input. To amortize the replay cost across these attempts, STAFF devises a *Multi-Staged Forkserver* that enables efficient checkpointing of protocol states at request boundaries. Traditional fuzzing frameworks such as AFL and AFL++ employ a *single-stage forkserver*, inserting a single fork point at a predetermined location (e.g., the first network-related system call). While effective for stateless programs, this design forces stateful fuzzers to replay entire request sequences from the beginning for every mutation. AFLNet and TriforceAFL inherit this limitation despite their protocol-aware capabilities. To mitigate such a limitation, STAFF exploits a *first-stage forkserver* that captures the initial HTTP service state, which is used when fuzzing an input for the first time. However, during the first fuzzing attempt over an input, STAFF replays the prefix of requests preceding the target request and dynamically establishes a *second-stage forkserver* at the boundary between prefix and target. This second-stage forkserver caches the intermediate protocol state reached after prefix execution. Any subsequent fuzzing attempt related to the same input and targeting the same request can then exploit the second-stage forkserver to skip prefix replay entirely, directly resuming from the cached state. This avoids re-executing identical prefix requests across mutation batches and dramatically reduces replay overhead. A detailed comparison with other snapshot- and checkpoint-based fuzzing approaches is provided in Section 7.

4.4.4. Whole-system code coverage tracing

Code coverage is a major feedback mechanism in fuzzing (see Section 2.1). A single code coverage bitmap for all processes, ignoring that the same (user-space) code address can be used by different processes, conflates identical virtual PCs that actually belong to different processes, producing many spurious collisions. To mitigate such a problem, STAFF tracks code module loading, recording for each process and each module the inode related to the loaded binary file, the start address, and the length of the text section. When a basic block executes, the code coverage bitmap is updated according to an index that depends on a combination of the normalized[11] program counter and the inode related to the module of currently running process. Moreover, code coverage of kernel space and common shared libraries is filtered out to avoid noisy and non-informative coverage.

4.4.5. Crash filtering & deduplication

Detecting and correctly triaging crashes in full-system fuzzing requires distinguishing true input-induced faults from unrelated emulation errors. To this end, a dedicated *Crash Filtering & Deduplication* pipeline was implemented.

STAFF identifies any user-space fatal fault inside the firmware emulation environment. Indeed, the Linux kernel used for emulation is instrumented to invoke a stub in `do_coredump()` whenever a process receives a fatal signal (e.g., `SIGSEGV`, `SIGBUS`). This guarantees that both long-running daemons and short CGI helpers are trapped immediately at the point of failure, without relying on delayed user-space detection. Moreover, throughout execution, STAFF maintains a circular buffer of the last N executed basic blocks for each distinct process. Upon trap, the content of the buffer is used as the *crash fingerprint*. An example of crash fingerprints is shown below:

```
=== Fingerprint #0 ===
Faulty process: setup.cgi
 [00] inode: 24746, pc: 0x00012244, module: setup.cgi
 [01] inode: 24746, pc: 0x00008384, module: setup.cgi
 [02] inode: 24746, pc: 0x000106d8, module: setup.cgi
 [03] inode: 24746, pc: 0x000106cc, module: setup.cgi
 [04] inode: 24746, pc: 0x00011d10, module: setup.cgi

=== Fingerprint #1 ===
Faulty process: potcounter
 [00] inode: 24728, pc: 0x0000070c, module: potcounter
 [01] inode: 24728, pc: 0x000006d0, module: potcounter
 [02] inode: 24728, pc: 0x000007c4, module: potcounter
 [03] inode: 24728, pc: 0x0000099c, module: potcounter
 [04] inode: 24728, pc: 0x00000a04, module: potcounter
```

As illustrated above, different processes may fault during a fuzzing attempt. In `Fingerprint #0`, the failure arises in the `setup.cgi` process, while in `Fingerprint #1` it occurs in the `potcounter` process. This highlights a key aspect of full-system fuzzing: a single mutated input may concurrently trigger distinct failures across multiple services running inside the emulated firmware. By fingerprinting each process's recent basic block history independently, the fuzzer is able to detect, classify, and deduplicate such multi-process crashes. Two

[11] To cope with address layout randomization.

crashes are thus considered equivalent when their *crash fingerprints* match, even across different fuzzing iterations. Unlike stack traces, which are often unavailable or unreliable in stripped firmware binaries, the histories of recently executed basic block are lightweight and consistently available.

One major complication is that some processes may fault deterministically during system initialization (e.g., due to missing peripherals or incomplete one-time setup). Such crashes are unrelated to input mutations and must not be reported as bugs. To filter them, a *dry-run phase* is performed before fuzzing: every seed in the initial corpus (assumed by definition unfaulty) is executed once, and all crash fingerprints observed during this phase are collected into an ignore list that is used to suppress spurious crashes. For example, the `potcounter` fault shown in `Fingerprint #1` above occurs consistently even during dry runs, and is therefore classified as an emulation artifact rather than a true crash.

However, this fingerprinting strategy may not always be sufficient for perfect deduplication. Indeed, two distinct crashes that follow different execution traces may converge on the same recent basic block history would lead to identical fingerprints. In such cases, different crashes are not reported separately but merged. This limitation is accepted as a trade-off: by restricting attention to the immediate control-flow execution history preceding the fault, the approach often provides good strategy to cheaply cluster duplicated instances of the same crash.

4.5. Discussion

Table 1 compares the features of STAFF with TriforceAFL and AFLNet, two prominent fuzzing frameworks relevant to our context.

Request modeling is crucial for achieving reproducible interactions in stateful applications. TriforceAFL is unaware of the underlying protocol logic, blindly breaking interactions into different inputs. AFLNet implements a session-aware mode but still depends on unreliable timeout heuristics to delimit responses. In contrast, STAFF exploits Session-Aware HTTP Request Modeling (Section 4.3.1) to reconstruct complete request/response exchanges using protocol-semantic rules rather than timing.

Sequence Minimization is a capability unique to STAFF (Section 4.4.1). It filters out irrelevant requests from the original interactions without harming intra- and inter-service dependencies. This minimization reduces replay cost, potentially increasing the number of fuzzing attempts.

Mutation strategies represent another essential difference. TriforceAFL integrates classical mutations from AFL, which do not always work well with stateful applications. AFLNet devises protocol-aware mutations that can drastically improve fuzzing effectiveness; however, it is not aware of which parts of the inputs affect the firmware execution. STAFF complements protocol-aware mutations with taint-guided mutations (Section 4.4.3), prioritizing inputs that impacted a large number of basic blocks and achieving finer-grained guidance than stateful heuristics alone.

Code coverage tracking also shows a fundamental divergence. TriforceAFL and AFLNet do not discriminate between different processes, experiencing an excessive number of collisions in their coverage bitmap. STAFF is process-aware and attempts to minimize bitmap collisions (Section 4.4.4).

Fuzzing efficiency is improved by STAFF's Multi-Staged Forkserver (Section 4.4.2), which supports forking even to intermediate execution states. This amortizes the cost of long input sequence replay and enables many mutation trials from the same intermediate execution state. AFLNet and TriforceAFL only support traditional single-stage forkservers.

Crash triage presents another key distinction. TriforceAFL and AFLNet treat faulty inputs as distinct crashes when their coverage bitmaps do not perfectly match. While this strategy is simple, it often fails to deduplicate crashes: small, early, and irrelevant divergences during execution may lead to retaining different crashing inputs even when they relate to the same bug. While state-of-the-art user-mode fuzzers often exploit stack traces to improve deduplication, system-mode emulation makes stack trace construction non-trivial and costly. STAFF performs crash triaging in distinct phases (Section 4.4.5): filtering and deduplication. In the first one, it discards spurious failures that do not depend on the mutated input. In the second one, it deduplicates crashes based on recent basic block execution history, which can significantly help group crashes related to the same bug.

5. Implementation

We developed STAFF using approximately 1100 lines of Python and 4000 lines of C/C++ code; in addition, the experimental and analysis setup comprises about 7000 lines of Python.[12] This section provides several details behind the implementation of STAFF, highlighting the key changes we made on top of existing rehosting, taint, and fuzzing frameworks, and the optimizations necessary to keep the system scalable.

5.1. Whole-system taint analysis

A major effort during the development of STAFF was targeted at improving the whole-system taint analysis layer based on QEMU (Bellard, 2005) from DECAF++ (Davanian et al., 2019). To catch taint as soon as values flow into userland, STAFF injects taint at the kernel-level write completion handler by instrumenting `address_space_write_continue()` from QEMU. This guarantees that values mapped into user memory from MMIO, kernel mediation, or other kernel sources are tainted immediately upon kernel-to-user transfer. Since QEMU emits multi-byte load and store operations, STAFF fragments them into byte-level sub-operations, emitting one tainted memory-access event per byte. We further added explicit support for unaligned partial-word semantics (SWL/SWR) and injected taint operations directly into the translation-block generation path through `tcg_emit_op()`. These steps reduced missed flows and significantly increased the fidelity of our byte annotations.

To capture filesystem operations (Section 4.3.2), STAFF instruments primary read and write syscalls, recording writer–reader relationships between requests. The resulting dependency map is filtered to ignore irrelevant files such as shared libraries or timezone files. This filtering mitigates over-dependencies and produces minimized sequences that remain compact and focused on causally relevant writers. In practice, this choice substantially reduced noisy prefix and suffix retention without missing genuine dependencies.

5.2. Byte annotation heuristic optimizations

Two components of our implementation are particularly sensitive to combinatorial blowup: the trie construction performed by the procedure *IndexRequestByteSequencesInTrie* in Algorithm 1 and the trie-based subsequence matching used to infer inter-request dependencies in Algorithm 5. We applied a number of optimizations to keep memory and CPU costs bounded while maximizing precision.

During trie construction, procedure *IndexRequestByteSequencesInTrie* considers request subsequences up to a capped maximum length. Moreover, when the memory usage during trie construction exceeds a user-defined threshold, the trie construction is aborted and re-attempted with a halved maximum length.

During trie-based subsequence matching, Algorithm 1 is quadratic with respect to $|E|$ and may quickly become intractable. We bound ℓ to an interval such that extremely short subsequences, which tend to match many positions, are avoided, while overly long ones are

[12] https://github.com/alessioizzillo/STAFF

Table 1
Comparison of core features across TriforceAFL, AFLNet, and STAFF.

Feature	TriforceAFL (Plc, 2017)	AFLNet (Pham et al., 2020)	STAFF
Request Modeling	No	Timeout	Session Aware
Mutations	AFL	State Aware	State Aware + Taint Guided
Sequence Minimization	No	No	Yes
Forkserver	Single Stage	Single Stage	Multi Stage
Code Coverage Tracing	Single process	Single process	Multiple processes
Crash Filtering	No	No	Spurious Crashes
Crash Deduplication	Coverage Bitmap	Coverage Bitmap	BB History

capped. A further optimization is to stop as soon as the first unique trie match is found (line 6). This early exit avoids unnecessary exploration of additional subsequences once a reliable mapping has already been discovered.

5.3. Taint hints ordering

The *SortFilterHints* procedure (Algorithm 7) performs the length-based ordering and the overlap filtering as described in Section 4.3.4. In our implementation, we further sort the filtered hints at the end of the procedure to later guide the fuzzing scheduler: STAFF prioritizes hints covering more basic blocks, with the intuition that mutating their bytes may impact a large number of basic blocks and possibly expose crashes faster. By applying this ordering we noticed that it may help reduce the time-to-exposure (TTE) of crashes on some firmware images.

6. Evaluation

This section evaluates the effectiveness of STAFF when considering real-world Linux-based MIPS firmware images. Our goal is to validate the ability of STAFF in discovering real-world firmware bugs and the efficiency of the proposed ideas. In more detail, we summarize the evaluation of STAFF around the following research questions:

- **RQ1:** How precise is the *Byte Annotation Heuristic* compared to traditional taint analysis, and what is the impact of *Sequence Minimization*?
- **RQ2:** Does STAFF improve crash discovery in terms of number of crashes found, crash reproducibility, and time-to-exposure, compared to the state of the art?
- **RQ3:** What types of bugs does STAFF discover, and how do case studies demonstrate the importance of dependency-aware fuzzing for multi-request inter-binary bugs?
- **RQ4:** Why is STAFF able to discover these crashes?
- **RQ5:** What is the individual contribution of each STAFF component (taint guidance, Sequence Minimization, Multi-Staged Forkserver) to crash discovery, consistency, and time-to-exposure?

Before presenting our experimental results, we describe our experimental setup, including the considered dataset.

6.1. Experimental setup

In the following, we describe the experimental environment and runtime parameters, the dataset construction and filtering criteria, and the selection of competitor baselines. We also summarize the modifications required to run all approaches in the same full-system emulation setting, ensuring an apples-to-apples comparison.

