GREYHOUND: Directed Greybox Wi-Fi Fuzzing

Matheus E. Garbelini, Chundong Wang, and Sudipta Chattopadhyay

Abstract—The recent rise in complex Wi-Fi vulnerabilities, such as KRACK and Dragonslayer, indicates the critical need for effective Wi-Fi protocol testing tools. In this paper, we conceptualize, design and implement a directed fuzzing methodology named GREYHOUND that automatically tests the Wi-Fi client implementations against vulnerabilities such as crashes or non-compliant behaviours. Leveraging a holistic Wi-Fi protocol model, GREYHOUND directs the fuzzer in specific states of target Wi-Fi client. By exchanging mutated packets with a Wi-Fi client, GREYHOUND aims to induce the client to exhibit anomalous behaviours that badly deviate from Wi-Fi protocols. We have implemented GREYHOUND and evaluated it on a variety of real-world Wi-Fi clients, including smartphone, Raspberry Pi, IoT device microcontrollers and a medical device. Our evaluation indicates that GREYHOUND not only automatically discovers known vulnerabilities (including KRACK and Dragonslayer) that would require specialized verification otherwise, but, more importantly, it also has uncovered four new vulnerabilities in popular Wi-Fi client devices. All discovered vulnerabilities have been confirmed by manufacturers and they have been assigned three different common vulnerability exposure (CVE) IDs. We also win a bug bounty of 2,200 USD for discovering the security vulnerabilities. Furthermore, our evaluation with three existing Wi-Fi fuzz testing tools reveals that all such tools fail to discover any of the vulnerabilities (including crashes) uncovered by GREYHOUND. Last but not the least, we have deployed GREYHOUND to test the Wi-Fi client implementation on automotive head units. GREYHOUND automatically discovers KRACK, Dragonslayer and other anomalies in these Wi-Fi implementations. Such a real world try-out justifies the necessity and efficacy of GREYHOUND.

Index Terms—Greybox Fuzzing, Wi-Fi Client, Software Security.

1 INTRODUCTION

With the rise of internet-of-things (IoT), most computing devices, including laptops, smartphones, and Raspberry Pi are now equipped with wireless network hardware. Using such hardware, these devices (called clients) set up connections with a network access point (AP) and transmit data via the network backbone. A number of wireless protocols, e.g., ones following IEEE 802.11 standards [1], have been defined so that the communication between a client and an AP remains secure and confidential. In particular, the designers of wireless protocols have acknowledged the essential of privacy in data transmission and hence employed a number of negotiation techniques such as handshaking to establish a reliable and secure network connection [2], [3], [4].

Unfortunately, the recent rise in complex Wi-Fi vulnerabilities has exposed the bleak side of wireless protocols. For example, Vanhoef and Piessens [5] uncovered a series of attacks named Key Reinstallation Attacks (KRACK) that adversaries can leverage to intercept and steal data across a Wi-Fi network under Wi-Fi Protected Access II (WPA2) security protocol. Moreover, even regarding a perfect wireless protocol, the specific implementation of the protocol in a wireless device, as being a hand-crafted software or firmware program, is probable to embrace vulnerabilities. For instance, recent implementation problems in Nest Cam Indoor cameras can lead to the leakage of user's privacy (e.g., the motion trace of user) [6]. Therefore, it is critical to validate the software or firmware implementations of wireless protocols against potential vulnerabilities.

• *M. E. Garbelini and S. Chattopadhyay are with the Singapore University of Technology and Design, Singapore. C. Wang is with ShanghaiTech University, China. The work was done when C. Wang was a research fellow at Singapore University of Technology and Design.*
E-mail: sudipta_chattopadhyay@sutd.edu.sg.

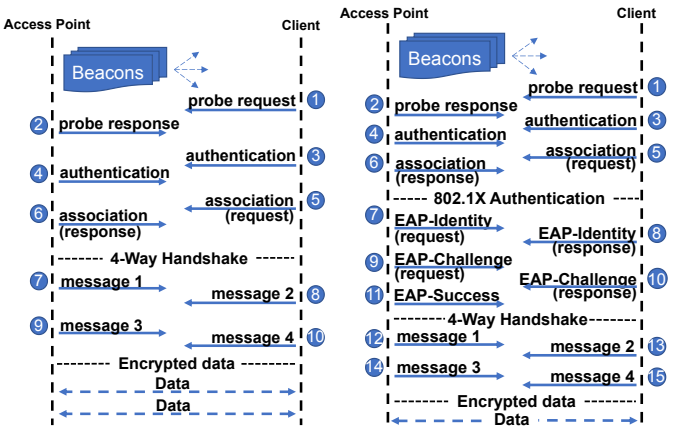
In this paper, we have considered *fuzzing* the implementation of wireless protocols in an arbitrary device to discover vulnerabilities or non-compliant behaviours. Fuzzing is an idea in software testing [7], [8], [9]. As most of the vendors do not disclose their source code of implementations, whitebox fuzzing is inapplicable. Moreover, treating each undisclosed implementation as a black box and naively using a blackbox fuzzer is likely to expose shallow bugs only. An interesting observation is that, the implementation of any Wi-Fi device should strictly follow the standardized wireless protocols that are clearly documented and publicly available. This motivates us to design a greybox fuzzing approach. With respect to the diversity of wireless devices as well as the complexity of wireless protocols, such a greybox fuzzer must be automated without much manual tuning or interference in the fuzzing process. Concretely, we propose GREYHOUND, one fully automated approach that contains a novel greybox fuzzer as its backbone, which references the wireless protocols to fuzz a specific implementation. GREYHOUND aims to discover both existing vulnerabilities as well as unknown non-compliant behaviours. This, in turn, allows us to attach any client (e.g., smartphone, Raspberry Pi, and wireless medical device) with GREYHOUND to validate the client's wireless protocol implementation.

Existing Wi-Fi fuzzing tools [10], [11], [12], are only capable to discover crashes and are incapable to discover more complex non-compliant behaviours. Besides, the existing fuzzing tools fail to explore deep states in the Wi-Fi protocol in a comprehensive fashion. To develop GREYHOUND that is comprehensive to discover as many as vulnerabilities and non-compliant behaviours, we faced several technical challenges. Firstly, fuzzing a wireless device that is connecting to network differs from fuzzing a standalone software program. Fuzzing a software program uses different inputs

to traverse execution paths of the program in order to find as many bugs as possible. To fuzz protocol implementations of a wireless device, GREYHOUND focuses on discovering crashes or non-compliant behaviours against protocol standard when the device is setting up connection with a network AP. As a result, GREYHOUND requires a distinct strategy. Secondly, the diversity of wireless devices is overwhelming while most of their implementations for wireless protocols are poorly documented. The only knowledge GREYHOUND can rely on is the standards defining those wireless protocols. To this end, GREYHOUND employs an abstraction of Wi-Fi protocols (via a state machine) to validate the corresponding unknown implementation behaviour in a specific wireless device. Thirdly, setting up a wireless connection involves multiple wireless protocols, which are layered to form a packet in transmission. These protocols have different likelihoods of being vulnerable. Thus, GREYHOUND requires systematic strategies to direct the fuzzing process in order to quickly and thoroughly discover vulnerabilities for different protocols.

To resolve the aforementioned challenges, GREYHOUND incorporates several novel methods as summarized below.

- GREYHOUND employs a greybox fuzzing approach operating at the side of network AP. Since a wireless protocol can be modeled as a state machine [5], GREYHOUND builds a protocol model that exactly resembles this state machine. GREYHOUND leverages such a model to automatically, *(i)* speculate about the state in which a wireless device is exchanging packets, *(ii)* generate and mutate packets for the purpose of fuzzing, and *(iii)* validate response packets received from the wireless device to check the occurrence of crash or non-compliant behaviours.
- GREYHOUND generates well-formatted (i.e., according to the protocol standard) but inappropriate packets to fuzz a wireless device. However, such a packet is delivered to a wireless device either redundantly or at a wrong time, or some fields of the packet are altered. Using these fuzzed packets, GREYHOUND aims to observe whether the device's implementation for wireless protocols makes the device exhibit crashes or non-compliant behaviours that badly deviate from the protocol standard.
- GREYHOUND uses different probabilities in mutating multiple protocol layers in a packet. This accelerates the fuzzing process to discover anomalous behaviours of wireless devices. Specifically, GREYHOUND leverages reports from the Common Vulnerability and Exposures (CVE) and assigns initial higher mutation probability to protocol layers that are prone to vulnerabilities. For instance, the recent revelation of Dragonslayer [13] directs GREYHOUND to assign a higher mutation probability to the Extensible Authentication Protocol (EAP) layer.
- GREYHOUND defines a number of cost functions along multiple dimensions to automatically monitor and in turn improve its efficacy. One of GREYHOUND's cost functions, for instance, is the number of vulnerabilities that it has discovered. GREYHOUND attempts to maximize this cost function value so as to promote its effectiveness.



(a) WPA2-Personal (b) WPA2-Enterprise

Fig. 1. An Illustration of WPA2-Personal and WPA2-Enterprise

We have implemented GREYHOUND and fuzzed a variety of real-world Wi-Fi clients, including smartphones, Raspberry Pi, ESP8266, ESP32, and medical device. Smartphone was chosen due to their popularity in daily usage. Raspberry Pi, ESP8266 and ESP32 were chosen due to their massive usage in IoT applications [14]. Finally, we chose a wireless medical device due to its criticality in terms of private data communication. Our thorough experiments confirm that, GREYHOUND not only automatically discovers reported vulnerabilities (including Dragonslayer and KRACK), but, more importantly, it also has uncovered at least four new vulnerabilities in popular Wi-Fi clients and three other new non-compliant behaviours. All vulnerabilities have been confirmed and they are assigned three new CVE IDs *CVE-2019-12586* [1], *CVE-2019-12587* [2] and *CVE-2019-12588* [3]. We also compare our GREYHOUND approach with three existing Wi-Fi testing tools: *wifuzzit* [10], *wifuzz* [11] and *IoTcube wifuzz* [12]. Our evaluation reveals that such tools are incapable of discovering any of the vulnerabilities discovered by GREYHOUND. Finally, we show the scalability and applicability of GREYHOUND by deploying it to test the Wi-Fi implementation on several automotive head units.

The remainder of this paper is organized as follows. In Section 2, we present the background of wireless connections. In Section 3, we show an overview of GREYHOUND. We detail the modules of GREYHOUND in Section 4. We describe the evaluation of GREYHOUND on a variety of Wi-Fi clients in Section 5. In Section 6, we compare GREYHOUND to other testing tools. We discuss the key factors of GREYHOUND in Section 7. We conclude the paper in Section 8.

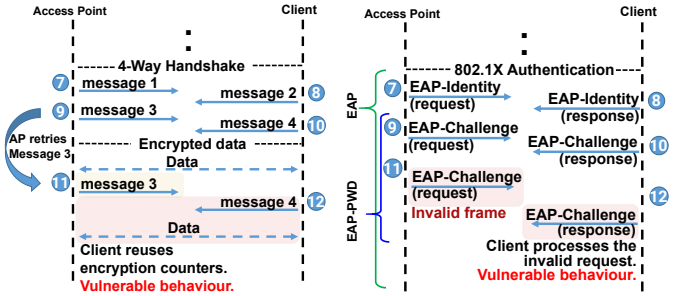
2 Background and Protocol Model

In this paper, we place emphasis on Wi-Fi wireless protocols defined in the family of IEEE 802.11 standards.

2.1 IEEE 802.11i Standard

The IEEE 802.11i standard [15] defines a set of protocols to provide Wi-Fi security, such as the widely-used WPA2-

1. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12586
2. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12587
3. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12588



(a) KRACK (b) Dragonslayer

Fig. 2. An Illustration of KRACK and Dragonslayer vulnerabilities

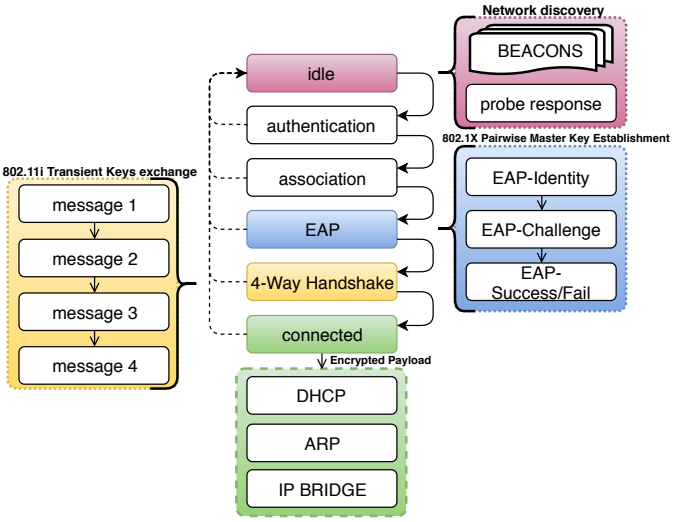


Fig. 3. Wi-Fi State Machine Model

Personal and WPA2-Enterprise. It describes how authentication between a Wi-Fi client and a network AP occurs. This includes the exchange of temporary encryption keys used in posterior data communication. The standard also defines how a protocol is structured in a normal Wi-Fi packet to enable different functionalities during the negotiation of keys. Figure 1(a) shows the packet exchanged between a Wi-Fi client and AP until a successful connection with WPA2-Personal. At start, an AP broadcasts beacon frames to advertise itself to nearby Wi-Fi clients. A client starts connecting to the AP by sending probe request (①), and waits for the response from AP (②). Then the Wi-Fi client initiates authentication (③ and ④) and association (⑤ and ⑥) with the AP. Once the AP accepts the association request, it registers the Wi-Fi client and allocates necessary resources. In WPA2-Personal, the AP further launches a strict 4-Way Handshake with the Wi-Fi client by exchanging four messages (⑦ to ⑩). The purpose is to establish Pairwise Transient Keys (PTK) between Wi-Fi client and AP. In particular, `message 3` ensures AP legitimacy to the Wi-Fi client while `message 4` completes 4-Way Handshake. Then data can be securely transmitted as assumed in WPA2-Personal.

