

WiDepFuzz: Finding Wi-Fi Protocol Bugs in Mobile Hotspot via Field Dependency Model

Yuanliang Chen

Renmin University of China
Beijing, China

Fuchen Ma

Tsinghua University
Beijing, China

Taotao Gu*

Tsinghua University
Beijing, China

Yu Jiang*

Tsinghua University
Beijing, China

Abstract

Wi-Fi is a widely used wireless protocol, and mobile hotspot functionality has become essential for on-the-go connectivity. Bugs in the hotspot protocol stack can cause crashes, performance degradation, and semantic inconsistencies. Fuzzing is a promising technique for detecting such vulnerabilities. However, existing fuzzers typically test by modeling protocol inputs but ignore some critical dependencies among Wi-Fi fields, leading to inefficient testing.

In this work, we present WiDepFuzz, a fuzzing framework designed to detect bugs in Mobile Hotspot implementations by modeling field dependencies in the Wi-Fi protocol. First, WiDepFuzz automatically identifies the semantic dependencies among Wi-Fi fields based on the IEEE 802.11 standard and constructs a Field Dependency Model (FModel) to characterize various direct and indirect dependencies. Second, leveraging the FModel, WiDepFuzz utilizes dependency-combination guided fuzzing to constantly generate Wi-Fi test frames that efficiently explore deep paths and trigger hidden bugs. Finally, we propose liveness and dependency detectors that enable real-time monitoring to identify abnormal states and uncover potential bugs in mobile hotspots. We implemented WiDepFuzz and evaluated it in six widely used mobile hotspots, covering major smartphone manufacturers, including Samsung, Xiaomi, OPPO, Honor, Huawei, and Nokia. Compared to SOTA Wi-Fi fuzzers like Peach, OwFuzz, wpaspy, and Greyhound, WiDepFuzz detects 15 more bugs and improves bug detection speed by 37% on average. In total, it uncovered 20 new bugs.

CCS Concepts

• Networks → Protocol testing and verification.

Keywords

Wi-Fi Protocol, IEEE 802.11, Bug Detection

ACM Reference Format:

Yuanliang Chen, Fuchen Ma, Taotao Gu, and Yu Jiang. 2026. WiDepFuzz: Finding Wi-Fi Protocol Bugs in Mobile Hotspot via Field Dependency Model. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3832783.3837483>

*Taotao Gu and Yu Jiang are Corresponding authors.



This work is licensed under a Creative Commons Attribution 4.0 International License. ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
<https://doi.org/10.1145/3832783.3837483>

1 Introduction

Wi-Fi has become the de facto standard for wireless communication, enabling seamless connectivity for billions of devices [29, 30]. Among its applications, mobile hotspot functionality, which allows smartphones to share cellular networks over Wi-Fi, is widely used for mobile work, travel, and emergency connectivity [7, 27]. This communication relies on the IEEE 802.11 protocol [17], which governs device discovery, association, and data exchange. Given its central role, ensuring the correctness and robustness of Wi-Fi implementations, particularly in mobile hotspots, is critical for both usability and security [4, 20].

However, the widespread adoption of mobile hotspots also makes them attractive attack targets. Vulnerabilities in the hotspot protocol stack can be triggered by crafted packets, leading to device crashes, service disruption, or unauthorized access [16, 28]. For example, CVE-2024-20816 [28] is a critical vulnerability in the Auto Hotspot feature of Samsung devices, allowing attackers to connect to the victim's hotspot without their knowledge, resulting in unauthorized network access and privacy breaches. Thus, ensuring mobile hotspot security is crucial to protect wireless networks and connected devices from potential threats.

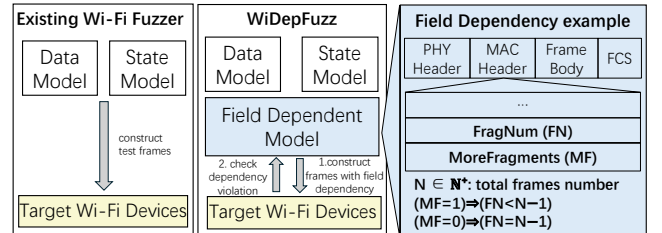


Figure 1: WiDepFuzz leverages the field dependency model not only to construct test frame inputs but also to validate the responses from the target.

Fuzzing is an effective technique for uncovering bugs in network protocols. Prior Wi-Fi fuzzers adopt different strategies and have discovered numerous vulnerabilities across wireless devices. In general, they generate test frames based on two models: a data model, which mutates packet payloads to explore the data plane, and a state model, which explores protocol states and transitions to exercise control logic. However, although these fuzzers (e.g., Greyhound [15]) leverage protocol state machines to produce valid packet sequences, their mutations are largely field-independent and do not explicitly coordinate inter-field constraints. As a result, they often generate inputs that violate dependency rules and lack precise oracles to detect such violations, as shown in figure 1. These relationships, referred to as **field dependencies**, define constraints where the value of one field depends on another. Such dependencies are prevalent in the 802.11 protocol. For example, in the MAC

header, when `MoreFragments` is 1, `FragNum` must be less than the total number of fragments; otherwise, it must equal the total number minus one. Ignoring these field dependencies has two major consequences: (1) it limits the fuzzer's ability to generate semantically valid but rare frame combinations, causing it to miss subtle bugs that only surface under protocol-compliant structures; and (2) it prevents the detection of semantic bugs in mobile hotspots that respond with frames violating these dependencies.

Some prior work has explored dependency awareness in other domains, such as `ViDeZZo` [21] for virtual devices, `TaintBFuzz` [41] for Zigbee stacks, and `FieldsFuzz` [44] for binary files. However, their notion and use of dependencies differ fundamentally from ours. These approaches typically rely on manually annotated grammars or runtime feedback (e.g., taint or coverage) to infer data and control-flow constraints for maintaining input validity. In contrast, such dependency modeling is either incomplete for protocol semantics or relies on assumptions, such as emulation visibility, instrumentation support, or static file structures, that do not hold in our black-box, over-the-air Wi-Fi hotspot setting with interactive, cross-frame, and temporal constraints. Moreover, prior work primarily uses dependencies for input generation, but does not leverage them as semantic oracles for detecting protocol-level violations, limiting their effectiveness in uncovering semantic bugs.

To effectively test and uncover bugs hidden in deep paths in mobile hotspots, there are three main challenges: (1) **The first challenge** is to automatically identify which frame fields exhibit semantic dependencies. Mobile hotspots involve a wide variety of frame field types, and the semantic relationships among them are complex, including explicit direct dependencies and implicit indirect ones. Thus, precisely modeling the field dependency is challenging yet essential. (2) **The second challenge** lies in generating high-quality test frames to efficiently trigger bugs in the target mobile hotspots. The test input space is already large due to the diversity of frame types, field formats, and protocol states. Incorporating field dependency combinations further expands this space, making it hard to explore. (3) **The third challenge** is designing precise detectors to determine whether the mobile hotspot behaves correctly. Some bugs, such as dependency violations, are silent, they do not exhibit obvious symptoms in the early stages after being triggered. As a result, even if such bugs are triggered, they are hard to detect – especially in black-box settings with no visibility into internal states.

To address these challenges, we propose `WiDepFuzz`, a fuzzing framework that detects Wi-Fi protocol bugs in mobile hotspots by constructing a **Field Dependency Model (FDModel)**. First, according to the IEEE 802.11 standard, `WiDepFuzz` automatically analyzes and extracts field-level semantic constraints (e.g., direct value dependencies, indirect state dependencies, etc.). Based on them, `WiDepFuzz` encodes them into a structured dependency graph to serve as the basis for both test generation and response validation. Second, considering that different field combinations in the responses of mobile hotspots reflect distinct semantic states, `WiDepFuzz` adopts a dependency-combination guided fuzzing strategy to efficiently explore the field dependency input space. Finally, to precisely detect bugs, `WiDepFuzz` introduces two anomaly detectors that monitor the runtime behavior of mobile hotspots. A liveness detector checks whether the hotspot remains online and continues

to provide service, while a dependency detector identifies violations of field dependency relationships in the hotspot's response.

We implemented `WiDepFuzz` and evaluated it in six widely used mobile hotspots from various manufacturers (e.g., Samsung [37], Xiaomi [38], OPPO [36], Honor [33], Huawei [34], and Nokia [35]). Compared with state-of-the-art Wi-Fi fuzzers, `Peach` [11], `OwFuzz` [9], `wpaspy` [42], and `Greyhound` [15], `WiDepFuzz` detects 15 more bugs and achieves a 37% improvement in bug detection speed. In total, `WiDepFuzz` discovered 20 previously unknown bugs.

In summary, we make three key contributions:

- We propose a field dependency model (FDModel) and design a dependency-combination guided fuzzing strategy to effectively explore this model.
- We propose two bug detectors for precisely identifying anomalies and bugs in mobile hotspots.
- We implement and evaluate `WiDepFuzz` in six widely used mobile hotspots. We will open-source `WiDepFuzz`¹.

2 Background of Mobile Wi-Fi Hotspot

Mobile Hotspot Protocol Stack: A mobile hotspot integrates multiple networking stack layers to share cellular connectivity over Wi-Fi [31]. At the bottom, the PHY and MAC layers handle wireless transmission, medium access, and management frames according to IEEE 802.11 [13, 17, 19]. Above them, the network layer (e.g., IP stack) provides routing, address configuration, and service discovery via DHCP and NAT. The device thus acts as a Wi-Fi access point and cellular gateway, enabling clients to access the internet [1, 6]. In practice, this architecture should remain reliable under dynamic conditions such as roaming, channel or power changes, fluctuating cellular backhaul, and bursty multi-client traffic. These dynamics stress state machines across layers (e.g., association and RSN handshakes at MAC, address assignment and forwarding at IP), where inconsistencies or semantic violations may cause connection drops, weakened protection, or service disruption. Ensuring semantic correctness and robustness is therefore critical for usability and security in real deployments.