6.1.1. Experimental environment

Experiments ran on dual-socket Intel Xeon Gold 6238R CPU (56 cores). To ensure reproducibility and safe parallelism, the FirmAE image-preparation pipeline was extended to emit per-tool artifacts and package each firmware run into isolated Docker environments: per-tool databases replaced the previous monolithic PostgreSQL store to remove global contention, and each fuzzer instance executes in its own container with an independent network namespace and resource limits.

Selected runtime parameters were chosen to balance fidelity, reproducibility, and throughput. The minimum subsequence length ℓ for the *Intra- and Inter-Service Dependency Analysis* was set to 3 to filter out 1- and 2-byte trivially short matches that produce most ambiguous trie hits. Tracing of shared libraries was disabled to avoid noisy coverage from commonly-invoked library code and focus instrumentation on application binaries. Each fuzzing trial ran for 24 h (per-trial budget), and individual test cases were bounded by a 150 s per-input timeout to prevent hung or excessively slow emulations from stalling the campaign. The time spent on the *Intra- and Inter-Service Dependency Analysis* is accounted for within STAFF's 24 h budget. As a result, STAFF has less effective fuzzing time than the baselines, which begin fuzzing immediately, making the comparison fair. Coverage bitmap sizes were tuned per firmware in the range 2^{22}–2^{25} to reduce collisions for larger code execution while keeping RAM usage reasonable. AFL's timing calibration was disabled and queue shuffling turned off to improve repeatability in our multi-process emulation environment, where scheduling nondeterminism otherwise produces unstable timing baselines. AFLNet inputs are delimited by the four-byte marker `0x1A1A1A1A` to separate request boundaries reliably. Each mutation was applied up to three times to elicit multiple behaviors from the same edit, and the fuzzing dictionary contained a single long token (1260 bytes) intended to exercise common lexical fields without introducing many short, extraneous tokens.

6.1.2. Dataset

The evaluation corpus (Table 2) is a carefully curated subset of firmware images drawn from the FirmAE project (Kim et al., 2020). Since STAFF targets MIPS-based firmware, we first excluded ARM-based images; after this architectural filtering, the FirmAE corpus contains 661 Linux-based MIPS firmware images spanning routers, range extenders, access points, and IP cameras from multiple vendors. We then selected a subset of firmware applying the following functional criteria: (i) successful boot and initialization without kernel panics or persistent service failures; (ii) a reachable administrative web interface excluding purely static landing pages; (iii) at least one complete web interaction sequence that can be recorded as a PCAP trace and deterministically replayed; (iv) transparent authentication mechanisms based on standard HTTP requests (e.g., GET/POST with credentials), excluding browser-specific dialogs or client-side encrypted tokens; (v) absence of selective client filtering such as user-agent checks or bot detection; and (vi) availability of valid administrative credentials beyond the login phase. Beyond these functional requirements, additional firmware images were excluded due to compatibility constraints introduced by whole-system instrumentation. As described in Section 4.2, enabling DECAF++-based taint analysis requires substituting FirmAE's default QEMU system binary and kernel images with instrumented versions. These substitutions caused initialization failures, kernel incompatibilities, or behavioral differences affecting firmware that passed functional filtering. These engineering constraints required additional filtering to ensure stability during intensive instrumentation and long-running fuzzing campaigns. These selection criteria yield a reproducible, realistic dataset appropriate for comparing stateful, multi-request fuzzing techniques.

Table 2
Dataset of Linux-based MIPS firmware images used in the evaluation.

Firmware	Vendor	Version	Type	Architecture	Captured sessions
RT_N10U	ASUS	300 437.3754	Router	MIPSEL	4
RT_N53		300 437.3754	Router	MIPSEL	4
DAP-2310	D-Link	v1.00_o772	Access Point	MIPSEB	4
DIR-300		v1.03_7c	Router	MIPSEB	4
DIR-815A1		FW104b03	Router	MIPSEL	5
RE1000	Linksys	1.0.02.001_US_20120214	Range Extender	MIPSEL	2
WRT320N		1.0.05.002_20110331	Router	MIPSEL	4
DGN3500	Netgear	V1.1.00.30_NA	Router	MIPSEB	5
DGND3300		V1.1.00.22_NA	Router	MIPSEB	5
JNR3210		V1.1.0.14	Router	MIPSEB	4
Archer C2	TP-Link	v1_160128	Router	MIPSEL	4
TL-WPA8630		V2_171011	Range Extender	MIPSEB	4
TV-IP121WN	TRENDnet	1.2.2	IP Camera	MIPSEB	4
TV-IP651WI		V1_1.07.01	IP Camera	MIPSEL	4
TEW-652BRU		1.00b12	Router	MIPSEB	4

6.1.3. Competitors

The evaluation compares STAFF against two representative protocol-aware and system-mode baselines, respectively, AFLNet (Pham et al., 2020) and TriforceAFL (Plc, 2017). Both baselines were adapted to run in a full-system, multi-process emulation context so that comparisons are fair and apples-to-apples.

AFLNet was ported to system-mode (referred to in the paper as AFLNet†) by replacing its original qemu-user workflow with a FirmAE-based system VM that hosts a forkserver. The port includes the harness which coordinates QEMU and the forkservers via signals to spawn/exit worker children that execute mutated inputs. This allowed AFLNet† to exercise full-system behavior (multiple daemons, inter-process interactions) rather than a single userland process.

TriforceAFL is derived from a full-system FirmAFL variant and was adapted to the multi-process setting by improving its `accept` syscall detection heuristic and multi-process-handling logic so that network-driven requests are recognized reliably in the emulated environment. These changes were necessary to make TriforceAFL operate robustly when multiple processes run concurrently.

These baselines are selected because they represent complementary dimensions of firmware fuzzing: TriforceAFL preserves full-system emulation but performs single-step mutations without protocol awareness, while AFLNet supports stateful multi-request fuzzing but operates within a single-process context. STAFF combines both capabilities to target stateful, multi-binary firmware bugs.

To ensure a fair comparison on coverage-driven feedback, both AFLNet† and TriforceAFL are aligned with STAFF's *Whole-system Code Coverage Tracing* (see Section 4.4.4). This unification prevents cross-process pollution and reduces bitmap collisions, so differences in results reflect methodological choices (taint guidance, Sequence Minimization, Multi-Staged Forkserver) rather than artifacts of mismatched tracing implementations.

Furthermore, all three approaches are initialized from the *same* manually collected seed corpus for each firmware image. This design choice ensures that the observed differences in bug discovery stem exclusively from each tool's fuzzing methodology rather than from seed quality or diversity. In particular, STAFF's improvements over the baselines cannot be attributed to richer starting inputs, since all tools operate on identical initial corpora.

6.2. RQ1: Byte annotation precision

An essential trait of STAFF is byte annotations, which serve as the backbone for taint hints, which in turn allow STAFF to perform targeted mutations based on firmware execution behavior. These byte annotations are obtained by combining a coarse-grained dynamic taint analysis (using a single taint source) with a value matching strategy designed to refine taints and recover fine-grained byte-level dependencies.

Table 3
Precision of the byte annotations before and after *Sequence Minimization*, and speedup of replaying the minimized sequence compared to the original one.

Firmware	Byte annotation	Seq. minimization	
	Precision	Speedup	Precision
RT-N10U	0.71	11.56×	0.92
RT-N53	0.87	8.28×	0.87
DIR-815	0.94	1.45×	0.97
DAP-2310	0.92	1.62×	0.93
DIR-300	0.88	4.61×	0.91
RE1000	0.99	4.96×	0.91
WRT320N	0.86	5.41×	0.90
DGN3500	0.93	9.29×	0.89
DGND3300	0.90	4.29×	0.73
JNR3210	0.88	10.22×	0.95
Archer C2	0.73	10.78×	0.84
TL-WPA8630	0.91	16.68×	0.92
TV-IP121WN	0.91	14.29×	0.94
TV-IP651WI	0.46	1.47×	0.95
TEW-652BRU	0.91	6.95×	0.90
Geo Mean	0.84	5.82×	0.90

This approach has been designed as a nice trade-off between accuracy and scalability compared to traditional dynamic taint analysis, which may assign distinct taint sources to each input byte but would suffer from scalability limitations. However, this raises a natural question: *how precise is the approximated taint analysis from* STAFF *compared to full taint analysis with byte-level taint sources?* In the remainder of this section, we aim to provide an answer, considering also the Sequence Minimization phase that further affects the taints.

In more detail, we designed an experiment that, given a taint hint h generated by STAFF for the input bytes $[h.\texttt{offset}, h.\texttt{offset} + h.\texttt{len} - 1]$, considers the related byte annotations that lead to hint creation, collecting all basic blocks PCs_{STAFF} impacted by the considered input bytes according to the byte annotations. We then run a dynamic taint analysis, marking each input byte in $[h.\texttt{offset}, h.\texttt{offset} + h.\texttt{len} - 1]$ with distinct taint sources, collecting all basic blocks PCs_{DTA} impacted by the considered input bytes according to the fine-grained dynamic taint analysis. To account for execution variability, we repeat the experiments five times and merge basic block lists from different runs. To avoid bias in the taint hint choice, we consider 10 randomly picked taint hints generated during a run over a randomly chosen seed input. Finally, we compute the mean precision across the 10 taints, defined as:

$$\text{precision} = \frac{|PCs_{\text{STAFF}} \cap PCs_{\text{DTA}}|}{|PCs_{\text{STAFF}}|}$$

Since Sequence Minimization may filter out byte annotations, we compute the precision before and after minimization.

Table 4
Average number of unique crashes found during 24-hour runs.

Firmware	TriforceAFL	AFLNet†	STAFF
DGN3500	1.0	4.0	5.2
DGND3300	1.4	3.8	5.2
TV-IP121WN	0.0	1.0	1.0
TV-IP651WI	0.0	0.0	1.0
WRT320N	2.8	1.4	4.8
JNR3210	2.0	2.0	3.6
TL-WPA8630	1.0	2.0	2.0
DAP-2310	0.0	1.0	3.2
DIR-300	0.0	1.0	4.4

Table 3 summarizes the results of the experiment. The mean precision of the byte annotations, before applying Sequence Minimization, is high for most firmware images, suggesting that byte annotations correctly represent, in the majority of cases, real taint dependencies between input bytes and influenced basic blocks and therefore are unlikely to be spurious associations.

When considering the impact of Sequence Minimization, the precision of the surviving byte annotations remains significantly high, with an increase in precision in several firmware images compared to the byte annotations before minimization. This result suggests that the taint hints considered by STAFF faithfully represent taint dependencies. Moreover, the filtering is clearly justified by significant execution speedups during the replay of the minimized sequences compared to the original ones, supporting our design choice.

Finally, we comment on the fact that the experiments do not focus on reporting the recall, i.e., how many taint dependencies are ignored by STAFF. Therefore, the presented results should not be interpreted as a complete evaluation of the heuristic, but rather as an assessment of the usefulness of the retained annotations. This choice is fundamental rather than incidental: measuring recall would require a ground truth enumerating the complete set of taint dependencies between all input bytes and all reachable code locations across stripped, closed-source firmware binaries. Such ground truth is not constructable without full source code and debug symbols, making recall an inherently unmeasurable quantity in this setting. Furthermore, by design, STAFF intentionally discards some dependencies to preserve scalability, so a fraction of false negatives is expected and accepted. To mitigate the lack of this evaluation, in the next research questions, we consider the practical value of the taint hints considered by STAFF when analyzing crashes and bugs, which, in the end, is the ultimate goal of STAFF.

Answer to RQ1

The *Byte Annotation Heuristic* provides a precise approximation of full per-byte dynamic taint analysis. *Sequence Minimization* preserves high precision (mean 0.9 across firmware images) while delivering substantial execution speedup (mean 5.82× improvement) during input replay, significantly enhancing the approach's scalability.

6.3. RQ2: Fuzzing effectiveness

In this section, we investigate whether the ideas proposed by STAFF can actually make it identify more crashes than TriforceAFL and AFLNet†, two existing representative system-mode firmware fuzzing solutions. To this aim, we executed the different fuzzing frameworks over our dataset for 24 h, repeating each run five times.

Table 4 presents the average number of unique crashes discovered by the different approaches during the five runs for each firmware from our dataset.[13] STAFF can consistently identify more crashes than the competitors, finding on average across the 15 firmware 2.03 crashes, compared to 1.08 and 0.55 for AFLNet† and TriforceAFL, respectively. STAFF discovered crashes in 9 out of the 15 firmware images in the dataset, demonstrating its effectiveness across diverse targets. These results suggest that the features of STAFF can indeed make a difference in crash discovery.

