**BRAKTOOTH: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing**

Matheus E. Garbelini  
*SUTD*

Vaibhav Bedi  
*SUTD*

Sudepta 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 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 fuzzier 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 fuzzers 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 fuzzier 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

![Common OSI Layers of Bluetooth Classic Stack](image)

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
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/transition 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.
Algo 1 get-state-label Procedure

1: Input: Packet P, Rules $M$: \{$(\Phi^i, F^i)$ \mid $i \in [1,N]$\}
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 $L \in$ layers($P$)
5: for each $L \in$ layers($P$) do
6: \hspace{1em} initialise state layer and type
7: \hspace{2em} state.layer := state.type := empty; $i := 1$
8: \hspace{1em} continue the mapping for $P$ until a state label is found
9: repeat
10: \hspace{2em} check whether the layer matches with the next rule
11: \hspace{3em} if $L$ satisfies $\Phi^i$ then
12: \hspace{4em} record matching layer and field names
13: \hspace{5em} state.layer := L.name; $F_{map}$ := $F[L] \cap F^i$
14: \hspace{4em} for each $f \in F_{map}$ do
15: \hspace{6em} $v$ := value of field $f$ in packet $P$
16: \hspace{6em} state.type := concat(state.type, lookup[v])
17: \hspace{4em} end for
18: \hspace{3em} end if
19: end if
20: $\Theta$ := (state.layer = empty \lor state.type = empty)
21: $i := i + 1$
22: until $(\Theta = false) \lor (i > N)$
23: if (state.layer \neq empty \land state.type \neq empty) then
24: \hspace{1em} state.name := state.dir \oplus state.layer \oplus state.type
25: \hspace{1em} return state.name
26: \hspace{1em} 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 $M_{d}$; a list of $N$ pairs $(\Phi^i, F^i)$ for $i \in [1,N]$. $\Phi^i$ 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 $M_{d}$: the first rule captures the protocol layer name (i.e., layer.name) as L2CAP and the packet type in L2CAP is identified in field $F^1$ = l2cap.code. Similarly, the second rule captures the protocol layer LMP and target fields \{lmp.eop, lmp.code\}. 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 $M_{d}$ (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).

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.
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.

![Diagram](image.png)

**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., $Y_i(S)$ 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 $Y_i(S)$ and $Y_i(S')$ respectively. We observe that $pr^1_i$ probability is assigned for mutating the Baseband (BB) layer, whereas the ACL Header is mutated with probability $pr^2_i$. Then, for each packet type, all the fields e.g., LT_ADDRESS, Type, receive the same probability $pr^1_i$ for mutation. Finally, the fuzzer iterates over all layers and if a layer $L_i$ hits its chance $pr^i_f$, each field of layer $L_i$ is mutated with a probability $pr^i_f$.

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 $Y = \bigcup_{S \in M_{ref}} Y_i(S)$ 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.

![Diagram](image.png)

**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 $r_{rel}$ for a packet $P$ to be duplicated ($r_{rel} \in [0, 1]$) and the time $t$ (in milliseconds) for which the packet is kept in a queue $Q_t$ before transmitting $P$ out-of-order. The time $t$ is chosen randomly between $[1, D_T]$. Adjusting $r_{rel}$ 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 $r_{rel}$ 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).

![Targets and Connections](image)

**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 simply created by 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_buff).

![Dissection Hook](image)