Field Dependencies in Wi-Fi Protocol: Wi-Fi protocols include management, control, and data frames with interdependent fields that must satisfy semantic constraints [24, 45]. For example, in authentication exchanges, the sequence number should follow the progression defined by the selected algorithm, while in QoS frames, the TID field should match the traffic category. These dependencies are implicit in the 802.11 specification and must be handled by transmitters and receivers to ensure protocol consistency and security [5]. However, existing Wi-Fi fuzzers mutate fields independently without coordinating related fields. Consequently, they generate syntactically valid but semantically inconsistent inputs that are often filtered early or fail to exercise deeper protocol logic, missing subtle bugs across state transitions. Moreover, prior work overlooks semantic defects in response frames. A device may accept and emit frames violating dependency constraints without immediate failure, yet these inconsistencies can later cause desynchronization, non-compliant behavior, weakened protection, or crashes. Therefore, a dependency-aware approach is needed to generate high-quality inputs and detect semantic violations in responses.

¹`WiDepFuzz` at: <https://doi.org/10.5281/zenodo.19219328>

3 Overview

3.1 Threat Model

Throughout this paper, we adopt the following threat model. We define a **Mobile Hotspot Protocol Stack Under Test** as $\mathcal{H} = \{F, D, S\}$, where $F = \{f_1, f_2, \dots, f_n\}$ represents the set of 802.11 management, control, and data frames processed by the hotspot device; $D = \{d_1, d_2, \dots, d_m\}$ denotes the set of dependent field relations among these frames, capturing semantic constraints (e.g., $d_i = \langle A \in \text{Set}(a) \Rightarrow B \in \text{Set}(b) \rangle$, which means that when A takes a value from $\text{Set}(a)$, B can only take a value from $\text{Set}(b)$ for the combination to be semantically valid); and $S = \{S_n, S_c, S_o\}$ denotes the state of $\mathcal{H}$: S_n represents the hotspot functioning normally, S_c indicates the hotspot has crashed and gone offline, and S_o indicates that the response frame from $\mathcal{H}$ violates the dependencies D .

Considering that most mobile hotspot implementations are closed-source, we tested $\mathcal{H}$ in a black-box manner using only over-the-air (OTA) frame injection. The attacker (or fuzzer) can inject arbitrary 802.11 frames with customized fields. A **bug** in $\mathcal{H}$ is defined as the mobile hotspot under test entering into State S_c or S_o , and cannot recover to the normal state S_n .

3.2 A Motivation Example

Bugs in mobile hotspot implementations can lead to severe failures such as connection drops, authentication hangs, or privacy leakage, significantly impacting user experience and posing serious security risks. These bugs are difficult to detect for two main reasons: (1) they are hidden in deep paths within the protocol, and (2) dependency violation bugs are often silent initially, making them hard to identify. One such example is found in Honor mobile hotspots, where incorrect SeqNum and Retry handling allows attackers to exploit the flaw, gain unauthorized access to other users' private data, bypassing the intended security measures. Figure 2 illustrates the six key steps to trigger this bug, and Figure 3 presents the core code snippet of this vulnerability.

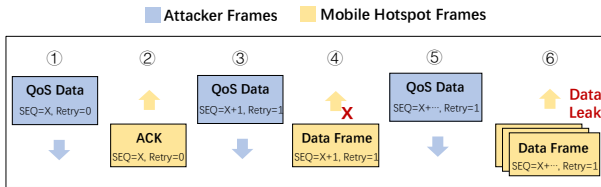


Figure 2: A protocol implementation bug in mobile hotspot. Incorrectly handling the Sequence Number and Retry fields, allowing unauthorized access to the privacy data.

In the Wi-Fi protocol, QoS Data frames carry prioritized traffic, ACK frames confirm successful reception, SEQ numbers track frame order, the Retry bit indicates retransmissions, and the `tx_buffer` temporarily holds frames awaiting transmission. As shown in Figure 2, when a QoS Data frame is transmitted, the hotspot stores it in the `tx_buffer` based on its SEQ number. If a QoS frame is retransmitted with the Retry bit set, the hotspot places it into the `retry_queue` for subsequent retransmission. According to the 802.11 standard, there is a semantic dependency between the Retry and SeqNum fields: when `Retry = 1`, the SeqNum should remain the same as the original frame. However, due to improper handling, when the hotspot

```

1 void retry_frame(uint16_t seqNum) {
2     ...
3     uint16_t index_ = retry_queue->pop();
4     retry_counter[index_]++;
5     -- if (tx_buffer[index_]) {
6     ++ if (tx_buffer[index_] && is_retry_valid(index_)) {
7         transmit_(tx_buffer[retry_index]);
8     }
9 }
10 bool is_retry_valid(uint16_t retry_index) {
11     // Ensure that the Sequence Number remains unchanged for
12     // retry frames
13     return tx_buffer[retry_index]->seq_num <= latest_seq_num;

```

Figure 3: Code snippet showing incorrect construction of protocol fields in the mobile hotspot.

receives a QoS frame with a large SeqNum (e.g., $X+1$), it mistakenly frees `tx_buffer` entries associated with other users' frames. This results in data leakage when these freed entries are later accessed again. This vulnerability is fixed by adding a dependency checker before transmitting a frame to ensure SeqNum remains unchanged for retry, as shown in Lines 6 and 10-13.

Mobile hotspots are commonly used in smartphones and IoT devices to share cellular data over Wi-Fi. Their vulnerabilities can lead to unauthorized access, data breaches, or service disruptions. We can draw **three key observations** from this case: (1) Wi-Fi frames contain many fields with semantic dependencies, which are often implicit, complex, and crucial to correct protocol stack behavior (e.g., Retry and SeqNum in this case). However, existing Wi-Fi fuzzers do not model inter-field dependencies as first-class constraints in either mutation or oracle design. To address this, WiDepFuzz performs static analysis of the Wi-Fi protocol specification (i.e., the IEEE 802.11 standard) to automatically identify such dependencies and construct a FModel that guides the overall testing process. (2) Some bugs are hidden in deep execution paths and can only be triggered by specific sequences of test frames (e.g., the six key steps shown in Figure 2), which significantly increases the input space. To address this challenge, WiDepFuzz adopts dependency-combination guided fuzzing, dynamically prioritizing high-quality test frames that explore diverse and meaningful field dependency relationships. (3) Some vulnerabilities are silent and exhibit no obvious symptoms. Without precise oracles to validate semantic dependency correctness, these bugs can be easily misclassified as benign failures or ignored as noise. To address this issue, WiDepFuzz incorporates liveness and dependency detectors that enable real-time monitoring to accurately identify bugs in mobile hotspots.

4 WiDepFuzz Design

4.1 WiDepFuzz Workflow

Figure 4 illustrates the workflow of WiDepFuzz, which consists of two main phases. The first phase is the FModel construction process. WiDepFuzz analyzes the IEEE 802.11 specification to automatically extract protocol rules, frame structures, and semantic constraints between fields. These constraints are encoded into a field dependency graph that captures both explicit and implicit relationships across different frame types, forming the FModel. The second phase is the testing process, which repeatedly executes

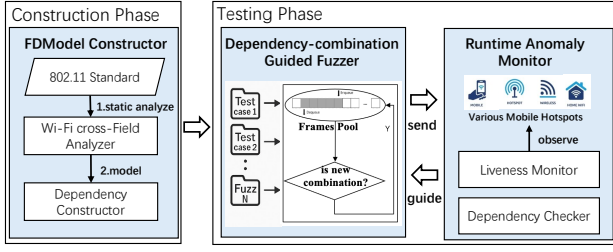


Figure 4: The workflow of WiDepFuzz.

the following six steps to explore the dependency-aware input space: (1) WiDepFuzz generates a batch of initial frame sequences based on the FDModel. These frames are crafted to cover diverse field combinations, including semantically valid and potentially violating inputs. (2) The generated frames are transmitted to the target mobile hotspot using standard Wi-Fi interfaces, emulating realistic interactions. (3) The mobile hotspot processes the incoming frames, triggering corresponding behaviors within its protocol stack, such as state transitions or responses. (4) Meanwhile, two anomaly detectors run in parallel to monitor the hotspot’s behavior: a liveness detector checks whether the hotspot remains online, and a dependency detector checks if the response frames violate known field dependency constraints. (5) WiDepFuzz analyzes the received responses to identify new field dependency combination states or detect semantic anomalies. (6) If a new bug or novel combination is detected, the corresponding input is marked as an interesting seed and prioritized in the frames pool to help guide subsequent test frame generation. WiDepFuzz then proceeds to the next iteration (from step 1 to step 6) until the testing process terminates.

4.2 Field Dependency Model Construction

Since Wi-Fi implementations are expected to follow the IEEE 802.11 standard for interoperable frame processing, WiDepFuzz uses the standard as the primary source for extracting field dependencies. The input is the PDF specification, and the output is an executable set of dependency rules that can be used by both the mutator and the runtime checker. Each rule is represented as a tuple $(f_{src}, f_{dst}, \phi, s)$, where f_{src} is the controlling field or condition, f_{dst} is the constrained field, ϕ is the dependency predicate, and s is an optional frame-type or protocol-state guard.

Types of Field Dependencies: Field dependencies in IEEE 802.11 can be categorized into two types.

(1) *Direct Dependencies.* Direct dependencies describe explicit constraints between fields within the same frame. Formally, a direct dependency exists between two fields f_i and f_j if: $\text{valid}(f_i) \iff C(f_j)$, where C defines the valid value or range of f_i according to f_j . For example, in fragmentation control, if $\text{MoreFragments} = 0$, then FragNum should indicate the last fragment.

