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.

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.

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, CVE-2019-12587, and CVE-2019-12588. We also compare our GREYHOUND approach with three existing Wi-Fi testing tools: wifiuzzit [10], wifiuzz [11], and IoTube wifiuzz [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.

Fig. 2: An illustration of KRACK and Dragonslayer vulnerabilities

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-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 an AP. The packet exchange is initiated by the AP with a beacon frame, which is followed by an association request from the Wi-Fi client. The AP responds with an authentication request, and the Wi-Fi client sends an authentication response. This exchange of messages establishes a simple open Wi-Fi connection, and the AP initiates the EAP (Extensible Authentication Protocol) to authenticate the Wi-Fi client. EAP initiates the Enterprise Authentication Protocol (EAP) exchange through exchanging EAP-Identity request and response, and EAP-Challenge request and response. 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 9), 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 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 in Figure 2(b) to the invalid EAP-Challenge request 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).
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 order to 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.

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 3 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 message 3 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 11-17). Once the AP receives a packet $P$, 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 23). 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 24 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 \( P_{\text{hist}} \) and selects a packet \( P_{\text{dup}} \) from \( P_{\text{hist}} \) to communicate with the Wi-Fi client (Lines 30-35). The rationale behind such a communication is to send a packet that is invalid regarding the current state of the protocol. Moreover, even though \( P_{\text{dup}} \) triggers a legitimate transition to the Wi-Fi client (e.g., when the reception of \( P_{\text{dup}} \) 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 57) 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 \( X_{i+1} \) is 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 13).

### Algorithm 1 Main Steps of GREYHOUND

```plaintext
1: Procedure: Greyhound_Core()
2: 3: \( i \leftarrow 0 \) \( \triangleright \) \( i \) captures fuzzing iteration
4: \( \triangleright \) generate Wi-Fi protocol model (cf. Figure 3)
5: \( M_{WIFI} \leftarrow \text{Generate_Protocol_Model}() \)
6: \( \triangleright \) wait to receive mutation probabilities from PSO
7: \( X_i \leftarrow \text{Particle_Swarm_Opt}() \)
8: \( \triangleright \) initialize history of sent packets
9: \( P_{\text{hist}} \leftarrow P_{\text{.dup}} \leftarrow \emptyset \)
10: repeat
11: \( \triangleright \) Wait for the AP to be in \( \text{idle} \) state
12: \( \triangleright \) assign expected layers for received packets
13: For each \( S \in M_{WIFI} \), assign \( \text{expected}(S) \)
14: repeat
15: \( \triangleright \) Wait for the Wi-Fi client to communicate
16: \( \text{goto} \) line 39 if timeout occurs at line 15
17: \( \triangleright \) Let the AP receive packet \( P_r \) from the Wi-Fi client
18: \( \triangleright \) monitors states and checks anomalies
19: \( \theta_{\text{anom}} \leftarrow \text{Run_Validation}(P_r, S) \)
20: \( \triangleright \) exits the fuzzing iteration in case of anomalies
21: if \( \theta_{\text{anom}} \) is false then
22: \( \text{goto} \) line 39
23: end if
24: \( S \leftarrow \text{Get_Current_State}(M_{WIFI}, P_r) \)
25: \( \triangleright \) generate a valid packet from the model
26: \( P \leftarrow \text{Get_Packet_from_Model}(M_{WIFI}, S) \)
27: \( \triangleright \) generate fuzzed packets from \( P \) via mutation
28: \( P' \leftarrow \text{Mutate_Packet}(P, X_i) \)
29: \( \triangleright \) send fuzzed packets \( P' \) to the Wi-Fi client
30: \( \triangleright \) Choose a packet \( P_{\text{dup}} \in P_{\text{hist}} \cup \{\} \) s.t. \( P_{\text{dup}} \neq P \)
31: \( \triangleright \) Send irrelevant packet \( P_{\text{dup}} \) to the Wi-Fi client
32: \( \triangleright \) reset expected layers after fuzzing
33: if \( P' \neq P \) then
34: \( \text{expected}(S) \leftarrow \emptyset \)
35: end if
36: \( P_{\text{hist}} \leftarrow P_{\text{hist}} \cup \{P\} \)
37: until AP does not reach the \( \text{idle} \) state
38: \( \triangleright \) measure cost function value for \( X_i \)
39: \( CF_i \leftarrow \text{Measure_Cost_Function}(X_i) \)
40: \( \triangleright \) send cost function value to PSO
41: \( \text{Particle_Swarm_Opt}() \leftrightarrow CF_i \)
42: \( \triangleright \) wait to receive new mutation probabilities from PSO
43: \( X_{i+1} \leftarrow \text{Particle_Swarm_Opt}() \)
44: \( i \leftarrow i + 1 \)
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 (≈ 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 [I]. 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 a fully automated approach that can easily be replaced to support fuzzing other protocols. In the following, we describe the core mechanism embodied in GREYHOUND.