WPA2-Enterprise serves in more complicated environment. It adds 802.1X authentication after association and before 4-Way Handshake. As shown in Figure 1(b), the AP initiates the Enterprise Authentication Protocol (EAP) exchange through exchanging `EAP-Identity` request and response (⑦ to ⑧), and `EAP-Challenge` request and response (⑨ to ⑩). Then the AP notifies the Wi-Fi client with an `EAP-Success` (⑪).

Wi-Fi Vulnerabilities: Figure 2(a) and Figure 2(b) outline two uncovered critical Wi-Fi vulnerabilities, namely KRACK [5] and Dragonslayer [13], respectively. KRACK is attributed to the 4-Way Handshake, while Dragonslayer is attributed to the EAP. For KRACK, the Wi-Fi client must not respond via `message 4` (⑫ in Figure 2(a)) in reply to the redundant `message 3` sent by the AP. For Dragonslayer, in Figure 2(b), the Wi-Fi client must not reply with an `EAP-Challenge` response (⑫ in Figure 2(b)) to the invalid `EAP-Challenge` request (⑪ in Figure 2(b)) from the AP.

The preceding examples capture some unique situations where the vulnerabilities appear. In the perspective of testing, it is thus crucial to generate a sequence of messages that might cause a Wi-Fi client to expose a vulnerability. Such a sequence of messages can only be generated with a comprehensive model of the targeted protocol. Thus, GREYHOUND incorporates a detailed model of specific protocols, via a state machine, to generate valid sequence of packets.

2.2 Protocol Model

Figure 3 provides a high-level overview of the state machine model used by GREYHOUND. The model captures the core design of IEEE 802.11, 802.11i and 802.1X, which enables us to test implementations of both WPA2-Personal and WPA2-Enterprise against possible undesirable Wi-Fi client behaviours, such as non-compliance with the standard.

In the middle of Figure 3, the three states, i.e., `idle`, `authentication` and `association`, are the minimum states required to establish a simple open Wi-Fi connection, that is, a non-protected, plain text communication between a Wi-Fi client and an AP. GREYHOUND emulates a fake AP to communicate with the Wi-Fi client. In `idle` state, such an AP actively broadcasts beacon frames every 100ms to announce itself towards Wi-Fi clients. The AP also responds to probes requested by the Wi-Fi client. The `authentication` state in our model consists of two messages: an authentication request from a Wi-Fi client and a success or failure authentication response sent to the Wi-Fi client. If the request from the client is valid, the AP responds with a successful authentication response and transits to `association` state. Next, in `association` state, the Wi-Fi client must send a correct association request, so the AP can reply with a successful association response.

EAP: With WPA2-Enterprise, the AP starts the EAP exchange after the `association` state. Our model handles three EAP sub-states, i.e., `EAP-Identity`, `EAP-Challenge` and `EAP-Success/Fail`. While `EAP-Identity` and `EAP-Success/Fail` are relatively simple messages to construct, `EAP-Challenge` messages require the usage of several cryptographic algorithms (e.g., elliptic curve crypto). On receiving a successful `EAP-Challenge` response, both the Wi-Fi client and AP share a Pairwise Master Key (PMK).

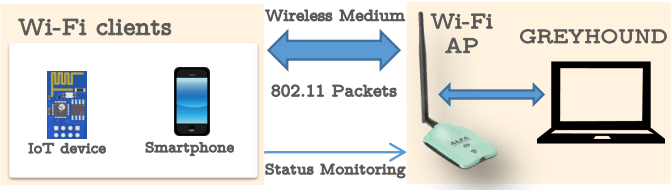


Fig. 4. An Illustration of Fuzzing Scenario for GREYHOUND

4-Way Handshake: `4-Way Handshake` state is essential for both WPA2-Enterprise and WPA2-Personal. As mentioned with Figure 1, in this state, four messages are exchanged between AP and Wi-Fi client with the goal of establishing PTK. PTKs are derived from the PMK during the state of `4-Way Handshake`.

After the `4-Way Handshake` state, the Wi-Fi client gets connected to the Internet and data packets can be exchanged between the AP and the Wi-Fi client. Moreover, all such data packets are encrypted. We note that GREYHOUND systematically generates fuzzed packets until the 4-Way Handshake is complete. GREYHOUND *does not plan to discover vulnerabilities in high-level protocols such as DHCP and ARP.*

3 OVERVIEW OF GREYHOUND

In this section, we introduce a high-level overview of our GREYHOUND approach and differentiate our approach with the state-of-the-art fuzzing methodologies.

Basic Design: Figure 4 shows the scenario in which GREYHOUND works. GREYHOUND can be used to test an arbitrary Wi-Fi client, such as a smartphone, computer, or Raspberry Pi. To establish a wireless connection under, say, the aforementioned WPA2-Personal, a Wi-Fi client needs to follow protocols to orderly exchange a number of packets with a network AP. As a greybox fuzzer, GREYHOUND models a wireless protocol as a state machine for reference. Through answering packets received from the Wi-Fi client and proactively sending out packets, GREYHOUND triggers state transitions at the AP side and in turn the Wi-Fi client side. To fuzz a Wi-Fi client, our GREYHOUND, residing in the AP, systematically generates packets via the protocol model (cf. Figure 3) and a test optimization algorithm embodied within GREYHOUND.

The Usage of GREYHOUND: GREYHOUND can be primarily used in two scenarios. Firstly, GREYHOUND can be used to systematically test the Wi-Fi client implementations of arbitrary Wi-Fi clients. To this end, GREYHOUND can discover non-compliant behaviours of the implementation (including crashes and vulnerabilities) with respect to the protocol standard. This is possible, as GREYHOUND employs a comprehensive state-machine model of Wi-Fi and continuously monitors the communication between Wi-Fi client and AP to check unexpected packets received in any state. Secondly, GREYHOUND can discover existing protocol design flaws, such as KRACK, through testing Wi-Fi client implementations. This is because the set of expected layers constituting a packet, as associated with each state in the Wi-Fi model (cf. Figure 3), already accounts for the unexpected behaviour in vulnerabilities such as KRACK. Yet GREYHOUND is not supposed to discover new vulnerabilities in the design of protocols.

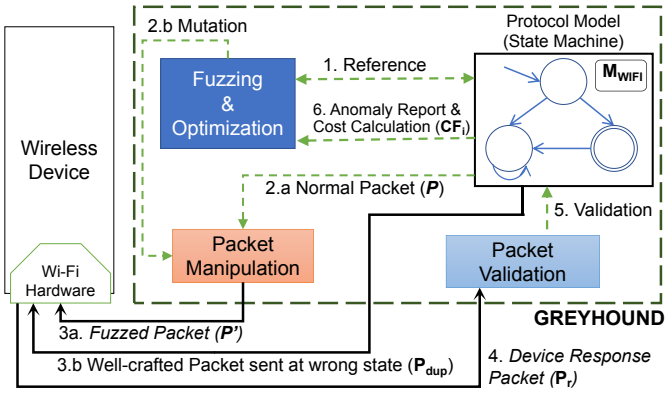


Fig. 5. An Illustration of the Architecture of GREYHOUND

GREYHOUND Workflow: Figure 5 illustrates the high-level workflow of GREYHOUND. As mentioned, GREYHOUND incorporates a protocol model, i.e., M_{WIFI}, at the upper right of Figure 5 to supervise and drive the process of greybox fuzzing. Besides the protocol model, there are three major modules in GREYHOUND, i.e., *(i)* fuzzing and test optimization, *(ii)* packet manipulation, and *(iii)* packet validation that validates the correctness of packets received from the Wi-Fi client. Periodically, the fuzzing and optimization module references M_{WIFI} to learn the current state of Wi-Fi client and advance the fuzzing process (1 in Figure 5). The protocol model also generates normal valid packets (2.a in Figure 5) to the packet manipulation module. Following the mutation instructions from the fuzzing and optimization module (2.b in Figure 5), a normal valid packet results in a fuzzed packet P', which is sent through the Wi-Fi hardware (3.a in Figure 5) to the Wi-Fi client. Concurrently, GREYHOUND may leverage a valid packet, i.e., P_{dup} to be sent from an inappropriate state of the Wi-Fi access point (AP). This feature is crucial to expose vulnerabilities such as KRACK, as KRACK appears when `message3` is sent from the `connected` state (cf. Figure 2(a)). GREYHOUND employs such a feature to observe the response of a Wi-Fi client when the client receives a well-formed packet at a wrong state. Once a response packet P_r to P' or P_{dup} is received (4 in Figure 5) from the Wi-Fi client, it is validated by the packet validation module by referring to the protocol model (5 in Figure 5). A non-compliant response implies the possible existence of vulnerabilities in the Wi-Fi client. The non-compliant responses are reported to the fuzzing and optimization module, and based on a quantitative cost calculation, GREYHOUND further optimizes the fuzzing process of the Wi-Fi client (6 in Figure 5). More detailed steps of GREYHOUND are outlined in Algorithm 1 (procedure `Greyhound_Core`).

Initially, the protocol model is generated and the mutation probabilities are initialized before fuzzing iterations begin (Lines 3-7 in Algorithm 1). Specifically, the mutation probabilities are initialized via the procedure `Particle_Swarm_Opt` (Algorithm 2). Then the network AP is at the `idle` state and waits for the Wi-Fi client to communicate with the AP (Lines 11, 15-17). Once the AP receives a packet P_r from the Wi-Fi client, it automatically checks

whether the packet is expected via the `Run_Validation` procedure (Lines 18-19, cf. Section 4.2). The crux of the `Run_Validation` procedure is based on a set of expected layers $expected(S)$, assigned to each state S of the protocol model M_{WIFI} (Line 13 in Algorithm 1). With the received packet P_r, the AP makes an appropriate state transition by referring to the protocol model (Line 24). Subsequently, GREYHOUND generates a valid packet P from the current protocol state S. However, before communicating with the Wi-Fi client, GREYHOUND mutates the fields of packet P and generates mutated packets P' (Lines 25-28) according to a set of mutation probabilities X_i (see Section 4.1 for details). X_i captures the set of mutation probabilities associated with all the states in the model (cf. Figure 3). It is worthwhile to mention that due to the probabilistic nature of mutation, the packet P' may differ from the original packet P. When the mutated packet P' indeed differs from the original packet P, the packet P' is deemed malformed. As a result, any response (from the Wi-Fi client) for such P' is considered as an anomaly. To this end, the expected set of layers in state S is momentarily cleared (Line 34 in Algorithm 1). This is for the validation process (Lines 18-19, cf. Section 4.2) to detect potential anomalies that result due to the response to malformed P'.

Additionally, GREYHOUND maintains a history of packets generated from the model in $\mathbb{P}_{hist}$ and selects a packet P_{dup} from $\mathbb{P}_{hist}$ to communicate with the Wi-Fi client (Lines 30-36). The rationale behind such a communication is to send a packet that is invalid regarding the current state of the protocol. Moreover, even though P_{dup} triggers a legitimate transition to the Wi-Fi client (e.g. when the reception of P_{dup} is delayed), it is not possible to trigger the same transition to the Wi-Fi client via another packet P. As a result, duplicate responses from a Wi-Fi client, during the same fuzzing iteration, are caught as anomalies. This is because such responses are received during different states of the AP state machine. In Section 4.2, we comprehensively discuss the different scenarios that appear during anomaly detection (i.e. procedure `Run_Validation`).

A fuzzing iteration finishes when the AP reaches `idle` state (Line 37) or an anomaly is detected (Line 22 in Algorithm 1). After a fuzzing iteration, the mutation probabilities to fuzz a packet are refined via a particle swarm optimization (Lines 38-43, Algorithm 2), which takes input as the value of a cost function measured during the communication with the targeted Wi-Fi client (cf. Section 4.1). We note that different mutation probabilities X_i may result in different fuzzed packets, thus leading to different cost (e.g., number of transitions covered in the model) CF_i. We aim to automatically maximize or minimize the value of the cost function by refining the mutation probabilities. This new set of mutation probabilities X_{i+1}, as computed iteratively via procedure `Particle_Swarm_Opt` are used in the next fuzzing iteration. Using such an approach makes GREYHOUND directed, with the aim to uncover anomalies in the protocol implementation.

Finally, while waiting for the response from a Wi-Fi client (Line 15 in Algorithm 1), GREYHOUND may time out. This mostly happens when the client drops mutated and invalid packets or becomes unresponsive due to crashes. In such cases, after GREYHOUND times out, the AP reaches the *idle* state and exits the inner loop in Algorithm 1. Then, the cost function value is computed and GREYHOUND starts a new fuzzing iteration (Line 11) via the refined mutation probabilities (Line 43).