(2) *Indirect Dependencies.* Indirect dependencies describe constraints whose validity depends on prior interactions or protocol states. Formally, for a field f_i and protocol state $s \in S$: $\text{valid}(f_i) \iff C(s)$, where C defines the valid value of f_i under state s . For example, in authentication frames, the valid sequence number depends on the selected authentication algorithm and the current step.

Automatic Extraction Pipeline. WiDepFuzz uses a PDF-aware static analyzer to extract such dependencies from the 802.11 specification through four steps.

Step 1: Specification parsing. WiDepFuzz first parses the IEEE 802.11 PDF to recover the document hierarchy, clause identifiers, tables, paragraphs, figures, and cross-references. It focuses on frame-format and information-element definitions, especially Clause 9.3 for frame formats and Clause 9.4 for management information elements. The analyzer identifies paragraphs and table rows that describe value ranges, field presence conditions, cross-field rules, and state-dependent behaviors. It searches for normative expressions such as “shall”, “only if”, “present when”, “set according to”, and “reserved”, and links each candidate statement to its clause identifier and nearby table context.

Step 2: Dependency candidate extraction. For each candidate statement, WiDepFuzz applies rule-based templates assisted by dependency parsing to extract dependency tuples. We use spaCy [12] as a natural-language parser to identify subjects, conditions, and constrained fields, but the final extraction is template-driven to avoid free-form NLP errors. Direct templates include patterns such as “ F is present only if C ”, “if C , then $F = v$ ”, and “when C holds, F is set to v ”. For example, Address 4 is present only when To DS= 1 and From DS= 1. Indirect templates capture state-dependent rules, such as “ E establishes state S ” and “frames in state S should satisfy $\psi(F)$ ”. For example, when the RSN IE indicates Management Frame Protection Required (MFPR= 1), subsequent protected management frames should set the Protected bit.

Step 3: Normalization and dependency graph construction. The extracted statements are normalized into a canonical field namespace. Specifically, WiDepFuzz canonicalizes field aliases and abbreviations by matching clause titles, table headers, and nearby field descriptions. For example, “Protected Frame”, “Protected bit”, and “Frame Protected field” are normalized to `FrameControl.Protected`. It then resolves each field to its enclosing frame or element to avoid name ambiguity, e.g., mapping MFPR to `RSN.Capabilities.MFPR` rather than treating it as a standalone field. WiDepFuzz constructs a directed dependency graph $G = (V, E)$, where each node $v \in V$ represents a concrete field or a virtual state node, and each edge $e = (f_{src} \rightarrow f_{dst}, \phi, s) \in E$ represents a semantic constraint from f_{src} to f_{dst} guarded by an optional state predicate s . When $s = \emptyset$, the edge represents a direct field-only dependency; otherwise, it represents a state-dependent or temporal dependency.

Step 4: Filtering and executable rule generation. To make the FD-Model executable, WiDepFuzz aligns canonical fields with Scapy [8] frame definitions. It builds a symbol map from canonical field names to Scapy paths, verifies bit widths, endianness, and TLV layouts, and adds custom Scapy layers when necessary. Rules that refer to unobservable fields, device-internal logic, unsupported proprietary elements, or non-serializable fields are filtered out. For temporal rules, WiDepFuzz binds state guards to session hooks so that sequence numbers, per-TID counters, replay counters, and association states can be tracked across frames. Finally, each dependency edge is compiled into two executable artifacts: a mutation recipe for generating satisfying or violating variants, and a check function for validating hotspot responses. We use template-driven extraction rather than direct LLM extraction because dependency rules are compiled into semantic oracles, where hallucinated or wrong rules may cause false reports, as discussed in Section 7.

Extraction Results. Overall, WiDepFuzz extracts 213 dependencies from the IEEE 802.11 specification, including 141 direct dependencies and 72 indirect/state-dependent dependencies. These dependencies cover management, control, and data frames, ranging from sequence-number/retry constraints to multi-field RSN, capability-information, and protected-management-frame constraints. Table 1 summarizes the extraction results. Manual extraction is possible in principle, but it is expensive and error-prone because 802.11 dependencies are scattered across clauses, tables, cross-references, and temporal rules involving prior frames or protocol states. WiDepFuzz systematizes this process and directly generates executable checkers and mutation recipes from the extracted FModel.

Table 1: Automatically extracted dependencies in FModel.

Type	Count	Example
Direct dependency	141	RSN IE $\Rightarrow$ Privacy bit
Indirect/state dependency	72	Retry $\Rightarrow$ previous SeqNum
Total	213	Mgmt./control/data frames

Initial Test Frame Sequence Generation: Before fuzzing begins, WiDepFuzz seeds initial Wi-Fi frame sequences using the FModel together with a traditional state model. Specifically, WiDepFuzz first generates sequences based on a state model (as commonly used in prior Wi-Fi fuzzers) that satisfy the basic workflow (scanning/probing $\Rightarrow$ authentication $\Rightarrow$ (re)association $\Rightarrow$ RSN handshake $\Rightarrow$ data). Instead of treating fields independently, WiDepFuzz uses the FModel to focus on fields that participate in semantic dependencies and initializes them according to the encoded constraints. For each frame type (e.g., Authentication, Association Request, QoS Data), WiDepFuzz queries the FModel for concrete field dependencies and then performs dependency-aware initialization and mutation on the relevant field groups, depending on whether the dependency is direct or indirect.

- Type direct dependency: WiDepFuzz mutates direct dependencies by jointly varying the controlling and constrained fields defined in the FModel to synthesize both rule-satisfying and rule-violating combinations, while keeping unrelated fields fixed to isolate semantic effects. E.g., for dependency: ‘RSN IE present $\Rightarrow$ CapabilityInfo.Privacy = 1’, two mutation operations on AssocReq/ReassocReq are proposed: (1) Satisfying: include a valid RSN IE (e.g., AKM = PSK, pairwise = CCMP) and set Privacy = 1 while holding SSID/SupportedRates constant. (2) Violating: keep the same RSN IE but set Privacy = 0. All other independent fields are mutated based on the traditional data model (as commonly used in prior Wi-Fi Fuzzer).
- Type indirect dependency: WiDepFuzz composes cross-state rules into minimal multi-frame sequences: it first establishes the prerequisite condition(s) and then perturbs downstream constrained fields to synthesize both satisfying and violating end-to-end behaviors while keeping unrelated context fixed. E.g., for dependency: ‘RSN.MFPR $\Rightarrow$ FrameControl.Protected’, two mutation operations on it are proposed: (1) Satisfying: send a Deauth/Disassoc with the ‘Protected’ bit set and include the PMF MIC/IE as required. (2) Violating: send the same Deauth/Disassoc but clear the ‘Protected’ bit (and omit PMF MIC).

Based on this approach, we design tailored and dependency-aware mutation strategies for different frame fields in the FModel

according to their semantic meanings. Since these semantics are derived from the IEEE 802.11 standard, which all Wi-Fi protocols follow, the mutation strategies proposed by WiDepFuzz are generalizable across different Wi-Fi devices. More implementation details and mutation strategies can be found in our Git repository¹.

4.3 Dependency-combination Guided Fuzzer

Due to its black-box design, WiDepFuzz cannot access internal runtime information of mobile hotspots under test (e.g., protocol states or coverage counters used by prior fuzzers). Meanwhile, randomly generating test frames often produces semantically invalid inputs that are filtered early, leading to low testing efficiency. To address this, WiDepFuzz employs a dependency-combination guided fuzzing strategy, using field dependency combinations in response frames as feedback to prioritize meaningful inputs. The key heuristic WiDepFuzz adopts is that different field dependency combinations correspond to distinct semantic execution states in the mobile hotspot. Unseen combinations therefore act as a proxy for new states, guiding WiDepFuzz to explore deeper protocol logic and expose bugs under valid but inconsistent conditions.

Dependency-combination signature. For each response frame, WiDepFuzz evaluates applicable FModel rules given the frame subtype and session phase, and produces a signature $\sigma = \langle t, s, \{(r_i, v_i)\} \rangle$, where t is the response subtype, s is the protocol phase, r_i is a rule identifier, and $v_i \in \{\text{satisfied}, \text{violated}\}$ indicates the rule outcome. Signatures are hashed and stored in *VisitedD*. A test sequence is marked *interesting* if it yields a previously unseen signature (i.e., $\sigma \notin \text{VisitedD}$), indicating a new semantic behavior.