To have a more fine-grained investigation of the unique crashes discovered by the different tools, Table 5 refines the analysis of the experimental results by reporting for each firmware, for each unique crash, and for each approach, the Time-To-Exposure (TTE) of the crash and the number of times the crash was identified during the five runs. Furthermore, in the table, each crash is identified by the binary and function where it is located and a categorization. Table 5 also reports the minimum number of requests required to reproduce each crash, obtained through manual minimization of the proof-of-concept sequences. This metric provides empirical validation of our crash categorization: single-request bugs require one request, while multi-request bugs require between 2 and 8 requests to trigger the crash, demonstrating the complexity of the attack chains involved. In particular, we consider the following crash categories:

- **OIB** *One-request, Intra-binary*. A single request is able to trigger the crash inside a network-facing process.
- **OID** *One-request, Inter-binary Direct*. A single request is able to trigger the crash inside a child spawned by a network-facing process.
- **OII** *One-request, Inter-binary Indirect*. A single request is able to trigger the crash inside a binary that is not spawned by the network-facing process.
- **MIB** *Multi-request, Intra-binary*. Several requests are needed to trigger the crash inside a network-facing process.
- **MID** *Multi-request, Inter-binary Direct*. Several requests are needed to trigger the crash inside a child spawned by a network-facing process.
- **MII** *Multi-request, Inter-binary Indirect*. Several requests are needed to trigger the crash inside a binary that is not spawned by the network-facing process.

These categories differ with respect to the number of requests needed to trigger the crash and where the crash occurs (the network-facing process, a child of the network-facing process, or another unrelated process). Fuzzers like TriforceAFL implicitly focus on OIB and OID crashes, where the crash is exercised by a single request and primarily affects the network-facing process (or its children). Protocol-aware fuzzers like AFLNet extend their attention to MIB crashes, which are triggered by multiple requests and affect the network-facing process. Our system-mode port of AFLNet, AFLNet†, should nonetheless identify MID and MII crashes. STAFF naturally considers all the categories but embraces design choices that focus on crashes triggered by multiple requests, regardless of the crashing process.

The per-crash breakdown in Table 5 confirms the fuzzers' nature. TriforceAFL can indeed find only OIB and OID crashes, with a TTE that can be for a few crashes even smaller than what is recorded for AFLNet† and STAFF. This is consistent with its lack of multi-request reasoning and single-process fuzzing scope. Overall, it finds 8 OIB crashes and 1 OID crash. However, it still misses 4 OIB crashes and 1 OID crashes that are found by STAFF, likely due to the lack of taint-guided mutations. Interestingly, TriforceAFL quickly finds one OIB crash in TL-WPA8630 that is missed by the other two fuzzers. No OII crashes have been identified by STAFF, TriforceAFL and AFLNet†.

AFLNet† is significantly more effective than TriforceAFL when considering crashes requiring multiple requests. Indeed, it can find 8 OIB crashes, 2 OID crashes, 15 MID crashes, and 3 MII crashes. This substantial improvement is likely due to being protocol-aware, which allows it to perform mutations on the multiple requests composing the interactions, preserving the protocol state.

STAFF further improves on AFLNet†, discovering 11 OIB crashes, 3 OID crashes, 21 MID crashes, and 7 MII crashes. The improvement

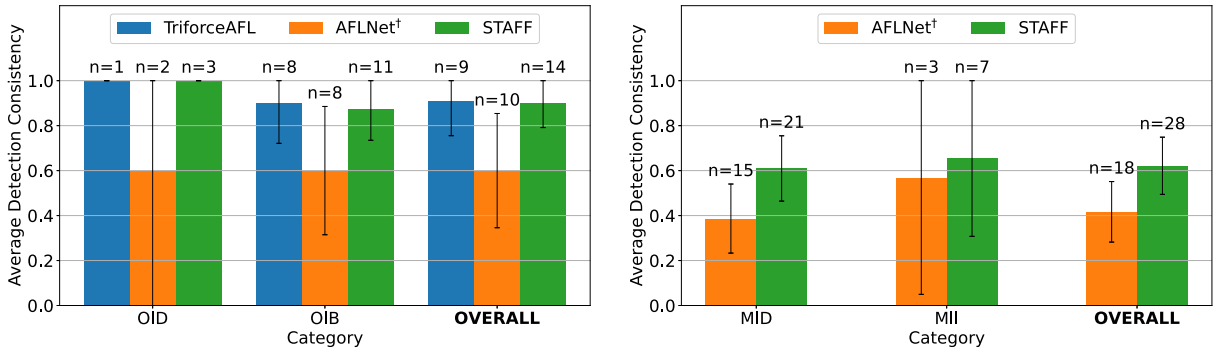
[13] Firmware images without a crash are omitted.

Table 5
Unique crashes found by the three approaches.

Firmware	Binary	Function	Crash category	# Min Reqs	TriforceAFL #	TriforceAFL TTE	AFLNet† #	AFLNet† TTE	STAFF #	STAFF TTE
DGN3500	mini_httpd	FUN_00410e6c	OIB	1	5	4 h 25 m	5	0 h 58 m	5	2 h 22 m
	rc	get_sd_info	MII	6	0		4	3 h 32 m	5	5 h 34 m
	setup.cgi	addkeyword	MID	2	0		0		3	16 h 39 m
		delete	MID	2	0		2	7 h 1 m	0	
		find_val	MID	2	0		2	10 h 18 m	1	22 h 31 m
		html_parser	MID	2	0		5	1 h 33 m	5	1 h 3 m
		save	MID	2	0		0		2	19 h 27 m
		set_SRouteMetric	MID	2	0		0		5	6 h 3 m
		set_TimeZone	MID	2	0		1	4 h 17 m	0	
		set_rule_out	MID	2	0		1	10 h 50 m	0	
DGND3300	mini_httpd	FUN_004036e8	OIB	1	2	0 h 34 m	0		0	
		FUN_0040427c	OIB	1	5	0 h 56 m	2	11 h 4 m	2	19 h 29 m
	setup.cgi	addkeyword	MID	2	0		1	13 h 19 m	3	12 h 53 m
		del_list	MID	2	0		3	2 h 20 m	3	13 h 56 m
		find_val	MID	2	0		0		1	22 h 27 m
		get_WAN_ipType	MID	2	0		1	17 h 51 m	1	18 h 47 m
		html_parser	MID	2	0		5	0 h 35 m	5	0 h 57 m
		save	MID	2	0		2	14 h 29 m	2	5 h 19 m
		set_SRouteMetric	MID	2	0		0		5	4 h 15 m
		set_TimeZone	MID	2	0		1	0 h 19 m	1	17 h 41 m
		set_WAN_ipType	MID	2	0		1	4 h 37 m	0	
		set_rule_in	MID	2	0		2	7 h 44 m	0	
		set_rule_out	MID	2	0		1	0 h 45 m	3	9 h 47 m
TV-IP121WN	network.cgi	main	OID	1	0		0		5	0 h 33 m
	view.cgi	main	OID	1	0		1	14 h 59 m	0	
TV-IP651WI	alphapd	websCgibinProcessor	MID	2	0		0		3	1 h 13 m
WRT320N	httpd	get_cgi	OIB	1	5	16 h 30 m	0		4	4 h 29 m
		valid_hwaddr	OIB	1	0		5	4 h 49 m	5	1 h 14 m
		validate_cgi	OIB	1	5	16 h 21 m	0		5	2 h 56 m
		validate_forward_single	OIB	1	0		2	6 h 55 m	5	2 h 17 m
		validate_merge_ipaddrs	OIB	1	4	16 h 38 m	0		5	0 h 52 m
JNR3210	mini_httpd	FUN_00403acc	OIB	1	0		0		3	9 h 35 m
		FUN_00405c88	OIB	1	5	1 h 1 m	5	0 h 30 m	5	2 h 25 m
	setup.cgi	html_parser	OID	1	5	0 h 34 m	5	1 h 31 m	5	7 h 37 m
		upgrade_main	OID	1	0		0		5	4 h 28 m
TL-WPA8630	httpd	FUN_004057d0	OIB	1	0		1	2 h 2 m	5	13 h 49 m
		FUN_0040edfc	OIB	1	0		2	4 h 38 m	4	10 h 55 m
		FUN_0041685c	OIB	1	0		1	4 h 36 m	0	
		FUN_00425f90	OIB	1	5	0 h 1 m	0		0	
	ledschd	FUN_00402320	MII	2	0		2	4 h 11 m	1	6 h 2 m
DAP-2310	atp	ExeShell	MID	2	0		0		2	8 h 56 m
		redirect_htm	MID	2	0		0		5	3 h 4 m
		rpsubname	MID	2	0		0		1	23 h 20 m
	udhcpd	FUN_00403448	MII	3	0		2	6 h 58 m	2	3 h 23 m
	xmldb	FUN_00408618	MII	3	0		0		5	2 h 31 m
		FUN_00408b50	MII	2	0		0		1	17 h 8 m
DIR-300	atp	ExeShell	MID	2	0		0		3	9 h 5 m
		do_xgi	MID	2	0		0		5	1 h 43 m
		redirect_htm	MID	2	0		1	11 h 34 m	5	1 h 19 m
	udhcpd	FUN_00402f70	MII	4	0		0		4	4 h 17 m
	xmldb	FUN_0040b7e4	MII	2	0		0		5	1 h 20 m

likely results from the effective combination of state-aware mutations and taint-guided mutations. Moreover, not only does it find more unique crashes, but it can find them more consistently across runs. Indeed, for several crashes, STAFF can find the crashes during most or all runs. However, there are still some crashes that it can find only within specific runs, highlighting the non-deterministic nature of fuzzing.

This residual run-to-run variability is a well-recognized property of full-system firmware fuzzing: kAFL (Schumilo et al., 2017) explicitly identifies asynchronous interrupts and kernel background threads as sources of non-determinism that perturb coverage and crash discovery, and Nyx-Net (Schumilo et al., 2022) describes persistent targets as "noisy" because background threads may be scheduled independently of the fuzzed input. To quantify the impact of this variability on our results, we computed 95% confidence intervals (CIs) for *detection consistency* using the t-distribution across the five independent runs. Detection consistency is the per-crash rediscovery rate across runs; for each crash category and method, we calculated the mean detection rate, standard error, and confidence bounds, clamped to $[0, 1]$. The results are presented in Fig. 10 with error bars showing the 95% CIs, separated by crash complexity, and each bar annotated with the number of unique crashes discovered by that method in the corresponding group. For one-request crashes (Fig. 10a), STAFF achieves a mean consistency of 0.90 (CI: [0.79, 1.00]) across 14 discovered crashes, compared to AFLNet†'s 0.60 (CI: [0.35, 0.85]) across 10 crashes. TriforceAFL shows high consistency (0.91, CI: [0.76, 1.00]) but discovers only 9



(a) One-request crashes (b) Multi-request crashes

Fig. 10. Detection consistency with 95% confidence intervals across crash categories (5 independent runs). Error bars show 95% CIs (clamped to $[0, 1]$). Each bar is annotated with the number of *unique crashes* contributing to that category for the given method. (a) One-request crashes: all three methods compared. (b) Multi-request crashes: only AFLNet† and STAFF are shown, since TriforceAFL fails to find multi-request crashes.

crashes, all in the simpler OIB category, demonstrating that while all methods perform well on simpler crashes, STAFF achieves both broader discovery and higher reliability. For multi-request crashes (Fig. 10b), STAFF's advantage is more pronounced: overall consistency reaches 0.62 (CI: [0.49, 0.75]) across 28 unique crashes versus AFLNet†'s 0.42 (CI: [0.28, 0.55]) across 18 unique crashes, with non-overlapping CIs confirming that the improvement is statistically robust and not an artifact of random variation.

Finally, STAFF maintains a nice balance between crashes discovery effectiveness and TTE efficiency since it finds significantly more crashes than the competitors and for most crashes exhibits on average comparable TTEs.

Although STAFF applies conservative approximations at both the dynamic taint analysis level (Section 4.3.2) and the heuristic level when generating taint hints (Section 4.3.3), our experimental results show that these approximations do not undermine crashes discovery in practice. Despite the possibility of false negatives in dependency tracking, STAFF consistently discovers more crashes than competing approaches across the evaluated Linux-based MIPS firmware images. This suggests that the retained taint hints, while not exhaustive, are sufficiently precise and informative to effectively guide mutation and exploration, achieving a practical balance between precision, scalability, and coverage.

Answer to RQ2

STAFF can find 367% and 50% more unique crashes than TriforceAFL and AFLNet†, respectively. This improvement does not come at the cost of efficiency, since, on average, it exhibits a comparable TTE. Compared to TriforceAFL (limited to one-request crashes), STAFF discovers substantially more crashes, albeit with slightly lower consistency on those simple cases. Compared to AFLNet†, STAFF finds more unique crashes across both categories (one-request: 14 vs. 10; multi-request: 28 vs. 18) and re-discovers them more consistently across runs (90.0% vs. 60.0% for one-request; 62.1% vs. 41.7% for multi-request), making it a more dependable solution for complex multi-request bugs.