Algorithm 1 Main Steps of GREYHOUND

```
1: Procedure: Greyhound_Core()
2:
3:  i ← 0  ▷ i captures fuzzing iteration
4: ▷ generate Wi-Fi protocol model (cf. Figure 3)
5:  M_WIFI ← Generate_Protocol_Model()
6: ▷ wait to receive mutation probabilities from PSO
7:  X_i ←-- Particle_Swarm_Opt()
8: ▷ initialize history of sent packets
9:  P_hist ← ∅; P_dup ← ∅
10: repeat
11:     Wait for the AP to be in idle state
12:     ▷ assign expected layers for received packets
13:     For each S ∈ M_WIFI, assign expected(S)
14:     repeat
15:         Wait for the Wi-Fi client to communicate
16:         goto line 39 if timeout occurs at line 15
17:         Let the AP receives packet P_r from the Wi-Fi client
18:         ▷ monitors states and checks anomalies
19:         θ_anom ← Run_Validation(P_r, S)
20:         ▷ exits the fuzzing iteration in case of anomalies
21:         if θ_anom is false then
22:             goto line 39
23:         end if
24:         S ← Get_Current_State(M_WIFI, P_r)
25:         ▷ generate a valid packet from the model
26:         P ← Get_Packet_from_Model(M_WIFI, S)
27:         ▷ generate fuzzed packets from P via mutation
28:         P' ← Mutate_Packet(P, X_i)
29:         Send fuzzed packets P' to the Wi-Fi client
30:         Choose a packet P_dup ∈ P_hist ∪ {∅} s.t. P_dup ≠ P
31:         Send irrelevant packet P_dup to the Wi-Fi client
32:         ▷ reset expected layers after fuzzing
33:         if P' ≠ P then
34:             expected(S) ← ∅
35:         end if
36:         P_hist ← P_hist ∪ {P}
37:     until AP does not reach the idle state
38:     ▷ measure cost function value for X_i
39:     CF_i ← Measure_Cost_Function(X_i)
40:     ▷ send cost function value to PSO
41:     Particle_Swarm_Opt() ←-- CF_i
42:     ▷ wait to receive new mutation probabilities from PSO
43:     X_{i+1} ←-- Particle_Swarm_Opt()
44:     i ← i + 1
45: until timeout
```

Handling retransmission of packets by Wi-Fi client: In our implementation of Algorithm 1, GREYHOUND discards duplicated packets sent by the client by checking the `Retry` field of a Wi-Fi packet. Thus, while waiting for the client (Line 17 in Algorithm 1), GREYHOUND only proceeds when the `Retry` flag is reset in P_r, otherwise P_r is dropped. It is also possible that the Wi-Fi client software may retransmit fresh `authentication-request` and `association-request` without the `Retry` flag being set. However, such retransmission happens after a substantial time interval ($\approx$ 1-2 seconds) and only when some malformed packet from GREYHOUND is legitimately dropped by the client. This time interval is large enough for the AP to be in the `idle` state (i.e. it exits the inner loop

of Algorithm 1). While the `authentication-request` is a valid packet to be received in the AP `idle` state, GREYHOUND discards (i.e. does not flag anomalies) any `association-request` packets in the `idle` state. This is to handle the retransmission of `association-request` packets from the previous fuzzing iteration.

How GREYHOUND differs from state-of-the-art fuzzers? The fuzzing strategy of GREYHOUND differs from conventional strategies for software fuzzing [16], [17], [18]. Firstly, traditional software fuzzing methodologies fuzz programs with valid and invalid inputs; by contrast, GREYHOUND has to acquire a deep understanding of the underlying network protocol to trigger appropriate state transitions with the protocol for finding vulnerabilities. A classic fuzzer is hence inapplicable to fuzzing Wi-Fi clients since, being ignorant of the protocol, it would get stuck in the initial state of the protocol model due to the invalid packets it usually generates. Secondly, to induce certain anomalies in a Wi-Fi protocol, it is often essential to monitor the timing for packet exchange and also minimize the timing for a Wi-Fi client reconnecting with the AP. This involves fairly complex test optimization strategy – a feature that seldom exists in the conventional fuzzing techniques. Finally, the test optimization of GREYHOUND critically depends on how it maps packets received from a Wi-Fi client to the states of a protocol model. Such a test optimization strategy is unique and significantly differentiates GREYHOUND from state-of-the-art software fuzzing techniques.

GREYHOUND also differs from previous works [7], [19], [20], [21] targeting network fuzzing. These works either focused on text-structured protocols such as `ftp` and `http`, or they are highly specific to certain fragments of Wi-Fi, such as handshake [21]. Thus, such fuzzing techniques are incapable of discovering a variety of vulnerabilities, including Dragonslayer [13] that might appear in different protocol states. Moreover, these works have limited automation as their test scenarios are often manually provided. On the contrary, GREYHOUND is a fully automated approach that comprehensively covers the states of an AP and a Wi-Fi client, when the client is trying to set up connection with the AP. Moreover, GREYHOUND is general in the sense that the protocol model can easily be replaced to support fuzzing a variety of other protocols.

4 FUZZING IN GREYHOUND

In this section, we discuss the fuzzing component of GREYHOUND in detail.

4.1 Fuzzing Component

Using the aforementioned protocol model, GREYHOUND generates packets compliant to the protocol standard. Subsequently, each such packet is forwarded to the fuzzing component embodied in GREYHOUND. The fuzzer systematically modifies the packets generated from the protocol model. The objective of such modification is to direct the fuzzer towards scenarios that are likely to lead to non-compliant behaviours, including but not limited to crashes. In the following, we describe the core mechanism embodied within our fuzzer.

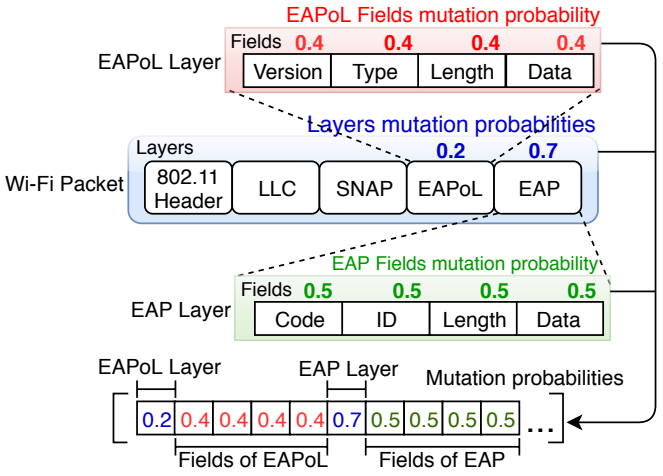


Fig. 6. An Illustration of GREYHOUND's Fuzzing

Mutation Probabilities: A packet generated from the protocol model has a fairly complex structure. The mutation of GREYHOUND occurs at two levels: each packet P contains multiple *layers* and each such layer may consist of multiple *fields*. Let L_P be the set of layers associated with the packet P. For each layer $l \in L_P$, let F_l be the set of associated fields. We associate each layer $l \in L_P$ with a probability pr_l^+. pr_l^+ captures the probability for GREYHOUND to mutate the layer l of packet P. Finally, we also associate a field mutation probability pr_l^- for each layer $l \in L_P$. Any field $f \in F_l$ is mutated by GREYHOUND with the probability pr_l^-. As each packet is associated with a protocol state s (cf. Figure 3), the number of mutation probabilities is bounded by $2 \times \sum_{s \in \mathbb{S}} L_s$, where $\mathbb{S}$ is the total number of states in the protocol model and L_s is the number of layers associated with the packet in state s. In Algorithm 1, each X_i captures a set of $2 \times \sum_{s \in \mathbb{S}} L_s$ such mutation probabilities.

Figure 6 captures the core mechanism of GREYHOUND's fuzzing module that mutates a Wi-Fi packet. The packet includes five different layers including `LLC`, `SNAP`, `EAPoL` and `EAP`. The probabilities of these layers can initially be assigned by the user or based on the reference of common vulnerability exposures (CVEs). Specifically, from the description of a CVE, we can figure out the Wi-Fi state where the respective vulnerability appears. As an example, consider the Dragonslayer vulnerability (*CVE-2019-9499*) that is associated with the `EAP` state. Thus, to assign mutation probabilities based on *CVE-2019-9499*, we assign high mutation probability values to all layers and fields related to the `EAP` state. The rest of the layers and fields are assigned lower mutation probability values. For instance, considering *CVE-2019-9499*, GREYHOUND assigns a higher probability of 0.7 to the `EAP` layer as shown in Figure 6. A layer with a higher probability means that it would be more frequently mutated by GREYHOUND. The intuition behind this is that, GREYHOUND is more likely to mutate layers that have been known to embrace vulnerabilities, and expects the implementation of the Wi-Fi client device to be more likely to exhibit non-compliant behaviours. Each layer in turn contains a set of fields. Take the `EAP` layer for example again. It contains four fields: `Code`, `ID`, `Length`, and `Data`.

GREYHOUND also needs to assign probabilities to them. To keep the design of GREYHOUND simple, we do not use different probabilities for fields of the same layer. However, this can be easily tuned.

Mutation Operators: GREYHOUND uses the following mutation operators to modify an arbitrary packet generated from the protocol model:

- **Random byte:** Each packet field selected by the fuzzer is assigned a randomly generated value. We choose this operator to induce stochastic behaviour in the protocol.
- **Zero filling:** The packet field is overwritten with all zeros. Such a mutation operator is chosen with the objective to induce potential buffer underflow during the sending and reception of Wi-Fi packets.
- **Bit setting:** For the chosen packet field, the most significant bit of a byte is set. For multi-byte fields, the most significant bit for each byte is set. GREYHOUND chooses this mutator with the goal to incur potential buffer overflows (e.g., setting higher bits correlates to setting high values for certain packet fields).

Given a packet P generated from the protocol model, GREYHOUND mutates the fields of it according to the probabilities attributed to pr_l^+ and pr_l^- for each layer l of P. Once GREYHOUND decides to mutate a packet field, we choose one of the mutation operators (i.e., random byte, zero filling and bit setting) in random. Thus, the outcome is a mutated packet P' after applying a respective mutation operator.

Refining Mutation Probabilities: In the preceding section, we show the mechanism to mutate the layers of a Wi-Fi packet based on certain probabilities that are initially assigned either with CVEs or randomly. In order to effectively discover anomalies (e.g., crashes or non-compliant behaviours against the protocol standard), GREYHOUND employs a set of *cost functions* to systematically refine the mutation probabilities. The cost functions considered by GREYHOUND are as follows.

- **Transitions:** This counts the number of state transitions in the model discussed in Figure 3. The number of such transitions are counted until the model goes back to the `idle` state. This cost function is chosen to maximize the number of state transitions that may potentially discover more anomalous behaviours across all states.
- **Anomaly period:** This captures the elapsed time between two discovered anomalies. Such a cost function value is minimized to ensure that GREYHOUND converges to the potential anomalous states faster.
- **Anomaly count:** We use the number of unique anomalies discovered as a cost function. This is to maximize the discovery of potential anomalies including crashes.
- **Iteration time:** This is the duration for an AP to return to its `idle` state after initiating a communication (i.e., the time it takes to complete the inner loop in Algorithm 1). GREYHOUND aims to minimize this time to do stress test onto Wi-Fi clients. Clients are more likely to crash while attempting to frequently re-interact with the AP.

The cost function value is measured for each individual capturing a set of mutation probabilities. We note that a different set of mutation probabilities may direct the fuzzer to generate different sets of mutated packets (Line 28 in Algorithm 1). As a consequence, the measured cost function value might vary with respect to the set of chosen mutation probabilities. The objective of refining the mutation probabilities is to minimize or maximize a chosen cost function value for next fuzzing iteration. In GREYHOUND, we used the custom generational particle swarm optimization (PSO). We chose this optimization due to its superior performance in the light of the non-linear and stochastic behaviour present in the protocol model. The PSO is accomplished via the procedure `Particle_Swarm_Opt` (cf. Line 41 in Algorithm 1). We describe this procedure next.