Algorithm 1: Dependency-combination Guided Fuzz

Input : *HS*: Mobile Hotspots under Test
Output : *B*: Bugs

```

1  $B \leftarrow \emptyset$ ;  $seedPool \leftarrow \text{initialSeeds}()$ ;
2  $detectors \leftarrow \text{setupDetectors}()$ ;
3  $FD \leftarrow \text{setupFModel}()$ ;
4  $VisitedD = HS.\text{process}(seedPool)$ ;
5 while true do
6    $frameSeq \leftarrow seedPool.\text{dequeue}()$ ;
7    $frameSeq' \leftarrow \text{mutate}(FModel, frameSeq)$ ;
8    $resp \leftarrow HS.\text{processOnHotspot}(frameSeq')$ ;
9    $\sigma \leftarrow \text{computeSignature}(FD, resp)$ ;
10  if  $\sigma \notin VisitedD$  then
11     $VisitedD \leftarrow VisitedD \cup \{\sigma\}$ ;
12     $seedPool.\text{enqueue}(frameSeq')$ ;
13  end
14  async:
15     $bugs \leftarrow detectors.\text{checkAnomalies}(FModel)$ ;
16     $B \leftarrow B \cup bugs$ ;
17    if  $bugs \neq \emptyset$  then
18       $seedPool.\text{enqueue}(frameSeq')$ ;
19      // re-explore neighborhood
20    end
21 end
```

Algorithm 1 shows the dependency-combination guided fuzzing process. In the initial phase (lines 1–4), two anomaly detectors are instantiated to monitor the mobile hotspot in real time, and the FDModel is configured to guide dependency-aware mutations and act as a semantic-bug oracle. The hotspot first processes initial test frame sequences from the seed pool (*seedPool*); during this phase, WiDepFuzz uses the FDModel to record, for each response frame, the observed field-dependency combinations into a set *Visited_D*. In each iteration, WiDepFuzz dequeues a frame sequence *frameSeq* from *seedPool* and mutates it into *frameSeq'* based on the strategy in Section 4.2. The hotspot processes the mutated sequence, and the *fdModel* analyzes the response frames and maps them to a signature σ . If $\sigma \notin \text{Visited}_D$, indicating a new dependency combination, *frameSeq'* is regarded as interesting and stored back into the seed pool to guide subsequent fuzzing (lines 10–13). Meanwhile, the anomaly detectors monitor the hotspot and check for abnormal behaviors (lines 14–20). If any anomalies are detected, WiDepFuzz logs the corresponding test frames and adds *frameSeq'* into the seed pool for further exploration.

Scope of Dependency Handling. WiDepFuzz does not aim to solve all constraints in the FDModel globally. Instead, it handles dependencies within the local scope affected by each mutation. For each fuzzing iteration, WiDepFuzz selects a target dependency rule, mutates the fields involved in that rule, and checks the affected dependency neighborhood rather than the entire FDModel. For example, if mutating field *A* affects a rule involving *B*, and *B* is also constrained by *C*, WiDepFuzz considers the local neighborhood $\{A, B, C\}$ instead of all fields and states. Rules are further filtered by frame type and protocol phase, so each generated or received frame only triggers a small subset of relevant dependency checks. This design avoids high-dimensional constraint solving while still preserving the dependency consistency needed for effective fuzzing and response validation.

Handling Dynamic and Random Fields. Similar to prior protocol fuzzers, WiDepFuzz preserves session-specific fields needed to maintain valid protocol interactions. However, it does not blindly mutate or replay dynamic fields. Instead, it handles session-specific or random fields based on their dependency relevance. For fields that are needed to keep valid interactions but are not targeted by the current dependency rule, such as nonces, timestamps, session counters, or negotiated capabilities, WiDepFuzz preserves values observed from legitimate sessions or lets Scapy/protocol hooks fill them automatically. For dependency-relevant dynamic fields, such as sequence numbers, retry bits, per-TID counters, RSN negotiation status, and replay-related values, WiDepFuzz maintains lightweight session state and mutates them only within FDModel constraints to preserve validity. Thus, WiDepFuzz tracks and replays necessary session context instead of randomly guessing dynamic values, reducing early-stage communication breakage and helping fuzzing reach deeper protocol states.

4.4 Runtime Anomaly Monitor

To ensure generality, WiDepFuzz employs a black-box runtime anomaly monitor that observes the behavior of mobile hotspots during fuzz testing without requiring intrusive access to internal data such as source code or proprietary interfaces. Figure 5 illustrates

the workflow. The monitor includes two detectors: (i) a liveness monitor that determines whether the hotspot becomes unreachable or nonfunctional (e.g., crash, hang, or persistent connection failure); and (ii) a dependency checker that detects violations of 802.11 field-dependency rules, i.e., whether the hotspot incorrectly accepts or processes frames with inconsistent or invalid field combinations, thereby revealing potential frame processing logic errors within the mobile hotspots under test.

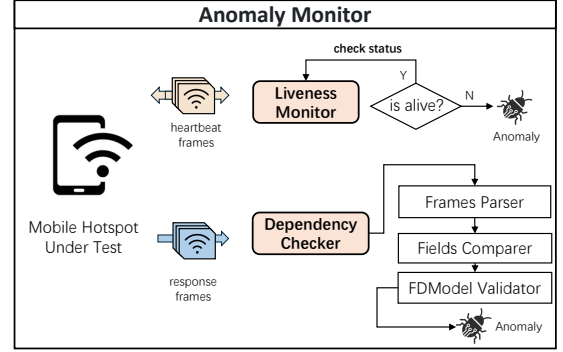


Figure 5: Anomaly Monitor in WiDepFuzz for monitoring the runtime status and identifying anomalies in real time.

Liveness monitor: The liveness monitor tracks the operational status of the target mobile hotspot to ensure it remains responsive during fuzzing. It periodically sends heartbeat probes and expects timely responses; if a valid response is not received for a long time, the detector retries up to three times, following common practice adopted by prior tools. If all retries fail, the mobile hotspot is considered offline. At this point, the liveness detector raises a liveness anomaly and logs the relevant timestamped transmit/receive Wi-Fi frames to facilitate diagnosis of the underlying issue.

Dependency checker: The dependency checker functions by intercepting Wi-Fi frames received from the mobile hotspots and verifying whether their field dependencies satisfy the constraints modeled in the FDModel. Specifically, it performs a four-stage check: (1) Frame parse: WiDepFuzz captures each frame via Radiotap and parses 802.11 headers/IEs into a normalized field map. (2) Field comparison: it filters dependency-relevant fields and correlates them with the FDModel. (3) FDModel validation: it selects the applicable rules for the frame type/state and evaluates both explicit and temporal dependencies. (4) Decision & logging: if a rule is violated, it flags a semantic anomaly and records the rule ID, fields, and timestamped frames into a log for subsequent reproduction.

Bug Reproduction and Deduplication: WiDepFuzz records all transmitted and received Wi-Fi frames during testing, orders them by timestamp, and reconstructs execution traces for analysis. It identifies the first state where the anomaly monitor reports an anomaly as the triggering state S_1 , with the initial hotspot state denoted as S_0 . To confirm reproducibility, WiDepFuzz replays the frame sequence from S_0 to S_1 multiple times and ensures the anomaly can be consistently triggered. Building on this, WiDepFuzz performs bug deduplication inspired by AFL’s testcase minimization [22]. It iteratively removes or merges frames while verifying that the anomaly remains reproducible, prioritizing frames involving field dependencies in the FDModel. The minimized sequences are then used for deduplication: two bugs are considered duplicates if their

sequences share the same ordered frame-type pattern, identical dependency relationships, and trigger the same anomaly type.

5 Implementation

Figure 6 presents the core components of WiDepFuzz, which are divided into three key parts. Dependency Analyzer is responsible for statically parsing the 802.11 specification to extract and model both explicit (direct) and implicit (indirect) field dependencies. Test Frame Generator parses, mutates, and generates concrete Wi-Fi frames as test inputs. Bug Detector monitors the behavior of the mobile hotspot in real time and detects abnormal states. The rest of the section describes notable implementation details.

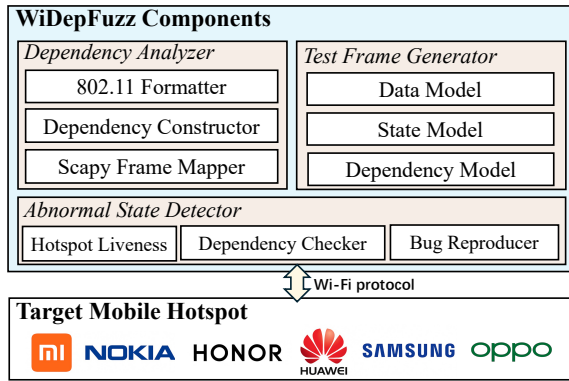


Figure 6: Core components of WiDepFuzz implementation.

Data Model & State Model: WiDepFuzz leverages the data and state models provided by Scapy [8], a widely used packet-generation tool in network testing and security research. Specifically, WiDepFuzz uses Scapy’s data model to parse and format Wi-Fi frames, and its state model to generate frame sequences and drive the Wi-Fi processing workflow, including probing, association, authentication, handshake, and data exchange. For every test frame in a sequence, WiDepFuzz performs pre-send and post-receive semantic checks against the FDModel to verify (or deliberately violate) field-dependency constraints, and continuously performs dependency-combination guided fuzzing. To enable fine-grained control and monitoring of frame transmission and reception, WiDepFuzz leverages the Radiotap interface [40]. It handles combined packets, Radiotap headers plus 802.11 frames, using raw sockets.

Adaptation to Diverse Mobile Hotspot Devices Thanks to the black-box design of WiDepFuzz, which offers high generality, the framework can be easily and quickly adapted to test a new mobile hotspot by following three steps: (1) connect the device under test to the cellular network, enable the hotspot, and allow peer devices to join; (2) in WiDepFuzz, configure the hotspot’s SSID, authentication method, and other basic options (e.g., band/channel, etc.); and (3) run WiDepFuzz to continuously exercise the hotspot and report potential bugs. No rooting or source access is required.

6 Evaluation

In this section, we evaluate WiDepFuzz to answer the following five research questions:

- **RQ1:** Is WiDepFuzz effective in detecting vulnerabilities in real-world Mobile Hotspot?

- **RQ2:** Does the Field Dependency Model effectively improve testing performance?
- **RQ3:** What is the accuracy of detectors and FDModel?
- **RQ4:** Can WiDepFuzz achieve higher code coverage than state-of-the-art methods on open-source hotspot implementations?
- **RQ5:** Can WiDepFuzz be applied to Wi-Fi client-side?