6.4. RQ3: Bug case studies

The 42 crashes reported in Table 4 represent the output of the automated crash filtering and deduplication pipeline (Section 4.4.5): each corresponds to a unique crash fingerprint— a distinct recent basic-block execution history at crash time — identified across the five fuzzing runs. Following the fuzzing campaigns, we performed a comprehensive crash triage analysis to consolidate the discovered crashes into distinct bugs through manual verification. During this analysis, multiple crashes carrying distinct fingerprints were found to share the same root cause (e.g., triggering the same memory corruption in the same function via different request prefixes or orderings) and were therefore merged into a single bug entry, yielding the 25 unique bugs presented in Table 6. Among these 25 bug entries, 24 were discovered by STAFF. ID 5 bug was not discovered by STAFF, but is included for completeness. To make this characterization more explicit, column STAFF Discovery from the table indicates whether the bug was discovered by STAFF, discovered exclusively by STAFF, or not discovered by STAFF. The difference between 42 crash fingerprints and 25 bugs thus reflects the natural convergence of different stateful execution paths onto the same underlying defect. For each identified bug, we attempted to determine whether it corresponded to a previously disclosed vulnerability with an assigned CVE identifier or represented a potentially new zero-day bug. The CWE classifications were assigned through a systematic analysis that combined multiple sources of evidence: (i) kernel log information capturing the signal that terminated the process (such as `SIGSEGV` for segmentation faults), (ii) the crash fingerprint derived using the methodology described in Section 4.4.5, (iii) and the actual proof-of-concept seed that triggered the crash during the fuzzing experimentation. This multi-faceted approach to classification enabled us to categorize bugs according to the CVE taxonomy, providing insight into the types of software flaws that STAFF successfully uncovered across the tested Linux-based MIPS firmware images. The complete results of this analysis are presented in Table 6. We conservatively omit from the table the crashes that cannot be reproduced in a deterministic manner, as further manual investigation is required.

In the following analysis, we analyze the bugs found by STAFF. For the sake of brevity, we focus on the OID, MID, and MII bug categories, which are the categories not explicitly considered by the state-of-the-art of firmware fuzzing and that were central in inspiring the design of STAFF. For each category, we provide one case study. The OID bug presented is a known vulnerability, while MID and MII bugs are previously unknown bugs submitted for disclosure.

6.4.1. [Oid] TV-IP121WN: stack BOF in network.cgi (Table 6, bug ID: 6)

The OID bug was found in the firmware image of the TRENDnet IP camera TV-IP121WN. The crash is triggered by a single HTTP POST processed by the webserver `boa` and then handled by the CGI binary `network.cgi`, thus it was categorized as an *inter-binary* issue (`boa` → `network.cgi`). In particular, the attacker can craft a malformed POST value `ip4`, which after being processed but not sanitized by `boa` and passed to `network.cgi`, results in a stack buffer overflow

Table 6
CVE and CWE summary.

ID	Firmware	Bug category	Binary	Functions	# Min Reqs	CVE	CWE	STAFF discovery
0	DGN3500	OIB	mini_httpd	FUN_00410e6c	1	/	476	Yes
1	DGN3500	MII	rc	get_sd_info	6	/	476	Yes
2	DGN3500	MID	setup.cgi	addkeyword, delete, find_val, html_parser, save, set_SRouteMetric, set_TimeZone, set_rule_out	2	/	476	Yes
3	DGND3300	OIB	mini_httpd	FUN_004036e8, FUN_0040427c	1	/	787, 476, 191, 20	Yes
4	DGND3300	MID	setup.cgi	addkeyword, del_list, find_val, get_WAN_ipType, html_parser, save, set_SRouteMetric, set_TimeZone, set_WAN_ipType, set_rule_in, set_rule_out	2	/	119	Yes
5	TV-IP121WN	OID	view.cgi	main	1	/	476, 20	No
6	TV-IP121WN	OID	network.cgi	main	1	2018–19240	119	Only
7	TV-IP651WI	MID	alphapd	websCgibinProcessor	2	/	125	Only
8	WRT320N	OIB	httpd	get_cgi, valid_hwaddr, validate_cgi, validate_forward_single, validate_merge_ipaddrs	1	/	476, 121, 822	Yes
9	JNR3210	OIB	mini_httpd	FUN_00403acc	1	/	476	Only
10	JNR3210	OIB	mini_httpd	FUN_00405c88	1	/	476	Yes
11	JNR3210	OID	setup.cgi	html_parser	1	/	476	Yes
12	JNR3210	OID	setup.cgi	upgrade_main	1	/	119	Only
13	TL-WPA8630	OIB	httpd	FUN_004057d0, FUN_0041685c	1	2024–57357	78	Yes
14	TL-WPA8630	OIB	httpd	FUN_0040edfc, FUN_00425f90	1	2023–27837	77	Yes
15	TL-WPA8630	MII	ledschd	FUN_00402320	2	/	476	Yes
16	DAP-2310	MID	atp	ExeShell	2	2016–1558	119	Only
17	DAP-2310	MID	atp	redirect_htm, rpsubname	2	/	121, 476, 20	Only
18	DAP-2310	MII	udhcpd	FUN_00403448	3	/	476	Yes
19	DAP-2310	MII	xmldb	FUN_00408618	3	/	121	Only
20	DAP-2310	MII	xmldb	FUN_00408b50	2	/	120, 121	Only
21	DIR-300	MID	atp	ExeShell, redirect_htm	2	/	120, 121, 787	Yes
22	DIR-300	MID	atp	do_xgi	2	/	119	Only
23	DIR-300	MII	udhcpd	FUN_00402f70	4	/	476	Only
24	DIR-300	MII	xmldb	FUN_0040b7e4	2	/	119	Only

within the function `ipCat()`, corrupting the return address and thus generating a `SIGSEGV`. Since `boa` respawns the CGI on exit, repeated POSTs can repeatedly crash the CGI process, yielding a reliable remote DoS. Unfortunately, the bug occurs even with unauthenticated requests: `network.cgi` crashes even without valid session credentials because authentication is enforced after CGI invocation.

In more detail, the unauthenticated POST request is:

❶ *Poisoned IP value.* The attacker issues an HTTP POST to `/admin/network.cgi` where `ip1`, `ip2`, `ip3` can be benign values but `ip4` is a large attacker-controlled string. An example of the request is:

```
POST /admin/network.cgi HTTP/1.1
Host: 127.0.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: <...>
iptype=0&ip1=192&ip2=168&ip3=10&ip4=
[>30 bytes attacker-controlled string]
&...other fields...
```

The webserver `boa` performs repeated `strlen`, `malloc`, and `memcpy` operations to build CGI inputs and then directly executes in a child the CGI process `network.cgi`. The `main` function from `network.cgi` loops over the form fields (executing several `strcmp` calls), records indices into an array, and then calls `ipCat()` to assemble dotted-IP strings into stack buffers:

```
/* stack locals in decompilation */
undefined auStack_348 [32]; /* destination buffer */
...
/* assembly of dotted IPs (called after parsing) */
ipCat(auStack_348,0x1e, ip1_ptr, ip2_ptr, ip3_ptr,
ip4_ptr);
```

The destination length argument is `0x1e` (30) while each octet pointer is taken verbatim from the CGI input prepared by `boa`; no per-octet length or character-class checks occur prior to the call. `ipCat()` uses unbounded formatting/concatenation (for example, `sprintf(dst, ‘‘%s.%s.%s.%s’’, . . . ))` and can therefore write arbitrarily long strings into the small `dst`. A long `ip4` component overruns `auStack_348`, corrupting adjacent stack data (including the saved return address), and causes a crash when control returns.

6.4.2. [MID] DAP-2310: BOF in atp (Table 6, bug ID: 17)

The MID bug was found in the firmware image of the D-Link DAP-2310 access point. The crash occurs inside the web/CGI handling process `xgi` but requires a multi-stage request flow: a web page emits an unescaped query, the CGI parser in the native `atp` module accumulates parameters without bounds checks, and finally `redirect_htm` formats an HTTP header using unsafe string functions. The exploit is deterministic and proceeds in two steps:

1. *Authentication.* An initial login POST is required to obtain a session, as in the motivating example bug (Section 3).
2. *Client-side redirect construction.* The attacker can cause the browser to issue a GET request to `cfg_valid .xgi` with query parameters of unbounded length. An example of the request is:

```
GET /cfg_valid.xgi?random= [>256 bytes]
&exeshell=submit%20COMMIT&exeshell= [>256 bytes]
%20WLAN&flag=1 HTTP/1.1
```

Both the `random` parameter and each `exeshell` argument may contain arbitrarily long attacker input. These strings are injected verbatim by `cfg_valid.php`, which generates the redirect without encoding or length checks:

```
var str="cfg_valid.xgi?random="+generate_random_str();
function exe_str(str_shellPath) {
 myShell = str_shellPath.split(";");
 for(i=0; i<myShell.length; i++) {
  temp_str+="&"+"exeshell="+myShell[i];
 }
 return temp_str;
}
...
echo "str+=exe_str(\"submit COMMIT;\".$SUBMIT_STR.\"\");\n";
```

Because the values are inserted unescaped, the CGI parser receives an oversized request. During parsing, the `atp` module copies names and values into fixed buffers using unsafe functions. Finally, `redirect_htm` builds an HTTP 302 response header using two adjacent stack buffers:

```
char acStack_37c[100];
char local_318;
/* 256-byte "value buffer" */
undefined auStack_317[255];
char local_218;
/* 516-byte "header buffer" */
undefined auStack_217[515];
strcat(&local_218,"HTTP/1.0 302 Found\n");
/* Overflow */
sprintf(&local_318,"Location: %s\r\n", param_1);
strcat(&local_218,&local_318);
fputs(&local_218,stdout);
```

The 256-byte value buffer is the first overflow point: a long `param_1` string from the query (e.g., oversized `random` or `exeshell`) causes `sprintf` to overrun it. The 516-byte header buffer is subsequently at risk when concatenating the expanded `Location:` line; once the value buffer is corrupted, the header assembly may propagate the corruption further. Thus the attacker controls the crash via unbounded query parameters that flow from PHP into the parser and finally into `redirect_htm`.

6.4.3. [MII] DIR-300: NULL dereference in udhcpd (Table 6, bug ID: 23)

The MII bug that we consider was found in the firmware images of the D-Link DIR-300. The chain of this bug spans several processes: the HTTP server, the CGI handler and the DHCP daemon. In particular, the crashing DHCP process `udhcpd` has no direct process relationship with the HTTP daemon `httpd`, since it was as a daemon from the `init` process. The bug exploit requires a four-step sequence:

1. *Authentication.* The attacker first obtains a valid web session by logging in:

```
POST /login.php HTTP/1.1
Host: 192.168.0.1
Content-Type: application/x-www-form-urlencoded
ACTION_POST=LOGIN&LOGIN_USER=admin&LOGIN_PASSWD=&login=
Log+In
```

2. *Poisoned in-memory runtime configuration.* Using the authenticated session, the attacker issues a POST that writes a set of data form fields into an in-memory runtime configuration stored under the `tmpfs`-mounted `/runtime`. On the DIR-300 this is handled by `set_temp_nodes.php`, which stores each `d_<i>_<j>` form field verbatim under paths such as `/runtime/post/session_<sid>/entry:<i>/data_<j>`. This files likely represent the internal XML database used to store the configuration. Example of the request (truncated):

```
POST /set_temp_nodes.php HTTP/1.1
...
TEMP_NODES=/runtime/post/session_1&data=4&start=1
&d_1_2=192.168.0.45&d_1_3=ce%3A92%3Abf%3Ad1%3A09%3Acf
...
&d_3_2=192.168.0.3&d_3_3= [attacker-controlled string]
&end=25
```

The script stores the posted values without server-side sanitization, so a crafted or overly long token injected here becomes persisted in the runtime configuration.

3. *Poisoned persistent configuration.* When the LAN/DHCP form is submitted through the request (simplified):

```
POST /bsc_lan.php HTTP/1.1
Host: 192.168.0.1
Content-Type: application/x-www-form-urlencoded

ACTION_POST=SOMETHING&...&startipaddr=192.168.0.100&
endipaddr=192.168.0.199&...
```

the execution of the `bsc_lan.php` handler is triggered, which copies the runtime configuration into the persistent configuration files under the NVRAM path `/lan` (e.g., `/lan/dhcp/server/pool:1/staticdhcp/entry:<N>/ {hostname, ip, mac}`) via the `set` function. Unfortunately, `bsc_lan.php` copies values read from `/runtime/post/...` into the `/lan/...` files without further validation, so any poisoned string from step (2) is now part of the device configuration.

