BRAKTOOTH: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing

Matheus E. Garbelini
SUTD

Vaibhav Bedi
SUTD

Sudipta Chattopadhyay
SUTD

Sumei Sun
*Institute for Infocomm Research, A*Star*

Ernest Kurniawan
*Institute for Infocomm Research, A*Star*

Abstract

In this paper we propose, design and evaluate a systematic directed fuzzing framework to automatically discover implementation bugs in arbitrary Bluetooth Classic (BT) devices. The core of our fuzzer is the first *over-the-air* approach that takes full control of the BT controller baseband from the host. This enables us to intercept and modify arbitrary packets, as well as to inject packets out-of-order in lower layers of closed-source BT stack, i.e., *Link Manager Protocol* (LMP) and *Baseband*. To systematically guide our fuzzing process, we propose an extensible and novel rule-based approach to automatically construct the protocol state machine during normal over-the-air communication. In particular, by writing a simple set of rules to identify protocol messages, we can dynamically construct an abstracted protocol state machine, fuzz packets resulting from a state and validate responses from target devices. As of today, we have fuzzed 13 BT devices from 11 vendors and we have discovered a total of 18 unknown implementation flaws, with 24 common vulnerability exposures (CVEs) assigned. Furthermore, our discoveries were awarded with six bug bounties from certain vendors. Finally, to show the broader applicability of our framework beyond BT, we have extended our approach to fuzz other wireless protocols, which additionally revealed 6 unknown bugs in certain Wi-Fi and BLE Host stacks.

1 Introduction

Bluetooth Classic (BT) is a wireless protocol that has been in use for more than twenty years. Although BT is gradually shifting to *Bluetooth Low Energy* (BLE), several IoT products, audio devices and smartphones still support BT communication. Unfortunately, recent vulnerabilities in BT design [2, 3] and implementation [41] highlight concrete threats that remain hidden from vendors in absence of rigorous testing.

In this paper, we propose a general and extensible fuzz testing framework to test arbitrary BT protocol implementations in the wild. Figure 1 illustrates the context of our

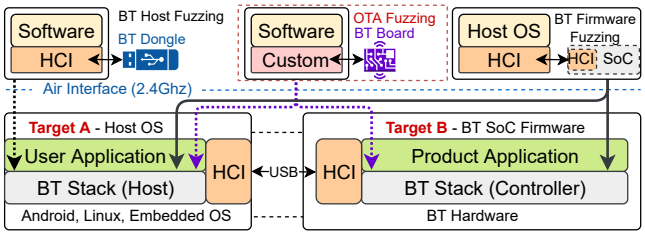

Figure 1: *Bluetooth Classic* protocol fuzzing approaches

BT fuzzer. Broadly, targets for BT stack fuzzing can be categorized into two: *(i)* host side of the BT stack that sits on the host operating system (`Target A`), and *(ii)* the BT controller stack that is implemented in the firmware (`Target B`). Due to the imposed isolation between the host and the BT controller (i.e., `Target A` and `Target B`), it is challenging to fuzz the controller BT stack directly from the host. Several fuzzing approaches [7, 20, 24], as shown by "BT Host Fuzzing" are only capable of fuzzing `Target A`. Emulation-based fuzzing [41, 58] ("BT Firmware Fuzzing" in Figure 1) is capable of fuzzing both `Target A` and `Target B`, but it requires reverse engineering the BT firmware and is not easily portable to any device. Moreover, emulation of a reversed engineered BT hardware relies on assumptions that may not correspond to exact features or timing behaviour of the real hardware [41].

Our proposed approach is shown via "OTA Fuzzing" (OTA stands for over-the-air) in Figure 1. In short, our BT fuzzer runs fuzzing campaigns directly on the host and it is capable to fuzz both `Target A` and `Target B` without modifying the firmware of the target device. Consequently, our BT fuzzer can be employed out-of-the-box to fuzz any BT controller stack. *To the best of our knowledge, our approach is the first* ***host-side*** *OTA fuzzer for BT Controller stack (i.e.* `Target B` *in Figure 1).* The fuzzing campaigns leverage the lowest data-link layer, thus bypassing existing firmware debloaters [55] that only consider the Host Controller Interface (HCI).

A key aspect of our OTA fuzzer is to keep the design general and extensible for many stateful protocols. Recent ap-

proaches on protocol fuzzing [17, 18, 38] suffer from several limitations to be generalizable. At one end, generational fuzzing approaches [17, 18] manually integrate custom state machines within the fuzzer for packet generation and mutation. At the other end, mutational fuzzers [38] leverage previously stored packet sequences as seeds for new input generation and fuzzing. Generational fuzzing approaches [17, 18] require significant work to support new protocols and is infeasible for complex protocols like BT which involve thousands of state transitions. Additionally, the state machine within such fuzzer is difficult to maintain upon new protocol features. In contrast, mutational fuzzers avoid the effort of creating the state machine, but are incapable to generate dynamic inputs that rely on data available only during live communication. For example, in BT, several packet fields in *L2CAP* contain data that are dynamically generated and exchanged. A mutational fuzzer relying on static seed inputs may often result in invalid data input, thus terminating the communication.

We employ a novel dynamic state mapping strategy to generalize OTA fuzzing. In particular, based on a few rules constructed only once per protocol, we dynamically map exchanged packets to protocol states during live communication. Then, we systematically refine the *mutation probabilities* assigned to a state for manipulating exchanged packets on-the-fly and maximize the state machine coverage to guide the fuzzing process. In such a fashion, we avoid manual hard-coding of state machines, yet direct the fuzzer to maximize coverage of protocol features captured by the state machine. Additionally, it is applicable to closed protocol stacks and is extensible to other protocols with a one-time effort of constructing the *mapping rules* and the *fuzzing interface*.

In the context of fuzzing BT stack, *we present the first technique to take full control of the communication in the BT controller (i.e.* `Target B` *in Figure 1) from the host.* While frameworks such as InternalBlue [29] allows packet injection, it is not able to control all the link manager procedures. To address this challenge, we develop a novel BT fuzzing interface by reverse engineering the BT stack of ESP32 [15]. This enables us to intercept and arbitrarily modify *any LMP or Baseband packet* exchanged with a target device. We note that the commodity ESP32 hardware costs below 20 USD, thus making our proposed fuzzer low cost and easily replicable. In summary, we make the following contributions in the paper:

1. We present the general design of our OTA fuzzer targeting stateful protocol stacks (Section 3). We also present our generic state mapping technique to dynamically create the state machine for fuzzing (Section 4).

2. We present the design of a *real-time* fuzzing interface by reverse engineering a commodity BT stack (Section 5).

3. We evaluate our fuzzer on 13 different BT devices (development boards, modules and consumer products). We discover 18 unknown implementation bugs, collectively named BRAKTOOTH (with 24 CVEs assigned) and six non compliances (Section 6). All bugs have been reported to the vendors, with several already patched. Moreover, six of the BRAKTOOTH bugs have received bug bounty from Intel, Espressif and Xiaomi. *An exploration on Bluetooth listing* [46] *reveals that* BRAKTOOTH *affects over 1400 product listings*.

4. We compare our fuzzer with four state-of-the-art BT fuzzers: BT Stack Smasher [5], Bluefuzz [7], IoTCube bfuzz [24] and Toothpicker [20]. We show that our fuzzer significantly outperforms all the competitors in terms of finding implementation flaws (Section 6).

5. We show the extensibility of our OTA fuzzer by extending it to fuzz arbitrary Wi-Fi and BLE Host protocol stacks. Our evaluation with eight Wi-Fi and BLE Host stacks reveals 6 unknown implementation flaws, none of which was discovered in comparable runs by state-of-the-art fuzzers for Wi-Fi and BLE i.e., Greyhound [17] and SweynTooth [18], respectively (Section 8).

2 Background and Motivation

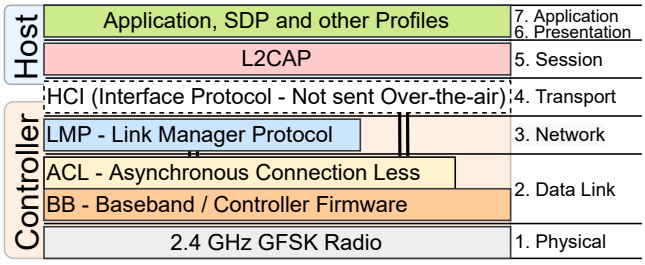

Figure 2: Common OSI Layers of Bluetooth Classic Stack

The BT stack employs several protocol interactions which are scattered across *OSI layers* as shown in Figure 2. Notably, there is an isolation between lower-layer protocols such as *Baseband*, *LMP*; and higher-layer protocols such as L2CAP and beyond [45]. While higher-layer protocols run on a host operating system (OS), lower-layer protocols run on a separate hardware named *controller*, which receives HCI packets from the OS rather than *Baseband* or *LMP* packets.

BT main procedures are shown in Figure 3(a). Each procedure contains one or more message exchanges between *master* and *slave* devices. Furthermore, relevant procedures are split into *inquiry*, *paging* and *connection*. While inquiry and paging are only related to BT discovery and connection establishment between the master and the slave, the *connection* state involves exchanging most of layer 2 and 3 messages. After the master discovers the slave address via *inquiry scan*, the master connects to the slave via *paging* procedure and establishes an Asynchronous Connection Less (ACL) *connection* as illustrated in Figure 3(a): **(I)** The *master*

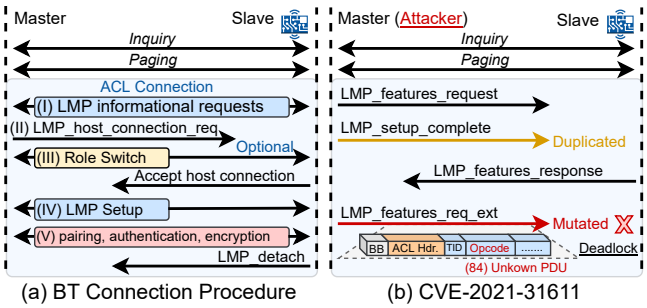

Figure 3: BT connection process and a BT stack vulnerability

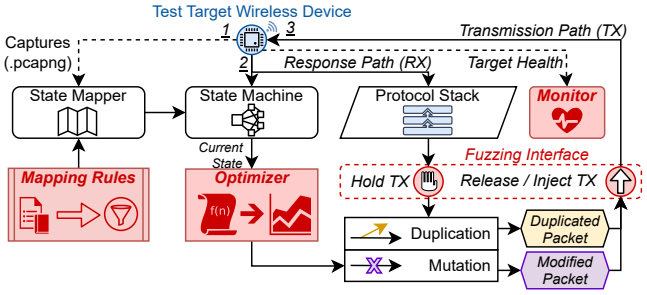

Figure 4: An Illustration of Our Fuzzing Architecture.

sends LMP *Informational Requests*, which includes mutual exchange of device features, version and name; **(II)** *master* requests *LMP_host_connection_req* to request communication with *slave* higher-layers; **(III)** *slave* optionally performs *role switch* if it needs to become the *master*. Finally, if the *slave* accepts host connection with the *master*, **(IV)** *LMP Setup* is mutually performed and **(V)** LMP procedures for *pairing*, *authentication* and *encryption* start. The *slave* or *master* can reject any request by sending *LMP_reject* and they may disconnect from an active connection by sending *LMP_detach*.