6.1 Experiment Setup

Subjects: We evaluated WiDepFuzz on 6 real-world mobile hotspots from different vendors, covering diverse operating systems, firmware versions, and hardware platforms. The devices were selected based on two criteria: (1) broad real-world usage, ensuring that discovered vulnerabilities affect a large user base, and (2) vendor and implementation diversity, demonstrating WiDepFuzz’s generality across heterogeneous mobile hotspot stacks.

Compared Tools: We compare WiDepFuzz with four state-of-the-art Wi-Fi fuzzers: OwFuzz [9], wpaspy [42], Greyhound [15], and Peach [11]. OwFuzz and Greyhound perform over-the-air fuzzing on real hotspots in a black-box setting; wpaspy enables targeted fuzzing across multiple frame types; and Peach is a general-purpose fuzzer adapted for Wi-Fi frame injection. These tools represent diverse fuzzing strategies (e.g., random, directed, mutation-based), providing a comprehensive baseline.

Metrics and Settings: We use two metrics: the number of unique bugs detected and the average time to discover them. For closed-source mobile hotspots in RQ1–RQ3, coverage is not available, so we use bug number and detection time. For open-source targets in RQ4 and RQ5, we additionally report branch coverage. All tools were executed under the same environment (Intel i9-10900 CPU, 32 GB RAM, Ubuntu 20.04), with hotspots in default configurations and an isolated wireless testbed. Each experiment was repeated 10 times, and the average results are reported.

6.2 Bug Detection in Mobile Hotspots

We applied WiDepFuzz, OwFuzz, Greyhound, wpaspy, and Peach on all six target mobile hotspots for vulnerability detection in 24 hours. In total, WiDepFuzz discovered 20 previously unknown bugs across the six mobile hotspots, significantly outperforming other state-of-the-art fuzzers, none of which detected more than 4 bugs. Detailed information on these bugs is summarized in Table 2.

When WiDepFuzz detects an anomaly (either liveness or dependency violation), it automatically records all Wi-Fi frames exchanged with the target hotspot, organizing them by timestamp into a reproduction log. This log is then provided to developers, and we replay the Wi-Fi frames to help reproduce the anomalies and facilitate root cause analysis. Once an anomaly is successfully reproduced and its root cause is identified, it is confirmed as a bug. As shown in Table 2, most findings are semantic bugs (12/20) that violate 802.11 field-dependency rules (e.g., inconsistent capability/IE combinations or invalid state transitions). These flaws destabilize the state machine, corrupt (re)association negotiation, and can cause unexpected crashes or silent acceptance of malformed frames. Because they are remotely triggerable via crafted Wi-Fi frames, attackers could forcibly disconnect clients, hijack association flows, and even obtain unauthorized network access to eavesdrop on user traffic – posing material risks to wireless network security. In addition, WiDepFuzz uncovered 8 liveness bugs that induce hangs and

Table 2: 20 new bugs detected within 24 hours. WiDepFuzz uncovered all 20 bugs, other SOTA tools found fewer: OwFuzz found 4, wpsapy found 2, Greyhound found 2, and Peach found 3.

#	Mobile Hotspot Model	Root Cause Analysis	Bug Type
1	SAMSUNG Galaxy C55	Incorrect validation of 'CapabilityInfo' and 'RSN' dependency, allowing malformed ReassocReq frames.	Semantic Bug
2	SAMSUNG Galaxy C55	Frame parsing logic mistakenly trusts spoofed Capability info without cross-checking RSN and HT fields.	Semantic Bug
3	SAMSUNG Galaxy C55	AP stops responding when receiving fragmented Auth frames due to improper reassembly timeout handling.	Liveness Bug
4	OPPO Find X7	Association response generation does not sanitize supported rates field, leading to frame crafting confusion.	Semantic Bug
5	OPPO Find X7	AP halts beacon transmission entirely after a burst of disassociation frames with invalid reason codes.	Liveness Bug
6	OPPO Find X7	Device enters idle state after RXing malformed Auth frames without retry mechanism, leading to soft-lock.	Liveness Bug
7	Honor 30	Capability negotiation logic fails to verify mutual HT/VHT features, allowing invalid connection setup.	Semantic Bug
8	Honor 30	Parsing error in RSN IE causes hotspot to skip encryption capability check, violating security policy.	Semantic Bug
9	Honor 30	Incorrect frame processing order in handshake phase leads to inconsistent key material handling.	Semantic Bug
10	Honor 30	Repeatedly dropped EAPOL frames are not retried, eventually stalling the authentication state machine.	Liveness Bug
11	Honor 30	Hotspot ignores probe request bursts during STA connection processes due to state desynchronization.	Liveness Bug
12	Xiaomi 14C	Beacon frame handler incorrectly parses overlapping IEs, resulting in malformed capability propagation.	Semantic Bug
13	Xiaomi 14C	Fails to detect contradictory SSID field dependencies in reassociation frame, causing state inconsistency.	Semantic Bug
14	Xiaomi 14C	Unacknowledged Auth response triggers premature timeout and aborts connection establishment.	Liveness Bug
15	Huawei P40 Pro	Incorrect fallback logic enables Association with unsupported cipher suites caused by dependency conflict.	Semantic Bug
16	Huawei P40 Pro	WPA2 handshake parsing fails to detect malformed MIC field dependency, leading to security bypass.	Semantic Bug
17	NOKIA 6 TA-1054	Capability field mismatch in reassociation is ignored, causing undefined negotiation behavior.	Semantic Bug
18	NOKIA 6 TA-1054	Over-accepted extended IEs bypass security checks due to incorrect dependent frame fields validation.	Semantic Bug
19	NOKIA 6 TA-1054	AP hangs after malformed probe request floods due to incorrect resource free, leading to exhaustion.	Liveness Bug
20	NOKIA 6 TA-1054	Unexpected disassociation frames with spoofed BSSID stall the connection state, disrupting sessions.	Liveness Bug

prolonged service unavailability, which an attacker could exploit to launch persistent remote denial-of-service attacks.

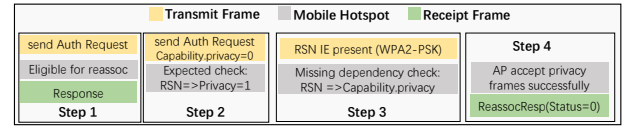
Responsible disclosure: All 20 discovered bugs were promptly reported to the respective vendors, enabling timely remediation before any potential exploitation. Currently, all bugs have been confirmed and patched by the corresponding maintainers.

Table 3: Bugs found by each testing tools.

Fuzzer Name	Bug Number	Bugs ID#
WiDepFuzz	20	#1 - #20
OwFuzz	4	#6,10,13,19
OwFuzz ⁺	11	#2, 5, 6, 8, 9, 10, 11, 13, 15, 17, 19
wpsapy	2	#11,19
wpsapy ⁺	6	#4, 5, 8, 11, 15, 19
Greyhound	2	#10,13
Greyhound ⁺	7	#2, 4, 8, 10,13,15,17
Peach	3	#10,11,19
Peach ⁺	9	#2, 5, 9, 10,11,13, 15, 18,19

Comparison with existing Fuzzers: In our 24-hour experiments, Greyhound, wpsapy, Peach, and OwFuzz identified only 2, 2, 3, and 4 liveness bugs, respectively, because they lack field dependency relationships to check dependency violation bugs. To make a fair comparison, we also enhanced these SOTA tools with WiDepFuzz's dependency checker (denoted Greyhound⁺, wpsapy⁺, Peach⁺, and OwFuzz⁺). With this enhancement, they successfully detected 7, 6, 9, and 11 bugs, respectively. These statistics demonstrate that the dependency checker is effective at exposing dependency violations. However, the remaining 8 bugs (#1,3,7,9,12,14,16,20) were still not detected. This is because they are hidden in deep execution paths. Triggering them requires exercising multiple Wi-Fi frames across multiple field dependencies. With the help of the FDModel and dependency-combination guided fuzzing strategy, WiDepFuzz successfully explored the field dependency handling logic hidden in deep paths and detected all 20 bugs, proving the effectiveness of WiDepFuzz in detecting Wi-Fi protocol bugs in real-world mobile hotspots, which adequately answers **RQ1**. Compared with other state-of-the-art Wi-Fi fuzzers, WiDepFuzz found all the bugs that other fuzzing tools found.

6.2.1 Case Study. We present a case study to illustrate how bugs uncovered by WiDepFuzz threaten wireless security and how WiDepFuzz detects them. This case corresponds to bug #1 in Table 2. The SAMSUNG Galaxy C55 contains a semantic bug: a violation of the 802.11 dependency between RSN and CapabilityInfo.Privacy. This inconsistency desynchronizes the security state machine and enables policy downgrades. Figure 7 shows the key triggering steps.

**Figure 7: A semantic bug that causes privacy leakage**

To trigger the bug, four interactions are required: (1) perform open authentication (or reuse an existing association context); (2) send a ReassocReq whose CapabilityInfo sets Privacy=0; (3) include a syntactically valid RSN IE (e.g., WPA2-PSK with supported cipher suites); (4) observe that the hotspot returns a successful ReassocResp (Status Code 0) and proceeds with the RSN handshake despite the violated dependency. This behavior creates an inconsistent security state: the Privacy bit indicates that no encryption is required, while the presence of the RSN IE triggers protected association procedures. As a result, the hotspot may incorrectly accept a reassociation without enforcing proper key establishment or policy checks. An attacker can exploit this inconsistency to establish a connection under weakened security conditions, potentially bypassing access control or causing traffic to be transmitted without proper protection. This can lead to unauthorized access to user traffic or privacy leakage, especially if subsequent data frames are processed under inconsistent security assumptions. In our experiments, this semantic bug was only detected by WiDepFuzz. It is difficult to uncover due to the specific stateful interaction sequence with cross-field dependencies: 'open authentication → ReassocReq(Privacy=0) → RSN IE present → hotspot returns success → bug triggered'. Existing Wi-Fi fuzzers miss this bug because they do not effectively explore semantic field-dependency relationships.

