# L2Fuzz: Discovering Bluetooth L2CAP Vulnerabilities Using Stateful Fuzz Testing

Haram Park, Carlos Kayembe Nkuba, Seunghoon Woo, Heejo Lee*
*Korea University*, {freehr94, carlosnkuba, seunghoonwoo, heejo}@korea.ac.kr

***Abstract*—Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR) is a wireless technology used in billions of devices. Recently, several Bluetooth fuzzing studies have been conducted to detect vulnerabilities in Bluetooth devices, but they fall short of effectively generating malformed packets. In this paper, we propose L2FUZZ, a stateful fuzzer to detect vulnerabilities in Bluetooth BR/EDR Logical Link Control and Adaptation Protocol (L2CAP) layer. By selecting valid commands for each state and mutating only the core fields of packets, L2FUZZ can generate valid malformed packets that are less likely to be rejected by the target device. Our experimental results confirmed that: (1) L2FUZZ generates up to 46 times more malformed packets with a much less packet rejection ratio compared to the existing techniques, and (2) L2FUZZ detected five zero-day vulnerabilities from eight real-world Bluetooth devices.**

***Index Terms*—Bluetooth, Fuzz Testing, Wireless Security.**

## I. INTRODUCTION

Bluetooth is a wireless communication technology that allows users to exchange various data in a short range, including Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR) and Bluetooth Low Energy (BLE). Owing to their convenience, billions of devices have adopted Bluetooth technologies [1]. Because Bluetooth is an open standard, most vendors install similar Bluetooth host stacks (*i.e.*, software stack) in their devices for interoperability among different vendors. This also implies that it is easy for malicious users to craft wireless attacks on Bluetooth devices, and even a single vulnerability has the risk of being exploited in billions of devices.

To address such undesirable situations, several studies have been conducted to detect unknown Bluetooth vulnerabilities. However, existing studies are limited in their ability to perform fuzz testing in various Bluetooth devices; they (1) required the Bluetooth pairing process (*e.g.*, Defensics [2]), (2) failed to generate valid malicious packets (*e.g.*, BFuzz [3]), (3) did not consider state information (*e.g.*, Bluetooth stack smasher [4]), or (4) were inefficient for testing various Bluetooth devices (*e.g.*, KNOB [5], BIAS [6], BlueMirror [7]), all of which impair the effectiveness of Bluetooth fuzzing in terms of detecting critical vulnerabilities (see Section VI).

To overcome these shortcomings, we propose L2FUZZ, a stateful fuzzer for Bluetooth host stacks, which targets the Logical Link Control and Adaptation Protocol (L2CAP) layer. Because all Bluetooth services use the L2CAP that is located in the lowest layer, L2CAP was chosen for our study in order to guarantee the root-of-trust of Bluetooth devices.

* Heejo Lee is the corresponding author.

L2FUZZ uses the following two key techniques: *state guiding* and *core field mutating*. Through *state guiding*, L2FUZZ maps valid commands for each L2CAP state based on their events, functions and actions. Here, the mapped commands can be used for state transitions or to test valid attacks against a specific state. Next, L2FUZZ mutates only the core (*i.e.*, critical) fields of L2CAP packets, which are in charge of the port and channel setting, through *core field mutating* technique. By mutating only core fields while not modifying other parts, L2FUZZ can generate more valid test packets, resulting in more efficient detection of potential vulnerabilities.

When we applied L2FUZZ to the selected eight test devices, L2FUZZ detected five zero-day vulnerabilities in three smartphones, one wireless earphone, and one laptop; we reported all detected vulnerabilities to the corresponding vendors.

To demonstrate the effectiveness of L2FUZZ, we compared it with existing Bluetooth fuzzing techniques [2]–[4]. To this end, we devised two novel metrics suitable for evaluating Bluetooth fuzzers: *mutation efficiency* and *state coverage*. The mutation efficiency includes the number of error-prone test packets that a fuzzer can generate, and the number of test packets that are rejected by the target. The state coverage represents the number of protocol states that a fuzzer can test.

From our experiments, we confirmed that L2FUZZ outperformed existing Bluetooth fuzzing techniques [2]–[4]. Compared with the existing techniques, L2Fuzz was able to: (1) cover and test up to 10 (out of 19) more L2CAP states, (2) generate up to 46 times more valid malformed packets, and (3) significantly reduce the probability that test packets would be rejected on the target device (up to 60% reduction).

This paper makes the following three main contributions:

- We present L2FUZZ, a stateful fuzzer for the Bluetooth host stack L2CAP layer. The key technical contributions is generating valid packets that are not likely to be rejected by the target device using *state guiding* and *core field mutating*. The source code of L2FUZZ is available at https://github.com/haramel/L2Fuzz.
- We devised two novel metrics that were suitable for evaluating Bluetooth fuzzers: *mutation efficiency* and *state coverage*, which can be used even in an environment where the target device is a black-box.
- When we applied L2FUZZ to eight Bluetooth devices, it detected five zero-day vulnerabilities, including a denial-of-service on Android devices and a crash on Apple devices.