All the aforementioned procedures are *usually* expected to be completed in a certain sequence. Since it is likely that such expected sequence of procedures had already been covered by compliance testing, our fuzzer aims to generate adversely crafted packet sequence. This is challenging due to the imposed requirements on hardware. For example, a test engineer may need to freely inject packets at any time during BT procedures for generating out-of-order packet sequences. Concretely, consider the *undefined* behaviour illustrated in Figure 3(b). First, the (malicious) *master* sends an out-of-order (*duplicated*) *LMP_setup_complete* during *LMP Informational requests* procedure to the *slave*. Next, the master waits for a reply from the *slave* and sends a *malformed LMP_features_req_ext* with an invalid *Opcode* value of 84, instead of the original opcode 3. This triggers a deadlock on the *slave*, requiring the user to manually reboot the *slave*.

The example depicted in Figure 3(b) requires sending to the *slave* a packet from procedure **(IV)** into procedure **(I)**, followed by mutating a packet in procedure **(I)** (c.f., Figure 3(a)). This, in turn, surfaces three crucial challenges for designing a comprehensive and effective fuzzer: **(a)** To have full freedom in *duplicating and injecting any packet at any point during the BT connection process*, **(b)** to mutate any field of an arbitrary packet and send it to a BT target in real-time, **(c)** to make the packet manipulation targeted such that the fuzzer steers the communication towards likely vulnerable scenarios.

3 Design Overview

In this section, we first discuss the key design concepts employed in our fuzzing framework. We then provide an overview of different design elements in the framework and justify our choices.

Design Novelty: We employ two key design concepts to generalize the fuzzing of stateful communication protocols: 1) Rule-based state mapping, and 2) dynamic fuzzing. Conceptually, the rules are provided by the designer to *locate the type field within an exchanged packet.* This is fundamentally different from writing an extensive set of rules (i.e., grammars) to *generate packets* for communication. Indeed, by allowing the target devices to communicate normally, we can intercept each exchanged packet and extract the type of the packet via the provided rules. This is then used to construct the normal flow of packet exchanges in the form of a state machine where each state is uniquely mapped to a packet type. During fuzzing, any exchanged packet is mapped on-the-fly to its respective state in this state machine for computing state/transitions coverage.

Concurrently, we employ dynamic fuzzing that *eliminates the need to integrate custom packet generation and handling within the fuzzer*. In other words, we simply intercept any exchanged packet and manipulate or inject packets as guided by the fuzzing process. This allows our fuzzing to account for all contextual information available *only during communication*, thus providing flexibility in reaching the deep states in the protocol implementation for fuzzing. Additionally, it makes our design lightweight and extensible to other stateful protocols with only a one-time effort per protocol on constructing the state mapping rules.

In the following, we discuss the key elements in our design and provide rationale behind our design choices.

Rule-based State Mapping and Fuzzing: Our proposed fuzzing architecture (see Figure 4) automatically generates the state machine from a few simple rules (“Mapping Rules" in Figure 4). In particular, the rules are used to map an exchanged packet to a particular state (“State Mapper" in Figure 4). As an example, by using just eight (8) rules, we construct a BT state machine with a hundred of states and over thousand transitions. We note that it is practically infeasible to manually construct such state machines like certain existing protocol fuzzers [4, 11, 17, 36]. An alternative approach is to use active automata learning to learn the behavior of

black-box systems. However, as expected and discussed in a recent approach [37], such learning requires significant time due to many queries to target devices. For example, even to learn a BLE state machine of around ten states, it might take an hour [37]. In contrast, we develop a lightweight approach that only requires a one-time effort from the designer to understand the protocol packet structure and devise the rules. Once the rules are devised, we can automatically construct the state machine for any device implementing the protocol and by mapping the states from a normal communication. We also show that such a rule-based approach is generic by employing it to diverse protocols i.e., BT, BLE (Host) and Wi-Fi.

Decoupling Packet Generation from Fuzzing: In our fuzzing architecture, we facilitate automated fuzzing during live communication between the target device and an arbitrary third-party stack ("Protocol Stack" in Figure 4). This is fundamentally different from conventional fuzzers that need to model the entire environment for communication (e.g., packet generation and handling) [4, 17, 36]. Our design facilitates fuzzing arbitrarily complex protocol stacks without modeling complex protocol characteristics within the fuzzer. While mutation-based fuzzing approaches [38] also fuzz packet sequences without packet generation effort, such approaches are not effective for complex protocols like Bluetooth Classic. This is because BT involves dynamic protocols e.g., L2CAP where certain contextual information (e.g., channel configuration) is only available *during communication* and used within generated packets. Thus, mutating packet sequences with *static* seed inputs (i.e., packet sequences stored from previous communications) will often terminate the communication due to the lack of contextual information in the packets. This results in ineffective fuzzing. Moreover, our decoupled design allows us to easily account for protocol changes, as such changes can be introduced to the third-party stack and thus reflect in the live communication for subsequent fuzzing.

Thus, our OTA fuzzer can be employed out of the box even if amendments are added to the target protocol. In contrast, when the packet generation is handled within the fuzzer [4, 17, 36], it is highly likely to require modification that accounts for new protocol features.

Generic and Efficient Packet Decoding: Unlike several protocol fuzzers [4, 18, 38, 40], the design of our OTA fuzzer avoids manual construction of packet decoders. Instead, we leverage on the rich body of packet decoders available in the community supported Wireshark project [35], which includes both wired and wireless protocols. Moreover, as Wireshark is actively maintained, it also includes protocols that are early in adoption such as 5G-NR [52] and BLE Audio [51]. In our OTA fuzzer, we directly expose all the decoders of Wireshark supported protocols to the fuzzing interface ("Fuzzing Interface" in Figure 4), allowing highly efficient packet handling and mutations. More importantly, our fuzzer eliminates the need for constructing packet decoders for a significant number (over 1000) of protocols already supported by Wireshark [53].

Feedback and Monitoring: Our approach involves target health monitoring ("Monitor" in Figure 4), which faces unique challenges to detect crashes for certain devices e.g., BT sound devices (see Section 4). Additionally, our approach aims to maximize the transition coverage of the reference state machine constructed via state mapping ("Optimizer" in Figure 4). To this end, it uses particle swarm optimization (PSO) to refine the probabilities for mutating packet fields and leverages the number of transition in the state machine as a cost function for PSO. We employ PSO due to its applicability in stochastic optimization scenarios in which the cost function may depict some randomness given the same decision vectors. This property translates well for our use in wireless fuzzing as communication over-the-air is imperfect and adds interferences that cause certain unpredictable behavior or delays during communication with the target. Such strategy was also used successfully in previous fuzzing works [17, 28]. However, to maneuver the cost function value of PSO within our framework involves additional challenges. This is because the packet generation is not controlled by our fuzzer. In particular, to compute the value of the cost function within PSO, we leverage our state mapper (see Figure 4) to return the state label of an exchanged packet on-the-fly. This is then used to update the state transition coverage.

In contrast, when the packet generation is completely controlled within the fuzzer [17], the computation of coverage is trivially attributed to the control flow hard-coded in the fuzzer state machine. Nonetheless, such coverage computation heavily depends on the nature of the state machine integrated within the fuzzer. This, in turn, makes the approach potentially challenging to extend and generalize for OTA fuzzing.

4 Methodology

The fuzzing process: Figure 5 captures some scenarios encountered during the fuzzing process. To start, we generate a reference state machine model M_{ref} (Figure 5(b)) from a few simple rules (see Section 4.1 for details). The OTA fuzzer intercepts all transmitted packets from the protocol stack (see Figure 4). Subsequently, these packets might be mutated (Figure 5(c)) or duplicated (Figure 5(d)) while sending them to the target device. For all the exchanged packets, either from target device or protocol stack, the state mapper is invoked to map a packet to a particular state in M_{ref} (Figures 5(b)-(d)). This, in turn, is used to monitor the current state of the protocol and to compute the transitions covered via the fuzzing process. Additionally, the state mapping is crucial to validate target response. For example, if the current state is S and the target response is mapped to state S', then we validate this response if and only if S' is the destination of any outgoing transitions from S. In the next section, we discuss our state mapping process in detail.

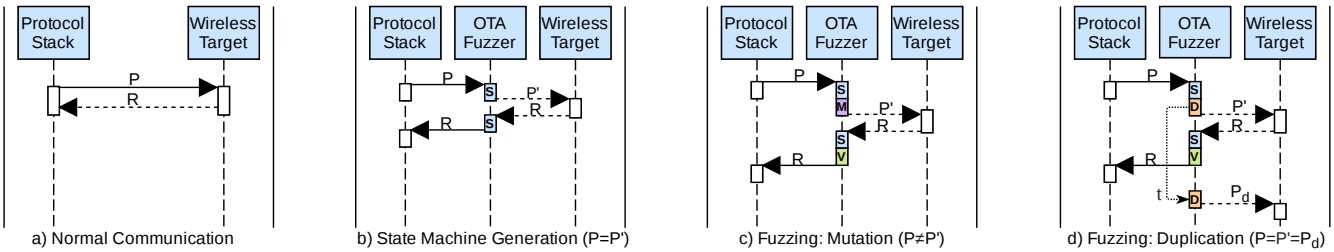

Figure 5: Different scenarios encountered during our fuzzing. **S:** State Mapping, **M:** Mutation, **D:** Duplication, **V:** Validation. P is the packet sent from the protocol stack whereas R is the response received from the target wireless device.

Algorithm 1 `get_state_label` Procedure

```
1:  Input: Packet P, Rules M_u : {⟨Φ^i, F^i⟩ | i ∈ [1,N]}
2:  Output: State label generated for packet P
3:  Parse packet P to get layers(P)
4:  Let F[L] be the set of packet fields for L ∈ layers(P)
5:  for each L ∈ layers(P) do
6:      ▷ initialise state layer and type
7:      state.layer := state.type := empty;  i := 1
8:      ▷ continue the mapping for P until a state label is found
9:      repeat
10:         ▷ check whether the layer matches with the next rule
11:         if L satisfies Φ^i then
12:             ▷ record matching layer and field names
13:             state.layer := L.name;  F_map := F[L] ∩ F^i
14:             for each f ∈ F_map do
15:                 v := value of field f in packet P
16:                 state.type := concat(state.type, lookup[v])
17:             end for
18:         end if
19:         Θ := (state.layer = empty ∨ state.type = empty)
20:         i := i + 1
21:     until ((Θ = false) ∨ (i > N))
22:     ▷ create the state label if a match was found
23:     if (state.layer ≠ empty ∧ state.type ≠ empty) then
24:         state.name := state.dir ⊕ state.layer ⊕ state.type
25:         return state.name
26:     end if
27: end for
```

4.1 Protocol State Mapping

The *state mapper* shown (see Figure 4) dynamically generates the state machine M_{ref} to capture the protocol between the devices. This can be performed either in real-time during the communication (c.f., Figure 5 (b)) or via previously stored capture files. We note that our state mapper constructs M_{ref} by inspecting exchanged packets and mapping each packet type to a unique state in M_{ref}. The exchanged packets depend only on the protocol and not on any device specific states. Thus, as long as the target device implements the considered protocol, we can always map an arbitrary packet P, exchanged from any target device, to a state in M_{ref}.

Mapping Rules: The core of state mapping is to create a state label for any packet exchanged during communication. This is accomplished via a set of rules M_u: a list of N pairs $\langle \Phi^i, F^i \rangle$ for $i \in [1,N]$. Φ^i denotes a condition (e.g., name of a layer) to identify the protocol layer (say L), whereas F^i captures the names of fields that identify the packet type within L. Figure 6 shows two such rules in M_u: the first rule captures the protocol layer name (i.e., *layer.name*) as `L2CAP` and the packet type in `L2CAP` is identified in field $F^1 \equiv$ `l2cap.code`. Similarly, the second rule captures the protocol layer `LMP` and target fields {`lmp.eop`, `lmp.opcode`}. We argue that such a set of rules is easy to construct for well-defined protocols (we needed only eight rules for BT) and that our mapping approach avoids manual construction of protocol state machines for fuzzing. Additionally, the mapping rules are significantly more lightweight as compared to typical grammar rules used for generating packets in model-based fuzzers [14]. In particular, the state machine generated by leveraging the mapping rules facilitate state monitoring and coverage computation to guide the fuzzing process. Such a state machine is not used for generating packets within our fuzzer. For reference, the mapping rules for BT, Wi-Fi and BLE Host Protocols are included in the Appendix.