6.3 Evaluating FModel and Fuzzing Strategy

To evaluate the effectiveness of the proposed FModel and the dependency-combination guided fuzzing strategy, we conducted an experiment comparing three variants: (1) WiDepFuzz: The full version with both FModel and dependency-combination guided fuzzing enabled. (2) WiDepFuzz_m−: A baseline version that disables the FModel, mutates Wi-Fi only based on traditional data model and state model, and disable the dependency checker. (3) WiDepFuzz_g−: A variant that retains the FModel for field dependency but disables the dependency-combination guided fuzzing algorithm, instead performing random generation of test frames. We measured the number of bugs detected by each variant over a 24-hour testing period across six target mobile hotspots. In addition, we recorded the detection latency for each identified bug to assess the testing efficiency under each tool.

Table 4: Comparison among different variants of WiDepFuzz.

Metric	WiDepFuzz _m −	WiDepFuzz _g −	WiDepFuzz
Bugs Found	5	14	20 (↑300%, ↑43%)
Avg Bug Detection Time (Min)	621	503	329 (↑89%, ↑53%)

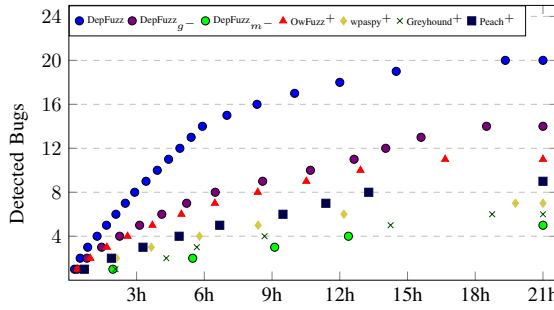


Figure 8: Number of bugs detected by WiDepFuzz and other SOTA tools enhanced with FModel over time.

As shown in Table 4, with the support of the FModel and dependency-combination guided fuzzing, WiDepFuzz successfully detects all 20 vulnerabilities within 24h. In contrast, WiDepFuzz_m− detects only 5 vulnerabilities, while WiDepFuzz_g− detects 14. The remaining 6 vulnerabilities (e.g., Bug #1, detailed in the case study) require execution sequences with specific field dependencies, which are difficult to uncover through random testing methods. In addition, as shown in Figure 8, WiDepFuzz demonstrates faster bug detection. On average, its vulnerability discovery speed is 53% faster than WiDepFuzz_g− and 89% faster than WiDepFuzz_m−. These results demonstrate the effectiveness of both components: compared to WiDepFuzz_m−, WiDepFuzz shows that the FModel enhances both vulnerability detection count and detection speed. Similarly, compared to WiDepFuzz_g−, the dependency-combination guided fuzzing strategy effectively improves these two metrics. These statistics provide a comprehensive answer to RQ2.

6.4 Accuracy of WiDepFuzz

To evaluate the accuracy of WiDepFuzz and assess the potential **false negatives** and **false positives** that may be introduced by the bug deduplication process, we collected all anomalies reported by detectors, along with their corresponding reproduction sequences

Table 5: Accuracy of WiDepFuzz’s Anomaly Detector.

	Before Deduplication		After Deduplication	
	Liveness	Semantic	Liveness	Semantic
Reported Number	19	27	8	12
True Positive	8	12	8	12
False Positive	3	0	0	0

during the testing phase. We then manually analyzed them to identify true positives and false positives.

Before bug deduplication, WiDepFuzz reported a total of 46 anomalies, including 19 from the Liveness Monitor and 27 from the Dependency Checker. Some of these were false positives, caused by background power-saving scheduling of mobile hotspots. To conserve energy, certain devices periodically suspend their Wi-Fi modules or reduce beacon transmission frequency, leading to a short-term absence of Beacon or Probe Response frames. This transient behavior was mistakenly identified by the liveness detector as an offline state, although the hotspot quickly resumed normal operation. After applying WiDepFuzz’s reproduction and deduplication process, all false positives were removed. The Liveness and Dependency Detectors ultimately reported 8 and 12 unique bugs, respectively, all of which were true positives, with no new false negatives introduced. In addition, we evaluated the accuracy of the FModel. We randomly sampled a subset of extracted field dependencies and manually validated them against the IEEE 802.11 specification. The results show that no incorrect dependency rules were observed in the extracted FModel, which aligns with the 0 false positives reported by the semantic detector. The recall is 87%, as some implicit dependencies that are not explicitly specified or require contextual interpretation may not be captured. Nevertheless, the current FModel is sufficiently accurate to guide fuzzing and support reliable semantic anomaly detection. Thus, the accuracy of WiDepFuzz is satisfactory, which adequately answers RQ3.

6.5 Effectiveness in Code Coverage

Table 6: Branch coverage on four open-source Hotspots in 24 hours. WiDepFuzz consistently covers more branches.

Tool	linux-wifi-hotspot	TetherFi	Termux-Hotspot	Linux-router
OwFuzz	2742	3812	1937	2193
wpaspy	2613	3768	1848	2087
Greyhound	2678	3907	1828	2013
Peach	2837	4136	2028	2287
WiDepFuzz	3349 ↑ 23%	4721 ↑ 21%	2318 ↑ 22%	2495 ↑ 17%

Since most mobile hotspots are closed-source, we selected four representative open-source implementations from GitHub based on popularity (i.e., linux-wifi-hotspot [3], TetherFi [39], Termux-Hotspot [10], and Linux-router [47]) to evaluate WiDepFuzz in terms of code coverage. We instrumented the code using gcov [14] and collected branch coverage over a 24-hour fuzzing period. The results are shown in Table 6. Overall, WiDepFuzz consistently outperforms all baselines across the four implementations. Compared to OwFuzz, wpaspy, Greyhound, and Peach, it achieves an average branch coverage improvement of 17%–21%. These results provide a clear answer to RQ4.

6.6 Effectiveness on Wi-Fi Clients.

Although WiDepFuzz mainly targets AP-side mobile hotspots, its dependency-guided design can also be adapted to client-side Wi-Fi implementations. For client-side fuzzing, WiDepFuzz runs on a separate Linux PC and acts as a malicious AP, while the target client acts as a station. The fuzzer uses an external Wi-Fi NIC in monitor/injection mode to send and receive 802.11 frames over the air, requiring no modification to the target client. WiDepFuzz generates dependency-aware Beacon, Probe Response, Authentication, Association Response, Reassociation Response, Deauthentication/Disassociation, and RSN-related frames, and monitors whether the client becomes unresponsive or violates protocol dependencies.

Table 7: Client-side fuzzing results for each tool.

Tool	wpa_supplicant		iwd		NetworkManager	
	Bugs	Branches	Bugs	Branches	Bugs	Branches
Peach ⁺	1	35163	1	21162	2	26492
OWFuzz ⁺	1	36878	1	21916	2	27421
wpaspy ⁺	1	34405	0	20869	1	24881
Greyhound ⁺	0	34070	0	20155	1	24473
WiDepFuzz	2	42881	1	25193	4	31168

We evaluate three representative open-source Wi-Fi client-side implementations: wpa_supplicant [48], iwd [18], and NetworkManager with the wpa_supplicant backend [26]. wpa_supplicant and iwd represent two widely used Linux Wi-Fi supplicant/daemon implementations, while NetworkManager represents a higher-level client-side Wi-Fi management stack that coordinates scanning, association, and supplicant interaction. For a conservative comparison, we adapt the enhanced baselines used in the AP-side evaluation, denoted Peach⁺, OWFuzz⁺, wpaspy⁺, and Greyhound⁺, to the same malicious-AP client-fuzzing setup.

Table 7 shows the client-side fuzzing results. WiDepFuzz identified 7 bugs across the three client-side targets, outperforming all baselines. It also achieved higher branch coverage on every target, improving over the best baseline by 16.3%, 14.9%, and 13.7%, respectively. Although the enhanced baselines can expose some client-side bugs, they cover fewer branches and find fewer bugs because they lack dependency-combination guidance. These results show that WiDepFuzz is not limited to AP-side hotspot testing: by switching the test harness from a client/attacker to a malicious AP, the same workflow can exercise client-side implementations.

7 Discussion

LLM-assisted Dependency Extraction. IEEE 802.11 is a multi-thousand-page specification with dense tables, scattered field definitions, and many cross-clause references. Directly feeding such long specifications to LLMs can easily produce hallucinated or incomplete dependency rules. In our preliminary attempts, we found that more than 60% of the LLM-extracted candidate rules were invalid or unverifiable. Such errors are risky because FDModel rules are compiled into both mutation recipes and response-side semantic oracles, potentially leading to invalid test cases or false positives.

As future work, we plan to explore a controlled LLM-assisted extraction pipeline instead of using LLMs as direct extractors. Specifically, the specification can first be split into clause-level and table-level chunks to reduce context size and retrieval noise. A retrieval component can then select dependency-relevant chunks, and LLMs

can summarize cross-references, normalize field aliases, and propose candidate predicates in a constrained schema. Finally, rule-based validators can check field existence, predicate consistency, Scapy mappings, and checker executability before adding the candidates into FDModel. Such a pipeline may improve extensibility to other protocols and support detailed bug triage by linking detected violations back to specification clauses.

Extending WiDepFuzz to Other Protocols. The core idea of WiDepFuzz is not Wi-Fi-specific: it extracts field dependencies from protocol specifications, uses them to guide input generation, and validates response-side dependency violations. To apply WiDepFuzz to another protocol, one can follow three steps: parsing the target specification to identify message formats, field definitions, and state-dependent constraints; extracting and normalizing these constraints into a protocol-specific FDModel; and mapping FDModel fields to a packet/message generator and response parser so that dependencies can be compiled into mutation recipes and check functions. For example, in our preliminary ZigBee experiment, we extracted dependencies from IEEE 802.15.4 frame definitions and ZigBee command specifications, mapped them to the open-source zigpy implementation [49], and applied dependency-guided fuzzing, successfully identifying 3 new bugs.