Algorithm 2 GREYHOUND Particle Swarm Optimization

1: **Procedure:** `Particle_Swarm_Opt()`
2: **Send:** Sets of mutation probabilities X_i
3: **Receive:** Cost function value CF_i for X_i
4:
5: ▷ g captures PSO generation number
6: $g \leftarrow 0$
7: Let $[\![X]\!]_g$ be a population of sets of mutation probabilities
8: For each $X_i \in [\![X]\!]_g$, assign X_i randomly
9: ▷ initialize velocity, mutation probabilities and cost
10: **for** each $X_i \in [\![X]\!]_g$ **do**
11: ▷ send X_i to Algorithm 1 for fuzzing
12: `Greyhound_Core()` `<--` X_i
13: ▷ wait to receive cost from Algorithm 1
14: CF_i `<--` `Greyhound_Core()`
15: $pb_i \leftarrow X_i$; $f_i \leftarrow CF_i$; $v_i \leftarrow 0$
16: **end for**
17: ▷ initialize gb with initial best mutation probabilities
18: $gb \leftarrow X_m$ such that $CF_m \leq CF_i$ for any $X_i \in [\![X]\!]_g$
19: **repeat**
20: **for** each $X_i \in [\![X]\!]_g$ **do**
21: ▷ randomize r_1 and r_2
22: $r_1, r_2 \leftarrow$ `rand(0,1)`
23: ▷ calculate velocity (to shift mutation probabilities)
24: $v_i \leftarrow w((v_i + \eta_1 r_1) \cdot (X_i - pb_i) + \eta_2 r_2 \cdot (X_i - gb))$
25: ▷ calculate new mutation probabilities
26: $X_i \leftarrow X_i + v_i$
27: ▷ send X_i to Algorithm 1 for fuzzing
28: `Greyhound_Core()` `<--` X_i
29: ▷ wait to receive cost from Algorithm 1
30: CF_i `<--` `Greyhound_Core()`
31: ▷ update mutation probabilities
32: **if** $CF_i < f_i$ **then**
33: $pb_i \leftarrow X_i$; $f_i \leftarrow CF_i$
34: **end if**
35: ▷ update global best mutation probabilities
36: **if** $f_i < CF_m$ **then**
37: $gb \leftarrow pb_i$; $CF_m \leftarrow f_i$
38: **end if**
39: **end for**
40: $g \leftarrow g + 1$
41: **until** $g \geq$ *max_generations*

Procedure `Particle_Swarm_Opt`**:**

We capture sets of mutation probabilities X_i, in any given fuzzing iteration i, as the particle in a swarm. Recall that each X_i is a vector of length $2 \times \sum_{s \in \mathbb{S}} L_s$. Therefore, X_i can be considered as the position of a particle in a $2 \times \sum_{s \in \mathbb{S}} L_s$ dimensional space. The goal of PSO is to optimize the value of a chosen cost function via regulating the positions of the swarm of particles. Since the position of the particles is captured via the mutation probabilities, we refine the mutation probabilities via PSO and optimize our chosen cost function (e.g. anomaly count). The overall

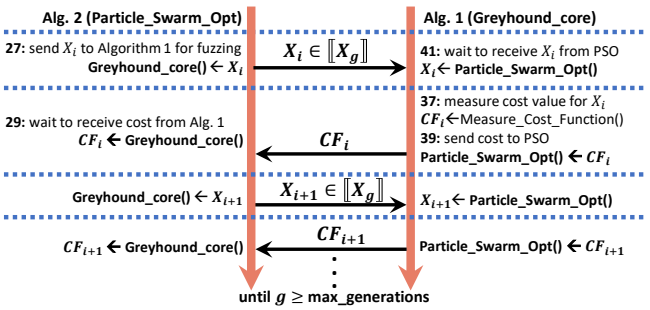


Fig. 7. Communication between `Greyhound_Core` fuzzing (Algorithm 1) and PSO (Algorithm 2)

process of refining the mutation probabilities is outlined in Algorithm 2. The positions of a swarm of particles $[\![X]\!]_g$ are first initialized (Line 8 in Algorithm 2). The position of each particle (i.e., X_i), which, in turn captures a set of mutation probabilities, is then sent to trigger a fuzzing iteration via procedure `Greyhound_core` (Line 12 in Algorithm 2). Once the fuzzing iteration in `Greyhound_core` finishes (i.e., the inner loop in Algorithm 1 terminates), the value of the cost function is measured (Line 14 in Algorithm 2) for the respective fuzzing iteration. It is worthwhile to note that the cost function value is also received from the procedure `Greyhound_Core` (Line 41 in Algorithm 1). Once the cost function value CF_i is computed for each $X_i \in [\![X]\!]_g$, the value of each X_i is refined via PSO to optimize the cost function value in next fuzzing iteration.

The refinement of mutation probabilities (i.e., X_i) is influenced via three crucial variables: the personal best mutation probabilities pb_i, velocity v_i and the global best mutation probabilities gb for the entire swarm $[\![X]\!]_g$ (see Lines 24-26 in Algorithm 2). The velocity component v_i is an offset to modify the mutation probabilities, whereas pb_i and gb components act as a memory to direct the search process towards optimal cost function value. Once the value of X_i is refined, X_i is sent to trigger the next fuzzing iteration via `Greyhound_core` (Line 28 in Algorithm 2) and the cost function value for the respective fuzzing iteration is measured accordingly (Line 30 in Algorithm 2). This process is repeated for each X_i in the swarm $[\![X]\!]_g$. During this process, the personal and global best mutation probabilities, i.e. pb_i and gb, respectively, are also updated according to the cost function value measured via `Greyhound_core` (Lines 33-37 in Algorithm 2). Finally, we repeat the process of refining the mutation probabilities and fuzzing for $max\_generations$ iterations. In our experiments, we set $max\_generations$ to be 200. Figure 7 illustrates the communication between the procedure `Greyhound_Core` and `Particle_Swarm_Opt` in our framework.

As observed in Algorithm 2, the refinement is also dictated via three configuration parameters: the inertia component w, the cognitive component η_1 and the social component η_2. w controls the speed that the particle X_i is originally heading to, whereas the cognitive component η_1 acts by increasing the tendency to follow its personal best position pb_i. Finally, the social component η_2 increases the tendency of X_i to follow the best position of the swarm (gb). In our experiments, we set the default values of w (0.729), η_1 (2.05) and η_2 (2.05) as embodied in the PAGMO toolkit [22].

Algorithm 3 GREYHOUND Validation Component

```
1: Procedure: Run_Validation()
2: Input: State S of Wi-Fi protocol model (cf. Figure 3)
3: Input: Packet P_r sent via Wi-Fi client
4: Output: Absence of anomaly (true or false)
5: ▷ Layers(P_r) is the set of layers constituting packet P_r
6: if ∃l ∈ Layers(P_r) s.t. l ∈ expected(S) then
7:     validated ← true
8: else
9:     validated ← false
10: end if
11: ▷ restore the set of expected layers if required
12: if expected(S) = ∅ then
13:     restore the original set of expected layers expected(S)
14: end if
15: return validated
```

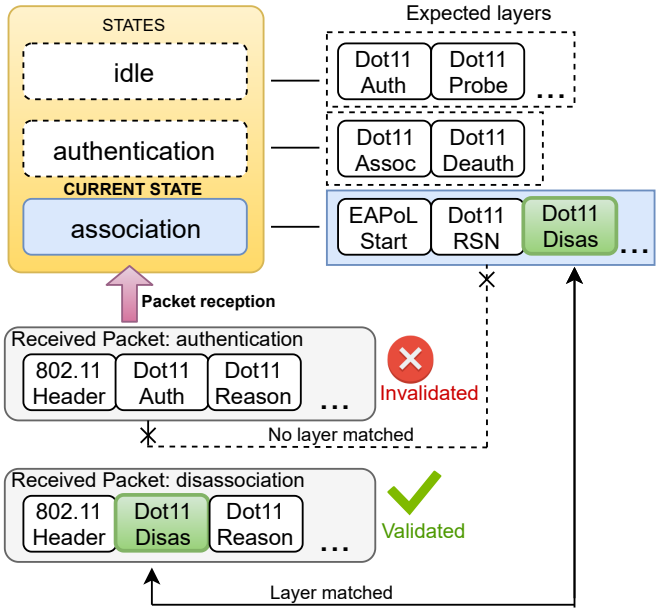


Fig. 8. Packet Dissection and Validation of GREYHOUND

4.2 Validation Component

GREYHOUND relies on the validation module to check whether the packets received from Wi-Fi client are compliant with the protocol model. To this end, GREYHOUND monitors the state of protocol model while interacting with the Wi-Fi client. Moreover, GREYHOUND assigns a predefined set of expected layers (that can be received from the Wi-Fi client) to each state (cf. Line 13 in Algorithm 1). When a packet is received from Wi-Fi client, its type is compared to the expected set of layers assigned with the current state of the protocol. Any mismatch is deemed as an anomaly. To discover crashes, GREYHOUND monitors whether the Wi-Fi client process was killed. To this end, GREYHOUND uses a global timer, which is initialized to a pre-determined threshold (large enough to detect crashes) and is reset to the initialized value each time a response is received from the Wi-Fi client. For the sake of brevity, this is not shown in Algorithm 1. Certain devices (e.g., ESP32 and ESP8266) also send a customized message to indicate a crash during communication.

Algorithm 3 outlines the `Run_Validation` module invoked in Algorithm 1. There is a special case where the expected set of layers was cleared during fuzzing. This happens when a mutated packet is sent from state S (Line 34 in Algorithm 1), as any response to such a mutated packet is considered an anomaly. Thus, at the end of the validation process, we restore the original set of expected layers (Line 13 in Algorithm 3).

Figure 8 provides an example of our validation component in practice. The current state of the protocol, as monitored by GREYHOUND, is `association`. At this state, the types of expected layers are shown alongside the state. As shown in Figure 8, two packets have been received from the Wi-Fi client via `association-Request` and `authentication`. We note that the types of packets in the `association-Request`, i.e., `Dot11 Assoc` and `Dot11 RSN` are compliant with the type of expected packets in the `association` state. By contrast, the types of packets in the `authentication`, i.e., `Dot11 Auth` and `Dot11 Reason`, are not compliant with the expected packet types in the `association` state and thus, GREYHOUND would flag them as anomalies.

Analysis of the validation component: It is worthwhile to note that we discover anomalies simply by monitoring the state of AP. This is possible due to the design of GREYHOUND and the unique nature of the Wi-Fi protocol. To consider the different situations that may arise while detecting anomalies, let us assume an arbitrary fuzzing iteration where GREYHOUND waits in state S after sending packets P' and P_{dup} to the Wi-Fi client (i.e Line 15 in Algorithm 1). The following scenarios may arise based on the response P_r received from the Wi-Fi client:

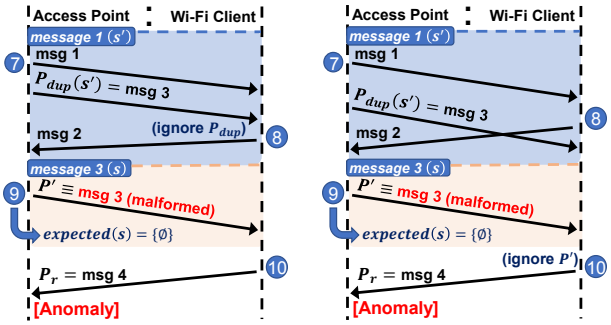
a) **Response for P':** If $P' = P$, it is a valid response and in Algorithm 3, $Layers(P_r) \cap expected(S) \neq \emptyset$. Therefore, no anomaly is detected. However, if $P' \neq P$ (i.e. P' was mutated), by design of `Greyhound_Core`, $expected(S) = \emptyset$. Therefore, response to $P' \neq P$ will be legitimately flagged as an anomaly via Algorithm 3.

b) **Response for P_{dup}:** Since $P_{dup} \neq P$ (cf. Algorithm 1), P_{dup} is always an invalid packet to be sent from state S. Thus, the response from the client for P_{dup} is not expected in state S of the AP. Due to our design, $Layers(P_r) \cap expected(S) = \emptyset$ (the expected layers in AP states are mutually exclusive) and therefore, an anomaly will be correctly highlighted via Algorithm 3. This case is void when $P_{dup} = \emptyset$.

c) **AP does not receive any response:** In this scenario, the validation in Algorithm 3 trivially passes, as $P_r = \emptyset$. This might happen if the client drops all invalid and mutated packets or crashes. We note that a crash is separately detected via a global timer. However, when the client legitimately drops invalid or mutated packets, the AP reaches idle state and eventually commences the next fuzzing iteration.

d) **Response for sent packets from previous fuzzing iteration:** We do not consider this case, as it is highly unlikely to occur in practice because of the timing constraints in Wi-Fi protocol.

e) **Response for sent packets during state $\neq S$:** This scenario occurs when packets sent from the Wi-Fi client



(a) Anomalous behaviour **(b)** Expected behaviour

Fig. 9. An Illustration of ambiguous scenarios during validation

are delayed. Specifically, let us assume the response P_r corresponds to the packet $P_{S'}$ sent via AP during the state $S' \neq S$. In an abuse of notation, we consider $P'(S')$ and $P_{dup}(S')$ to capture the sent packets P' and P_{dup} in state S'. Given these notations, there are three possible situations as follows:

- $\mathbf{P_{S'} = P'(S')}$ **and** $\mathbf{P'(S')}$ **was mutated:** Since $S \neq S'$, $expected(S) \cap expected(S') = \emptyset$. This holds even if $P' \neq P$, as $expected(S) = \emptyset$ in such case. Therefore, Algorithm 3 will legitimately catch an anomaly for the response to $P'(S')$ albeit being at a different state S.
- $\mathbf{P_{S'} = P'(S')}$ **and** $\mathbf{P'(S')}$ **was not mutated:** Since $P'(S')$ was a legitimate packet sent during state S', a response to this packet was valid during state S'. We note that the AP state machine waits for the response during fuzzing (Lines 15-17 in Algorithm 1). Thus, if the state machine had moved to a different state S', then it must be due to a response to some $P_{dup}(S'')$ sent in state $S'' \neq S'$ and $S'' \neq S$. Since $S \neq S'$, $expected(S) \cap expected(S') = \emptyset$ and this also holds when $P' \neq P$. Thus, Algorithm 3 will catch the response as an anomaly. We believe this should be highlighted as an anomaly, as the Wi-Fi client should not have responded to both the packets $P_{dup}(S'')$ and $P'(S')$ that trigger the same state transitions at the client.
- $\mathbf{P_{S'} = P_{dup}(S')}$: Let us first consider the case where $P' = P$. Thus, the set of expected layers $expected(S)$ is not cleared. If $Layers(P_r) \cap expected(S) \neq \emptyset$, then P_r is validated via Algorithm 3 and GREYHOUND will proceed to the next state. It is worthwhile to mention that despite $P_{dup}(S')$ being sent during an invalid state S', the client response arrives at the correct state S. Thus, the AP does not need to flag the response P_r as an anomaly in state S. If $Layers(P_r) \cap expected(S) = \emptyset$, an anomaly will be caught via Algorithm 3. Nonetheless, such an anomaly is caught in state S instead of state S' (as $Layers(P_r) \cap expected(S) = \emptyset$ by design).
  If $P' \neq P$, $expected(S) = \emptyset$. If $expected_{orig}(S)$ is the original set of expected layers at state S and $Layers(P_r) \cap expected_{orig}(S) = \emptyset$, Algorithm 3 will legitimately highlight an anomaly. However, if $Layers(P_r) \cap expected_{orig}(S) \neq \emptyset$, Algorithm 3 will

still catch an anomaly, as $expected(S) = \emptyset$. Nonetheless, we consider it to be an anomaly, as it cannot be inferred whether the client responds for the mutated packet P' or $P_{dup}(S')$. Due to such ambiguity, this needs to be further investigated by the developer. Figure 9(a) captures a scenario where the situation is an anomaly while Figure 9(b) is an expected scenario. As these scenarios cannot be differentiated by GREYHOUND and thus, both are flagged as anomalies conservatively.