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.

**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 $pr^f_l$ for each layer $l \in L_P$. Any field $f \in F_l$ is mutated by GREYHOUND with the probability $pr^f_l$. As each packet is associated with a protocol state $s$ (cf. Figure 6), the number of mutation probabilities is bounded by $2 \times \sum_{s \in S} L_s$, where $S$ 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 $2 \times \sum_{s \in S} L_s$ 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 underflows 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 \( p_{\text{r}} \) and \( p_{\text{z}} \) 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 \( X_s \)
3. **Receive:** Cost function value \( CF_i \) for \( X_i \)
4. 5. \( g \) captures PSO generation number
6. \( g \leftarrow 0 \)
7. Let \([X_{i}]_{g}\) be a population of sets of mutation probabilities
8. For each \( X_i \in [X_{i}]_{g} \), assign \( X_i \) randomly
9. Initialize velocity, mutation probabilities and cost
10. For each \( X_i \in [X_{i}]_{g} \), do
11. \( \triangleright \) Send \( X_i \) to Algorithm 1 for fuzzing
12. Greyhound_Core() \( \leftarrow X_i \)
13. \( \triangleright \) Wait to receive cost from Algorithm 1
14. \( CF_i \leftarrow \) Greyhound_Core()
15. \( pb_i \leftarrow X_i; f_i \leftarrow CF_i; v_i \leftarrow 0 \)
16. End for
17. \( \triangleright \) Initialize \( gb \) with initial best mutation probabilities
18. \( gb \leftarrow X_m \) such that \( CF_m \leq CF_i \) for any \( X_i \in [X_{i}]_{g} \)
19. Repeat
20. For each \( X_i \in [X_{i}]_{g} \), do
21. \( \triangleright \) Randomize \( r_1 \) and \( r_2 \)
22. \( r_1, r_2 \leftarrow \text{rand}(0,1) \)
23. \( \triangleright \) 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. \( \triangleright \) Calculate new mutation probabilities
26. \( X_i \leftarrow X_i + v_i \)
27. \( \triangleright \) Send \( X_i \) to Algorithm 1 for fuzzing
28. Greyhound_Core() \( \leftarrow X_i \)
29. \( \triangleright \) Wait to receive cost from Algorithm 1
30. \( CF_i \leftarrow \) Greyhound_Core()
31. \( \triangleright \) 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. \( \triangleright \) 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 \( g \geq \text{max generations} \)
our chosen cost function (e.g., anomaly count). The overall 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 31 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 $p_b_i$, velocity $v_i$, and the global best mutation probabilities $g_b$ 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 $p_b_i$ and $g_b$ 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 31 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., $p_b_i$ and $g_b$, respectively, are also updated according to the cost function value measured via Greyhound_core (Lines 33-35 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 $\eta_1$, and the social component $\eta_2$. $w$ controls the speed that the particle $X_i$ is originally heading to, whereas the cognitive component $\eta_1$ acts by increasing the tendency to follow its personal best position $p_b_i$. Finally, the social component $\eta_2$ increases the tendency of $X_i$ to follow the best position of the swarm ($g_b$).