Applying WiDepFuzz to complex protocols presents challenges. Some protocols have incomplete or ambiguous specifications, proprietary extensions, or dependencies hidden in implementation-specific state machines. Moreover, valid interactions may require lengthy setup, capability negotiation, or environment-specific states, complicating dependency extraction, state tracking, and response validation. However, unlike mobile hotspots, many protocol implementations are open source, enabling fine-grained runtime feedback such as code coverage, state transitions, or protocol-handler traces. Future work will extend WiDepFuzz to gray- and white-box settings through selective instrumentation, preserving its dependency-driven workflow.

Limitations of Specification-derived Dependencies. WiDepFuzz currently focuses on dependencies that can be extracted from the public IEEE 802.11 specification and validated through observable over-the-air frames. This scope enables black-box testing without firmware, source code, or device instrumentation, but it also limits WiDepFuzz in handling implementation-specific behaviors. Some vendors implement hotspot logic beyond IEEE 802.11, such as customized authentication flows, vendor-specific information elements, proprietary validation logic, and hidden dependencies involving device-internal states. Such dependencies may not be captured by the current FDModel, causing WiDepFuzz to miss certain vendor-specific bugs, especially when proprietary logic affects security decisions or response generation.

A possible extension is to combine specification-derived modeling with dynamic dependency inference. For example, differential probing can perturb candidate fields across controlled trials and infer stable input-output relationships from observed responses. Such inferred dependencies could complement FDModel by capturing implicit or vendor-specific behaviors. However, dynamic inference may introduce additional overhead and noise, especially in unstable wireless environments. Balancing inference precision, runtime cost, and false positives remains future work.

Table 8: Comparison with field-constrained, semantic, and Wi-Fi fuzzers.

Approach	Dependency Source	Dependency Scope	Black-box	Feedback	Oracle
Peach [11]	Manual model	Intra-input constraints	Limited	✗	Crash/timeout
ViDeZZo [21]	Emulation	Device semantics	✗	Coverage	Crash
TaintBFuzz [41]	Taint/coverage	Field constraints	✗	Taint/coverage	Crash
FieldsFuzz [44]	Field inference	Structural field relations	✗	Limited	Crash
Greyhound [15]	Manual model	Frame sequences	✓	State/response	Liveness
Wpaspay [42]	Manual model	Operation-level testing	✓	Limited	Liveness
OWFuzz [9]	Manual model	Frame-level mutation	✓	✗	Liveness
WiDepFuzz	802.11 specification	Cross-field/frame/state dependencies	✓	Dependency combination	Dependency violation & Liveness

MAC-layer Timing and Dynamic Fields Limitations. WiDepFuzz does not reimplement timing-critical MAC-layer logic in user space. Low-level operations such as carrier sensing, contention-window control, and frame scheduling are handled by commodity Wi-Fi NICs, drivers, and firmware. Instead, WiDepFuzz controls the fuzzing-relevant protocol-level behaviors: constructing 802.11 frames, injecting them in monitor/injection mode, capturing responses through Radiotap, and maintaining protocol-level session state. To keep over-the-air interactions stable, WiDepFuzz uses heartbeat probes, retries, and resynchronization. Therefore, WiDepFuzz may miss vulnerabilities that require fine-grained MAC-layer timing control, such as sub-millisecond scheduling behaviors. Detecting such vulnerabilities requires complementary techniques such as SDR-based testing, firmware-level fuzzing, driver instrumentation, or hardware-assisted testing.

A further limitation arises for challenge-response or cryptographically bound dynamic fields. When a message contains a fresh random challenge and the valid response depends on secret keys, negotiated cryptographic material, or device-internal state, simply replaying previously observed session values is insufficient and may break the communication before deeper states are reached. Applying WiDepFuzz to such protocols would require complementary support, such as a protocol-specific responder, access to negotiated keys, cooperation from one endpoint, or implementation instrumentation to generate valid responses.

8 Related Work

Wi-Fi Fuzzing Techniques: Prior Wi-Fi fuzzers have uncovered numerous vulnerabilities across different components of the Wi-Fi stack. Wdev-Fuzzer [23] targets kernel-space drivers via malformed packet injection. Greyhound [15] is a directed greybox fuzzer that leverages protocol state machines to generate valid sequences, but does not model inter-field dependencies. Wpaspay [42] enables targeted testing of protocol operations, while OWFuzz [9] performs over-the-air fuzzing of management and control frames. Peach [11] provides a general protocol fuzzing framework and can express manually specified field constraints, but it does not automatically extract protocol-level dependencies or use them as response-side semantic oracles. In contrast, WiDepFuzz models 802.11 field dependencies and leverages them to guide fuzzing and detect semantic violations in black-box over-the-air settings.

Modeling in Protocol Fuzzing: Most protocol fuzzers rely on data models and state models. Data models describe message formats and enable syntactically valid input generation, as used by Peach [11], Boofuzz [32], and SPIKE [2]. State models capture interaction sequences and guide exploration of protocol behaviors,

as in StateAFL [25], Atheris [43], and SymTCP [46]. These models improve input validity and protocol-state exploration, but they usually do not encode semantic dependencies among fields across frames and protocol states, limiting their ability to expose subtle dependency-violation bugs.

Dependency-aware Fuzzing: Prior work explores dependency awareness in other domains. ViDeZZo [21] learns dependencies for virtual devices under emulation. TaintBFuzz [41] infers constraints for Zigbee stacks using taint and coverage feedback. FieldsFuzz [44] models structural dependencies in binary files. These works show the value of dependency-aware fuzzing, but they target different domains and often rely on assumptions unavailable in black-box Wi-Fi hotspot testing, such as emulation, instrumentation, taint tracking, or coverage feedback.

Main Differences: Table 8 summarizes how WiDepFuzz differs from related fuzzers. First, existing Wi-Fi fuzzers mainly focus on frame mutation, state exploration, or liveness anomalies. They do not explicitly model 802.11 dependencies across fields, frames, and protocol states. Thus, they may miss bugs where the hotspot remains alive but returns protocol-invalid frames. Second, field-constrained and semantic fuzzers model dependencies in other domains, but their dependencies are usually intra-input, structural, or execution-feedback driven. For example, Peach relies on manually specified intra-input constraints, while ViDeZZo, TaintBFuzz, and FieldsFuzz depend on emulation, taint/coverage feedback, or file-structure inference. These assumptions are unavailable in black-box over-the-air hotspot testing. In contrast, WiDepFuzz extracts specification-derived 802.11 dependencies and uses them as a unified mechanism for input generation, black-box feedback, and response-side semantic oracles. This enables WiDepFuzz to explore cross-field/frame/state interactions and detect dependency violations in hotspot-generated responses.

9 Conclusion

In this paper, we propose WiDepFuzz, a field-dependency-guided fuzzing framework for testing Wi-Fi implementations. We evaluated WiDepFuzz on six widely used mobile hotspots and discovered 20 previously unknown bugs. Future work will explore LLM-assisted FModel construction, extend FModel to proprietary behaviors, and adapt WiDepFuzz to more network protocols.

Acknowledgments

This research is sponsored by the NSFC Program (No.62525207, U2441238).

Data Availability Statement

All research artifacts, including datasets, scripts, binaries, and source code, are publicly available in the anonymous Zenodo repository¹.