State Mapping Process: The state mapper (Algorithm 1) creates a *state label* for any exchanged packet P. Then, a transition between states s and s' is created in M_{ref} when s and s' correspond to consecutive packets. The *state label* contains the direction of the packet (i.e., TX for transmitted and RX for received), the name of the layer (e.g., LMP) and the type of the packet (e.g., `Features Req.`). To create the *state label*, we first dissect the packet P in real-time and obtain its inherent layers and fields (see Lines 3-4 in Algorithm 1). Subsequently, we navigate through all layers of packet P (i.e., $layers(P)$) to identify the layer name in the set of rules M_u (see Lines 5-13 in Algorithm 1). Once the layer is identified, we extract the values from the relevant fields in the layer to obtain the type of packet P (see Lines 13-15 Algorithm 1). Finally, the packet type is also used to generate the state label (Line 16 in Algorithm 1). We note that `lookup` dictionaries map the field value to a name e.g., the value `0x04` for the field `lmp.opcode` is mapped to type "`Features Res.`" (see Figure 6).

Although M_{ref} might be incomplete in nature, we argue that it is sufficient for effective fuzzing. Moreover, our state mapping architecture allows us to augment or modify the state mapping rules on-the-fly. This allows identification of new packet types as the protocol evolves, thus providing flexibility to the designer for selectively fuzzing protocol layers.

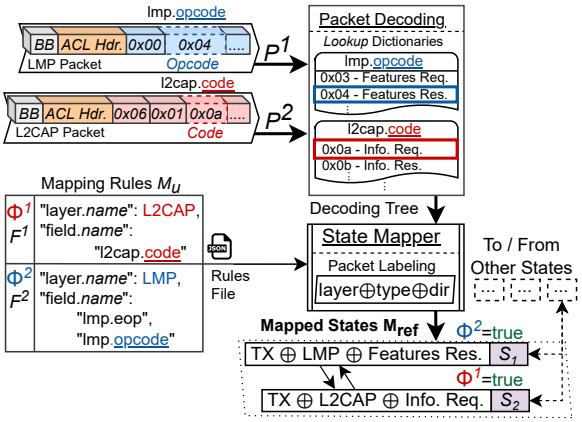

Figure 6: An example of the Protocol State Mapper.

4.2 Fuzzing and Optimization

As discussed in Section 3, we expose Wireshark protocol decoders to the fuzzer. In the following, we discuss how such is leveraged for packet mutation and duplication (Figure 5).

Mutation: Since a packet structure contains many fields, choosing *what* to mutate at *which* state is critical to diversify the fuzzing process. Consider an intercepted packet P and the mapped protocol state for packet P is S. Packet P can be mutated only if $S \in M_{ref}$ (protocol state machine). We capture the mutation probability at state S via $Y_i(S)$ in the i-th fuzzing iteration. $Y_i(S)$ contains a layer mutation probability for each layer in P and a field mutation probability for all the fields in P. Therefore, $|Y_i(S)| = N_L(P) + 1$ where $N_L(P)$ is the number of layers in packet P. The initial mutation probabilities at state S i.e., $Y_1(S)$ are *randomly* assigned. Figure 7 illustrates how the mutation is applied for two different packets P^1 and P^2, which have a number of layers. Firstly, a *Decoding Tree* is generated for each packet and fed to the *State Mapper*, which outputs states S for P^1 and S' for P^2. Then, such states are separately associated to mutation probabilities $Y_1(S)$ and $Y_1(S')$ respectively. We observe that pr_l^1 probability is assigned for mutating the *Baseband* (*BB*) layer, whereas the *ACL Header* is mutated with probability pr_l^2. Then, for each packet type, all the fields e.g., `LT_ADDRESS`, `Type`, receive the same probability pr_f for mutation. Finally, the fuzzer iterates over all layers and if a layer L_i hits its chance pr_l^i, each field of layer L_i is mutated with a probability pr_f.

In subsequent fuzzing iterations, we systematically refine the mutation probabilities to guide the fuzzing process via M_{ref}. To this end, we use an objective function to quantify the effectiveness of $Y_i = \bigcup_{S \in M_{ref}} Y_i(S)$ in terms of the coverage of M_{ref}. Therefore, by maximally covering the number of transitions in M_{ref}, we aim to maximize the number of anomalies found. Given this intuition, we use the number of newly covered transitions in M_{ref} as the objective function. We note that computing this coverage information requires repeated invocation of our state mapping algorithm (Algorithm 1).

We apply particle swarm optimization (PSO) to refine Y_i after each fuzzing iteration. Our choice of PSO is motivated by its effectiveness in a wireless environment that involves non-linear and stochastic behaviour [17, 39]. Intuitively, PSO optimizes the value of our objective function via iteratively modifying the positions of particles in the swarm. In our fuzzer, the position of a particle is the chosen probabilities Y_i in a given fuzzing iteration. Furthermore, each particle in a swarm is associated with a different set of mutation probabilities to apply PSO in our context.

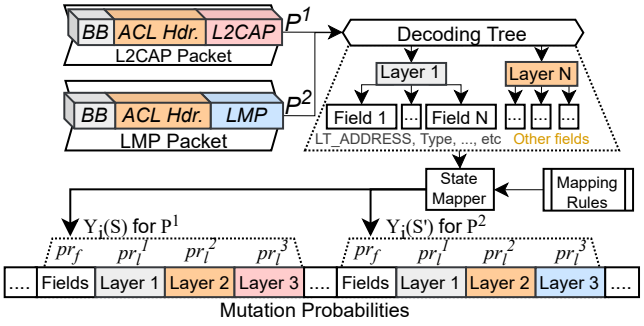

Figure 7: An illustration of mutation probabilities distributed across layers and fields of the dissected packet decoding tree.

Mutation Operators: The mutation operator is applied according to the type of the chosen field and it avoids mutating in valid ranges (normally certification covers it), resulting in packets ideally being rejected or timed out.

Integer fields are mutated with a random value, but since a field may not be byte-aligned, its value is overwritten according to a bit-mask and its bit-length; Bitfield types are toggled using XOR operation and byte arrays (i.e., strings) receive a random byte value at a random position in the array.

Duplication: As shown in Figure 5(d), our fuzzer duplicates packets to send them at inappropriate times. The duplication revolves around two parameters: The chance r_{sel} for a packet P to be *duplicated* ($r_{sel} \in [0,1]$) and the time t (in milliseconds) for which the packet is kept in a queue Q_s before transmitting P out-of-order. The time t is chosen randomly between $[1, D_T]$. Adjusting r_{sel} and D_T influences *when* the fuzzer sends out-of-order packets to the target. A high value for D_T allows the fuzzer to send out-of-order packets more scattered across deeper states. In contrast, a higher r_{sel} may create flooding situations, which might prevent the fuzzer to reach deeper states. The implication of choosing these parameters is discussed in our empirical evaluation (Section 6).

Target Monitoring: Monitoring a target device over-the-air is not as direct as compared to monitoring classic program crashes. Whenever a BT system-on-chip (SoC) crashes, its internal watchdog timer detects unresponsiveness and hard resets the SoC. We employ several schemes to listen for messages indicative of crash & restart, across different types of BT devices (see "*Monitor*" in Figure 4).

For BT development boards or modules, the *Target Monitor* collects *startup messages* through their exposed *Serial Port* via USB cable. Similarly, Android smartphones require *ADB* connection to collect crash messages from system logs via USB. Furthermore, *SSH* allows collecting error messages from the kernel ring buffer via Ethernet or Loopback. Finally, the *Microphone* can be used to identify whether a BT sound device plays a *startup sound* above a user-defined volume threshold. Considering that such sound is usually played only when the device is manually powered on, the event of repeated startup sounds indicates a crash. Figure 8 outlines the different external connections and techniques to monitor each target type. We note that in cases when the watchdog timer fails to detect SoC hangs, we employ a global timer to detect reconnection timeout (and hence a possible deadlock).

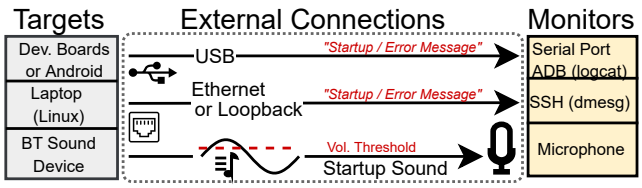

Figure 8: An illustration of different BT *Target Monitors*.

4.3 Exploitation via Dissection Hooks

Once the fuzzer gets a vulnerability (crash or deadlock) or non-compliance, the user can replicate the scenario via our exploitation framework. This is done using TX/RX *dissection hooks* to inject or mutate packets during communication.

For example, test cases for CVE-2021-31611 (see Figure 3(b)), which would otherwise require reverse engineering to create, are now created by simply manipulating packets via our exploitation framework (see Figure 9).

Specifically, Figure 9 highlights intercepting packets and checking whether they match `LMP_features_req` or `LMP_features_req_ext` by using the Wireshark filters. Then the packet `LMP_setup_complete` is injected after `LMP_features_req` and `LMP_features_req_ext` is mutated by modifying the raw packet contents (i.e., `pkt_buf`).

5 Achieving Real-Time Fuzzing Interface

Due to the lack of open-source *Bluetooth Classic* controller implementations, it is challenging to circumvent the HCI barrier such that the *layer 2* (see Figure 2) is fully controlled by

```
int tx_post_dissection(pkt_buf, pkt_length) {
  //detect LMP_features_req
  if (packet_has_condition("btbrlmp.op == 39")) {
    //Send LMP_setup_complete after
    send_packet(LMP_setup_complete); return 0;
  }
  //detect LMP_features_req_ext
  if (packet_has_condition("btbrlmp.eop == 5")) {
    //Mutate LMP_features_req_ext to unkown opcode
    pkt_buf[4]=0xa9; return 1;
  }
}
```

Figure 9: CVE-2021-31611 exploit via TX *dissection hook*.

the host. Moreover, since the Baseband packets are sent in a time window multiple of 625*us* [45], handling Baseband in the host is not straightforward as the OS scheduler cannot normally operate in such a strict timing constraint.

We have the following options to address this challenge: (1) Implementing a BT controller from scratch; (2) using existing frameworks for BT firmware patching [29], or (3) designing a patched firmware to enable Baseband manipulation from the host using popular and low-cost IoT SoC. Option (1) is extremely labour intensive to implement due to the complexity of Baseband and LMP handling [45, p. 150]. Furthermore, there is no wireless SoC that openly exposes documentation of its *Bluetooth Classic* radio registers. Concurrently, option (2) does not offer much flexibility in host-side fuzzing as discussed next (Section 5.1). In contrast, we choose option (3), as it allows us to create a patched firmware *only once* and (re)use this out-of-the-box for fuzzing arbitrary Bluetooth classic devices.

Next, we highlight limitations of option (2) i.e., the state-of-art BT sniffing and injection (InternalBlue [29]) and discuss the conceptual differences that enable our fuzzing interface to overcome such shortcomings, while still keeping our approach generically applicable to fuzz other wireless protocols.