In our experiments, we set the default values of $w$ (0.729), $\eta_1$ (2.05) and $\eta_2$ (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: if $\exists l \in \text{Layers}(P_r)$ s.t. $l \in \text{expected}(S)$ then
6: $\text{validated} \leftarrow \text{true}$
7: else
8: $\text{validated} \leftarrow \text{false}$
9: end if
10: if $\text{expected}(S) = \emptyset$ then
11: restore the set of expected layers if required
12: if $\text{expected}(S) = \emptyset$ then
13: restore the original set of expected layers $\text{expected}(S)$
14: end if
15: return validated

### 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 association-Request packets \( P \) are compliant with the type of expected packets in the state of AP. This is possible due to the design of Greyhound_Core. By design, the response to \( P \) is always an invalid packet to be sent from state \( S \). Thus, the response from the client for \( P \) is not expected in state \( S \) of the AP. Due to our design, \( \text{Layers}(P_r) \cap \text{expected}(S) = \emptyset \) (the expected layers in AP states are mutually exclusive) and therefore, any anomaly will be correctly highlighted via Algorithm 3.

This case is void when \( P \) is a legitimate packet sent during state \( S \).

c) Response for sent packets during state \( S \neq S' \): This scenario occurs when packets sent from the Wi-Fi client 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 \( P_{\text{dup}}(S') \) to capture the sent packets \( P' \) and \( P_{\text{dup}} \) in state \( S' \). Given these notations, there are three possible situations as follows:

- \( P_{S'} = P'(S') \) and \( P'(S') \) was mutated: Since \( S \neq S' \), \( \text{expected}(S) \cap \text{expected}(S') = \emptyset \). This holds even if \( P' \neq P \), as \( \text{expected}(S) = \emptyset \) in such case. Therefore, Algorithm 3 will legitimately catch an anomaly for the response to \( P'(S') \) albeit being at a different state \( S \).

- \( P_{S'} = P'(S') \) and \( P'(S') \) 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 \( S'' \), it must be due to a response to some \( P_{\text{dup}}(S'') \) sent in state \( S'' \neq S' \) and \( S'' \neq S \). Since \( S \neq S' \), \( \text{expected}(S) \cap \text{expected}(S') = \emptyset \) and this also holds when \( P' = P \). 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 \( P_{\text{dup}}(S'') \) and \( P'(S') \) that trigger the same state transitions at the client.

- \( P_{S'} = P_{\text{dup}}(S') \): Let us first consider the case where \( P' = P \). Thus, the set of expected layers \( \text{expected}(S) \) is not cleared. If \( \text{Layers}(P_r) \cap \text{expected}(S) \neq \emptyset \), then \( P_r \) is validated via Algorithm 3 and GREYHOUND will proceed to the next state. It is worthwhile to mention that despite \( P_{\text{dup}}(S') \) 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 \( \text{Layers}(P_r) \cap \text{expected}(S) = \emptyset \), an anomaly will be caught via Algorithm 3. Nonetheless, such an anomaly is caught in state \( S \) instead of state \( S' \) (as \( \text{Layers}(P_r) \cap \text{expected}(S) = \emptyset \) by design).

If \( P' \neq P \), \( \text{expected}(S) = \emptyset \). If \( \text{expected}_{\text{orig}}(S) \) is the original set of expected layers at state \( S \)
and $\text{Layers}(P_r) \cap \text{expected}_\text{org}(S) = \emptyset$, Algorithm \[2\] will legitimately highlight an anomaly. However, if $\text{Layers}(P_r) \cap \text{expected}_\text{org}(S) \neq \emptyset$, Algorithm \[3\] will still catch an anomaly, as $\text{expected}(S) = \emptyset$. Nonetheless, we consider it to be an anomaly, as it cannot be inferred whether the client responds for the mutated packet $P'$ or $P_{\text{dup}}(S')$. 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 fuzzers 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 fuzzers 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 time 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 \[3\], 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
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.