References

- [1] Khaldoun Al Agha, Guy Pujolle, and Tara Ali-Yahiya. 2016. *Mobile and Wireless Networks*. Wiley-ISTE. doi:10.1002/9781119007548
- [2] Dave Aitel. 2002. An Introduction to SPIKE, the Fuzzer Creation Kit. Black Hat USA. <https://blackhat.com/html/bh-usa-02/bh-usa-02-speakers.html> Presentation slides, accessed July 31, 2026.
- [3] Lakindu Akash. 2025. Feature-rich wifi hotspot creator for Linux. <https://github.com/lakindukash/linux-wifi-hotspot>. Accessed on September 1, 2025.
- [4] Khoula Al-Harthi, Nizar Shah, and Arun N. S. Shankarappa. 2016. Smart-phone's Hotspot Security Issues and Challenges. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 113–118. doi:10.1109/ICITST.2016.7856680
- [5] Giuseppe Anastasi, Marco Conti, and Enrico Gregori. 2004. IEEE 802.11 Ad Hoc Networks: Protocols, Performance, and Open Issues. In *Mobile Ad Hoc Networking*, Stefano Basagni, Marco Conti, Silvia Giordano, and Ivan Stojmenovic (Eds.). Wiley-IEEE Press, 69–116. doi:10.1002/0471656895.ch3
- [6] Android Open Source Project. 2026. Tethering. <https://source.android.com/docs/core/ota/modular-system/tethering> Accessed July 31, 2026.
- [7] Anand Balachandran, Geoffrey M. Voelker, and Paramvir Bahl. 2003. Wireless Hotspots: Current Challenges and Future Directions. In *Proceedings of the 1st ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH '03)*. ACM, 1–9. doi:10.1145/941326.941328
- [8] Philippe Biondi and Scapy Project. 2026. Scapy: Interactive Packet Manipulation Program. <https://github.com/secdev/scapy> Accessed July 31, 2026.
- [9] Hongjian Cao, Lin Huang, Shuwei Hu, Shangcheng Shi, and Yujia Liu. 2023. OWFuzz: Discovering Wi-Fi Flaws in Modern Devices through Over-the-Air Fuzzing. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '23)*. ACM, 263–273. doi:10.1145/3558482.3590174
- [10] CPScript. 2025. A wireless access point orchestration framework that leverages Linux networking subsystems. <https://github.com/0u44/Termux-Hotspot>. Accessed on July 31, 2026.
- [11] Michael Eddington. 2014. Peach Fuzzing Platform. <https://peachtech.gitlab.io/peach-fuzzer-community/Introduction.html> Peach Fuzzer Community Edition; accessed July 31, 2026.
- [12] Explosion. 2025. spaCy: Industrial-Strength Natural Language Processing in Python. <https://spacy.io/> Accessed July 31, 2026.
- [13] Mohammad H. Firooz, Zhiyong Chen, Sumit Roy, and Hui Liu. 2013. Wireless Network Coding via Modified IEEE 802.11 MAC/PHY: Design and Implementation on SDR. *IEEE Journal on Selected Areas in Communications* 31, 8 (Aug. 2013), 1618–1628. doi:10.1109/JSAC.2013.130823
- [14] Free Software Foundation. 2025. Gcov - A Test Coverage Program. <https://gcc.gnu.org/onlinedocs/gcov/Gcov.html>.
- [15] Mathews E. Garbelini, Chundong Wang, and Sudipta Chattopadhyay. 2022. GREY- HOUND: Directed Greybox Wi-Fi Fuzzing. *IEEE Transactions on Dependable and Secure Computing* 19, 2 (2022), 817–834. doi:10.1109/TDSC.2020.3014624
- [16] Ahmad K. Al Hwaitat, Hussam N. Fakhour, Moatsum Alawida, Mohammed S. Atoum, Bilal Abu-Salih, Imad K. M. Salah, Saleh Al-Sharaeh, and Nabil Alassaf. 2024. Overview of Mobile Attack Detection and Prevention Techniques Using Machine Learning. *International Journal of Interactive Mobile Technologies* 18, 10 (2024), 125–157. doi:10.3991/ijim.v18i10.46485
- [17] IEEE. 2025. IEEE Standard for Information Technology—Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks—Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Std 802.11-2024 (Revision of IEEE Std 802.11-2020). doi:10.1109/IEEESTD.2025.10979691
- [18] iwd contributors. 2026. iwd: Wireless Daemon for Linux. <https://kernel.googlesource.com/pub/scm/network/wireless/iwd/>. Accessed: 2026-07-12.
- [19] Raja Karmakar, Samiran Chattopadhyay, and Sandip Chakraborty. 2017. Impact of IEEE 802.11n/ac PHY/MAC High Throughput Enhancements on Transport and Application Protocols—A Survey. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2050–2091. doi:10.1109/COMST.2017.2745052
- [20] Kecheng Liu, Wenlong Shen, Yu Cheng, Lin X. Cai, Qing Li, Sheng Zhou, and Zhisheng Niu. 2019. Security Analysis of Mobile Device-to-Device Network Applications. *IEEE Internet of Things Journal* 6, 2 (2019), 2922–2932. doi:10.1109/JIOT.2018.2877174
- [21] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. 2023. ViDeZZo: Dependency-Aware Virtual Device Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3228–3245. doi:10.1109/SP46215.2023.10179354
- [22] American Fuzzy Lop. 2025. Performance Tips - AFL (american fuzzy lop) - Keep your test cases small. <https://lcamtuf.coredn.cx/afl/latest/tips.html>. Accessed on March 29, 2025.
- [23] Manuel Mendonça and Nuno Neves. 2008. Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities. In *2008 Seventh European Dependable Computing Conference*. IEEE, 110–119. doi:10.1109/EDCC-7.2008.22
- [24] Erfan Mozaffariahrar, Fabrice Théoleyre, and Michael Menth. 2022. A Survey of Wi-Fi 6: Technologies, Advances, and Challenges. *Future Internet* 14, 10 (2022), 293. doi:10.3390/fi14100293
- [25] Roberto Natella. 2022. StateAFL: Greybox Fuzzing for Stateful Network Servers. *Empirical Software Engineering* 27, 7 (2022), 191. doi:10.1007/s10664-022-10233-3
- [26] NetworkManager contributors. 2026. NetworkManager: Linux Network Configuration Tool Suite. <https://github.com/NetworkManager/NetworkManager>. Accessed: 2026-07-12.
- [27] Ana Nika, Asad Ismail, Ben Y. Zhao, Sabrina Gaito, Gian Paolo Rossi, and Haitao Zheng. 2016. Understanding and Predicting Data Hotspots in Cellular Networks. *Mobile Networks and Applications* 21, 3 (2016), 402–413. doi:10.1007/s11036-015-0648-6
- [28] National Institute of Standards and Technology. 2024. CVE-2024-20816 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2024-20816>. Accessed on September 1, 2025.
- [29] Edward J. Oughton, William Lehr, Konstantinos Katsaros, Ioannis Selinis, Dean Buble, and Julius Kusuma. 2021. Revisiting Wireless Internet Connectivity: 5G vs. Wi-Fi 6. *Telecommunications Policy* 45, 5 (2021), 102127. doi:10.1016/j.telpol.2021.102127
- [30] Kaveh Pahlavan and Prashant Krishnamurthy. 2021. Evolution and Impact of Wi-Fi Technology and Applications: A Historical Perspective. *International Journal of Wireless Information Networks* 28, 1 (2021), 3–19. doi:10.1007/s10776-020-00501-8
- [31] Eranga Perera, Henrik Petander, Kun-Chan Lan, and Aruna Seneviratne. 2005. An Implementation and Evaluation of a Mobile Hotspot. In *Proceedings of the 3rd ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH '05)*. ACM, 104–106. doi:10.1145/1080730.1080750
- [32] Joshua Pereyda and boofuzz contributors. 2023. boofuzz: Network Protocol Fuzzing for Humans. <https://github.com/jtpereyda/boofuzz> Version 0.4.2; accessed July 31, 2026.
- [33] Honor Mobile Phone. 2025. Buy Honor mobile officially online at a good price. <https://www.honor.com/global/phones/>. Accessed on September 1, 2025.
- [34] Huawei Mobile Phone. 2025. Explore the latest technologies in smartphone with HUAWEI. <https://consumer.huawei.com/en/phones/>. Accessed on September 1, 2025.
- [35] Nokia Mobile Phone. 2025. Nokia licensed products. <https://www.nokia.com/support/licensed-products/>. Accessed on September 1, 2025.
- [36] OPPO Mobile Phone. 2025. Discover powerful and the latest smartphones with OPPO. <https://www.oppo.com/en/smartphones/>. Accessed on September 1, 2025.
- [37] Samsung Mobile Phone. 2025. Explore the Latest Smartphones. <https://www.samsung.com/us/smartphones/>. Accessed on September 1, 2025.
- [38] Xiaomi Mobile Phone. 2025. Discover a mobile phone. <https://www.mi.com/global/phone/>. Accessed on September 1, 2025.
- [39] pyamsoft. 2025. TetherFi: Internet Sharing without Root. <https://github.com/pyamsoft/tetherfusenet/releases/tag/v1.61>. Version 61, accessed on September 1, 2025; the project was later renamed TetherFuseNet.
- [40] Radiotap Project. 2025. Radiotap: De Facto Standard for 802.11 Frame Injection and Reception. <https://github.com/radiotap>. Accessed on July 31, 2026.
- [41] Mengfei Ren, Haotian Zhang, Xiaolei Ren, Jiang Ming, and Yu Lei. 2023. Intelligent Zigbee Protocol Fuzzing via Constraint-Field Dependency Inference. In *Computer Security—ESORICS 2023 (Lecture Notes in Computer Science, Vol. 14345)*. Springer, 467–486. doi:10.1007/978-3-031-51476-0_23
- [42] Domien Scheepers, Mathy Vanhoef, and Aanjan Ranganathan. 2021. A Framework to Test and Fuzz Wi-Fi Devices. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21)*. ACM, 368–370. doi:10.1145/3448300.3468261
- [43] Google Security. 2025. Atheris: A Coverage-Guided, Native Python Fuzzer. <https://github.com/google/atheris>. Accessed on September 1, 2025.
- [44] Yinghao Su, Dapeng Xiong, Kechang Qian, Yu Wang, and Qingyao Zeng. 2025. FieldsFuzz: Implement Efficient Fuzzing Based on Grammar-Aware Mutation Strategy. *Journal of Systems and Software* 230 (Dec. 2025), 112557. doi:10.1016/j.jss.2025.112557
- [45] Milorad Tosic, Valentina Nejkovic, Filip Jelenkovic, Nenad Milosevic, Zorica Nikolic, Nikos Makris, and Thanasis Korakis. 2016. Semantic Coordination Protocol for LTE and Wi-Fi Coexistence. In *2016 European Conference on Networks and Communications (EuCNC)*. IEEE, 69–73. doi:10.1109/EuCNC.2016.7561007
- [46] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. 2020. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. In *27th Annual Network and Distributed System Security Symposium (NDSS 2020)*. The Internet Society, San Diego, California, USA. doi:10.14722/ndss.2020.24083
- [47] Garry Will. 2025. Set Linux as router in one command to provide Internet, or create WiFi hotspot. <https://github.com/garywill/linux-router>. Accessed on September 1, 2025.

- [48] wpa_supplicant contributors. 2026. Linux WPA/WPA2/WPA3/IEEE 802.1X Supplicant. https://w1.fi/wpa_supplicant/. Accessed: 2026-07-12.
- [49] zigpy contributors. 2026. zigpy: Library implementing a Zigbee stack. <https://github.com/zigpy/zigpy>. Accessed: 2026-07-12.

Received 2026-03-24; accepted 2026-06-18