**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

```c
int tx_post_dissection(pkt_buf, pkt_length) {
    //Detect LMP_features_req
    if (packet_has_condition("bttlmajo=")) {
        //Send LMP_setup_complete after
        send_packet(LMP_setup_complete); return 0;
    }
    //Detect LMP_features_req_ext
    if (packet_has_condition("bttlmajo=")) {
        //Mutate LMP_features_req_ext to unknown opcode
        pkt_buf[0]=0x03; 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 625us [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 labor 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
results in a loss of generality of our OTA fuzz 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 10: Packet Hook strategy employed by our BT Fuzzing Interface and InternalBlue framework.](image)

![Figure 11: ESP-IDF runtime patching to fix ROM functions.](image)

### 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 `calls`, 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.](image)
Using BTPatcher, we introduce hooks to relevant libbtm.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 firmwares (i.e., user code) as discussed in Section 5.3.

**ROM Patcher**: Enables the host to mutate Baseband TX Hold/Release:

**TX Injector**

**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 1.25ms), facilitatingflooding attacks such as AU Rand Flooding (see Section 6).

**ROM Patch**: 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 Graphiv++ [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 sdprfcomm_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 sdprfcomm_query with each target device for about 1h. 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.

**BRKTooth 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.

<table>
<thead>
<tr>
<th>BT SoC / Vendor</th>
<th>BT SoC</th>
<th>Dev. Kit / Product</th>
<th>Sample Code</th>
<th>Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>AX2100</td>
<td>Laptop Forge15-R</td>
<td>N.A</td>
<td>SSB</td>
</tr>
<tr>
<td>Qualcomm</td>
<td>WCN399X</td>
<td>Xamii Pocophone F1</td>
<td>N.A</td>
<td>ADB</td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>CC2564C</td>
<td>CC2532XQSN-EM</td>
<td>SPPDMultiDemo</td>
<td>Serial</td>
</tr>
<tr>
<td>Cypress</td>
<td>CYW20735B1</td>
<td>CYW20735Q6060V-01</td>
<td>rcomms_serial_port</td>
<td>Serial</td>
</tr>
<tr>
<td>Bluefruit Technology</td>
<td>AB3501A</td>
<td>AB3520G1</td>
<td>Default</td>
<td>N.A</td>
</tr>
<tr>
<td>Zhuhai Jiei Technology</td>
<td>AC952C</td>
<td>XY-WRBT Module</td>
<td>N.A</td>
<td>Mic.</td>
</tr>
<tr>
<td>Actions Technology</td>
<td>ATS281X</td>
<td>Xiaomi MID-B3-64-DB</td>
<td>N.A</td>
<td>Mic.</td>
</tr>
<tr>
<td>Zhuhai Jiei Technology</td>
<td>AC9005X</td>
<td>BT Audio Receiver</td>
<td>N.A</td>
<td>Mic.</td>
</tr>
<tr>
<td>Espressif Systems</td>
<td>ESP32</td>
<td>ESP-WROVER-KIT</td>
<td>btsr_acceptor</td>
<td>Serial</td>
</tr>
<tr>
<td>Harman International</td>
<td>JX25X</td>
<td>BTL TUNE560BT</td>
<td>N.A</td>
<td>Mic.</td>
</tr>
<tr>
<td>Qualcomm</td>
<td>CSR 8811</td>
<td>Land DVK-BT900-SA</td>
<td>vsopp_server.at</td>
<td>Serial</td>
</tr>
<tr>
<td>Beken</td>
<td>BK3260N</td>
<td>HC-05</td>
<td>V1.0.2</td>
<td>Serial</td>
</tr>
<tr>
<td>Bluetooth 4.0 + HS</td>
<td>WT32I</td>
<td>DKWT32I-A</td>
<td>ai:6.3.0-1149</td>
<td>Serial</td>
</tr>
</tbody>
</table>

We chose 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 DWKT32I-A depicted lower time to complete 1000 iterations than BT Products or Modules such as JBLTUNES500BT 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 JBLTUNES500BT, 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 4 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 $r_{rel} = 0.3$ and low $D_T = 100 ms$, the highest number of crashes were detected due to LMP flooding (V5). However, when decreasing $r_{rel}$ 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 $r_{rel} = 0.1, D_T = 6000 ms$ 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 Toole-
This page contains a table summarizing vulnerabilities in Bluetooth devices and their implementations. The table lists the CVE IDs, target layers, state types, and compliance violations. The vulnerabilities include issues such as feature requirements, protocol layer non-compliance, and baseband role switch crashes. The table also notes the impact of these vulnerabilities, whether they are LMP or Baseband related, and compliance violations such as RCE.

Another table provides a summary of vulnerabilities found in SDK/Firmware versions of vendors. This includes details such as the Dev. Kit/Product, Total Time, 1st Vulnerability, 1st Non-compl., and Model Coverage.

For comparison, the page also mentions the use of tools like bfuzz for fuzz testing Bluetooth stacks and highlights the differences between tools like bfuzz and iWRAP. The page concludes by discussing the challenges in maintaining a database of vulnerabilities and the importance of continuously testing and updating protocols to prevent new vulnerabilities.

The text also mentions the use of other tools like ESP32, MDZ-36-DB, JBL TUNE500BT, and AB32VG1, which were found to have vulnerabilities. These tools are essential for understanding the current state of Bluetooth security and for developing robust defense mechanisms against new threats.
model $M_{ref}$ affects the fuzzing process. We generated five reference models \{$M^{i}_{ref}$ | $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 $M_{ref}$ in a fuzzing session (i.e., 1000 iterations). Figure 15(a) showcases the number of ESP32 vulnerabilities per model over a fuzzing session 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 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 $M^{45}_{ref}$ 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 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...
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 ms 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_age. Additionally, general-purpose input/output (GPIO) can be controlled if the attacker knows addresses to functions controlling actuators attached to ESP32.

(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 WCN3990 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 Infini-

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 Dealock) 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 1.25ms 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.

![Diagram](image)

Figure 19: An Illustration of Truncated LMP Accepted and Feature Response 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 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.

<table>
<thead>
<tr>
<th>Extension</th>
<th>Stack</th>
<th>Target</th>
<th>Coverage</th>
<th>Vulnerability</th>
<th>CVE (New)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLE Host</td>
<td>Bluekitchen</td>
<td>ESP32</td>
<td>91.4%</td>
<td>DoS</td>
<td>CVE-2022-26594</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TI CC2540</td>
<td>67.9%</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Wi-Fi AP</td>
<td>Hostapd</td>
<td>ESP32</td>
<td>63.5%</td>
<td>EAP Heap Overwrite</td>
<td>CVE-2022-26503</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ESP8266</td>
<td>70.8%</td>
<td>Association DoS</td>
<td>CVE-2022-26609</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rasp Pi 3 B</td>
<td>57.6%</td>
<td>–</td>
<td>CVE-2022-26601</td>
</tr>
<tr>
<td></td>
<td></td>
<td>One Plus 5T</td>
<td>84.1%</td>
<td>–</td>
<td>CVE-26599</td>
</tr>
</tbody>
</table>

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


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.