5 EVALUATION

In this section, we describe our experimental setup and the evaluation findings in detail.

5.1 Evaluation Setup

We have implemented GREYHOUND with a Wi-Fi AP, i.e., Ralink RT3070 dongle. In order to improve overall responsiveness of GREYHOUND, we have developed a custom *RT3070USB* driver patch. Before GREYHOUND starts fuzzing, the dongle is configured to enable raw data reception and transmission. In data reception and transmission components, the fuzzer can interact with the Wi-Fi hardware using standard libraries (e.g., *packets socket*) available for most operating systems. Since GREYHOUND is targeted to discover existing and new vulnerabilities, we generically use the term *anomaly* to refer to both existing vulnerabilities and new non-compliant behaviours including crashes. The fuzzer and the validation components of GREYHOUND run on a machine with Intel i7 8-th generation CPU, 16GB DRAM memory, and Ubuntu 16.04 operating system. As to the four cost functions discussed in Section 4.1, we have used each of them to test the effectiveness of GREYHOUND against all subject Wi-Fi clients.

In evaluation, we have chosen five representative Wi-Fi clients: smartphone (Oneplus 5T), Raspberry Pi 3 Model B+, ESP8266, ESP32, and a medical device. Smartphones are widely used in daily life. Raspberry Pi is a popular embedded computing platform. ESP8266 and ESP32 are Wi-Fi modules playing important roles in IoT applications, such as energy metering, fire hazard alarm, and motion-based security camera [23], [24], [25]. The medical device is a smart syringe pump with Wi-Fi connection, which demands high-level data privacy in transmitting healthcare information of patients. We do not give the exact name of medical device due to a non-disclosure agreement signed with the company selling the medical device.

5.2 Summary of results

Table 1 outlines an overall summary of the results obtained through GREYHOUND. Each row in Table 1 lists a possible vulnerability, inconsistency (w.r.t. protocol specification) or crash discovered by GREYHOUND. It is worthwhile to emphasize that all such anomalies were discovered automatically by GREYHOUND. *Overall,* GREYHOUND *discovered four existing vulnerabilities across different devices (e.g., Raspberry Pi and Oneplus 5T) and seven new anomalies including three crashes*. These anomalies were discovered by GREYHOUND while communicating with the Wi-Fi client and in different states of the protocol model. The state, in which the respective anomaly was discovered, is also listed in the second column of Table 1. We also indicate, for each affected device, the fuzzing iteration number with which an anomaly was discovered (cf. Table 1). Finally, Table 1 precisely captures the set of our subject devices that are affected by a discovered anomaly. We note that the discovered vulnerabilities by GREYHOUND have been assigned three CVEs (common vulnerability exposures) already: *CVE-2019-12686*, *CVE-2019-12687* and *CVE-2019-12688*. At the timing of writing this paper, the details of these vulnerabilities are undisclosed for confidentiality. Moreover, the vulnerability associated with *CVE-2019-12587* has been awarded a bug bounty of 2,200 USD by the manufacturer of ESP8266 and ESP32.

Furthermore, we evaluated GREYHOUND to answer the following research questions.

RQ1: How effective is GREYHOUND in finding anomalies? We measure the effectiveness of GREYHOUND by analysing the total number of anomalies found over a limited number of iterations (cf. Table 2). After evaluating each device, GREYHOUND generates a report for each cost function to summarize the discovered anomalies in three categories: *(i) inconsistencies* with respect to protocol specification, *(ii) crashes* and *(iii) vulnerabilities*. While each anomaly discovered by GREYHOUND is automatically summarized as either a *crash* or an *inconsistency*, we manually perform a further in-depth analysis to identify whether an inconsistency is indeed a security *vulnerability*.

In Table 2, each cost function used by GREYHOUND has shown a different level of effectiveness in terms of discovered anomalies. Meanwhile, the effectiveness of a cost function is not consistent across all devices, because the implementations of Wi-Fi protocols differ significantly across different Wi-Fi clients. Therefore, it is desirable to use multiple cost functions to direct the fuzzing process of GREYHOUND. For ESP8266 and ESP32, the higher number of anomalies indicates that their protocol implementations are fragile and untrustworthy. By contrast, the medical device embraces more secure implementation, as the device contains a security patch for KRACK. In addition, it only includes implementations of EAP-TLS, EAP-PEAP, and EAP-TTLS that do not suffer from the Dragonslayer vulnerability, which refrains the device from being affected by Dragonslayer. However, GREYHOUND discovered that the medical device suffers from an old vulnerability $\mathfrak{A}\mathfrak{o}$, as captured by the CVE id *CVE-2008-5230*. This vulnerability allows conducting ARP poisoning.

RQ2: How do the different directed strategies converge in GREYHOUND?

We measure the convergence of GREYHOUND across different cost functions for all the devices (cf. Figure 10). All devices were run for 1,000 iterations except the medical device that was run for $\approx$ 200 iterations. This is due to the long testing time incurred for the medical device. Figure 10 exhibits the unique number of anomalies discovered with respect to the number of iterations. These diagrams confirm that all the cost functions direct GREYHOUND to discover unique anomalies.

The effectiveness of different cost functions varies in terms of finding anomalies across different devices. As an

TABLE 1
A summary of vulnerabilities / inconsistencies uncovered by GREYHOUND. The number beside each Affected tag captures the fuzzing iteration number in which the respective anomaly was discovered.

Vulnerabilities / Inconsistencies	Wi-Fi State	OnePlus 5T	Raspberry Pi 3	ESP8266	ESP32	Medical device
𝔄0 CVE-2008-5230	connected	Not affected	Not affected	Not affected	Not affected	**Affected** (91)
𝔄1 CVE-2017-13077 (KRACK)	message 3	Not affected	**Affected** (48)	Not affected	Not affected	Not affected
𝔄2 CVE-2017-13078 (KRACK)	message 3	Not affected	**Affected** (48)	Not affected	Not affected	Not affected
𝔄3 CVE-2017-13080 (KRACK)	connected	Not affected	**Affected** (50)	Not affected	Not affected	Not affected
𝔄4 CVE-2019-9499 (Dragonslayer)	EAP-Challenge	**Affected** (307)	**Affected** (95)	Not affected	Not affected	Not affected
𝔄5 New - Non compliance	EAP-Challenge	**Affected** (3)	**Affected** (9)	**Affected** (26)	**Affected** (2)	**Affected** (2)
𝔄6 New - Non compliance	EAP-Success/Fail	**Affected** (23)	**Affected** (13)	**Affected** (20)	**Affected** (7)	**Affected** (10)
𝔄7 New - Implementation issue (**New CVE-2019-12587**)	EAP-Success/Fail	Not affected	Not affected	**Affected** (91)	**Affected** (64)	Not affected
𝔄8 New - Crash (**New CVE-2019-12588**)	ALL States	Not affected	Not affected	**Affected** (2)	Not affected	Not affected
𝔄9 New - Crash (**New CVE-2019-12586**)	EAP-Identity EAP-Challenge	Not affected	Not affected	Not affected	**Affected** (421)	Not affected
𝔄10 New - Crash	EAP-Identity EAP-Challenge	Not affected	Not affected	**Affected** (228)	Not affected	Not affected
𝔄11 New - Malformed response	EAP-Identity EAP-Challenge	Not affected	Not affected	Not affected	**Affected** (261)	Not affected

TABLE 2
A summary of results per cost function

Discovery	Vulnerabilities / Inconsistencies / Crashes			
Cost Function	Iteration Time	Anomaly Period	Transitions	Anomaly Count
OnePlus 5T	0/1/0	1/2/0	0/1/0	0/2/0
Raspberry Pi 3	1/1/0	3/2/0	2/2/0	1/2/0
ESP8266	1/2/5	0/2/5	0/2/3	0/2/4
ESP32	0/2/2	0/1/0	1/2/2	1/1/0
Medical Device	0/1/0	0/2/0	1/2/0	1/2/0

TABLE 3
A summary of evaluation time for each device

Device	Max Iterations	Best case	Worst case
OnePlus 5T	1000	5 h. 50 min	6 h 40 min.
Raspberry Pi 3	1000	3 h. 32 min.	4 h. 52 min.
ESP8266	1000	1 h. 8 min.	1 h. 17 min.
ESP32	1000	1 h. 35 min.	1 h. 47 min.
Medical device	200	1 h. 52 min.	3 h. 41 min.

example, from Figure 10(c), it is feasible to conclude that the cost *iteration time* was best suited for ESP8266; while the Wi-Fi client implementation of ESP32 (cf. Figure 10(d)) was best directed with the cost *transitions*. Raspberry Pi 3, however, (cf. Figure 10(b)) is better tested via the cost *anomaly period*.

RQ3: How efficient is GREYHOUND with respect to generating error-prone inputs?

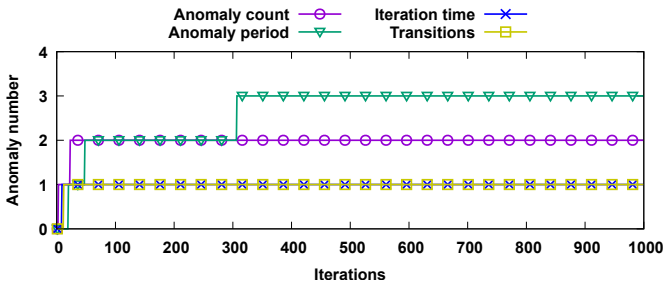
Timing between the AP and a Wi-Fi client significantly affects the evaluation process. For example, when GREYHOUND minimizes the cost *iteration time* (cf. Section 4.1), it is expected to decrease the average time to complete each evaluation. This may not hold while GREYHOUND aims to optimize other costs. Thus, to evaluate the efficiency of GREYHOUND, we measure both the best case and the worst case completion time of GREYHOUND after 1000 iterations. The best and the worst case evaluation time are captured in Table 3. From Table 3, we note that the timing behaviour is heavily dependent on the target Wi-Fi client implementation. This is because implementations in different devices use different thresholds for iteration time. For more specialized devices, such as the medical device, such thresholds are significantly higher and negatively affect the efficiency of GREYHOUND. As we had a limited time budget for accessing the medical device, we tested it for a maximum of 200 iterations. Nevertheless, the usage of *iteration time* as a cost aims to minimize the time to re-interact with the AP. By using this cost function, GREYHOUND improves the evaluation time for the medical device by a factor of two.

We report the results of the following three research questions for all devices except the medical device. The device was returned to the manufacturer due to the loan agreement signed earlier on.

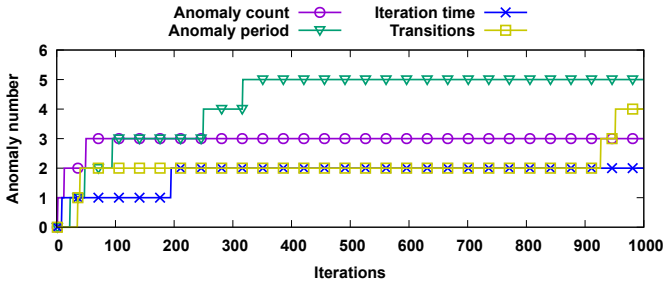
RQ4: Does the effectiveness of GREYHOUND depend on initial mutation probabilities?

As discussed in Section 4.1, GREYHOUND assigns a set of initial mutation probabilities to direct fuzzing specific layers and fields within each layer (cf. Figure 6). These mutation probabilities can be assigned randomly or they can be based on existing CVEs. Without loss of generality, we evaluate whether the effectiveness of GREYHOUND improves when the mutation probabilities are assigned based on CVE-2019-9499 for Dragonslayer. To this end, GREYHOUND assigned high mutation probability values for states only related to EAP. Specifically, all layers and fields that belong to the EAP state received an initial mutation probability of 80%. The rest of the layers and fields received zero probability for mutation. Such a configuration allows us to start the fuzzing process with a population that are likely to discover vulnerabilities similar to CVE-2019-9499.