4. *Configuration commit and crash.* Finally, a GET request to `bsc_lan.xgi` issues the commit of the configuration through `exeshell=submit%20COMMIT` and restarts the DHCP service through `exeshell=submit%20DHCPD`:

```
GET /bsc_lan.xgi?random_num=...&exeshell=submit%20COMMIT
&exeshell=submit%20DHCPD HTTP/1.1
Host: 192.168.0.1
```

This script triggers the execution of the template engine `/etc/templates/dhcp/dhcpd.php`, which reads the persistent `/lan/dhcp/...` files and writes a new `udhcpd` configuration file `/var/run/udhcpd-br0.conf` using unchecked `fwrite/fwrite2` calls, then restarts the DHCP daemon `udhcpd`.

When `udhcpd` parses the poisoned file, it calls its configuration reader (`read_config`) which extracts key–value pairs from each line. For certain malformed or empty values in the configuration file, the pointer to the value returned by the internal lookup function can be `NULL`. This pointer is then passed directly to a resolver wrapper (decompiled as `FUN_00402f70`) and into `inet_aton(param_1, . . . )` without any check for `NULL`, producing a NULL-pointer dereference and causing a `SIGSEGV`:

Table 7
Causality scores per crash category.

Crash category	# Found crashes	Causality score	# Full-causality crashes
OIB	11	0.68	7
OID	3	0.8	2
MID	21	0.83	15
MII	7	0.57	4
ALL	**42**	**0.74**	28

```
undefined4 FUN_00402f70(char *param_1, in_addr *param_2)
{
 int i = inet_aton(param_1, param_2); /* NO NULL CHECK */
 ...
}
```

Answer to RQ3

The bug case studies considered in this section demonstrate that STAFF successfully discovers interesting and non-trivial bugs arising from multi-request sequences and inter-binary interactions, including 20 previously unknown bugs submitted for disclosure (9 discovered exclusively by STAFF) and 4 previously disclosed CVEs (2 found only by STAFF), confirming that our approach achieves the initial goals outlined in this work.

6.5. RQ4: Taint hint causality

In this section, we investigate *why* STAFF was able to discover the crashes discussed in the previous sections. In particular, we analyze the role of taint hints to understand their impact during fuzzing by defining the *causality score*. For each crashing input, we assign a causality score of 1.0 if the input results from at least one taint-guided mutation in the chain of mutations that transformed the original seed into the crashing input, and 0 otherwise. We then compute the average across different runs.

6.5.1. Aggregate analysis

Among all 42 crashes found by STAFF, we found that 66.7% (28) of crashes have full taint causality (score 1.0), indicating they are consistently discovered, even across different runs, through taint-guided mutations. This high proportion of full-causality crashes demonstrates the systematic nature of taint-guided mutation rather than chance discovery. Examining the distribution across crash categories reveals that full causality is particularly prevalent in MID crashes (15 out of 21, 71.4%) and OIB crashes (7 out of 11, 63.6%), while OID (2 out of 3, 66.7%) and MII (4 out of 7, 57.1%) categories also show substantial full-causality rates. Another 9.5% (4) show high causality (scores 0.5–0.99), 7.1% (3) show low causality (scores 0.1–0.49), and 16.7% (7) show no causality (score < 0.1), suggesting these crashes could be discovered through state-aware mutations alone.

Table 7 shows the average causality score when grouping crashes by category. The OIB, OID, and MID categories exhibit high causality, demonstrating that taint analysis is particularly crucial for discovering inter-binary crashes that require precise dependency tracking. The lower MII score reflects cases where complex multi-process chains can sometimes be triggered through state-aware mutations.

6.5.2. Case studies

We now analyze the crashing inputs for the three case studies from RQ3 to understand the precise mechanisms by which taint hints enable crash discovery, examining mutation patterns across all runs for each crash.

For the OID crash from TV-IP121WN, we measured a full causality score of 1.0. The stack buffer overflow demonstrates full causality with consistent targeting of the `ip4` parameter across all 5 runs. In 4 runs, taint hints targeted offset 889 (the `3` in `ip4=30`), while in 1 run they targeted offset 890 (the `0`). Both mutations inserted 1260-byte havoc strings into the IP parameter, triggering the overflow in `ipCat()`. The dependency analysis consistently identified only vulnerable request 18 as required, filtering out all 21 other requests. This precision demonstrates how taint analysis enables surgical targeting of the exact parameter bytes that flow to the vulnerable function, explaining why random mutation of the 399-byte POST body would have minimal success probability.

For the MID crashes from DAP-2310, we also observed a full causality score of 1.0. Full causality manifests through diverse but consistent targeting of query parameters across all 5 runs. The taint hints targeted different offsets within the `exeshell` parameter: offset 113 targeting `DELAY_LAN`, offset 64 and 116 targeting `COMMIT`, and offset 69 targeting the parameter name itself. Crucially, one run (request 34, offset 69) achieved the optimal dependency minimization described in RQ3, filtering out 11 of 14 requests to retain only the essential causal chain. The other runs filtered out 7 requests, still providing significant sequence reduction. This variability shows taint analysis identifying multiple vulnerable injection points within the same attack vector while consistently maintaining the multi-request dependencies necessary for exploitation.

For the MII crashes from DIR-300, we also obtained a full causality score of 1.0. Full causality across all 5 runs shows systematic targeting of MAC address fields in the configuration data (request 32). Taint hints consistently targeted URL-encoded MAC addresses: offsets 679 and 680 in `d_3_3` field, and offset 604 in `d_2_3` field, applying single-byte bitflip mutations. All runs achieved optimal dependency minimization, filtering out 69 of 79 requests to preserve only the 10-request causal chain spanning authentication, configuration poisoning, and commit operations. The precision of targeting specific bytes within hex-encoded MAC addresses (e.g., changing a byte to an invalid one) while maintaining the complex multi-stage dependency chain demonstrates taint analysis' capability to track data flows across the complete 4-step attack sequence.

Answer to RQ4

Taint-guided mutations are responsible for discovering 83.3% of crashes found by STAFF, demonstrating highest effectiveness for multi-request, inter-binary crashes. STAFF's effectiveness stems from systematic identification of relevant input bytes and inter-request dependencies, enabling surgical precision in targeting vulnerable parameters while maintaining necessary multi-request chains.

6.6. RQ5: Ablation study

To isolate the contribution of individual STAFF components, we conducted a systematic ablation study comparing five configurations:

- **NoTaint**: AFLNet† state-aware baseline (i.e., without taint guidance, Sequence Minimization, or Multi-Staged Forkserver);
- **NoOpt**: STAFF with taint guidance only, without Sequence Minimization or Multi-Staged Forkserver;
- **SeqOnly**: STAFF with taint guidance and Sequence Minimization, without the Multi-Staged Forkserver;
- **MultiStageOnly**: STAFF with taint guidance and the Multi-Staged Forkserver, without Sequence Minimization;
- **FullOpt**: STAFF with all components enabled (taint guidance, Sequence Minimization, and Multi-Staged Forkserver).

Table 8
Ablation study: percentage change in crash Discovery, detection Consistency, and TTE relative to the NoTaint baseline, reported separately for One-Request and Multi-Request crash categories. Positive values indicate improvement over NoTaint baseline for all three metrics.

Category	Discovery	Consistency	TTE
NoOpt (Taint Only)			
One-Request	+20.00%	+18.59%	−17.49%
Multi-Request	+41.67%	+52.94%	−63.11%
Overall	+31.82%	+35.03%	−40.32%
SeqOnly (Taint + Sequence Min.)			
One-Request	+20.00%	+31.41%	−17.93%
Multi-Request	+41.67%	+58.82%	−18.95%
Overall	+31.82%	+44.14%	−16.79%
MultiStageOnly (Taint + Multi-Staged Forkserver)			
One-Request	+10.00%	+46.85%	+15.86%
Multi-Request	+66.67%	+45.00%	−53.66%
Overall	+40.91%	+41.94%	−19.69%
FullOpt (Taint + Sequence Min. + Multi-Staged Forkserver)			
One-Request	+20.00%	+53.85%	+39.14%
Multi-Request	+41.67%	+58.82%	−6.55%
Overall	+31.82%	+54.76%	+16.73%

The ablation study was performed on a six-firmware subset: JNR3210, DGND3300, TL-WPA8630, WRT320N, DIR-300, and DAP-2310. For each firmware and configuration, we performed 5 independent 24-hour fuzzing runs (30 runs per configuration).

Table 8 reports three metrics computed relative to NoTaint:

- **Discovery**: percentage change in unique crashes found across all runs;
- **Consistency**: percentage change in mean per-crash detection rate (crashes found/total runs);
- **TTE**: percentage change in geometric mean time-to-exposure. Positive values mean faster discovery.

Metrics are reported for three crash categories: One-Request (OID + OIB), Multi-Request (MID + MII), and Overall.

Comparing NoTaint to NoOpt reveals that taint guidance is essential for multi-request crashes: Discovery improves by +41.67% and Consistency by +52.94% for Multi-Request crashes. However, taint guidance introduces a TTE overhead on the crashes commonly found with the NoTaint baseline (−63.11% for Multi-Request, −17.49% for One-Request), because the duration of the *Intra- and Inter-Service Dependency Analysis* phase, which must complete before fuzzing begins, is counted against the fuzzing budget, effectively delaying the point at which the first mutations are issued.

SeqOnly demonstrates that eliminating redundant prefix and suffix requests significantly mitigates the TTE penalty introduced by taint guidance: Multi-Request TTE improves from −63.11% (NoOpt) to −18.95% (SeqOnly). Consistency also increases (+58.82% vs. +52.94%), indicating that shorter sequences improve reliability, without however affecting Discovery (+41.67%).

MultiStageOnly achieves the highest single-component Discovery improvement for Multi-Request (+66.67%) by enabling faster state restoration and broader exploration of multi-step interaction chains. However, it shows lower Discovery for One-Request (+10.00% vs. +20.00% for NoOpt and SeqOnly). Despite this trade-off, MultiStageOnly achieves a positive TTE for One-Request (+15.86%), indicating that when one-request crashes are found, the snapshot mechanism accelerates their discovery.

The complete configuration FullOpt combines the strengths of all previously discussed configurations by enabling all STAFF components together (taint guidance, Sequence Minimization, and Multi-Staged Forkserver): it matches the Discovery improvements of the individual components (+31.82% Overall) while achieving the highest Consistency (+54.76% Overall, +53.85% One-Request, +58.82% Multi-Request). Most importantly, STAFF achieves positive TTE for Overall (+16.73%) and One-Request (+39.14%), nearly eliminating the Multi-Request TTE overhead (−6.55% vs. −63.11% for NoOpt). This synergy arises from the interplay between the two optimizations: Sequence Minimization reduces the replay cost of taint-tracked sequences, while the Multi-Staged Forkserver amortizes the remaining overhead through fast state restoration. The result is a system that discovers more crashes, does so more reliably, and exposes them faster.

Answer to RQ5

Taint guidance (NoOpt) is essential for multi-request crashes (+41.67% Discovery, +52.94% Consistency) but introduces significant overhead (TTE −63.11%). Sequence minimization (SeqOnly) mitigates this overhead (TTE −18.95%) without sacrificing reliability (+58.82% Consistency). The Multi-Staged Forkserver (MultiStageOnly) maximizes exploration (+66.67% Multi-Request Discovery) but trades off one-request consistency. FullOpt achieves synergy: it matches the best Discovery and Consistency improvements while turning TTE positive (Overall: +16.73%; One-Request: +39.14%), demonstrating that the combination provides the strongest balance between discovery, reliability, and speed.

7. Related work

Software fuzzing is a popular technique that has seen a large body of works in the last two decades. A recent survey (Zhu et al., 2022) systematizes the design space of fuzzing techniques, considering dimensions such as input generation, mutation strategies, feedback signals, and target domains, within which STAFF falls under stateful and protocol-aware fuzzing. More recently, fuzzing research has also explored LLM-assisted approaches (Zhu et al., 2025; Meng et al., 2024), where large language models are used to generate or refine test inputs. While this direction is promising and complementary, STAFF primarily focuses on execution-grounded feedback, leveraging whole-system dynamic taint analysis to guide multi-request mutations in rehosted firmware.

In the remainder of this section, we consider works spanning four main areas closely related to the design of STAFF: protocol-aware and stateful fuzzing, firmware emulation and fuzzing, dataflow-guided analysis for multi-step vulnerabilities, and checkpoint approaches for fuzzing.