<table>
<thead>
<tr>
<th>Vulnerabilities / Inconsistencies</th>
<th>Wi-Fi State</th>
<th>OnePlus 5T</th>
<th>Raspberry Pi 3</th>
<th>ESP8266</th>
<th>ESP32</th>
<th>Medical device</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2008-5290</td>
<td>connected</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Affected (91)</td>
</tr>
<tr>
<td>CVE-2017-13077 (KRACK)</td>
<td>message 3</td>
<td>Not affected</td>
<td>Affected (48)</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Not affected</td>
</tr>
<tr>
<td>CVE-2017-13078 (KRACK)</td>
<td>message 3</td>
<td>Not affected</td>
<td>Affected (48)</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Not affected</td>
</tr>
<tr>
<td>CVE-2019-9499 (Dragonslayer)</td>
<td>connected</td>
<td>Not affected</td>
<td>Affected (50)</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Not affected</td>
</tr>
<tr>
<td>New - Non compliance</td>
<td>EAP-Challenge</td>
<td>Affected (30)</td>
<td>Affected (95)</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Not affected</td>
</tr>
<tr>
<td>New - Non compliance</td>
<td>EAP-Challenge</td>
<td>Affected (3)</td>
<td>Affected (9)</td>
<td>Affected (26)</td>
<td>Affected (2)</td>
<td>Affected (2)</td>
</tr>
<tr>
<td>New - Crash</td>
<td>ALL States</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Affected (91)</td>
<td>Affected (64)</td>
<td>Not affected</td>
</tr>
<tr>
<td>New - Crash</td>
<td>EAP-Identity</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Affected (2)</td>
<td>Not affected</td>
<td>Not affected</td>
</tr>
<tr>
<td>New - Crash</td>
<td>EAP-Challenge</td>
<td>Not affected</td>
<td>Affected (421)</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Not affected</td>
</tr>
<tr>
<td>New - Crash</td>
<td>EAP-Challenge</td>
<td>Not affected</td>
<td>Affected (228)</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Not affected</td>
</tr>
<tr>
<td>New - Malformed response</td>
<td>EAP-Challenge</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Affected (261)</td>
<td>Not affected</td>
<td>Not affected</td>
</tr>
</tbody>
</table>

TABLE 2: A summary of results per cost function

<table>
<thead>
<tr>
<th>Discovery</th>
<th>Cost Function</th>
<th>Vulnerabilities / Inconsistencies / Crashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration Time</td>
<td>Anomaly Period</td>
<td>Transitions</td>
</tr>
<tr>
<td>OnePlus 5T</td>
<td>0/1/0</td>
<td>1/2/0</td>
</tr>
<tr>
<td>Raspberry Pi 3</td>
<td>1/1/0</td>
<td>3/2/0</td>
</tr>
<tr>
<td>ESP8266</td>
<td>1/2/5</td>
<td>0/2/5</td>
</tr>
<tr>
<td>ESP32</td>
<td>0/2/2</td>
<td>0/1/0</td>
</tr>
<tr>
<td>Medical Device</td>
<td>0/1/0</td>
<td>0/2/0</td>
</tr>
</tbody>
</table>

TABLE 3: A summary of evaluation time for each device

<table>
<thead>
<tr>
<th>Device</th>
<th>Max Iterations</th>
<th>Best case</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnePlus 5T</td>
<td>1000</td>
<td>5 h. 50 min</td>
<td>6 h. 40 min.</td>
</tr>
<tr>
<td>Raspberry Pi 3</td>
<td>1000</td>
<td>3 h. 32 min.</td>
<td>4 h. 32 min.</td>
</tr>
<tr>
<td>ESP8266</td>
<td>1000</td>
<td>1 h. 8 min.</td>
<td>1 h. 17 min.</td>
</tr>
<tr>
<td>ESP32</td>
<td>1000</td>
<td>1 h. 35 min.</td>
<td>1 h. 47 min.</td>
</tr>
<tr>
<td>Medical device</td>
<td>200</td>
<td>1 h. 52 min.</td>
<td>3 h. 41 min.</td>
</tr>
</tbody>
</table>

unique anomalies.