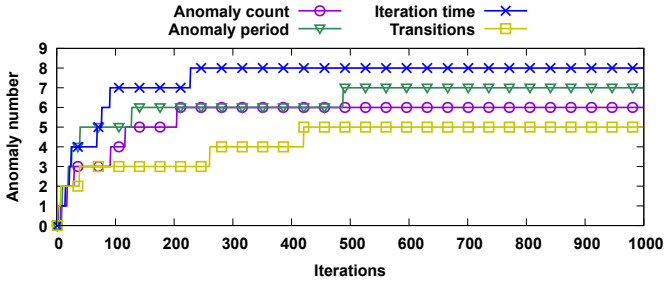
Table 4 outlines our findings. When evaluating with each cost function, we check the iteration number (i.e., the value alongside the X axis in Figure 10) where the first anomaly was discovered. We make several observations from Table 4. Firstly, a majority of the devices discover 𝔄5 as the first anomaly when mutation probabilities are assigned based on CVE-2019-9499 (row labelled "with CVE"). This is because 𝔄5 appears in the same state (i.e., EAP) as the Dragonslayer vulnerability. Secondly, 𝔄5 is discovered much faster (i.e., fewer fuzzing iterations) when the mutation probabilities are assigned based on CVE-2019-9499. This is expected, as GREYHOUND fuzzing process initially mutates the EAP



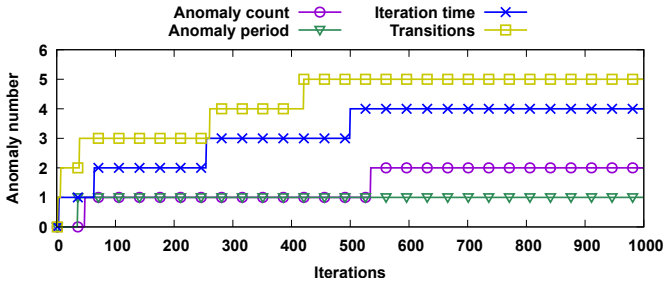
(a) Anomaly number w.r.t fuzzing iterations for OnePlus 5T



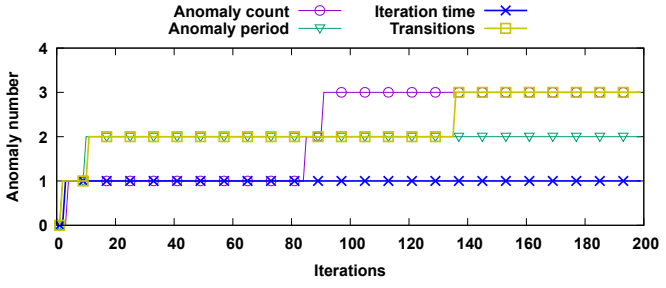
(b) Anomaly number w.r.t fuzzing iterations for Raspberry Pi 3



(c) Anomaly number w.r.t fuzzing iterations for ESP8266



(d) Anomaly number w.r.t fuzzing iterations for ESP32



(e) Anomaly number w.r.t fuzzing iterations for medical device

Fig. 10. The Convergence of GREYHOUND

TABLE 4
The effect of initializing mutation probabilities-based CVEs. For each device, we show the first anomaly discovered and the number of fuzzing iterations taken to discover the first anomaly.

Cost Function	Population Initialization	Max iterations until 1st anomaly is found / 1st anomaly			
		ESP8266	ESP32	Raspberry Pi 3	OnePlus 5T
Anomaly count	**with CVE**	**10 / 𝔄5**	**27 / 𝔄5**	**10 / 𝔄5**	**24 / 𝔄5**
	Randomly	07 / 𝔄8	45 / 𝔄5	56 / 𝔄5	51 / 𝔄5
Anomaly period	**with CVE**	**05 / 𝔄8**	**06 / 𝔄5**	**35 / 𝔄5**	**46 / 𝔄5**
	Randomly	16 / 𝔄8	31 / 𝔄5	64 / 𝔄5	105 / 𝔄5
Transitions	**with CVE**	**06 / 𝔄5**	**04 / 𝔄5**	**39 / 𝔄1,𝔄2**	**18 / 𝔄5**
	Randomly	02 / 𝔄8	21 / 𝔄5	48 / 𝔄5	44 / 𝔄5
Iteration time	**with CVE**	**04 / 𝔄5**	**07 / 𝔄5**	**25 / 𝔄5**	**40 / 𝔄5**
	Randomly	06 / 𝔄8	13 / 𝔄6	103 / 𝔄5	61 / 𝔄5

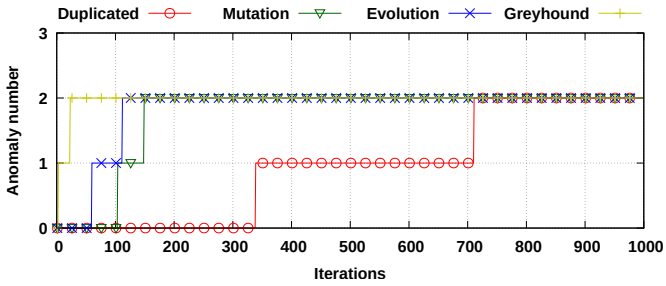
layer more frequently compared to other layers. Thirdly, for ESP8266, assigning mutation probabilities based on CVE resulted in more iterations to find the first anomaly, when compared to randomly initializing the probabilities. This is possible as the random initialization does not restrict the initial fuzzing iterations to find the anomaly 𝔄8 (beacon crash), which in turn, might appear in any state. Finally, for Raspberry Pi 3, we note that GREYHOUND discovers 𝔄1 and 𝔄2 as the first anomaly. This occurs when GREYHOUND uses the transitions in the state machine as the cost function. In such a case, despite mutating `EAP` layer more frequently than other layers, GREYHOUND still aims to cover the states due to the chosen cost function. As a result, it is possible for GREYHOUND to reach the `connected` state and send `message3` to trigger 𝔄1 and 𝔄2. In our evaluation, this happens even before 𝔄5 was triggered on Raspberry Pi 3.

Table 4 clearly captures that assigning the mutation probabilities from the CVEs helps GREYHOUND to converge faster in discovering anomalies that are similar to the respective CVEs (e.g., anomalies appearing in the same state).

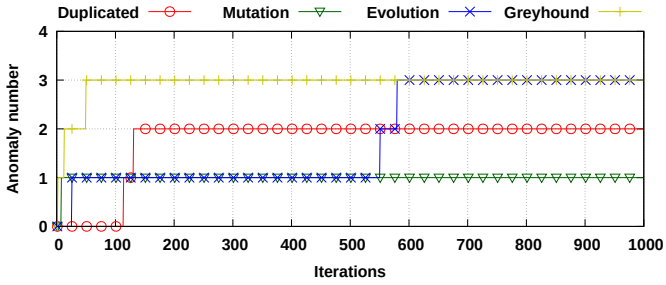
RQ5: How do the different design choices contribute in the effectiveness of GREYHOUND?

In this research question, we evaluate the rationale and contribution of the specific design choices made for GREYHOUND. To this end, we develop three additional variants of GREYHOUND: *(i)* GREYHOUND without any mutation and evolutionary components. Intuitively, this means the variant only generates packets from the state machine model and send these packets to the Wi-Fi client. The Wi-Fi client, however, may get these packets unexpectedly. Technically, in Algorithm 1, we make $P' = P$ in all iterations and disable lines 38-41 of Algorithm 1. *(ii)* GREYHOUND only with mutation. This means, GREYHOUND generates packets from the state machine model and mutates them before sending to the Wi-Fi client. However, these packets are never sent to the client unexpectedly and the mutation probabilities are not refined with the evolutionary approach. As a result, in Algorithm 1, we disable lines 30-36 and lines 38-41. *(iii)* The third variant of GREYHOUND considers both mutation and evolutions to send packets to the Wi-Fi client. However, all mutated packets are sent to the client at expected states. Formally, we disable lines 30-36 in Algorithm 1.

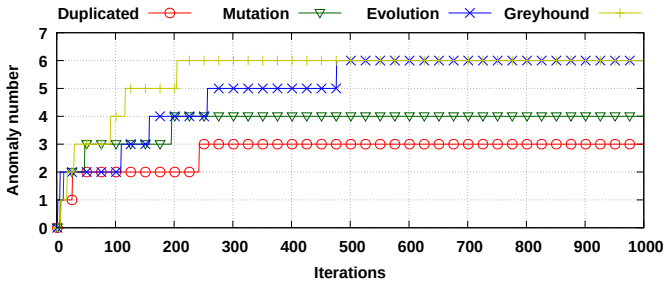
Figures 11(a)-(d) show our findings. For all the results reported in Figure 11, we keep the cost function to be the "Anomaly count" and we compare the number of anomalies discovered for each GREYHOUND variant. Variants *(i)*, *(ii)* and *(iii)* are labelled as "Duplicated", "Mutation" and



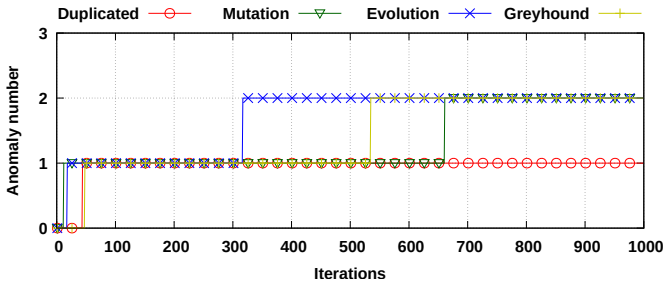
(a) Anomaly number w.r.t fuzzing iterations for OnePlus 5T



(b) Anomaly number w.r.t fuzzing iterations for Raspberry Pi 3



(c) Anomaly number w.r.t fuzzing iterations for ESP8266



(d) Anomaly number w.r.t fuzzing iterations for ESP32

Fig. 11. The impact of different design components in GREYHOUND

"Evolution", respectively. We make the following crucial observations from Figure 11. Firstly, we observe that it is crucial to mutate the packets and refine the mutation probabilities to discover anomalies. Except for OnePlus 5T, the variant "Duplicated" was incapable of discovering the same number of anomalies as other variants until 1000 iterations. For OnePlus 5T, the variant "Duplicated" took much longer iterations to discover the same number of anomalies as the other variants. Secondly, except for ESP32, GREYHOUND discovered the anomalies much faster than the other variants. This is expected as GREYHOUND employs a systematic combination of techniques embodied in the other three variants. For ESP32, it is noticeable that the variant

TABLE 5
GREYHOUND compared to other public tools

Comparison		Vulnerabilities / Inconsistencies / Crashes			
Tools	Maximum state	One Plus 5T	Raspberry Pi 3	ESP8266	ESP32
wifuzzit	association	0/0/0	0/0/0	0/0/0	0/0/0
wifuzzit (Client)	idle	0/0/0	0/0/0	0/0/0	0/0/0
wifuzz	association	0/0/0	0/0/0	0/0/0	0/0/0
IoTcube wfuzz	association	0/0/0	0/0/0	0/0/1	0/0/1
Greyhound	connected	1/2/0	3/2/0	1/2/2	1/2/1

TABLE 6
The time taken by GREYHOUND to discover each security vulnerability. "-" indicates that the respective tool was unable to discover the security vulnerability within the time budget.

Vulnerability	Device	Greyhound	Wifuzzit	Wifuzz	IoTcube
$\mathfrak{A}0$	Medical Device	55 min.	-	-	-
$\mathfrak{A}1$, $\mathfrak{A}2$	Raspberry Pi 3	3 min.	-	-	-
$\mathfrak{A}3$		5 min.	-	-	-
$\mathfrak{A}4$	OnePlus 5T	37 min.	-	-	-
	Raspberry Pi 3	11 min.	-	-	-
$\mathfrak{A}7$	ESP8266	22 min.	-	-	-
	ESP32	16 min.	-	-	-
$\mathfrak{A}8$	ESP8266	$<$ 1 min.	-	-	-
$\mathfrak{A}9$	ESP32	37 min.	-	-	-
$\mathfrak{A}10$	ESP8266	29 min.	-	-	-
New	ESP8266	N.A	-	-	**8 min.**
	ESP32	N.A	-	-	**29 min.**

"Evolution" got an anomaly faster than GREYHOUND. We argue that the additional introduction of the duplicated packets can sometimes delay the finding of anomalies that can appear by normal evolutionary mutation (i.e. the variant "Evolution"). However, it is expected that GREYHOUND will eventually trigger such anomaly with more test iterations, as observed in Figure 11(d). Finally, we note that it is not sufficient to simply use mutation to trigger all the anomalies. Specifically, for Raspberry Pi 3 and ESP8266, the variant "mutation" cannot discover all the anomalies in 1000 iterations. For the rest of the devices, this variant discovers anomalies much slower than GREYHOUND.

RQ6: How effective is GREYHOUND with respect to existing Wi-Fi Fuzzing tools?

We choose three freely available Wi-Fi fuzzers, namely *wifuzzit* [10], *wifuzz* [11] and *IoTcube wfuzz* [12] (state-of-the-art) to compare the effectiveness of GREYHOUND with respect to existing works. The choice of these fuzzing tools is motivated by the rationale that they are the closest to the objective of our work, when compared with the other available fuzz testing methodologies. Nonetheless, such tools can detect only crashes and therefore, they are fundamentally incapable of discovering other security vulnerabilities, such as *CVE-2019-12587* (cf. Table 1) and anomalous behaviours. Moreover, these tools are limited in regards by the depth of states that can be fuzzed. For example, while GREYHOUND can quickly reach states such as `EAP` and `connected`, we observe that the existing Wi-Fi fuzzers are only able to reach the `association` state at best.