7.1. Protocol-aware and stateful fuzzers

Early protocol fuzzers, such as Sulley (Amini et al., 2017) or Boo-Fuzz (Pereyda, 2015), relied on manually written grammars or templates. AFLNet (Pham et al., 2020) introduced coverage- and state-feedback for network protocols by replaying and mutating recorded message sequences and using server responses to guide exploration. StateAFL (Natella, 2022) extends this idea by hashing memory snapshots to infer protocol states. SGFuzz (Ba et al., 2022) further advances automatic state identification by detecting enumerated-type state variables through static analysis and constructing a State Transition Tree at runtime to steer the fuzzer towards under-explored state sequences, without requiring manual annotations. Recent embedded- and web-focused tools improve the quality of inputs by exploiting request correlations or semantics: CinfoFuzz (Feng and Dong, 2022) leverages web-service correlation information for embedded web interfaces, SR-Fuzzer (Zhang et al., 2019) applies semantic models and automated recovery to test physical SOHO routers, and app-driven or blackbox approaches, such as IoTFuzzer (Chen et al., 2018), Diane (Redini et al., 2021), and Snipuzz (Redini et al., 2021), use companion apps or

response-based inference to produce higher-quality test cases without firmware access.

These systems improve per-service state exploration and input validity, but typically operate on a single daemon or device channel. STAFF generalizes protocol-aware ideas to whole, rehosted Linux firmware by discovering dependencies across services (network, filesystem, IPC) and synthesizing multi-request, cross-component sequences for stateful fuzzing.

7.2. Firmware emulation and fuzzing

Large-scale firmware testing depends on robust rehosting. Firmadyne (Chen et al., 2016) pioneered automated extraction and system-mode emulation for Linux firmware; FirmAE (Kim et al., 2020) boosted emulation success through arbitration heuristics. TriforceAFL (Plc, 2017), FirmFuzz (Srivastava et al., 2019), and related projects integrate fuzzing into QEMU-based environments to allow full-system testing. Among these, TriforceAFL represents a canonical baseline for full-system firmware fuzzing, as it preserves realistic execution environments but performs single-step mutations without protocol or state awareness. In contrast, AFLNet (Pham et al., 2020) targets stateful network protocols but operates within a single-process context. These two tools therefore capture complementary dimensions of firmware fuzzing that STAFF aims to unify. Throughput-oriented efforts — FIRM-AFL (Zheng et al., 2019), EQUAFL (Zheng et al., 2022), SAFIREFUZZ (Seidel et al., 2023) — explore hybrid or near-native execution to reduce emulator overhead and speed fuzzing campaigns. For MCU or bare-metal firmware, automated peripheral-modeling/re-hosting — Laelaps (Cao et al., 2020), P2IM (Feng et al., 2020), Jetset (Johnson et al., 2021), µEmu (Zhou et al., 2021), HALucinator (Clements et al., 2020), Fuzzware (Scharnowski et al., 2022) — infer or synthesize peripheral behavior to make firmware executable without per-device engineering. Firmulti (Cheng and Cheng, 2023) demonstrates system-level VMI monitoring for multi-process firmware analysis.

These works solve orthogonal problems of making firmware executable and improving fuzzing scalability or throughput. In particular, Greenhouse (Tay et al., 2023) and EQUAFL (Zheng et al., 2022) deliberately avoid full-system complexity by rehosting individual services in user space, which improves performance but prevents modeling vulnerabilities that span multiple cooperating binaries. FirmSolo (Angelakopoulos et al., 2023) focuses on kernel module analysis rather than user-space services, while AflIot (Du et al., 2022) performs on-device single-binary fuzzing without inter-process modeling. FirmAEHF (Zhao and Zhang, 2025) targets verification of known vulnerabilities using predefined proof-of-concept inputs and does not automatically discover novel multi-step inter-binary bugs. FIRMCORN (Gui et al., 2020) applies evolutionary fuzzing strategies to single binaries without protocol-state awareness. Firmulti (Cheng and Cheng, 2023) extends crash detection across processes but does not address stateful multi-request scenarios. HouseFuzz (Xiao et al., 2025) performs multi-process service fuzzing with protocol-aware Token Dependency Graphs that model control-flow and parameter dependencies within protocol parsing logic, enabling generation of syntactically valid multi-request sequences. However, it does not track how specific input bytes propagate across process boundaries in full-system firmware environments. Pandawan (Angelakopoulos et al., 2024) advances rehosting completeness through improved initialization detection but does not address the fuzzing methodology challenges of multi-request dependency-aware testing. In contrast, STAFF assumes Linux-based firmware can be rehosted and focuses on exercising interactions among co-running services by combining whole-system taint-driven dependency discovery with stateful multi-request fuzzing.

7.3. Dataflow-guided fuzzing

Process-level taint- and dataflow-guided fuzzers, such as VUzzer (Rawat et al., 2017), Angora (Chen and Chen, 2018), RedQueen (Aschermann et al., 2019), markedly improve the ability to bypass complex checks by identifying influential input bytes. Whole-system taint engines such as DECAF++ (Davanian et al., 2019) enable VM-level tracking across kernel and userspace, while static multi-binary analyses — Karonte (Redini et al., 2020), SaTC (Chen et al., 2024) — surface insecure inter-binary flows in firmware. Recent work stresses the importance of multi-step, cross-component vulnerabilities: HOFF (Yu et al., 2021a) detects higher-order memory corruption by tracking data flows from attacker-controlled stores to memory-critical operations, and ReLink (Yu et al., 2021b) uncovers higher-order command injection by linking requests across interfaces. Complementary to taint analysis, lightweight runtime sanitizers — e.g., DFirmSan (Yang et al., 2025) — aim to increase observable fault signals in firmware fuzzing with reduced overhead.

Unlike single-process taint-guided fuzzers or tools that only detect multi-step issues, STAFF integrates whole-system taint into an active fuzzing loop: it uses cross-component taint to (i) identify which bytes and requests influence code regions, (ii) prioritize mutations accordingly, and (iii) construct minimal, checkpointed multi-request sequences that exercise multi-step behaviors across daemons.

7.4. Multi-staged and snapshot-based fuzzing

Efficient state restoration is essential for fuzzing throughput, especially for stateful targets requiring complex initialization or protocol state traversal. AFL's original forkserver (Zalewski, 2021) established a single-stage design by forking at a fixed execution point (e.g., program entry or first syscall), yielding modest speedups over full re-execution. AFL++ extends this model with persistent mode (Fioraldi et al., 2020b), eliminating fork overhead for stateless targets and achieving 10–20× speedups. Several works optimize snapshot placement for network services: SnapFuzz (Andronidis and Cadar, 2022) introduces deferred forkservers via binary rewriting (62.8× speedup for LightFTP), while Snappy (Geretto et al., 2022) uses dynamic taint analysis to identify optimal checkpoint locations. However, these approaches rely on a single checkpoint per execution and thus remain inefficient for multi-step protocol fuzzing. VM-level snapshot systems offer coarser-grained state restoration. Nyx-Net (Schumilo et al., 2022) achieves up to 300× speedup over AFLNet using incremental whole-VM snapshots, and Agamotto Song et al. (2020) introduces hierarchical checkpoint trees with delta-based restoration. These techniques typically checkpoint at system-level events (e.g., boot or packet boundaries) rather than fine-grained protocol states. STAFF's Multi-Staged Forkserver differs in three key aspects. First, instead of a single fork point or coarse VM snapshot, it employs hierarchical two-tier checkpointing dynamically placed at HTTP request boundaries. Second, unlike user-space checkpointing mechanisms (e.g., CRIU, DMTCP) limited to single processes, it preserves complete QEMU system state, including all daemons, IPC channels, filesystem modifications, and network sockets. Third, unlike single-binary multi-state fuzzers such as yFuzz (Chang et al., 2024), STAFF targets multi-binary firmware and uses whole-system taint analysis to identify inter-binary interactions spanning HTTP servers, CGI handlers, and backend daemons.

8. Limitations

While STAFF demonstrates significant improvements in firmware fuzzing effectiveness, several limitations constrain its applicability and scope.

Limited device scope. Our evaluation focuses on MIPS-architecture Linux-based firmware from routers, range extenders, and IP cameras. This represents only a subset of the embedded device ecosystem, excluding ARM/x86 architectures, RTOS/bare-metal systems, and other device categories. The generalizability to modern firmware with security features like encryption remains unvalidated. Beyond architectural and ecosystem constraints, the evaluation set was further reduced by engineering limitations related to whole-system taint analysis. As described in the *Dataset* paragraph of Section 6.1, DECAF++ instrumentation requires substituting FirmAE's standard QEMU binary and kernel images with ABI-compatible instrumented versions. Some otherwise functional firmware images exhibited boot failures or instability under this instrumentation and were therefore excluded, resulting in a final dataset of 15 firmware targets. The dataset also exhibits a temporal bias: images range from 2011 to 2017, as older firmware is more reliably emulated by FirmAE (Kim et al., 2020) due to better-understood kernel versions and hardware assumptions. Nevertheless, drawing from the FirmAE corpus is standard practice in the field — FIRM-AFL (Zheng et al., 2019), Greenhouse (Tay et al., 2023), EQUAFL (Zheng et al., 2022), HouseFuzz (Xiao et al., 2025), and Pandawan (Angelakopoulos et al., 2024) all use or directly extend it — and the exposed vulnerability classes remain prevalent in contemporary IoT security research.

Limited service entry-point coverage. STAFF currently targets only the HTTP daemon as the fuzzing entry point; backend daemons are exercised indirectly through HTTP-driven inter-process interactions. Extending the approach to multiple simultaneous entry-point services (e.g., SNMP alongside HTTP) would require coordinating request sequences across daemons with asynchronously evolving internal states, introducing non-trivial scheduling challenges that go beyond the scope of this work. Moreover, the overhead of whole-system taint analysis is expected to grow with the number of concurrently active services. We acknowledge this as a current limitation and identify multi-entry-point scheduling as a key direction for future work.

Emulation and protocol dependencies. STAFF inherits the fundamental challenges of firmware rehosting, limiting applicability to devices requiring specialized peripherals or precise timing.

Moreover, in its current implementation, STAFF targets management traffic exchanged over plaintext HTTP, which allows it to focus on dependency discovery and stateful multi-request fuzzing without introducing additional protocol-specific complexity. The current HTTP-focused implementation excludes firmware using other protocols such as FTP, DHCP, or SMTP, for which protocol-specific session modeling would be required to correctly capture request boundaries, state transitions, and dependencies. In principle, the same approach could be extended to encrypted HTTP traffic by introducing a decryption layer upstream to STAFF's workflow.

Taint analysis and byte annotation heuristic approximations. STAFF's dependency analysis deliberately trades precision for scalability and therefore introduces approximations at two levels. At the dynamic taint analysis (DTA) level, DECAF++ tracks taint presence rather than fine-grained per-byte provenance. This binary taint propagation may miss some true dependencies due to incomplete instruction-level propagation rules, conservative handling of implicit flows (i.e., control-flow dependencies), or architectural corner cases. These limitations may lead to false negatives when tainted data influences execution without being explicitly propagated. Quantifying such missed dependencies is extremely difficult in practice, as it would require complete ground-truth knowledge of all data dependencies within complex firmware binaries. At the heuristic level, STAFF's Byte Annotation Heuristic further introduces conservative filtering to improve precision and scalability when converting taint information into actionable taint hints. In particular, dependencies are retained only when a unique byte-pattern match exists, very short matches are discarded, cross-request dependencies are preserved only under a first-writer condition, and overlapping hints are filtered. These design choices intentionally reduce noise and false positives but may discard valid dependencies, introducing additional false negatives. False positives may also arise at both levels. At the DTA level, STAFF inherits the known limitations of DECAF++'s coarse-grained taint representation, which can conservatively over-approximate data influence. At the heuristic level, value-based matching with a single taint source may incorrectly associate input bytes with code locations when similar byte patterns appear in unrelated contexts or when execution exhibits minor non-determinism. We evaluate the impact of such heuristic-level false positives empirically in Section 6.2, using full (non-approximated) dynamic taint analysis as ground truth. Overall, these approximations represent deliberate design trade-offs. Although STAFF may miss some dependencies, implementing a fully precise variant (e.g., many taint sources and no filtering) proved prohibitively slow and failed to scale across firmware images. Despite these limitations, the results in Section 6.3 show that STAFF consistently outperforms prior approaches in vulnerability discovery, indicating that the retained taint hints remain sufficiently informative in practice.

Computational overhead. Despite optimizations, whole-system taint analysis introduces substantial overhead compared to application-level fuzzing, potentially limiting adoption in resource-constrained environments. The dependency analysis phase requires significant computational resources for large firmware images.