## II. Background and Motivation

In this section, we first introduce an overview of Bluetooth BR/EDR 5.2, mainly focusing on L2CAP, and then discuss the motivation behind the development of L2Fuzz.

### A. Overview of Bluetooth BR/EDR and L2CAP

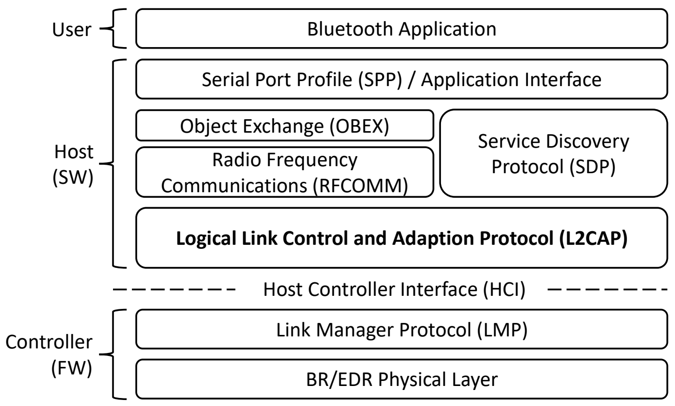

User
Bluetooth Application
Host (SW)
Serial Port Profile (SPP) / Application Interface
Object Exchange (OBEX)
Radio Frequency Communications (RFCOMM)
Service Discovery Protocol (SDP)
Logical Link Control and Adaption Protocol (L2CAP)
Host Controller Interface (HCI)
Controller (FW)
Link Manager Protocol (LMP)
BR/EDR Physical Layer

**Fig. 1:** Illustration for Bluetooth protocol stack.

Bluetooth BR/EDR is a wireless technology for short-range communication [8]. A Bluetooth network consists of a master device (*i.e.*, initiator) and a slave device (*i.e.*, acceptor). Both devices use a Bluetooth protocol stack, which consists of a controller stack (*i.e.*, firmware) and a host stack (*i.e.*, software), for communication (see Figure 1). Bluetooth BR/EDR is also referred to as Bluetooth in this paper. To use the service provided by the Bluetooth application, the master and slave devices should start the pairing process using their controller stack. Then, both devices create a connection between the L2CAP layer, which is the lowest layer of the host stack, enabling the connection between the upper layers. Since L2CAP is the lowest layer of the host stack, secure use of higher-layer protocols in Bluetooth applications requires a security assessment of L2CAP to ensure a root of trust.

**Introduction to L2CAP.** L2CAP is a core and essential protocol in Bluetooth because all Bluetooth applications require an L2CAP connection between the master and slave devices [9], [10]. To use Bluetooth applications, the master must know the service ports and channels of the slave services, which are handled by L2CAP. For example, suppose we intend to use a Bluetooth file transfer service. During this process, the master and slave devices first exchange an encrypted key using the controller stack. Thereafter, they share service ports and channels through the L2CAP layer. Based on these ports and channels, they create Radio Frequency Communications (RFCOMM) and Object Exchange (OBEX) connections to use file transfer applications.

**L2CAP states.** L2CAP is based on the concept of channels and consists of 19 states in Bluetooth 5.2 (see Figure 2). These states cover various scenarios that can occur during the L2CAP communication process. Each state has *event* and *action* for the state transition. To transit from the current state to the next state, a specific request should be sent to the device (*i.e.*, *event*) and an appropriate response to that request should be output from the device (*i.e.*, *action*).

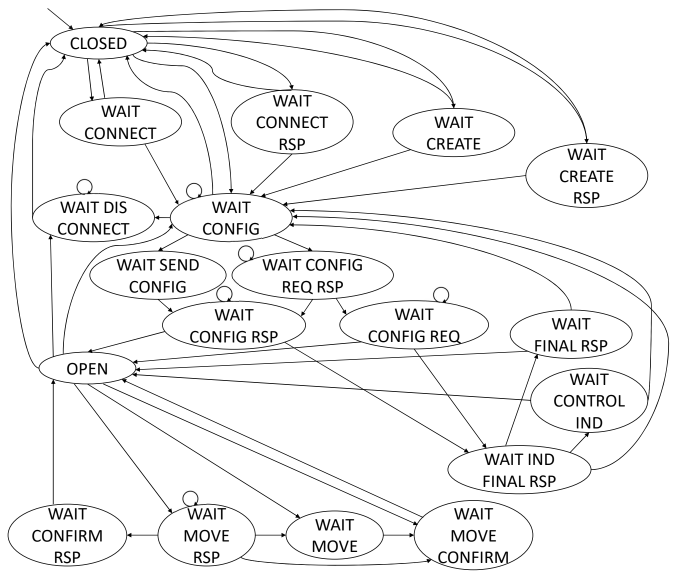
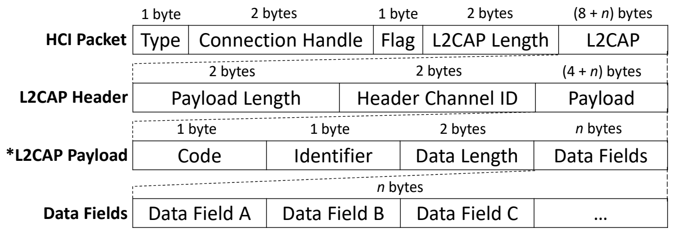