To evaluate the comparative effectiveness of GREYHOUND, each competitive fuzzing tool is tested on our subject devices. Except for *wfuzz*, which finishes the fuzzing in 30 minutes, other tools are evaluated until the worst-case time taken by GREYHOUND, as outlined in Table 3. This is required for a fair comparison with GREYHOUND.

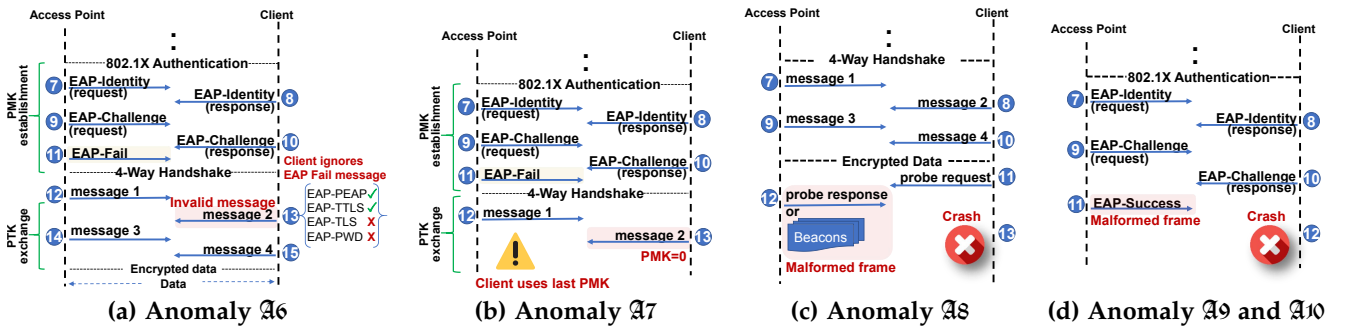


Fig. 12. An Illustration of New Anomalies (A6 to A10) Discovered by GREYHOUND

In general, *wifuzzit* and *wifuzz* require more than one day to finish their respective fuzzing processes. Despite the fast fuzzing session conducted by *IoTcube wfuzz* (i.e. 30 minutes), it is worthwhile to note that its users must keep a database with up-to-date test cases. However, GREYHOUND does not need to maintain any such database, all test cases are automatically generated via the approach outlined in Algorithm 1.

Table 5 outlines the summary of unique vulnerabilities, anomalies and crashes triggered by each competitive tool and GREYHOUND. It is worthwhile to mention that only *wifuzzit (Client)* is able to fuzz the Wi-Fi client, whereas the AP implementation is fuzzed by *wifuzzit*, *IoTcube wfuzz* and *wifuzz*. We note that the state machine model embodied in GREYHOUND facilitates us to fuzz the implementation of Wi-Fi client. Consequently, the overall result clearly shows that our GREYHOUND approach not only detects more crashes for ESP8266, but also indicated vulnerabilities and anomalies across all devices. Although *wifuzzit (Client)* can fuzz the Wi-Fi client implementation, it failed to discover any anomaly across all the test devices. This is because the crashes and vulnerabilities might appear deep in the Wi-Fi state, such as `EAP`. Our evaluation indicates that *wifuzzit (Client)* was unable to reach such state during its fuzzing process. Specifically, *wifuzzit (Client)* failed to trigger a crash caused by malformed beacon frame or association response. This further shows the rationale behind the mutation operators and the comprehensive Wi-Fi model embodied within GREYHOUND.

Table 6 compares the time taken by GREYHOUND in discovering each anomaly. It is worthwhile to mention that A0-A4 are existing security vulnerabilities, whereas A6-A10 are new vulnerabilities discovered by GREYHOUND (cf. Table 1). As observed from Table 6, GREYHOUND is not only capable to discover new vulnerabilities, but it also discovers existing vulnerabilities effectively in contrast to state-of-the-art fuzzers. Finally, as shown in Table 5 and Table 6, *IoTcube wfuzz* discovered a new anomaly shared across both ESP32 and ESP8266 AP implementations. GREYHOUND does not have a Wi-Fi AP model yet, thus it does not fuzz AP implementations to detect the crash found by *IoTcube wfuzz*.

5.3 Description of Anomalies

In this section, we will briefly present the anomalies discovered by GREYHOUND. GREYHOUND automatically discovers critical vulnerabilities such as variants of KRACK and Dragonslayer A4 in multiple Wi-Fi clients (cf. Table 1). Besides these known vulnerabilities, GREYHOUND has also discovered seven new anomalies as discussed in the following.

Anomaly A5: Anomaly A5 indicates the absence of verification on the "Version" field in the `EAPoL` layer of a Wi-Fi packet (cf. Figure 6). The lack of such verification results in the communication to proceed even with an arbitrary value in the "Version" field. Albeit not raising a security concern, such a behaviour captures a violation of the protocol standard.

Anomaly A6: A6 captures a non-compliant behaviour of Wi-Fi implementation in Wi-Fi clients that support EAP-PWD and EAP-TLS (cf. Figure 12(a)). Specifically, according to the RFC 3748 [26], following an `EAP-Fail` message after `EAP-Challenge`, the client must terminate the respective communication. GREYHOUND discovers a scenario (cf. Figure 12(a)) where several Wi-Fi clients, though implementing EAP-PWD and EAP-TLS, simply ignore the `EAP-Fail` message and proceed to the `4-way Handshake`. Although this is highlighted as a non-compliant behaviour, GREYHOUND does not consider it as a vulnerability, because the affected Wi-Fi clients proceed with the correct PMK to the `4-way Handshake` stage (cf. Section 2). Such a PMK was already established before the reception of the `EAP-Fail` message.

Anomaly A7: A7 is similar to A6, but A7 exposes a vulnerability for ESP8266 and ESP32. As shown in Figure 12(b), despite receiving an `EAP-Fail` message, both ESP8266 and ESP32 proceed to the `4-Way Handshake` stage. Worse, both devices surprisingly drop the PMK obtained during `EAP-Challenge` state and proceed with the `4-Way Handshake` using the last PMK. We investigated and discovered that the last PMK is always zero at start. Thus, it is possible for a rogue AP to easily hijack any ESP32/8266 connectivity to an enterprise network. This is doable by sending an `EAP-Fail` message before the original `EAP-Success` message is sent out by the authentic AP. As a result, we consider this to be a vulnerability of the Wi-Fi client implementations in ESP32/8266. *This vulnerability is now assigned a CVE id CVE-2019-12587* [27]. We also win a bug bounty of 2.2K USD for reporting this security vulnerability to the manufacturer.

Anomalies A8-A10: GREYHOUND identified three crashes mostly affecting low-power devices. A8 affects only ESP8266 and can be practically triggered in all states (cf. Figure 12(c)): a crash is triggered when a malformed beacon frame with

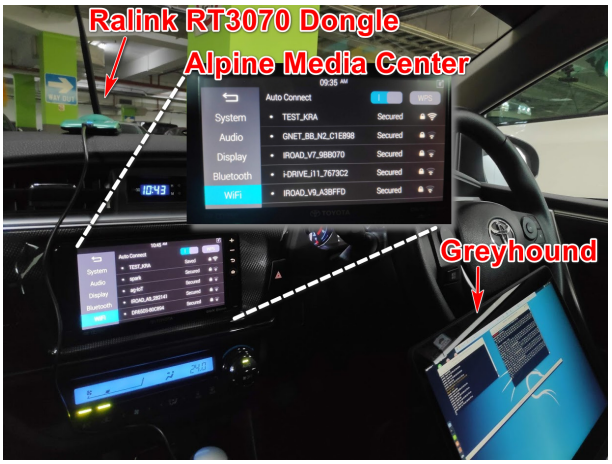


Fig. 13. GREYHOUND Setup with Toyota Altis

incorrect length is broadcast to ESP8266. 𝔄9 and 𝔄10 manifest other crashes during `EAP-Challenge` state of ESP32 and ESP8266, respectively (cf. Figure 12(d)). Specifically, GREYHOUND found that sending an `EAP-Success` message with additional padding caused ESP32 to crash. ESP8266 crashes in similar scenarios even without the padding. Note that RFC 3748 clearly addresses the proper handling of `EAP-Success` message in an `EAP-Challenge` state. According to RFC 3748, a Wi-Fi client being in `EAP-Challenge` state must always discard `EAP-Success` message. *These crashes are now assigned CVE ids CVE-2019-12586* [28] *and CVE-2019-12588* [29].

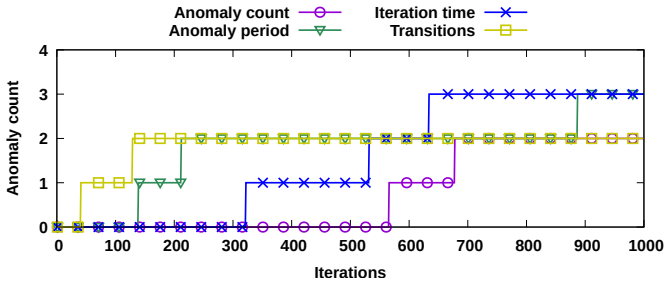
Anomaly 𝔄11: Finally, we discovered a non-compliant behaviour 𝔄11, which causes ESP32 to communicate with a malformed EAP response during the `EAP-Identity` and `EAP-Challenge` states. According to RFC 3748, Wi-Fi clients must not respond to malformed messages.

5.4 Case Study with Automotive Head Units

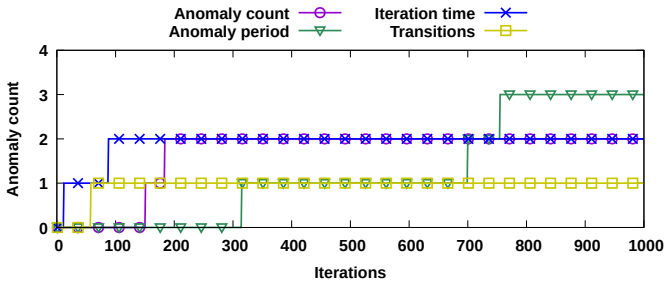
In this study, we choose three car automotive head units to validate the scalability and deployment potential of GREYHOUND. To this end, we first target the head unit of Toyota Altis, dated from 2016 with firmware version *01.04.1600*. The car's head unit contains a system that supports internet access through its Wi-Fi implementation. Moreover, the presence of old firmware, which is dated before KRACK vulnerabilities were discovered, gave us an incentive to perform tests against the car media center. The configuration of GREYHOUND is captured in Figure 13. A notebook running GREYHOUND (to the right) is used. GREYHOUND, in turn, generates an access point with the name *TEST_KRA* (shown in the media center screen). Subsequently, GREYHOUND starts fuzzing in line with the methodologies outlined in Algorithm 1. This is accomplished via leveraging the RT3070 dongle (on top of the car panel) to communicate systematically fuzzed packets.

Similar to the medical device, the car head unit exhibits a long delay between reconnection with the fuzzer, thus, making it impractical, during our limited access to the car, to complete substantial number of fuzzing iterations. Nevertheless, after running GREYHOUND for about ten minutes, it was already possible to discover anomaly 𝔄3 (CVE-2017-13080 KRACK). We note that the car Toyota Altis features a head unit from the vendor Alpine, which provides custom media center solutions for other car manufacturers.

We have extended our study to two different automotive head units, i.e., UGAR EX8-L and XY Auto (used by different vendors such as Podofo A2222A1, Bingfan YT9216B-07 and Midcourse MC7023E-N1), both running on Android 8.1 operating system. These are sold separately from a car. Such a study allowed us to perform a more complete analysis of the automotive head units. We evaluated both head units against each cost function and we compute the number of anomalies with respect to fuzzing iteration (cf. Figures 14(a) and 14(b)). We discovered that both head units are vulnerable to Dragonslayer (𝔄4). However, we noted both units are resilient to KRACK, as their respective Android systems are updated with security patches from November, 2017 and May, 2018. It is worthwhile to mention that these results are expected as their system is similar to OnePlus 5T, which also runs Android 8.1 with a security patch from October, 2018.



(a) Anomaly number w.r.t fuzzing iterations for UGAR



(b) Anomaly number w.r.t fuzzing iterations for XY Auto

Fig. 14. The Convergence of GREYHOUND

The outcome of our study concludes that the owners of Alpine head units are vulnerable to anomaly 𝔄3 if the security patch for KRACK is not applied. Moreover, owners of the aforementioned UGAR and XY Auto head units are vulnerable to anomaly 𝔄4 if they have not updated their respective units.

6 RELATED WORK

Wi-Fi Vulnerabilities: GREYHOUND is inspired from recent works [5] [13] that have discovered vulnerabilities in wireless protocols. KRACK [5] discovers implementation and design flaws in the Wi-Fi 4-Way Handshake. Dragonslayer

indicates a vulnerability in the `EAP-Challenge` state for EAP-PWD authentication method [13], affecting many devices that rely on an enterprise network for improved security. These works require careful manual investigation of the protocol standard to discover design and implementation flaws. By contrast, GREYHOUND is an automated technique to streamline the discovery of vulnerabilities including but not limited to KRACK and Dragonslayer. Moreover, GREYHOUND can easily be extended for testing a variety of protocols. This can be accomplished by providing the state machine model of the targeted protocol.