Seed collection limitations. STAFF's effectiveness depends on the quality and diversity of the initial seed corpus. As no pre-existing seed sets were available for the evaluated firmware images, seeds were collected via extensive manual interaction with each device's HTTP management interface. One author acted as a black-box security analyst, systematically exploring exposed functionalities without relying on internal device knowledge or reverse engineering artifacts. Seed collection followed common security testing practices: enumerating accessible web pages, exercising representative workflows (e.g., wireless configuration, firewall rules, user management, firmware updates), recording complete multi-request interaction sequences, and validating deterministic replay across executions. To avoid shallow seeds limited to authentication or navigation, we invested several hours per firmware image to capture deeper, multi-step interactions exercising diverse functionality. Despite this effort, manual seed collection has inherent limitations. Coverage may remain incomplete for undocumented features, hidden endpoints, or device-specific workflows, and alternative device states (e.g., factory reset or recovery mode) may expose additional code paths not represented in our corpus.

9. Conclusions

This article presents STAFF, a fuzzing framework aimed at Linux-based MIPS firmware images exposing HTTP-based services that control and interact with multiple cooperating daemons. STAFF analyzes the firmware execution to generate *taint hints*, which represent the influence of bytes from specific network requests on specific code portions within the firmware. Taint hints are used to guide the fuzzing by focusing mutations on relevant byte subsequences and replaying only a reduced subset of the original requests necessary to preserve execution context.

An experimental evaluation across a diverse dataset of real-world Linux-based MIPS firmware images shows that STAFF consistently outperforms representative full-system multi-process firmware fuzzing frameworks in both the number and reproducibility of discovered crashes. Through systematic crash triage and manual verification as detailed in Section 6.4, STAFF's 42 discovered crashes consolidate into 20 previously unknown bugs submitted for disclosure (9 of which were discovered exclusively by STAFF) and 4 previously disclosed CVEs (2 of which were found only by STAFF). In particular, STAFF can identify crashes corresponding to bugs requiring complex sequences of network

requests involving different firmware daemons.[14] We investigated several case studies to demonstrate that the discovered crashes and their corresponding bugs indeed depend on the taint hints constructed by STAFF.

Three main future directions emerge from this work. First, STAFF could be extended beyond HTTP to other management protocols (e.g., SNMP), by adding protocol-specific request modeling and replay while preserving the same whole-system dependency discovery and multi-stage checkpointing workflow. Second, migrating the fuzzing engine from AFL to AFL++ (Fioraldi et al., 2020b) would provide access to modern mutation strategies and performance enhancements; AFL was retained in the current implementation solely to maintain a fair, apples-to-apples comparison with TriforceAFL and AFLNet[†], both of which derive from AFL, ensuring that observed differences in results reflect methodological choices rather than differences in the underlying fuzzing infrastructure. Third, extending support to ARM-based firmware — which represents a substantial share of the FirmAE corpus and of the broader IoT device ecosystem — would substantially broaden the evaluation scope; the current restriction to MIPS arises not from a fundamental limitation of DECAF++'s taint engine, which is designed to be architecture-agnostic, but from the engineering effort required to produce ABI-compatible, DECAF++-instrumented QEMU system binaries and kernel images for ARM targets within the FirmAE emulation framework, and we identify this porting and validation work as a priority direction for future work.

CRediT authorship contribution statement

Alessio Izzillo: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Conceptualization. **Riccardo Lazzeretti:** Writing – review & editing, Writing – original draft, Supervision, Investigation. **Emilio Coppa:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Emilio Coppa reports financial support was provided by Italian NRRP MUR program funded by the European Union - NextGenerationEU. Riccardo Lazzeretti reports financial support was provided by Italian NRRP MUR program funded by the European Union - NextGenerationEU. Alessio Izzillo reports financial support was provided by Italian NRRP MUR program funded by the European Union - NextGenerationEU. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by :

- Project FARE (PNRR M4.C2.1.1 PRIN 2022, Cod. 202225BZJC, CUP D53D23008380006, Avviso D.D 104 02.02.2022) under the Italian NRRP MUR program funded by the European Union - NextGenerationEU.
- Project SETA (PNRR M4.C2.1.1 PRIN 2022 PNRR, Cod. P202233 M9Z, CUP B53D23026000001, Avviso D.D 1409 14.09.2022) under the Italian NRRP MUR program funded by the European Union - NextGenerationEU.
- Project SERICS (PE00000014) under the NRRP MUR program funded by the EU-NGEU

Data availability

Code will be released on GitHub.

References

Amini, P., Portnoy, A., Sears, R., 2017. Sulley: Pure python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley. (Online; Accessed 17 September 2025).

Andronidis, A., Cadar, C., 2022. SnapFuzz: high-throughput fuzzing of network applications. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA '22, http://dx.doi.org/10.1145/3533767.3534376.

Angelakopoulos, I., Stringhini, G., Egele, M., 2023. FirmSolo: enabling dynamic analysis of binary linux-based IoT kernel modules. In: Proceedings of the 32nd USENIX Security Symposium. URL https://dl.acm.org/doi/10.5555/3620237.3620518.

Angelakopoulos, I., Stringhini, G., Egele, M., 2024. Pandawan: Quantifying progress in linux-based firmware rehosting. In: Proceedings of the 33rd USENIX Security Symposium. URL https://dl.acm.org/doi/10.5555/3698900.3699228.

Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T., 2019. REDQUEEN: Fuzzing with input-to-state correspondence. In: Proceedings of the 26th Annual Network and Distributed System Security Symposium. NDSS '19, URL https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/.

Asmita, A., Tsang, R., Ghimire, S., Salehi, S., Homayoun, H., 2025. Bare-metal firmware fuzzing: A survey of techniques and approaches. IEEE Access http://dx.doi.org/10.1109/ACCESS.2025.3575691.

Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A., 2022. Stateful greybox fuzzing. In: Proceedings of the 31st USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity22/presentation/ba.

Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I., 2018. A Survey of Symbolic Execution Techniques. ACM Comput. Surv. http://dx.doi.org/10.1145/3182657.

Bekrar, S., Bekrar, C., Groz, R., Mounier, L., 2012. A taint based approach for smart fuzzing. In: Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation. ICST '12, http://dx.doi.org/10.1109/ICST.2012.182.

Bellard, F., 2005. QEMU, a fast and portable dynamic translator. In: Proceedings of the 2005 USENIX Annual Technical Conference. ATEC '05, URL https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

Binosi, L., Mazzini, P., Sanna, A., Carminati, M., Giacinto, G., Lazzeretti, R., Zanero, S., Polino, M., Coppa, E., Maiorca, D., 2024. Do you Trust your Device? Open Challenges in IoT Security Analysis. In: Proceedings of the 21st International Conference on Security and Cryptography. SECRYPT '24, http://dx.doi.org/10.5220/0012856200003767.

Borzacchiello, L., Coppa, E., Demetrescu, C., 2021. Fuzzing Symbolic Expressions. In: Proceedings of the 43rd International Conference on Software Engineering. ICSE '21, http://dx.doi.org/10.1109/ICSE43902.2021.00071.

Broekman, B., Notenboom, E., 2003. Testing Embedded Software. Addison-Wesley.

Cao, C., Guan, L., Ming, J., Liu, P., 2020. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In: Proceedings of the 36th Annual Computer Security Applications Conference. ACSAC '20, http://dx.doi.org/10.1145/3427228.3427280.

Chang, Y., Huang, C.C., Mori, T., Hsiao, H.C., 2024. Poster: Yfuzz: Data-driven fuzzing. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. CCS '24, http://dx.doi.org/10.1145/3658644.3691420.

Chen, P., Chen, H., 2018. Angora: Efficient fuzzing by principled search. In: Proceedings of the 39th IEEE Symposium on Security and Privacy. http://dx.doi.org/10.1109/SP.2018.00046.

Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W.C., Sun, M., Yang, R., Zhang, K., 2018. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium. NDSS '18, http://dx.doi.org/10.14722/ndss.2018.23166.

Chen, L., Wang, Y., Linghu, J., Hou, Q., Cai, Q., Guo, S., Xue, Z., 2024. SaTC: Shared-keyword aware taint checking for detecting bugs in embedded systems. IEEE Trans. Dependable Secur. Comput. http://dx.doi.org/10.1109/TDSC.2023.3307430.

[14] All vulnerability reports, proof-of-concept inputs, and responsible disclosure artifacts are publicly available at https://github.com/alessioizzillo/STAFF/tree/master/found_bugs, organized as one directory per bug. The current CVE triage status as coordinated by VulnCheck is maintained at https://github.com/alessioizzillo/STAFF/blob/master/found_bugs/vulncheck_bug_status.md.

Chen, D.D., Woo, M., Brumley, D., Egele, M., 2016. Towards automated dynamic analysis for linux-based embedded firmware. In: Proceedings of the 23th Annual Network and Distributed System Security Symposium. NDSS '16, http://dx.doi.org/10.14722/ndss.2016.23415.

Cheng, Y.T., Cheng, S.M., 2023. Firmulti fuzzer: Discovering multi-process vulnerabilities in IoT devices with full system emulation and VMI. In: Proceedings of the 5th Workshop on CPS&IoT Security and Privacy. http://dx.doi.org/10.1145/3605758.3623493.

Clause, J., Li, W., Orso, A., 2007. Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. ISSTA '07, http://dx.doi.org/10.1145/1273463.1273490.

Clements, A.A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., Payer, M., 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In: Proceedings of the 29th USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity20/presentation/clements.

Coppa, E., Izzillo, A., 2024. Testing concolic execution through consistency checks. J. Syst. Softw. http://dx.doi.org/10.1016/j.jss.2024.112001.

Coppa, E., Izzillo, A., Lazzeretti, R., Lenti, S., 2023. FuzzPlanner: Visually Assisting the Design of Firmware Fuzzing Campaigns. In: Proceedings of the 20th IEEE Symposium on Visualization for Cyber Security. http://dx.doi.org/10.1109/VizSec60606.2023.00007.

Coppa, E., Yin, H., Demetrescu, C., 2022. SymFusion: Hybrid Instrumentation for Concolic Execution. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE '22, http://dx.doi.org/10.1145/3551349.3556928.

D-Link, 2025. TSD firmware download. https://tsd.dlink.com.tw/. (Online; Accessed 17 September 2025).

Davanian, A., Qi, Z., Qu, Y., Yin, H., 2019. DECAF++: Elastic whole-system dynamic taint analysis. In: Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses. RAID '19, URL https://www.usenix.org/conference/raid2019/presentation/davanian.

Di Bartolomeo, L., Moghaddas, H., Payer, M., 2023. ARMore: Pushing Love back into binaries. In: Proceedings of the 32nd USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity23/presentation/di-bartolomeo.

Dinesh, S., Burow, N., Xu, D., Payer, M., 2020. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In: Proceedings of the 41st IEEE Symposium on Security and Privacy. http://dx.doi.org/10.1109/SP40000.2020.00009.

Du, X., Chen, A., He, B., Chen, H., Zhang, F., Chen, Y., 2022. AflIot: Fuzzing on linux-based IoT device with binary-level instrumentation. Comput. Secur. http://dx.doi.org/10.1016/j.cose.2022.102889.

Duck, G.J., Gao, X., Roychoudhury, A., 2020. Binary rewriting without control flow recovery. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '20, http://dx.doi.org/10.1145/3385412.3385972.

Feng, Q., Dong, W., 2022. CinfoFuzz: Fuzzing method based on web service correlation information of embedded devices. In: Proceedings of the 10th IEEE International Conference on Information, Communication and Networks. ICICN '22, http://dx.doi.org/10.1109/ICICN56848.2022.10006492.

Feng, B., Mera, A., Lu, L., 2020. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: Proceedings of the 29th USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity20/presentation/feng.

Feng, X., Zhu, X., Han, Q.L., Zhou, W., Wen, S., Xiang, Y., 2022. Detecting vulnerability on IoT device firmware: A survey. IEEE/CAA J. Autom. Sin. http://dx.doi.org/10.1109/JAS.2022.105860.

Fioraldi, A., D'Elia, D.C., Coppa, E., 2020a. WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA '20, http://dx.doi.org/10.1145/3395363.3397372.

Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M., 2020b. AFL++: combining incremental steps of fuzzing research. In: Proceedings of the 14th USENIX Conference on Offensive Technologies. WOOT '20, http://dx.doi.org/10.5555/3488877.3488887.

Fredkin, E., 1960. Trie memory. Commun. ACM http://dx.doi.org/10.1145/367390.367400.

Gao, Z., Dong, W., Chang, R., Wang, Y., 2022. Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware. Concurr. Comput.: Pr. Exp. http://dx.doi.org/10.1002/cpe.5756.