5.1 Comparison with InternalBlue

Figure 10 showcases how each approach manipulates or injects BT packets based on *where* the *packet hook* is located. Our BT fuzzing interface (Figure 10 (a)) manipulates packets in the host *TX Hook* via *Hold / Release* operations. In contrast, *Internalblue* (Figure 10 (b)) uploads and runs specific assembly patches (*ASM Hook*) in the controller firmware before any packet injection or manipulation can be done. The latter approach, while flexible for BT patching experimentation, introduces the following shortcomings for OTA fuzzing:

(I) *Hook Location*: InternalBlue *packet hook* runs in the controller firmware and requires a new patch to be applied *before* a BT exchange happens. For an ongoing BT exchange, such an approach only allows *fixed* packet manipulation, whereas our OTA fuzzer demands *arbitrary* packet manipulations to diversify the fuzzing process. Additionally, integration of fuzzing components and protocol decoding inside the firmware only

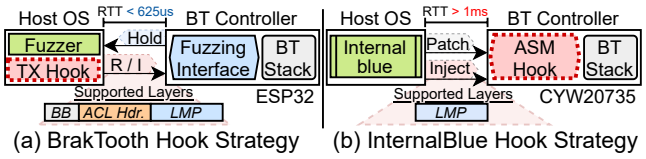

Figure 10: *Packet Hook* strategy employed by our BT Fuzzing Interface and InternalBlue framework. **R/I:** Release/Inject.

results in a loss of generality of our OTA fuzzer design. This is due to the requirement of custom patches for the InternalBlue target firmware. Instead, we leverage *Hold/Release* operations for fuzzing, which can be employed beyond BT.

(II) *Packet Sniffing/Injection*: InternalBlue only exposes LMP layer sniffing or injection to the user due to its reliance on Broadcom Diagnostics protocol. Moreover, such protocol only injects packets *after stable BT connection*, which undermines fuzzing attempts to inject packets *during early BT procedures* (see Figure 3(a)). This limitation is also discussed earlier [33]. To overcome this, we design a custom, yet simple protocol that encapsulates all layers, starting from *Baseband*.

(III) *Hardware Limitations*: InternalBlue target platforms do not support high-speed USB, which is critical for a low round trip time (RTT in Figure 10) between host and BT controller. Therefore, we shift to a platform such as ESP32-WROVER that allows manipulation of packets with $RTT < 625us$.

We note that authors of InternalBlue recently had replicated a specific BRAKTOOTH flaw (i.e., **V14** in Table 2). While doing so, it was mentioned that InternalBlue was not stable to reproduce exploits that target the Baseband layer, as the framework cannot easily modify Baseband headers [42].

We now describe how we design a patched firmware using *ESP32* IoT SoC to manipulate or inject Baseband packets.

5.2 Espressif Bluetooth Internals

We reverse engineer relevant parts of Espressif's ESP32 proprietary BT library *libbtdm_app.a* to investigate how BT packets are transmitted (TX) and received (RX). This is with the objective to modify TX/RX data/control flow to facilitate fuzzing from the host. ESP32 BT library is closed source and it is available in *Espressif IoT Development Framework* (ESP-IDF) [15]. Nonetheless, ESP-IDF exposes partial BT symbols in the compiled BT sample code image (i.e., the user code) and also provides a partial ELF image of ESP32 ROM for debugging purposes. Thus, importing both images into *Ghidra 9.1.2* [1] with a third-party Tensilica Xtensa CPU plug-in [13], allows us to investigate Baseband packet handling.

Espressif Patches: ESP32 ROM needed to be patched to solve issues with Wi-Fi coexistence, new BT features and vulnerabilities. Therefore, Espressif BT controller is split between ROM and the static library *libbtdm_app.a*. Such library is linked to the user application, which runs in flash memory.

As illustrated in Figure 11, the ESP-IDF applies a number of runtime patches during BT initialization by redirecting some ROM functions to *libbtdm_app.a*. This is silently accomplished by *config_funcs_reset*, which overwrites global pointers that are originally mapped to ROM during boot. However, certain fixes, which need to be introduced inside ROM functions, do not have a corresponding pointer. Addresses of such functions are intercepted at runtime in the BT frame interrupt function *r_ld_fm_frame_isr*. This is then replaced with function wrappers containing tiny fixes (i.e., *ld_inq_frm_cbk_wrapper*). When a simple fix is not possible, certain functions are completely reimplemented in *libbtdm_app.a*, such as *ld_acl_frm_isr*. Next, we describe our modifications to the ESP-IDF BT library that enable fuzzing from the host.

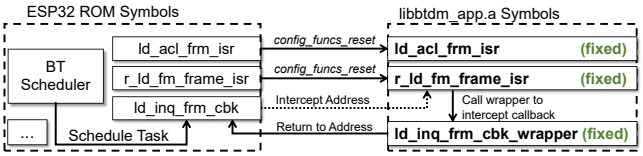

Figure 11: ESP-IDF runtime patching to fix ROM functions.

5.3 Patching from Inside and Outside

We design a mixed patching framework (Figure 12) for ESP32. It introduces custom function hooks in a user code that intercept or inject ACL/LMP packets on either ROM or any ESP32 proprietary library at runtime.

Since the static library *libbtdm.a* is independent from the user code (i.e. `firmware.c` in Figure 12) before the linking process, it is not reasonable to patch the library from ***outside*** before linking. This is because the call instruction `call8`, as used by the BT library, is relative to the program counter, but the address of our desired hook function in the user code is not known until linking. Specifically, we introduce our *BTPatcher* to ESP-IDF build pipeline. We patch the generated firmware ELF by overwriting specific user-defined symbol addresses with a call to custom hook in the user code (i.e. `firmware.c` in Figure 12). For instance, the interception hook `r_ld_acl_data_tx+609` (see Figure 12) is placed at offset 609 relative to the function `r_ld_acl_data_tx` and the hook is used to redirect the control flow to a custom function in `firmware.c`. Such a custom function, in turn, is used to intercept L2CAP packets, including its *Baseband* headers.

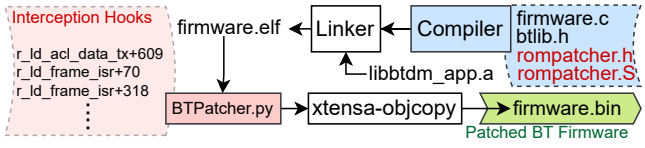

Figure 12: Workflow of BRAKTOOTH patching framework.

Our approach in Figure 12 significantly reduces time to test hooks, as opposed to manually applying patches via Ghidra.

Using *BTPatcher.py*, we introduce hooks to relevant *libbtdm.a* functions such as `r_ld_frame_isr` and `r_ld_acl_data_tx` to capture the reception of LMP/L2CAP packets and intercept the transmission of L2CAP packets, respectively. However, to intercept most LMP packets, it is necessary to patch `r_ld_acl_lmp_tx` located in ROM. We patch this function at runtime (i.e., from ***inside***) via the pointer redirection strategy discussed in Section 5.2. The redirected functions then implement Baseband interception for host-side mutation.

The remaining Baseband packets are handled entirely in ROM functions that do not have a global pointer, such as paging and enquiry related functions. For these functions, we create a *rompatcher* which installs ROM hooks via Xtensa breakpoint registers *IBREAK0-1*. This, in turn, enables our hooks to be called upon any ROM address, and hence enables interception of packets that are only generated in ROM.

5.4 Fuzzing Interface Firmware Design

Our BT fuzzing interface provides the following set of features (exposed via USB) to the host-side fuzzer:

TX Hold/Release: Allows the host to mutate Baseband TX packets in real-time before over-the-air transmission. According to *Core Specifications* [45], the time gap between each packet exchange is always multiple of 625*us*. Therefore, in the worst-case, the interception hook needs to forward a BT packet to the host and receive it back before a channel hop in 625*us*. Such low-latency is achieved by enabling the *"0ms"* latency mode of *FT2232H USB Driver* [16], which pools messages from the host USB port every 125*us*. This is the equivalent to forcing the host to busy-wait on the USB port and achieves an average RTT of 281*us* (see Figure 13).

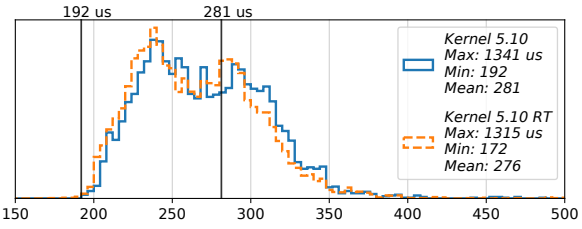

Figure 13: Histogram of Interception Round Trip Time (*us*) for Linux Kernel variants on AMD 3900X 2.6Ghz.

TX Injector: This allows the host to inject Baseband packets after the BT paging procedure (See Figure 3(a)). This is essential for the *duplication* component to work.

RX/TX Bypass: Effectively "blinds" ESP32 BT stack from receiving or transmitting LMP packets after the *paging* procedure. Combined with *TX Injector*, the host can inject packets on every BT transmission slot (i.e. every 1.25*ms*), facilitating flooding attacks such as *AU Rand Flooding* (see Section 6).

ROM Patcher: Installs ROM hooks at run-time from the firmware (i.e., user code) as discussed in Section 5.3.

6 Evaluation

Implementation: The fuzzer is split in two codebases: (I) The fuzzer software running on the host is mostly written in C++ with few lines of Python (16431 lines of code (LOC)) and (II) the BT fuzzer interface, discussed in Section 5.4, is written in C for ESP32 (3094 LOC). Moreover, the *BTPatcher* discussed in Section 5.3 has 371 LOC. Our *state mapper* and *fuzzing* was built around *Wireshark 3.3.0* Epan library and *Graphviz++* [27]. The Wireshark plug-in that enables decoding of *Baseband* packets is based on the InternalBlue Broadcom decoder [10]. Specifically, we extended such decoder to support custom ESP32 BT metadata and additional packet headers that were missing from the original project. The optimizer uses the generational PSO implementation [12] from PyGMO library with the default *pygmo.pso_gen* optimization parameters. Finally, we slightly modify Bluekitchen sample program *sdp_rfcomm_query* to start layer four (4) communication whenever the fuzzer starts a new iteration.

Evaluation Setup: Table 1 outlines all devices that we have tested. Each device can either be a BT development kit or an end product such as a laptop, smartphone, wireless speaker, etc. While products have a BT SoC pre-programmed for the product's application, development boards require an initial firmware to start. The latter is accomplished by programming a *sample code* provided by the SoC Software Development Kit (SDK). Since our fuzzer targets the BT stack, a vulnerability in such indicates that other devices relying on the same BT stack are also potentially vulnerable due to the shared code base of the SDK.

Since we need to have a reference model for the fuzzing, we let the *state mapper* (Section 4.1) learn the reference model by running Bluekitchen BT stack sample program *sdp_rfcomm_query* with each target device for about 1*h*. The final protocol model M_{ref} is the combined reference model of each individual device. This ensures that we can use a single model for different devices during the fuzzing campaign.

We now answer the following research questions (RQs):

RQ1: How effective is our fuzzer in terms of generating error-prone inputs? A summary of our evaluation appears in Table 2. In each row, we use the prefix **V** to identify a vulnerability and **A** to indicate a non-compliance (i.e., faulty target responses) that deviates from the *Core Specifications* [45]. Moreover, Table 2 outlines the respective CVEs, affected devices, protocol layers and the violated compliance. *Overall, we discovered 18 unknown implementation bugs and coordinated a 90-day disclosure period with all vendors.*

BRAKTOOTH flaws trigger (I) *crashes* and (II) *deadlocks*. *Crashes* generally trigger fatal assertion, segmentation faults due to buffer or heap overflow within the SoC firmware. *Deadlocks*, in contrast, lead the target to a condition in which no further BT communication is possible. This may happen due to the *paging scan* being forcibly disabled (**V17**), state machine corruption on **V7** or entirely disabling BT functionality

Table 1: Devices used for evaluation. The sample code is provided by vendor to test the development board. This is not applicable (N.A) on products running a fixed application.