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

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 layer more frequently compared to other layers. Thirdly,
(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.

<table>
<thead>
<tr>
<th>Cost Function</th>
<th>Population Initialization</th>
<th>Max iterations until 1st anomaly is found</th>
<th>1st anomaly found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anomaly count</td>
<td>Randomly with CVE</td>
<td>40 / 45</td>
<td>0 / 25</td>
</tr>
<tr>
<td></td>
<td>Randomly</td>
<td>5 / 45</td>
<td>10 / 25</td>
</tr>
<tr>
<td>Anomaly period</td>
<td>Randomly with CVE</td>
<td>5 / 45</td>
<td>20 / 25</td>
</tr>
<tr>
<td></td>
<td>Randomly</td>
<td>5 / 45</td>
<td>30 / 25</td>
</tr>
<tr>
<td>Transitions</td>
<td>Randomly with CVE</td>
<td>5 / 45</td>
<td>40 / 25</td>
</tr>
<tr>
<td></td>
<td>Randomly</td>
<td>5 / 45</td>
<td>50 / 25</td>
</tr>
<tr>
<td>Iteration time</td>
<td>Randomly</td>
<td>5 / 45</td>
<td>60 / 25</td>
</tr>
</tbody>
</table>

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 (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 A3 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 “Evolution”, respectively. We make the following crucial
Fig. 11: The impact of different design components in GREYHOUND

TABLE 5: GREYHOUND compared to other public tools

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Vulnerabilities / Inconsistencies / Crashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools</td>
<td>One Plus 5T</td>
</tr>
<tr>
<td>wifuzzit</td>
<td>association</td>
</tr>
<tr>
<td>wifuzzit (Client)</td>
<td>idle</td>
</tr>
<tr>
<td>wifuzzit</td>
<td>association</td>
</tr>
<tr>
<td>IoTcube wifuzz</td>
<td>association</td>
</tr>
<tr>
<td>Greyhound</td>
<td>connected</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Device</th>
<th>Greyhound</th>
<th>Wifuzzit</th>
<th>Wifuzz</th>
<th>IoTcube</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Medical Device</td>
<td>55 min.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2, A3</td>
<td>Raspberry Pi 3</td>
<td>3 min.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>OnePlus 5T</td>
<td>37 min.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>Raspberry Pi 3</td>
<td>11 min.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td>ESP8266</td>
<td>22 min.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td>ESP8266</td>
<td>&lt; 1 min.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td>ESP32</td>
<td>37 min.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A10</td>
<td>ESP8266</td>
<td>29 min.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New</td>
<td>ESP8266</td>
<td>N/A</td>
<td></td>
<td></td>
<td>8 min.</td>
</tr>
<tr>
<td></td>
<td>ESP32</td>
<td>N/A</td>
<td></td>
<td></td>
<td>29 min.</td>
</tr>
</tbody>
</table>

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 wifuzz [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 to 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 wifuzz, 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.

In general, wifuzzit and wifuzz require more than one day to finish their respective fuzzing processes. Despite the fast fuzzing session conducted by IoTcube wifuzz (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 A6-A9 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 [25], 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 connection 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 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 after an EAP-Failure 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 A2: Finally, we discovered a non-compliant behaviour A2, 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 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.

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 5). 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 sveynfothb@gmail.com.

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

References


Matheus E. Garbelini graduated with a bachelor of engineering degree in Electronics from Pontificial Catholic University of Paraná in Brazil. Currently Mathes 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.