Fuzzing Network Protocols: Directed greybox fuzzing is a well known software testing strategy [9], [16], [17], [18]. However, there has been limited to no attempt to adapt such methodologies for testing complex systems such as Wi-Fi. GREYHOUND is the first approach towards fuzzing complex wireless protocols. Existing works on fuzzing [30], [31], [32] wireless protocols only support testing driver implementations against buffer overflow or null pointer deference. These works are not capable to detect vulnerabilities such as KRACK and Dragonslayer, as they only support a limited number of states to fuzz, namely `idle`, `authentication` and `association` (cf. Figure 3). Moreover, tests need to be configured manually (e.g., fields to fuzz) for the fuzzing to progress. In contrast to these works, GREYHOUND provides a holistic and fully automated approach to fuzz all the states associated with Wi-Fi protocol. GREYHOUND is also orthogonal to other works on network protocol testing [7], [19], [20] that focus on text structured protocols such as `ftp` and `http`, but ignores wireless protocols. Finally, the objective of a recent work on IoT fuzzing [33] is orthogonal to the approach proposed in GREYHOUND. Specifically, the work [33] aims to discover memory corruptions in IoT devices via fuzzing the application layers (e.g., `JSON` and `XML` formats) through the mobile app. By contrast, GREYHOUND aims to discover vulnerabilities in wireless protocols via fuzzing Wi-Fi packets, it does not intend to fuzz application layers and it does not need to rely on mobile apps. Moreover, by design and as shown in our evaluation, GREYHOUND can discover security vulnerabilities beyond memory corruptions.

Fuzzing Optimization: A number of works have been proposed to improve the effectiveness of fuzzing process [34], [35], [36]. These approaches primarily target fast generation of interesting inputs via application-aware data and control flow features [36], using static and dynamic analysis to leverage information about program states [35] and via optimizing the choice of effective mutation operators [34]. None of these works are directly applicable to fuzzing Wi-Fi protocols. Moreover, these works are primarily targeted to discover crashes and they often require instrumenting program binaries to extract program state information. In contrast, our work is generally targeted to discover anomalous protocol behaviours including but not limited to crashes (e.g. encryption bypass). Moreover, our GREYHOUND approach does not involve any instrumentation and works in a fully automated fashion.

Model-based Testing: Model-based testing has recently been applied to validate IoT communication [37]. Specifically, it applies active automata learning to understand IoT communication protocol. The learned protocol model is then used for testing. This work does not employ an evolutionary approach. As shown in our evaluation, an evolutionary approach is critical for effective fuzzing of Wi-Fi protocol implementations. Finally, the automata learning approach of existing model-based testing framework [37] is complementary to our approach. Specifically, such automata learning can be considered to learn the Wi-Fi protocol model in GREYHOUND. Model-based security testing has also been studied for OAuth 2.0 implementations [38]. However, this work is not directly applicable for fuzzing Wi-Fi protocols and it does not involve any evolutionary method in fuzzing.

Fuzzing Wi-Fi Handshake: The closest to GREYHOUND is a work related to the testing of 4-Way Handshake [21]. In this work, a model of the 4-Way Handshake was constructed to test Wi-Fi client implementations. However, in contrast to GREYHOUND, such a model ignores the behaviour of the Wi-Fi client. Moreover, GREYHOUND differs from this approach in multiple other aspects. Firstly, the proposed approach [21] is incomplete in the sense that it only considers the model of *handshake*. Thus, it is incapable of handling other layers of wireless protocols, failing to discover vulnerabilities such as Dragonslayer. Secondly, all the test strategies in the approach are manually defined and specifically crafted for testing handshaking mechanism only. By comparison, GREYHOUND is a more generic approach for testing arbitrary wireless protocols and all the tests in GREYHOUND are generated automatically. Thirdly, GREYHOUND is modular and can easily be extended for fuzzing other protocols via adding protocol models [39].

Verification of Network Protocols: Our work is orthogonal to the chain of works that aim to perform verification of network protocols [40]. In contrast to these works, GREYHOUND has a significant testing flavour and it can be used to provide a concrete test case that exhibits vulnerabilities or non-compliant behaviours in Wi-Fi client implementations.

To the best of our knowledge, GREYHOUND is the first holistic approach to automatically test arbitrary Wi-Fi clients.

7 THREATS TO VALIDITY

GREYHOUND's effectiveness depends on some key factors:

Choice of initial probabilities: Incorrectly assigning mutation probabilities to basic Wi-Fi layers, such as *Wi-Fi Mac header*, can result in several iterations of GREYHOUND being stuck in the `idle` state of the protocol model. In such cases, GREYHOUND could cause a Wi-Fi client to never attempt a connection to the AP, as packets received by the Wi-Fi client might be invalid when the *Wi-Fi Mac header* is mutated.

Choice of cost function: GREYHOUND does not guarantee the accomplishment of finding the maximum anomalies using one single cost function in testing a Wi-Fi client. Each cost function focuses on a different interaction with the Wi-Fi client while the number of anomalies may highly depend on the specific Wi-Fi client implementation in a device. Thus, it is important to use different cost functions to cover different characteristics of Wi-Fi client implementations.

Model robustness: The protocol model must be well implemented in order to correctly interact with Wi-Fi clients. An incorrect or incomplete model may lead GREYHOUND

to misinterpret anomalies or limit its coverage in finding anomalies.

Validation: Currently, GREYHOUND's packet validation relies mostly on the checks implemented to investigate the expected set of layers in each state of the protocol model. An incomplete set of packets may miss potential anomalies attributed to the protocol. To mitigate this, we carefully followed the protocol standards and included the type of expected layers in each state of the protocol model.

8 CONCLUSION

We propose GREYHOUND, an automated and directed testing methodology to discover anomalies in the implementation of wireless protocols. It employs a holistic model of Wi-Fi protocol to systematically interact with Wi-Fi compliant devices. An appealing feature of GREYHOUND is that it can optimize the testing process via multiple cost functions and with the objective to discover anomalies in arbitrary Wi-Fi client implementation. We extensively evaluated the effectiveness of GREYHOUND with five representative Wi-Fi compliant devices. Our evaluation reveals that GREYHOUND discovers critical security vulnerabilities such as KRACK and Dragonslayer. Moreover, it uncovers seven new anomalies including three crashes and one new vulnerability. All these new anomalies have been confirmed by the respective manufacturers. In the future, we plan to extend GREYHOUND for Bluetooth compliant devices.

GREYHOUND is just one step forward to push the state-of-the-art in automated validation of current and next-generation wireless protocols. Given the critical vulnerabilities discovered in such protocols, we believe the need for automated testing is urgent and concrete. We hope that GREYHOUND provides a baseline to extend and streamline the research in systematic testing of wireless protocol implementations. For reproducibility and research, GREYHOUND source code is available upon request to *sweyntooth@gmail.com*.

Acknowledgement: We thank the anonymous reviewers for their insightful comments. The research is partially supported by Keysight Technologies grant no. RTKS171003.

REFERENCES

[1] Brian P Crow, Indra Widjaja, Jeong Geun Kim, and Prescott T Sakai. Ieee 802.11 wireless local area networks. *IEEE Communications magazine*, 35(9):116–126, 1997.

[2] Changhua He and John C Mitchell. Analysis of the 802.11 i 4-way handshake. In *Proceedings of the 3rd ACM workshop on Wireless security*, pages 43–50. ACM, 2004.

[3] CHJC Mitchell and Changhua He. Security analysis and improvements for IEEE 802.11 i. In *The 12th Annual Network and Distributed System Security Symposium (NDSS'05) Stanford University, Stanford*, pages 90–110. Citeseer, 2005.

[4] Wi-Fi Alliance. WPA3 specification v1.0. https://www.wi-fi.org/downloads-registered-guest/WPA3_Specification_v1.0.pdf, January 2018. https://www.wi-fi.org/file/wpa3-specification-v10.

[5] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1313–1328, New York, NY, USA, 2017. ACM.

[6] Noah Apthorpe, Dillon Reisman, and Nick Feamster. A smart home is no castle: Privacy vulnerabilities of encrypted IoT traffic. *arXiv preprint arXiv:1705.06805*, 2017.

[7] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. SNOOZE: Toward a stateful network protocol fuzzer. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security*, pages 343–358, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2329–2344, New York, NY, USA, 2017. ACM.

[9] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[10] Laurent Butti. wifuzzit: a 802.11 wireless fuzzer. https://github.com/0xd012/wifuzzit, January 2013. https://www.blackhat.com/presentations/bh-europe-07/Butti/Presentation/bh-eu-07-Butti.pdf.

[11] Roberto Paleari. wifuzz, March 2011. https://code.google.com/archive/p/wifuzz/.

[12] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Poster: Iotcube: an automated analysis platform for finding security vulnerabilities. In *2017 IEEE Symposium on Poster presented at Security and Privacy (SP). IEEE*, 2017.

[13] CVE-2019-9499. ble from MITRE, CVE-ID CVE-2019-9499., March 2019.

[14] Kiran Jot Singh and Divneet Singh Kapoor. Create your own internet of things: A survey of IoT platforms. *IEEE Consumer Electronics Magazine*, 6(2):57–68, 2017.

[15] Jyh-Cheng Chen, Ming-Chia Jiang, and Yi-wen Liu. Wireless lan security and ieee 802.11 i. *IEEE Wireless Communications*, 12(1):27–36, 2005.

[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[17] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *ASE*, pages 543–553, 2016.

[18] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *ICSE*, 2019.

[19] Serge Gorbunov and Arnold Rosenbloom. AutoFuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.

[20] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz, April 2017. https://wpa3.mathyvanhoef.com/.

[21] Mathy Vanhoef, Domien Schepers, and Frank Piessens. Discovering logical vulnerabilities in the Wi-Fi handshake using model-based testing. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 360–371. ACM, 2017.

[22] Francesco Biscani, Dario Izzo, and Marcus Märtens. esa/pagmo2: pagmo 2.6, 2017.

[23] Win Hlaing, Somchai Thepphaeng, Varunyou Nontaboot, Natthanan Tangsunantham, Tanayoot Sangsuwan, and Chaiyod Pira. Implementation of WiFi-based single phase smart meter for internet of things (IoT). In *2017 International Electrical Engineering Congress (iEECON)*, pages 1–4. IEEE, 2017.

[24] Lakshmana Phaneendra Maguluri, Tumma Srinivasarao, R Ragupathy, Maganti Syamala, and NJ Nalini. Efficient smart emergency response system for fire hazards using IoT. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 9(1):314–320, 2018.

[25] Gopinath Muruti, Fiza Abdul Rahim, Md Zawawi, and Nabil Ahmad. Motion activated security camera using Raspberry Pi: An IoT solution for room security. *Advanced Science Letters*, 24(3):1698–1701, 2018.

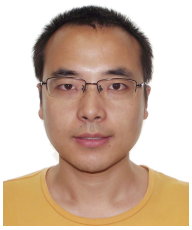
[26] Bernard Aboba, L Blunk, J Vollbrecht, James Carlson, and Henrik Levkowetz. RFC 3748-extensible authentication protocol (EAP). *Network Working Group*, 2004.

[27] CVE-2019-12587. Available from MITRE, CVE-ID CVE-2019-12587., June 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12587.

[28] CVE-2019-12586. Available from MITRE, CVE-ID CVE-2019-12586., June 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12586.

[29] CVE-2019-12588. Available from MITRE, CVE-ID CVE-2019-12588., June 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12588.
[30] Manuel Mendonça and Nuno Neves. Fuzzing wi-fi drivers to locate security vulnerabilities. In *2008 Seventh European Dependable Computing Conference*, pages 110–119. IEEE, 2008.
[31] Laurent Butti and Julien Tinnes. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology*, 4(1):25–37, 2008.
[32] Sylvester Keil and Clemens Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.
[33] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.
[34] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1949–1966, 2019.
[35] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 627–637, 2017.
[36] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.
[37] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. Model-based testing iot communication via active automata learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 276–287, 2017.
[38] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on oauth 2.0 implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 651–662, 2016.
[39] Debiao He, Neeraj Kumar, Huaqun Wang, Lina Wang, Kim-Kwang Raymond Choo, and Alexey V. Vinel. A provably-secure cross-domain handshake scheme with symptoms-matching for mobile healthcare social network. *IEEE Trans. Dependable Sec. Comput.*, 15(4):633–645, 2018.
[40] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *CAV*, pages 696–701, 2013.

Matheus E. Garbelini graduated with a bachelor of engineering degree in Electronics from Pontifical Catholic University of Paraná in Brazil. Currently Matheus is a PhD student at Singapore University of Technology and Design (SUTD). His research interests include Wireless Security, cyber-physical Systems and IoTs and embedded systems.



Chundong Wang received the Bachelors degree in computer science from Xian Jiaotong University in 2008, and the Ph.D. degree in computer science from National University of Singapore in 2013. Currently he is an assistant professor at ShanghaiTech University. Before joining ShanghaiTech University, he was a research fellow in Singapore University of Technology and Design (SUTD). Before joining SUTD, he worked as a scientist in Data Storage Institute, A*STAR, Singapore. He has published a number of papers in IEEE TC, ACM TOS, DAC, DATE, LCTES, USENIX FAST, etc. His research interests include data storage, non-volatile memory and computer architecture.

Sudipta Chattopadhyay received the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2013. He is an Assistant Professor with the Information Systems Technology and Design Pillar, Singapore University of Technology and Design, Singapore. In his doctoral dissertation, he researched on Execution-Time Predictability, focusing on Multicore Platforms. He seeks to understand the influence of execution platform on critical software properties, such as performance, energy, robustness, and security. His research interests include program analysis, embedded systems, and compilers. Mr. Chattopadhyay serves in the review board of the IEEE Transactions on Software Engineering.