BT SoC Vendor	BT SoC	Dev. Kit / Product	Sample Code	Monitor
Bluetooth 5.2				
Intel	AX200	Laptop Forge15-R	N.A	SSH
Qualcomm	WCN399X	Xioami Pocophone F1	N.A	ADB
Bluetooth 5.1				
Texas Instruments	CC2564C	CC256XCQFN-EM	SPPDMMultiDemo	Serial
Bluetooth 5.0				
Cypress	CYW20735B1	CYW920735Q60EVB-01	rfcomm_serial_port	Serial
Bluetrum Technology	AB5301A	AB32VG1	Default	Serial
Zhuhai Jieli Technology	AC6925C	XY-WRBT Module	N.A	Mic.
Actions Technology	ATS281X	Xiaomi MDZ-36-DB	N.A	Mic.
Bluetooth 4.2				
Zhuhai Jieli Technology	AC6905X	BT Audio Receiver	N.A	Mic.
Espressif Systems	ESP32	ESP-WROVER-KIT	bt_spp_acceptor	Serial
Bluetooth 4.1				
Harman International	JX25X	JBL TUNE500BT	N.A	Mic.
Bluetooth 4.0				
Qualcomm	CSR 8811	Laird DVK-BT900-SA	vspspp.server.at	Serial
Beken	BK3260N	HC-05	V1.0.2	Serial
Bluetooth 3.0 + HS				
Silabs	WT32i	DKWT32I-A	ai-6.3.0-1149	Serial

via remote code execution (RCE) on **V1**. Our results affect popular BT vendors (i.e., Intel, Qualcomm, Cypress, Texas Instruments) and relatively less known (i.e., Bluetrum, Jieli, Harman), which are still employed in many consumers products such as BT speakers, keyboards, toys, etc.

V1 affects ESP32, which is used in many products ranging from consumer electronics to industrial equipments such as programmable logic controllers (PLCs). Hence, the impact is significant, as the attacker only requires knowledge of the target *BDAddress* to launch the attack. Indeed, all the vulnerabilities **V1-V18** can be triggered without any previous pairing or authentication. Moreover, the impact of **V1-V18** reaches beyond the devices listed in Table 2, since any other BT product employing an affected SoC is also vulnerable.

Multiple LMP flooding attacks (e.g., **V5**, **V13**) and **V16** were found in SoCs from different vendors including majors e.g., Intel and Qualcomm. This indicates the lack of flexible tools for over-the-air testing even in 2022. Besides, the *Core Specifications* only allows a limited "LMP test mode" [45] that prevents the SoC from operating in many LMP procedures.

RQ2: How efficient is our fuzzer? Each fuzzing iteration results in a reconnection between the host and target, thus efficiency depends on how often the *target* starts the *paging* procedure (c.f., Figure 3) with the *master*. The time to reconnect can be decreased in Linux and in some development boards, but BT products do not offer such options. Table 4 showcases the total time to evaluate 1000 iterations for each target. In general, all development boards except for *DVK-BT900-SA* and *DKWT32I-A* depicted lower time to complete 1000 iterations than BT Products or Modules such as *JBLTUNE500BT* and *XY-WRBT*, respectively. Nevertheless, the time to find the first vulnerability (i.e., crash or deadlock) was generally within 10-30 minutes with few exceptions such as *Pocophone F1* and *JBLTUNE500BT*, which did not encounter crash or deadlock within 1000 iterations. Despite this, most devices had **A2** (c.f., Table 2) triggered within few minutes.

The column "Model Coverage" in Table 4 captures the number of unique valid transitions in M_{ref} (i.e., the protocol state machine) that are traversed by the respective target device during the fuzzing process. We note that each BT vendor implements BT devices differently and the vendor may choose scenarios that do not trigger all 1106 possible transitions in M_{ref}. In particular, target devices that perform *role switch* with the *master* yield more transitions in M_{ref}. For example, *XY-WRBT* and *BT Audio Receiver* explored fewer transitions due to the lack of *role switch* and overall LMP exchanges as compared to the other devices e.g., *Pocophone F1*.

RQ3: How do the different design choices contribute to the effectiveness of our fuzzer? To evaluate this, we first create three variants of the fuzzer that disables part of its components: (I) Only *Duplication* is enabled (c.f., Figure 4). This means that only out-of-order packets are sent to the target. (II) Packet *Mutation* without optimization or duplication. Thus, packets do not reach the target out-of-order and mutation probabilities are *not* refined. (III) We enable both *Mutation* and *Optimization*, but *Duplication* is disabled.

Figure 14 depicts the total number of unique crashes or deadlocks obtained during 1000 fuzzing iterations with *ESP32* as the target. We choose *ESP32* for this experiment as it yields higher number of crashes and deadlocks. The labels *"Duplication"*, *"Mutation"* and *"Evolution"* refer to variants (I), (II) and (III) respectively. The last label *"All"* is the fuzzer with all components enabled. The results yield the following observations: The *Duplication* returned crashes due to **V5**, **V4** being triggered after *paging* procedure (c.f., Figure 3). However, such a variant could not find **V1, V2** which require packet mutation. Likewise, *Mutation* and *Evolution* could only find **V1, V2** due to **V5** and **V4** requiring packet duplication to be triggered. Moreover, it was notable that *Evolution* was able to trigger **V1, V2** before variant *Mutation* due to its better exploration of the target. Nevertheless, in 1000 iterations, only the variant *All* triggered each of ESP32 vulnerabilities **V1-V5**. However, the variant *All* usually takes more time to trigger issues in *deeper* states (i.e., *bounding*) since *duplication* and *mutation* are competing during the fuzzing session.

Table 5 evaluates the *Duplication* variant of the fuzzer with respect to the tunable parameters r_{sel} (selection probability) and D_T (time to schedule). We note that for a higher probability $r_{sel} = 0.3$ and low $D_T = 100ms$, the highest number of crashes were detected due to LMP flooding (**V5**). However, when decreasing r_{sel} to 0.1 and increasing D_T to $6000ms$, crashes are triggered less frequently, as out-of-order packets are delivered into *deeper* states. This contributes to the highest number of average transitions and reduces flooding behaviour. We chose $r_{sel} = 0.1, D_T = 6000ms$ to evaluate other devices, as such configuration is less aggressive in terms of flooding and hence it does not overshadow the *mutation* component.

RQ4: How effective is our fuzzer compared to existing BT fuzzers? We compare our fuzzer against state-of-the-art tools that most closely match the objective of our fuzzer i.e., *bfuzz* [24], *BT Stack Smasher* [5], *Bluefuzz* [7], and *Tooth-*

Table 2: Summary of unknown implementation bugs and other anomalies found (**Vx:** Vulnerability, **Ax:** Non-compliance).

Anomalies	CVE ID(s)	Device(s)	State(s)	Target Layer(s)	Impact Type	Compliance Violated
V1 Feature Pages Execution	CVE-2021-28139	ESP-WROVER-KIT	*Feature Exchange*	LMP	RCE	[V.1] Part E, Sec. 2.7
V2 Invalid Public Key	CVE-2021-28138	ESP-WROVER-KIT	*Bounding*	LMP	Crash	[V.2] Part C, Sec. 5.1
V3 Feature Req. Ping-Pong	CVE-2021-28137	ESP-WROVER-KIT	*Feature Exchange*	LMP	Crash	[V.1] Part E, Sec. 2.7
V4 Duplicated IOCAP	CVE-2021-28136	ESP-WROVER-KIT	*Bounding*	LMP	Crash	[V.2] Part C, Sec. 4.2.7.1
V5 Feature Resp. Flooding	CVE-2021-28135 CVE-2021-28155 CVE-2021-31717	ESP-WROVER-KIT JBL TUNE500BT Xiaomi MDZ-36-DB	***After Paging***	LMP	Crash	[V.1] Part E, Sec. 2.7
V6 LMP Auto Rate Overflow	CVE-2021-31609 CVE-2021-31612	DKWT32I-A BT Audio Receiver	Data Rate Change	Baseband	Crash	[V.2] Part B, Sec. 6.6.2
V7 LMP 2-DH1 Overflow	CVE-2021-35093	DVK-BT900-SA	After EDR Change	Baseband	Deadlock	[V.2] Part C, Sec. 2.3
V8 LMP DM1 Overflow	CVE-2021-34150	AB32VG1	***Many***	Baseband	Deadlock	[V.2] Part B, Sec. 6.5.4.1
V9 Truncated LMP Accepted	CVE-2021-31613	BT Audio Receiver XY-WRBT Module	***Many***	LMP	Crash	[V.2] Part C, Sec. 5.1
V10 Invalid Setup Complete	CVE-2021-31611	BT Audio Receiver XY-WRBT Module	*Feature Exchange*	LMP	Deadlock	[V.1] Part E, Sec. 2.7
V11 Host Conn. Flooding	CVE-2021-31785	Xiaomi MDZ-36-DB	*Host Connection*	LMP	Deadlock	[V.1] Part E, Sec. 2.7
V12 Same Host Connection	CVE-2021-31786	Xiaomi MDZ-36-DB	*Host Connection*	LMP	Deadlock	[V.1] Part E, Sec. 2.7
V13 AU Rand Flooding	CVE-2021-31610 CVE-2021-34149 CVE-2021-34146	AB32VG1 CC256XCQFN-EM CYW920735Q60EVB	***After Paging***	LMP	Crash Deadlock	[V.1] Part E, Sec. 2.7
V14 Invalid Max Slot Type	CVE-2021-34145	CYW920735Q60EVB	*After Setup Complete*	Baseband	Crash	[V.1] Part E, Sec. 2.7
V15 Max Slot Length Overflow	CVE-2021-34148	CYW920735Q60EVB	*After Setup Complete*	Baseband	Crash	[V.1] Part E, Sec. 2.7
V16 Invalid Timing Accuracy	CVE-2021-34147 CVE-2021-30348 CVE-2021-33139	CYW920735Q60EVB Pocophone F1 Intel AX200	*Timing Accuracy*	LMP, Baseband	Crash	[V.1] Part E, Sec. 2.7
V17 Paging Scan Deadlock	CVE-2021-33155	Intel AX200	*After Host Connection*	LMP, Baseband	Deadlock	[V.1] Part E, Sec. 2.7
V18 SDP Element Size Overflow	Pending	Beken BK3260N	*SDP Exchanges*	SDP	Deadlock	[V.2] Part C, Sec. 4.2.5.2
A1 Accepts Lower LMP Length	N.A	**All, except ESP32**	***Many***	Baseband	Non-Compliance	[V.2] Part C, Sec. 5.1
A2 Accepts Higher LMP Length	N.A	**All tested devices**	***Many***	Baseband	Non-Compliance	[V.2] Part C, Sec. 5.1
A3 Multiple Encryption Start	N.A	Xiaomi MDZ-36-DB	*After Encryption Start*	LMP	Non-Compliance	[V.2] Part C, Sec. 4.2.5.3
A4 Ignore Role Switch Reject	N.A	Pocophone F1	*Role Switch*	LMP	Non-Compliance	[V.2] Part C, Sec. 4.4.2
A5 Invalid Response	N.A	Intel AX200 DVK-BT900-SA	*Feature Exchange*	LMP	Non-Compliance	[V.2] Part C, Sec. 4.3.4
A6 Ignore Encryption Stop	N.A	CYW920735Q60EVB	*After Encryption Start*	LMP	Non-Compliance	[V.2] Part C, Sec. 4.2.5.4

Table 3: Vulnerabilities: SDK/Firmware versions of vendors

SoC or Module Vendor	BT SoC	Firmware or SDK Ver.	Vuln. / Non-compl.
Intel	AX200	Linux - iwlwifi-cc-46.3 Windows - 22.40.0	V16-17 / A1-2
Texas Instruments	CC2541C	cc256xc_bt_sp_v1.4	V13 / A1-2
Cypress	CYW20735B1	WICED SDK 2.9.0	V13-16 / A2,A6
Bluetrum Technology	AB32VG1	1.0.5	V8,V13 / A1-2
Zhuhai Jieli Technology	AC6925C	N.A	V9-10 / A1-2
Zhuhai Jieli Technology	AC6905X	N.A	V6,V9-10 / A1-2
Actions Technology	ATS281X	N.A	V5,V11-12 / A1-2
Qualcomm	WCN399X	crbtfw21.tlv, patch 0x0002	V16 / A1-2,A4
Espressif Systems	ESP32	esp-idf-4.4	V1-V5 / A1
Harman International	JX25X	N.A	V5 / A1-2
Laird Connectivity	BT900 (**CSR8811**)	v9.1.12.14	V7 / A1-2
Silabs	WT32i	iWRAP 6.3.0 build 1149	V6 / A1-2
Beken	HC-05 (BK3260N)	V1.0.2	V18 / A1-3

Table 4: Timing of 1000 fuzzing iterations for each device.