Geretto, E., Giuffrida, C., Bos, H., Van Der Kouwe, E., 2022. Snappy: Efficient fuzzing with adaptive and mutable snapshots. In: Proceedings of the 38th Annual Computer Security Applications Conference. ACSAC '22, http://dx.doi.org/10.1145/3564625.3564639.

Godefroid, P., 2020. Fuzzing: hack, art, and science. Commun. ACM http://dx.doi.org/10.1145/3363824.

Google, 2023. Taking the next step: OSS-fuzz in 2023. https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html. Online; (Accessed 17 September 2025).

Gui, Z., Shu, H., Kang, F., Xiong, X., 2020. FIRMCORN: Vulnerability-oriented fuzzing of IoT firmware via optimized virtual execution. IEEE Access http://dx.doi.org/10.1109/ACCESS.2020.2973043.

Hernandez, G., Muench, M., Maier, D., Milburn, A., Park, S., Scharnowski, T., Tucker, T., Traynor, P., Butler, K., 2022. FirmWire: Transparent dynamic analysis for cellular baseband firmware. In: Proceedings of the 29th Annual Network and Distributed System Security Symposium. NDSS '22.

Holt, A., Huang, C.Y., 2014. Embedded Operating Systems. Springer, http://dx.doi.org/10.1007/978-1-4471-6603-0.

Johnson, E., Bland, M., Zhu, Y., Mason, J., Checkoway, S., Savage, S., Levchenko, K., 2021. Jetset: Targeted firmware rehosting for embedded systems. In: Proceedings of the 30th USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity21/presentation/johnson.

Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., Kim, Y., 2020. FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis. In: Proceedings of the 36th Annual Computer Security Applications Conference. ACSAC '20, http://dx.doi.org/10.1145/3427228.3427294.

Liang, J., Wang, M., Zhou, C., Wu, Z., Jiang, Y., Liu, J., Liu, Z., Sun, J., 2022. PATA: Fuzzing with path aware taint analysis. In: Proceedings of the 43rd IEEE Symposium on Security and Privacy. http://dx.doi.org/10.1109/SP46214.2022.9833594.

Lin, C., Li, V., 2024. Botnets continue exploiting CVE-2023-1389 for wide-scale spread. https://www.fortinet.com/blog/threat-research/botnets-continue-exploiting-cve-2023-1389-for-wide-scale-spread. (Online; Accessed 17 September 2025).

Liu, Q., Zhang, C., Ma, L., Jiang, M., Zhou, Y., Wu, L., Shen, W., Luo, X., Liu, Y., Ren, K., 2022. FirmGuide: boosting the capability of rehosting embedded linux kernels through model-guided kernel execution. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering. ASE '21, http://dx.doi.org/10.1109/ASE51524.2021.9678653.

LLVM-project, 2023. LibFuzzer. https://llvm.org/docs/LibFuzzer.html. (Online; Accessed 17 September 2025).

Maier, D., Seidel, L., Park, S., 2020. BaseSAFE: baseband sanitized fuzzing through emulation. In: Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks. http://dx.doi.org/10.1145/3395351.3399360.

Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M., 2019. The art, science, and engineering of fuzzing: A survey. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/TSE.2019.2946563.

Meng, R., Mirchev, M., Böhme, M., Roychoudhury, A., 2024. Large language model guided protocol fuzzing. In: Proceedings of the 31st Annual Network and Distributed System Security Symposium). NDSS '24, URL https://www.ndss-symposium.org/ndss-paper/large-language-model-guided-protocol-fuzzing/.

Nagy, S., Nguyen-Tuong, A., Hiser, J.D., Davidson, J.W., Hicks, M., 2021. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In: Proceedings of the 30th USENIX Security Symposium. URL https://www.usenix.org/system/files/sec21fall-nagy.pdf.

Natella, R., 2022. StateAFL: Greybox fuzzing for stateful network servers. Empir. Softw. Eng. http://dx.doi.org/10.1007/s10664-022-10233-3.

Networks, Z., 2024. Zyxel security advisory for OS command injection vulnerability in aps and security router devices. https://www.zyxel.com/global/en/support/security-advisories/zyxel-security-advisory-for-os-command-injection-vulnerability-in-aps-and-security-router-devices-09-03-2024. (Online; Accessed 17 September 2025).

Newsome, J., Song, D., 2005. Dynamic taint analysis for automatic detection, analysis, and SignatureGeneration of exploits on commodity software. In: Proceedings of the 12th Annual Network and Distributed System Security Symposium. NDSS '05.

Organisation, E.C.S., 2022. Technical paper on internet of things (IoT). https://ecs-org.eu/?publications=technical-paper-on-internet-of-things-iot. (Online; Accessed 17 September 2025).

Pereyda, J., 2015. Boofuzz: A fork and successor of the sulley fuzzing framework. https://github.com/jtpereyda/boofuzz. (Online; Accessed 17 September 2025).

Pham, V.T., Böhme, M., Roychoudhury, A., 2020. AFLNET: A greybox fuzzer for network protocols. In: Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification. ICST '20, http://dx.doi.org/10.1109/ICST46399.2020.00062.

Plc, N.G., 2017. Triforceafl. https://github.com/nccgroup/TriforceAFL. (Online; Accessed 17 September 2025).

Produit, B., Glockow, L., Shriwas, R., 2025. Hexagon fuzz: Full-system emulated fuzzing of qualcomm basebands. https://www.srlabs.de/blog-post/hexagon-fuzz-full-system-emulated-fuzzing-of-qualcomm-basebands. (Online; Accessed 17 September 2025).

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H., 2017. Vuzzer: Application-aware evolutionary fuzzing. In: Proceedings of the 24th Annual Network and Distributed System Security Symposium. NDSS '17, URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/.

Redini, N., Continella, A., Das, D., De Pasquale, G., Spahn, N., Machiry, A., Bianchi, A., Kruegel, C., Vigna, G., 2021. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices. In: Proceedings of the 42nd IEEE Symposium on Security and Privacy. http://dx.doi.org/10.1109/SP40001.2021.00066.

Redini, N., Machiry, A., Wang, R., Spensky, C., Continella, A., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2020. KARONTE: Detecting insecure multi-binary interactions in embedded firmware. In: Proceedings of the 41st IEEE Symposium on Security and Privacy. http://dx.doi.org/10.1109/SP40000.2020.00036.

Research, G.V., 2024. General purpose operating system (GPOS) - embedded software market statistics. https://www.grandviewresearch.com/horizon/statistics/embedded-software-market/operating-system/general-purpose-operating-system-gpos/global. (Online; Accessed 17 Septenber 2025).

Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., Abbasi, A., 2022. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In: Proceedings of the 31st USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski.

Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T., 2017. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In: Proceedings of the 26th USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

Schumilo, S., Aschermann, C., Jemmett, A., Abbasi, A., Holz, T., 2022. Nyx-net: network fuzzing with incremental snapshots. In: Proceedings of the 17th European Conference on Computer Systems. EuroSys '22, http://dx.doi.org/10.1145/3492321.3519591.

Seidel, L., Maier, D.C., Muench, M., 2023. Forming faster firmware fuzzers. In: Proceedings of the 32nd USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity23/presentation/seidel.

Serebryany, K., 2017. OSS-Fuzz - google's continuous fuzzing service for open source software. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany. (Online; Accessed 17 September 2025).

Shen, Y., Xu, Y., Sun, H., Liu, J., Xu, Z., Cui, A., Shi, H., Jiang, Y., 2022. Tardis: Coverage-guided embedded operating system fuzzing. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. http://dx.doi.org/10.1109/TCAD.2022.3198910.

Song, D., Hetzelt, F., Kim, J., Kang, B.B., Seifert, J.P., Franz, M., 2020. Agamotto: accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In: Proceedings of the 29th USENIX Security Symposium. URL https://dl.acm.org/doi/10.5555/3489212.3489355.

Srivastava, P., Peng, H., Li, J., Okhravi, H., Shrobe, H., Payer, M., 2019. FirmFuzz: Automated IoT firmware introspection and analysis. In: Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things. http://dx.doi.org/10.1145/3338507.3358616.

Tay, H.J., Zeng, K., Vadayath, J.M., Raj, A.S., Dutcher, A., Reddy, T., Gibbs, W., Basque, Z.L., Dong, F., Smith, Z., Doupé, A., Bao, T., Shoshitaishvili, Y., Wang, R., 2023. Greenhouse: Single-service rehosting of Linux-Based firmware binaries in User-Space emulation. In: Proceedings of the 32nd USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity23/presentation/tay.

Tekeoglu, A., Tosun, A.Ş., 2016. A testbed for security and privacy analysis of IoT devices. In: Proceeding of the IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems. MASS '16, http://dx.doi.org/10.1109/MASS.2016.051.

TP-Link, 2025. Firmware download center. https://www.tp-link.com/us/support/download/. (Online; Accessed 17 September 2025).

Wright, C., Moeglein, W.A., Bagchi, S., Kulkarni, M., Clements, A.A., 2021. Challenges in firmware re-hosting, emulation, and analysis. ACM Comput. Surv. http://dx.doi.org/10.1145/3423167.

Xiao, H., Wei, Z., Dai, J., Li, B., Zhang, Y., Yang, M., 2025. HouseFuzz: Service-aware grey-box fuzzing for vulnerability detection in linux-based firmware. In: Proceedings of the 46th IEEE Symposium on Security and Privacy. http://dx.doi.org/10.1109/SP61157.2025.00213.

Yang, S., Gao, Y., Kuang, B., Yang, Y., Fu, A., 2025. DFirmSan: A lightweight dynamic memory sanitizer for linux-based firmware. Comput. Secur. http://dx.doi.org/10.1016/j.cose.2025.104467.

Yang, Z., Yang, M., 2012. LeakMiner: Detect information leakage on android with static taint analysis. In: Proceedings of the 3rd World Congress on Software Engineering. WCSE '12, http://dx.doi.org/10.1109/WCSE.2012.26.

Yu, L., Li, L., Wang, H., Wang, X., He, H., Gong, X., 2021a. Towards automated detection of higher-order memory corruption vulnerabilities in embedded devices. In: Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition. DATE '21, http://dx.doi.org/10.23919/DATE51398.2021.9473979.

Yu, L., Wang, H., Li, L., He, H., 2021b. Towards automated detection of higher-order command injection vulnerabilities in IoT devices: Fuzzing with dynamic data flow analysis. Int. J. Digit. Crime Forensics http://dx.doi.org/10.4018/IJDCF.286755.

Yun, J., Rustamov, F., Kim, J., Shin, Y., 2022. Fuzzing of embedded systems: A survey. ACM Comput. Surv. http://dx.doi.org/10.1145/3538644.

Zalewski, M., 2021. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. (Online; Accessed 17 September 2025).

Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C., 2019. The Fuzzing Book. https://www.fuzzingbook.org/.

Zhang, Y., Huo, W., Jian, K., Shi, J., Lu, H., Liu, L., Wang, C., Sun, D., Zhang, C., Liu, B., 2019. SRFuzzer: an automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities. In: Proceedings of the 35th Annual Computer Security Applications Conference. ACSAC '19, http://dx.doi.org/10.1145/3359789.3359826.

Zhang, X., Zhang, C., Li, X., Du, Z., Mao, B., Li, Y., Zheng, Y., Li, Y., Pan, L., Liu, Y., Deng, R., 2024. A survey of protocol fuzzing. ACM Comput. Surv. http://dx.doi.org/10.1145/3696788.

Zhao, X., Zhang, G., 2025. FirmAEHF: A dynamic analysis method for embedded IoT firmware based on simulation and hybrid fuzzing. In: Proceedings of the 6th International Conference on Computer Science, Engineering, and Education. CSEE '25, http://dx.doi.org/10.1109/CSEE64583.2025.00018.

Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L., 2019. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In: Proceedings of the 28th USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.

Zheng, Y., Li, Y., Zhang, C., Zhu, H., Liu, Y., Sun, L., 2022. Efficient greybox fuzzing of applications in linux-based IoT devices via enhanced user-mode emulation. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA '22, http://dx.doi.org/10.1145/3533767.3534414.

Zhou, W., Guan, L., Liu, P., Zhang, Y., 2021. Automatic firmware emulation through invalidity-guided knowledge inference. In: Proceedings of the 30th USENIX Security Symposium. URL https://www.usenix.org/conference/usenixsecurity21/presentation/zhou.

Zhu, X., Wen, S., Camtepe, S., Xiang, Y., 2022. Fuzzing: A survey for roadmap. ACM Comput. Surv. http://dx.doi.org/10.1145/3512345.

Zhu, X., Zhou, W., Han, Q.L., Ma, W., Wen, S., Xiang, Y., 2025. When software security meets large language models: A survey. IEEE/CAA J. Autom. Sin. http://dx.doi.org/10.1109/JAS.2024.124971.