Dev. Kit / Product	Total Time	1st Vulnerability	1st Non-compl.	Model Coverage
Laptop Forge15-R	~3 h.	27 min.	5 min.	243 (21.9%)
Pocophone F1	2 h. 48 min.	>~2 h. 48 min.	6 min.	258 (23.3%)
CC256XCQFN-EM	3 h. 46 min.	2h. 34 min.	9 min.	105 (9.5%)
CYW920735Q60	3 h. 19 min.	12 min.	20 min.	197 (17.8%)
AB32VG1	3 h. 08 min.	11 min.	7 min.	140 (12.7%)
XY-WRBT Module	4 h. 12 min.	35 min.	29 min.	94 (8.5%)
BT Audio Receiver	3 h. 48 min.	26 min.	11 min.	99 (8.9%)
MDZ-36-DB	4 h. 27 min.	6 min.	1 min.	150 (13.5%)
ESP-WROVER-KIT	~3 h.	10 min.	42 min.	244 (22.1%)
JBL TUNE500BT	~5 h.	>~5h.	26 min.	153 (13.8%)
DVK-BT900-SA	3 h. 50 min.	~1 h.	26 min.	119 (10.7%)
DKWT32I-A	~4 h.	13 min.	8 min.	143 (12.9%)
HC-05 (BK3260N)	~5 h.	1 h. 12 min.	1 min.	95 (8.6%)

picker [20]. These tools target BT higher layers e.g., *L2CAP*, *SDP*, while only bfuzz supports multiple BT protocols.

For a fair comparison, we run all fuzzing tools for *three hours* (unless the fuzzer terminates before). Table 6 shows the number of non-compliances (**Ax**) and crashes (**Vx**) triggered by each fuzzing tool. It also lists the products or SoCs that exhibit at least one crash on the rightmost columns. The competitor tools discovered new crashes on a few devices such as ESP32, MDZ-36-DB, JBL TUNE500BT and AB32VG1. These crashes were found either in *L2CAP* or *RFCOMM* layer. In contrast to these tools, our fuzzer enables full control to fuzz the *LMP* and *Baseband* layers, and hence we discovered critical vulnerabilities in such layers (see Table 2). Although *bfuzz* performed the fastest *L2CAP* fuzzing (130 packets/second), it must keep a database with up to date test cases and does not reach *LMP* or *Baseband* layers. In contrast, our fuzz tests are generated automatically via the approach illustrated in Section 4 and we do not need to maintain any database.

Overall the competitor tools can only detect a few crashes as compared to our fuzzer and they are incapable of finding layer two and layer three vulnerabilities such as **V1**-**V17** or non-compliances **A1**-**A6** (cf. Table 2). Our fuzzer has superior performance for two reasons: first, our fuzzer intercepts ESP32 BT stack for fuzzing during early LMP procedures. Secondly, the competitor tools neither send duplicated packets nor do they optimize the mutation probabilities.

Finally, we compare how the time of training the reference

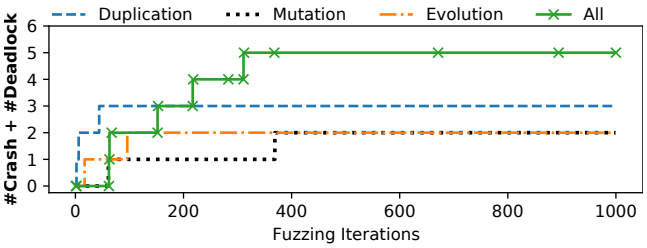

Figure 14: Unique crashes/deadlocks w.r.t ESP32 fuzzing iterations. Parameters: $r_{sel} = 0.1$ $D_T = 6000$

Table 5: Evaluation summary w.r.t. different r_{sel} and D_T.

# Crash (# Average Transitions) during 200 Iterations			
r_{sel}	$D_T = 100ms$	$D_T = 1000ms$	$D_T = 6000ms$
0.1	12 (116 ± 74)	6 (113 ± 82)	7(107 ± 81)
0.2	14 (118 ± 57)	10 (108 ± 75)	7(110 ± 84)
0.3	15(107 ± 57)	18(115 ± 64)	13(145 ± 74)

model M_{ref} affects the fuzzing process. We generated five reference models $\{M^i_{ref} \mid i \in \{1, 15, 30, 45, 60\}\}$ by running the state mapper against ESP32 for one, 15, 30, 45 and 60 minutes, respectively. We then employ M^i_{ref} in a fuzzing session (i.e., 1000 iterations). Figure 15(a) showcases the number of ESP32 vulnerabilities per model over a fuzzing and Figure 15(b) depicts the number of states, anomalies (**A**) and coverage (**C**:%) of each model during their fuzzing session.

In general, the more the time a model is trained, the more the states and transitions it includes, which contributes to finding vulnerabilities in less frequent states. However, Figure 15(a) highlights that even a model trained for 1 *min* discovers higher vulnerabilities (*#Crash + #Deadlock*) in a fuzzing session than a more complete model (e.g., M^{60}_{ref}). This means that a model employing more states than necessary to find certain vulnerable states, can lead the fuzzer to take more time to reach a higher number of vulnerabilities (e.g., M^{45}_{ref}). Nonetheless, such a model can explore more states for fuzzing.

The number of vulnerable states (*"Crash" States*) are shown in Figure 15(b). Notably, most of the mapped states are assigned to LMP, which grows from 62 in M^1_{ref} to 87 in M^{60}_{ref}. While M^1_{ref} finds ten *"Crash" States*, it yields higher coverage (C : 86%) and number of anomalies (A : 62) at the end of the fuzzing session due to its smaller number of states. On the contrary, M^{60}_{ref} results in less coverage (C : 64.6%), but it can guide the fuzzer to explore more states and transitions.

For reference, our full BT model resulted in a coverage of 22.1% for ESP32 and it has the following number of states: *LMP (125), L2CAP (16), SDP (5), RFCOMM (9).* For simplicity, such reference model is illustrated in Figure 16 with a few transitions between certain LMP and L2CAP states.

7 Attacks Exploiting BRAKTOOTH

BRAKTOOTH presents a set of implementation bugs previously unknown to BT vendors. While BRAKTOOTH does not

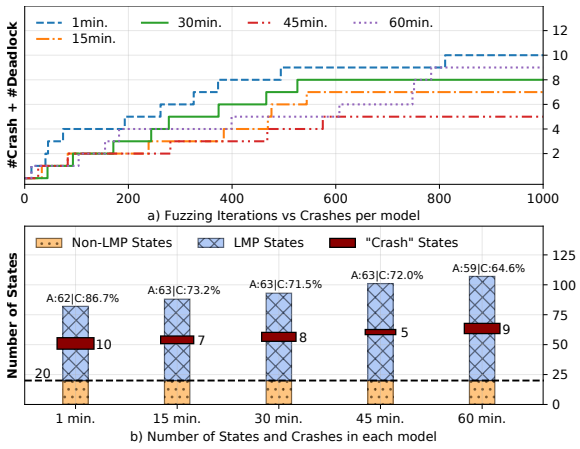

Figure 15: Evaluation of different state machine models.

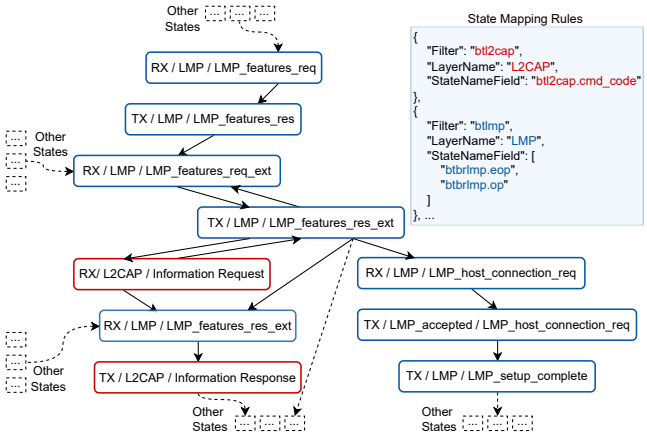

Figure 16: An illustration of a simplified BT state machine and corresponding state mapping rules for LMP and L2CAP.

present a novel class of security vulnerabilities, patching the affected devices is crucial to avoid BT exploitation in the wild. Broadly, BRAKTOOTH captures two classes of vulnerabilities: 1) crashes, and 2) deadlocks. While crashes may trigger fatal assertion, heap overflows or segmentation faults, deadlocks typically involve a power cycle to continue normal BT operation. In the following, we discuss a representative set of vulnerabilities captured by BRAKTOOTH and outline how such vulnerabilities were exploited to launch remote code execution or denial-of-service (DoS) attacks.

(1) Remote Code Execution in IoTs: The most critical vulnerability (**V1** *Feature Pages Execution*) found by our fuzzer affects *ESP32* SoC, which is used in many Wi-Fi and Bluetooth IoT appliances such as Industry Automation, Smart Home and Fitness. The attack is illustrated in Figure 17. A lack of out-of-bounds check in ESP32 *BT Library* allows the reception of a *mutated LMP_feature_response_ext*. This results in injection of eight bytes of arbitrary data outside the bounds of *Extended Feature Page Table* ("E. Features Table" in Figure 17). An attacker, which knows the firmware layout

Table 6: A Comparison among different fuzzing tools.

Comparison		Vulnerabilities / Non-compliances				
# Tools (# Fuzzing Strategy)	Fuzzable Layer(s)	ESP32	MDZ.	JBL.	AB32VG1	Others
bfuzz (IotCube) **(Random + Test Database)**	L2CAP/SDP/RFCOMM	3/0	1/0	1/0	1/0	0/0
Stack Smasher **(Random)**	L2CAP	0/0	0/0	0/0	0/0	0/0
Bluefuzz **(Random)**	RFCOMM	0/0	0/0	0/0	0/0	1/0
ToothPicker **(Random)**	L2CAP	0/0	0/0	1/0	0/0	0/0
Our Fuzzer **(Evolutionary)**	**BB/LMP**/L2CAP/SDP/ RFCOMM	4/2	3/3	1/2	2/2	8/4

of target device, can write a known function address (*JMP Addr.*) to the offset pointed by *Features Page* ("Feat. Page" in the *LMP_feature_response_ext* packet) field. It turns out that the *BT Library* stores some callback pointers within the out-of-bounds *Features Page* offset and such a callback is eventually invoked during the BT connection. While exploiting this vulnerability, we forced *ESP32* into erasing its NVRAM data (normally written during product manufacturing) by setting *JMP Addr.* to the address of *nvs_flash_erase*, which is always included in *ESP32* firmware. Similarly, disabling BT or BLE can be done via *esp_bt_controller_disable* and Wi-Fi via *disable_wifi_agc*. Additionally, general-purpose input/output (GPIO) can be controlled if the attacker knows addresses to functions controlling actuators attached to *ESP32*.

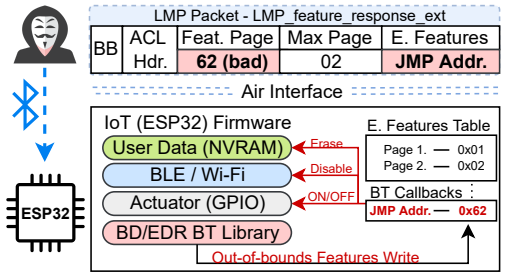

Figure 17: An Illustration of CVE-2021-28138.

(2) DoS in Laptops & Smartphones:

We discuss two sample DoS attacks discovered by our fuzzer on laptops and smartphones. These attacks were launched on laptops employing *Intel AX200* SoCs and smartphones employing *Qualcomm WCN399X* SoC family, among many others [34]. Due to the amount of smartphones and laptops vulnerable to such attacks, and the common use of BT connectivity during video conference calls and music streaming, updating the affected devices is essential.

The first DoS (**V16** *Invalid Timing Accuracy*) is due to a failure in the SoC to validate timer resources upon receiving an invalid *LMP_timing_accuracy_response* from a BT slave. As shown in Figure 18 (a), the attacker performs a loop of connection and injection of the malformed *LMP_timing_acc_response* (i.e., wrong *type* and *opcode* fields) until the target SoC gets unstable. These steps are repeated with a different BT address (i.e., BDAddress) until the SoC is exhausted from accepting new connections. This triggers a firmware crash in Qualcomm WCN3990 and Infi-

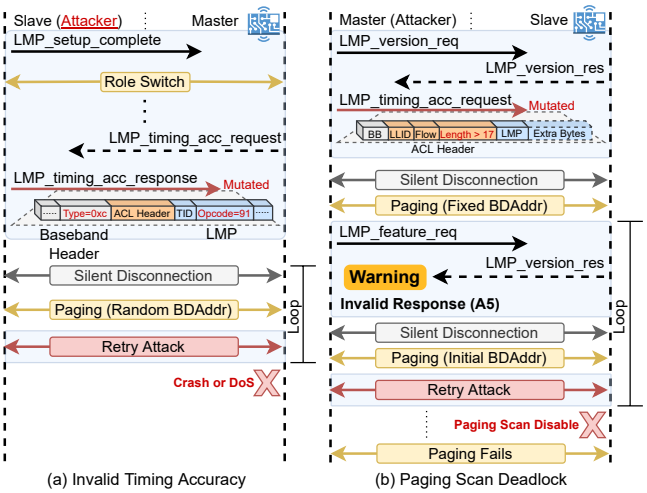

Figure 18: An Illustration of *Invalid Timing Accuracy* and *Paging Scan Dealock* attacks.

neon CYW20735B1 SoCs, or it leads the Intel AX200 SoC to disconnect other BT devices currently connected to *AX200*. In reality, users with the affected Qualcomm Android phones or Intel Laptops connected to BT Headsets experience audio to be continuously "cut" during the attack. This also results in firmware crashes and restart of the BT Service in WCN3990 and CYW20735B1.

The second DoS (**V17** *Paging Scan Deadlock*) affects only devices using Intel *AX200* and it is triggered when an oversized *LMP_timing_accuracy_request* (>17 bytes) is sent to *AX200 slave* over multiple reconnections, as illustrated in Figure 18 (b). Interestingly, during a reconnection, AX200 sends invalid response (*LMP_version_res*) upon receiving a feature request from the attacker. This depicts anomaly **A5** as listed in Table 2. Eventually, after a number of attacks, *AX200* disables its *paging* procedure (cf., Figure 3) and the target is unable to initiate multiple BT connections. Thus, scanning from *AX200* works, but other devices cannot initiate connection to *AX200*. This behavior can be used to trick a user to connect to the attacker's BT hardware with a spoofed BDAddress instead of the legitimate device since paging scan is disabled. During the attacks, firmware crashes may be sporadically triggered on AX200, but no specific scenario was found to reliably trigger such crashes all the time. Ultimately, the user needs to manually re-enable BT to restore functionality.

(3) Freezing Audio Products: Many vulnerabilities were discovered while testing our fuzzer with a BT Speaker (*Xiaomi MDZ-36-DB*), BT Headphone (*JBL TUNE 500BT*) and BT Audio Modules (XY-WRBT and BT Audio Receiver). The discovered vulnerabilities arise from target's firmware failure when the attacker *(i)* sends certain truncated packets with LMP *length field* set to one (**V9**: *Truncated LMP Accepted*) as shown in Figure 19(a) or oversized packets with LMP *length field* greater than 17 (**V6**: *LMP Auto Rate Overflow*), *(ii)* start-

ing procedures out-of-order (**V10**: *Invalid Setup Complete*) as highlighted in Figure 3(b) and finally by *(iii)* flooding LMP packets (**V5**: *Feature Response Flooding*) at every 1.25*ms* BT *transmission slot* as illustrated in Figure 19(b).

The vulnerabilities can *"freeze" Xiaomi MDZ-36-DB* and completely shutdown *JBL TUNE 500BT*. This requires the user to *manually* turn on the unresponsive devices. Since both devices accept multiple BT connections, an attack can be triggered while the user is playing some music. As an exception, *XY-WRBT* and *BT Audio Receiver* accept only one connection, which avoids an attack to be launched during an active BT connection with the user. Nevertheless, different products employing the same SoC may enable multiple BT connections depending on the product requirements.

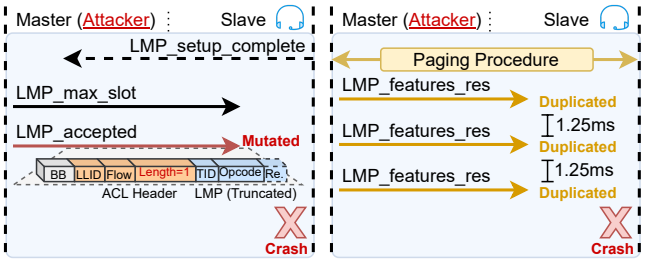

Figure 19: An Illustration of *Truncated LMP Accepted* and *Feature Repsonse Flooding* attacks

8 Fuzzing Other Wireless Protocols

The fuzzing architecture (Figure 4) can be applied to other wireless protocols beyond BT by altering some of its components. To exemplify such extensibility, we modify the components `Protocol Stack`, `Mapping Rules` and `Fuzzing Interface` to support a subset of *(i)* **Wi-Fi Access Point (AP)**, and *(ii)* **Bluetooth Low Energy (BLE) Host** protocols.

To support the extensions mentioned in the preceding paragraph, we made the following changes: firstly, the `Protocol Stack` component was entirely switched to *hostapd* to support *(i)*. In contrast, the `Protocol Stack` for *(ii)* reused the stack employed in our main BT Classic work (i.e., Bluekitchen) since it supports BLE host protocols. Secondly, the `Mapping Rules` (see Section 4.1) were formulated according to the protocols layer of *(i)* and *(ii)*, resulting in four and five rules assigned to extensions *(i)* and *(ii)*, respectively. Lastly, the `Fuzzing Interface` for extension *(i)* was enabled by implementing *Hold/Inject TX* block in the `Realtek RT8XXAU` Wi-Fi USB driver. Concurrently, adding *Hold/Inject TX* to extension *(ii)* simply required creating an HCI pseudoterminal between the Bluekitchen stack and the rest of the framework.

The summary of extensions, vulnerabilities, model states/-transitions and coverage (for 1000 iterations) are depicted in Table 7. Overall, since the reference model for each extension

Table 7: Summary of unknown flaws found by extension. *#S/#T* captures *#States/#Transitions* in reference models.

Extension	Stack	#S/#T	Target	Coverage	Vulnerability	CVE (New)
BLE Host	Bluekitchen	62/139	ESP32	75.4%	Null Dereference	CVE-2022-26604
			Telink TLSR8258	71.1%	Re-Advertisement DoS	CVE-2022-26602
			NXP KW41Z	81.0%	–	–
			TI CC2540	67.5%	–	–
Wi-Fi AP	Hostapd	33/100	ESP32	63.5%	EAP Heap Overflow	CVE-2022-26603
					Association Deadlock	CVE-2022-26600
			ESP8266	70.8%	Association Crash	CVE-2022-26601
			Rasp. Pi 3 B	57.8%	Probe Resp. Deadlock	CVE-2022-26599
			One Plus 5T	84.1%	–	–

is simpler (has lower number of states and transitions) than the BT model, the coverage obtained for each tested target was relatively higher. Notably, our extension allowed us to find six unknown implementation bugs in popular IoT devices such as ESP32/8266 and Raspberry Pi 3. This highlights the feasibility to extend our framework to other wireless protocols. Furthermore, we compared our fuzzing extensions with model-based fuzzers for BLE [18] and Wi-Fi AP [17]. To this end, we run our fuzzer and all competitors for the same amount of time (max. six hours). The competitors did not discover the implementation bugs reported in Table 7 for the same targets. This is because the packet generation models of these competitors have not evolved as compared to a full stack, therefore missing certain fuzzing scenarios. In contrast, the state machine in our framework, although incomplete, is constructed via mapping packets from normal communication. Thus, potentially such a mapping strategy evolves the state machine closer to a full stack as compared to a hard-coded state machine within a generational fuzzer [17, 18].

9 Related Work

Bluetooth vulnerabilities such as Blueborne [43], KNOB [3], BIAS [2] and BleedingBit [44] may allow unauthorized remote access or launch remote code execution. These works require extensive manual effort (e.g., reverse engineering and code inspection). A recent work BLESA [54] discovers a vulnerability in BLE *specification* via formal analysis. In contrast to these works, our fuzzer finds security issues directly in the *implementation* by learning the protocol states automatically.

Classic greybox fuzzing [6, 23, 25, 26, 28, 49] faces significant challenges in testing wireless targets: firstly, most greybox fuzzers instrument code to optimize code coverage. Such is not possible for commercial and closed wireless stacks. Secondly, classic greybox fuzzers aim to generate a single input leading to crashes. For wireless protocols, often a sequence of packets with strict timing constraints triggers crashes. Thirdly, it is beyond the capability of conventional fuzzers to break the isolation between the host and BT controller (see Section 5) for effective fuzzing. In summary, extending a classic greybox fuzzer e.g. AFL [56] for OTA fuzzing is non-trivial and even a lose adoption will be insufficient to detect non-compliances (see Table 2). Moreover, due to code instrumentation, this extension is inappropriate for fuzzing closed protocol stacks.

Prior works on host-side BT fuzzing [5, 7, 20, 24] are unable to fully break the isolation between the host and the BT controller. We show many link manager vulnerabilities that these works fail to discover. InternalBlue [29] allows LMP injection, but such only works *after the connection is set up* [33] and accessibility to Baseband fields is limited for fuzzing. In contrast, our fuzzer works *during* the connection process and it allows arbitrary packet duplication and manipulation.

Emulation based fuzzing [41, 57, 58] involves extensive reverse engineering of the firmware (if available at the first place) to obtain coverage information. For example, Frankenstein [41] works with specific Cypress/Broadcom hardware and it requires additional engineering effort to adopt for other devices. Moreover, Frankenstein did not find vulnerabilities **V3-V6** on CYW20735B1 SoC. Finally, emulation may lead to inaccurate fuzzing results, as the fuzzing is not run *in situ*.

A recent work on BLE firmware static analysis [50] does not focus on packet handling vulnerabilities. OTA fuzzing for BLE and Wi-Fi [17, 18] involve manual construction of the respective protocol state machines for packet generation and handling. Thus, these works are not extensible to complex protocols e.g., BT. In contrast, apart from enabling a novel BT fuzzing interface, our approach is generic and extensible for fuzzing other wireless protocols, as showcased in Section 8.

Works on static analysis and verification [21, 31, 32, 48] do not generate packets to trigger vulnerabilities in real devices. Additionally, none of these works targets BT. Works on testing text structured protocols [4, 19, 36] e.g. `ftp`, `http`, are not directly applicable for wireless fuzzing. Other works targeting network protocols [8, 22, 47] either require access to the source code [47] or lack test generation and fuzzing [8, 22]. Another approach discovers memory corruptions on IoT devices [9] via mobile apps. Our fuzzer does not rely on mobile apps as it targets the data link layer instead of the application layer.

In summary, we develop a generic wireless fuzzing approach and instantiate the fuzzer for three different protocols. Additionally, we present the first comprehensive approach to *fully control the BT link manager from the host.*

10 Discussion and Conclusion

Limitations: Our OTA fuzzer does not map device specific states and may also miss fuzzing certain states if such states are not mapped during reference model generation. Additionally, even though our BT fuzzing approach is generalizable to fuzz all BT layers, we prioritize malformed packet generations in LMP that contains many states. As a result, our fuzzer may repeatedly reconnect with the target when the target becomes unresponsive, resulting in slow progress reaching higher BT layers and discover bugs in such layers. In terms of manual effort, our fuzzing architecture involves the construction of the mapping rules and the fuzzing interface once per target protocol. Furthermore, even though our health monitoring process allows us to validate responses from a comprehensive set of wireless devices, there might be additional monitoring methods needed for other devices. Finally, our fuzzer does not automatically discover the precise attack vector of a vulnerability upon triggering a firmware crash or deadlock in the target. This limitation is widely common to any OTA fuzzer since it does not have access to the target's memory content during the fuzzing session.

Impact: In this paper, we have proposed a general wireless protocol fuzzing architecture and the instantiation of this architecture allows us to fuzz arbitrary devices implementing any of the three wireless protocols: Bluetooth Classic (BT), Wi-Fi and BLE Host. This opens up significant opportunities and flexibilities to test wireless stacks at low cost. Additionally, as evident from the effectiveness of our OTA fuzzer extension to Wi-Fi and BLE Host, our fuzzing architecture can be used by the community as a platform to contribute and extend for other protocols not considered in this paper. Apart from uncovering 24 unknown bugs that affect several thousands of wireless devices, our tool has been used by the community. Notably, Samsung and Mediatek have independently used our exploits to discover multiple vulnerabilities in their SoCs leading to additional CVE assignments [30].

Availability: Our fuzzer and exploits are publicly available: https://github.com/Matheus-Garbelini/braktooth_esp32_bluetooth_classic_attacks. Source code is freely available for academic research upon request to braktooth@gmail.com.

Acknowledgement: We thank the anonymous reviewers and our shepherd Daniele Antonioli for their insightful comments. This work is partially supported by the Singapore International Graduate Award (SINGA) and NRF National Satellite of Excellence in Trustworthy Software Systems (Project no. RGNSOE2001 and RGNSOE2101).

References

[1] National Security Agency. Ghidra Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra, 2020.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. BIAS: bluetooth impersonation attacks. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 549–562. IEEE, 2020.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. The KNOB is broken: Exploiting low entropy in the encryption key negotiation of Bluetooth BR/EDR. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1047–1061, Santa Clara, CA, August 2019. USENIX Association.

[4] 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.

[5] Pierre Betouin. Bluetooth stack smasher version 0.6. http://www.secuobs.com/news/05022006-bluetooth10.shtml, May 2006.

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

[7] Luca Boni. BlueFuzz. https://github.com/lucaboni92/BlueFuzz, 2017.

[8] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 213–218, San Jose, CA, 2013. USENIX.

[9] 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 *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA*, February 2018.

[10] Jiska Classen. Bluetooth Wireshark plugin from the InternalBlue project. https://github.com/seemoo-lab/h4bcm_wireshark_dissector, September 2019.

[11] Baojiang Cui, Shurui Liang, Shilei Chen, Bing Zhao, and Xiaobing Liang. A Novel Fuzzing Method for Zigbee Based on Finite State Machine. *International Journal of Distributed Sensor Networks*, 10(1):762891, 2014.

[12] Pagmo development team. Pagmo & Pygmo. https://esa.github.io/pagmo2/, 2019.

[13] Ebiroll. Fork of Tensilica Xtensa module for Ghidra. https://github.com/Ebiroll/ghidra-xtensa, 2021.

[14] Michael Eddington. Peach fuzzer. http://peachfuzzer.com/.

[15] Espressif. Espressif IoT Development Framework. https://github.com/espressif/esp-idf, 2020.

[16] FTDI. FT2232H Product Overview Webpage. https://ftdichip.com/products/ft2232hq/, 2018.

[17] Matheus E Garbelini, Chundong Wang, and Sudipta Chattopadhyay. Greyhound: Directed greybox wi-fi fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[18] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. Sweyntooth: Unleashing mayhem over bluetooth low energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 911–925. USENIX Association, July 2020.

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

[20] Dennis Heinze, Jiska Classen, and Matthias Hollick. Toothpicker: Apple picking in the ios bluetooth stack. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[21] Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 627–638, June 2017.

[22] Samuel Jero, Hyojeong Lee, and Cristina Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *DSN*, pages 1–12, June 2015.

[23] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening pandora's box through atfuzzer: dynamic analysis of at interface for android smartphones. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 529–543, 2019.

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

[25] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. CAB-Fuzz: Practical concolic testing techniques for COTS operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 689–701, Santa Clara, CA, July 2017. USENIX Association.

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

[27] Lin. Graphviz++. https://github.com/compilin/gvpp, July 2020.

[28] 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 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[29] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - Bluetooth binary patching and experimentation framework. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 79–90, New York, NY, USA, 2019. ACM.

[30] Mediatek. January 2022 Product Security Bulletin. https://corp.mediatek.com/product-security-bulletin/January-2022, 2022.

[31] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[32] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, page 12, USA, 2004. USENIX Association.

[33] Tom Nijholt, Erik Poll, and Frits Vaandrager. BlueSpec: Development of an LMP state machine and a stateful black-box BR/EDR LMP fuzzer. *Radboud Univeristy*, 2020. https://www.ru.nl/publish/pages/769526/tom_nijholt.pdf.

[34] National Institute of Standards and Technology (NIST). Database Entry for CVE-2021-30348. https://nvd.nist.gov/vuln/detail/CVE-2021-30348, 2021.

[35] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

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

[37] Andrea Pferscher and Bernhard K. Aichernig. Fingerprinting bluetooth low energy devices via active automata learning. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 524–542, 2021.

[38] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[39] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, Jun 2007.

[40] Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. Z-fuzzer: Device-agnostic fuzzing of zigbee protocol implementation. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '21, page 347–358, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36. USENIX Association, August 2020.

[42] The Secure Mobile Networking Lab (SEEMOO). Hunting ghosts in bluetooth firmware: Braktooth meets frankenstein. https://naehrdine.blogspot.com/2021/09/hunting-ghosts-in-bluetooth-firmware.html, 2021.

[43] Ben Seri and Gregory Vishnepolsky. Blueborne: Unveiling zero day vulnerabilities and security flaws in modern bluetooth stacks. https://armis.com/blueborne/, 2017.

[44] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. BleedingBit: The hidden attack surface within BLE chips. https://armis.com/bleedingbit/, 2018.

[45] Bluetooth SIG. Bluetooth Core Specification v5.2, December 2019. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[46] Bluetooth SIG. View previously qualified designs and declared products, January 2020. https://launchstudio.bluetooth.com/Listings/Search.

[47] JaeSeung Song, Cristian Cadar, and Peter Pietzuch. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, July 2014.

[48] Octavian Udrea, Cristian Lumezanu, and Jeffrey S Foster. Rule-based static analysis of network protocol implementations. *Information and Computation*, 206(2-4):130–157, 2008.

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

of the 41st International Conference on Software Engineering, ICSE '19, page 724–735. IEEE Press, 2019.

[50] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 167–180, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Wireshark. Bluetooth HCI ISO Packet Filter Reference. https://www.wireshark.org/docs/dfref/b/bthci_iso.html, 2021.

[52] Wireshark. MAC-NR Packet Filter Reference. https://www.wireshark.org/docs/dfref/m/mac-nr.html, 2021.

[53] Wireshark. Wireshark Filters Reference. https://www.wireshark.org/docs/dfref/, 2021.

[54] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave Jing Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. *BLESA*: Spoofing Attacks against Reconnections in Bluetooth Low Energy. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[55] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. Lightblue: Automatic profile-aware debloating of bluetooth stacks. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[56] Michal Zalewski. American fuzzy lop. https://github.com/google/AFL, April 2017.

[57] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[58] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

Appendix

State Mapping Rules

In this section, we provide the mapping rules used within our fuzzer. Specifically, Figure 20, Figure 21 and Figure 22 capture the state mapping rules for BT, BLE (Host) and Wi-Fi protocols, respectively.


```
"Mapping": [
   {
        "LayerName": "SDP",
        "StateNameField": "btsdp.pdu",
   },
   {
        "LayerName": "A2DP",
        "StateNameField": "bta2dp.codec",
   },
   {
        "LayerName": "AVRCP",
        "StateNameField": "btavrcp.notification.event_id",
   },
   {
        "LayerName": "RFCOMM",
        "StateNameField": "btrfcomm.frame_type",
   },
   {
        "LayerName": "L2CAP",
        "StateNameField": "btl2cap.cmd_code",
   },
   {
        "LayerName": "LMP_accepted",
        "StateNameField": "btbrlmp.opinre",
   },
   {
        "LayerName": "LMP",
        "StateNameField": [
            "btbrlmp.eop",
            "btbrlmp.op"
        ],
   },
   {
        "LayerName": "Baseband",
        "StateNameField": "btbbd.type",
   }
],
```


Figure 20: BT State Mapping Rules in JSON format


```
"Mapping": [
    {
        "LayerName": "GATT",
        "StateNameField": "btatt.uuid16",
    },
    {
        "LayerName": "ATT Error",
        "StateNameField": "btatt.error_code",
    },
    {
        "LayerName": "ATT",
        "StateNameField": "btatt.opcode.method",
    },
    {
        "LayerName": "SMP",
        "StateNameField": "btsmp.opcode",
    },
    {
        "LayerName": "L2CAP",
        "StateNameField": "btl2cap.cmd_code",
    }
],
```


Figure 21: BLE (Host) State Mapping Rules in JSON format


```
"Mapping":    [
    {
        "LayerName": "EAP",
        "StateNameField": [
             "eap.type",
             "eap.code"
         ],
    },
    {
        "LayerName": "802.1X",
        "StateNameField": [
             "eapol.keydes.type",
             "eapol.type"
         ],
    },
    {
        "LayerName": "Action",
        "StateNameField": "wlan.fixed.action_code",
    },
    {
        "LayerName": "802.11",
        "StateNameField": "wlan.fc.type_subtype",
    }
],
```


Figure 22: Wi-Fi State Mapping Rules in JSON format