**Fig. 2:** Bluetooth 5.2 L2CAP state machine.



| | 1 byte | 2 bytes | 1 byte | 2 bytes | (8 + *n*) bytes |
|---|---|---|---|---|---|
| **HCI Packet** | Type | Connection Handle | Flag | L2CAP Length | L2CAP |

| | 2 bytes | 2 bytes | (4 + *n*) bytes |
|---|---|---|---|
| **L2CAP Header** | Payload Length | Header Channel ID | Payload |

| | 1 byte | 1 byte | 2 bytes | *n* bytes |
|---|---|---|---|---|
| ***L2CAP Payload** | Code | Identifier | Data Length | Data Fields |

| | *n* bytes | | | |
|---|---|---|---|---|
| **Data Fields** | Data Field A | Data Field B | Data Field C | … |

*L2CAP Payload can be up to 65,535 bytes.

**Fig. 3:** Bluetooth L2CAP protocol format.

**L2CAP format.** L2CAP consists of a header and a payload. The payload can have multiple `Data Fields` depending on the command being used (see Figure 3). To transmit an L2CAP packet, the packet must be configured in the order of an HCI packet, an L2CAP header, and an L2CAP payload.

The HCI packet is used to communicate between the host and the controller. The L2CAP header consists of a `Payload Length` and `Header Channel ID`; the `Header Channel ID` identifies the destination channel endpoint of the packet and transmits L2CAP commands using the signaling channel (*i.e.*, 0x0001). The L2CAP payload consists of `Code`, `Identifier`, `Data Length`, and `Data Fields`. `Code` and `Identifier` indicate the command code and packet ID, respectively. Next, `Data Fields` vary depending on the L2CAP command; there are 26 L2CAP commands in Bluetooth 5.2, and each command has different `Data Fields`. For example, "L2CAP connection request" has two `Data Fields`: Protocol/Service Multiplexer (PSM, port number) and Source Channel ID (SCID). Meanwhile, "L2CAP connection response" has four `Data Fields`: Destination Channel ID (i.e., DCID), SCID, Result and Status.

## B. Problem Statement

In this paper, we focus on detecting L2CAP vulnerabilities in a Bluetooth host stack. Because L2CAP is the lowest layer of the host stack, it is necessary to ensure a root of trust. In particular, L2CAP vulnerabilities (*e.g.*, BlueBorne [11] and SweynTooth [12]) can compromise the security of the entire system, such as causing denial-of-service and remote code execution. Therefore, to improve the security of Bluetooth applications, an effective technique that can detect vulnerabilities in the L2CAP layer is required.

Several existing approaches have attempted to resolve such undesirable situations; in particular, fuzz testing (i.e., fuzzing) was mainly performed for security verification of wireless communications [2]–[4]. However, they are limited in detecting potential vulnerabilities in Bluetooth applications owing to the following two main challenges. Failure to overcome these challenges can collectively impair the effectiveness of fuzzing.

**Challenge 1: Increasing the L2CAP state coverage.** Because Bluetooth is a stateful protocol, it follows a specific state machine (see Figure 2) and moves from one state to another [13], each of which contains its own functions to perform the desired operation (*e.g.*, a connection-related operation is conducted in the WAIT CONNECT state). The L2CAP provides state-transition functions to enter each state.

Because vulnerabilities are highly likely to occur in (1) the state transition process and (2) the functions of each state, we need to validate the security of as many states as possible. However, increasing the state coverage of a fuzzer is difficult owing to the complexity of the protocol and various implementations of the Bluetooth stack. In fact, implementations using Bluetooth specifications are conducted differently according to the vendors preferences. Therefore, it is challenging to cover several L2CAP states on such a diverse implementation of Bluetooth applications (*e.g.*, BSS [4] can cover only three out of 19 L2CAP states).

**Challenge 2: Generating valid malformed packets.** A malformed packet refers to a packet wrapped with malicious information or data [14]. Because malformed packets have a higher chance of causing crashes that lead to fatal vulnerabilities (*e.g.*, denial-of-service and buffer overflows) [15]–[17], an effective Bluetooth fuzzer needs to generate valid malformed packets that are not rejected by the target device.

However, the method of simply mutating any or all fields of L2CAP packets without considering the characteristics of each field, which is used in the existing Bluetooth fuzzing techniques, results in most of the generated packets being rejected by the target devices [18].

In addition, it is challenging to apply the existing mutating algorithms of file fuzzing studies (*e.g.*, AFL [19]) to Bluetooth fuzzing; because there are as many Bluetooth stacks as the number of manufacturers, and most of them are not open-sourced, the technology of existing file fuzzing studies cannot be applied.

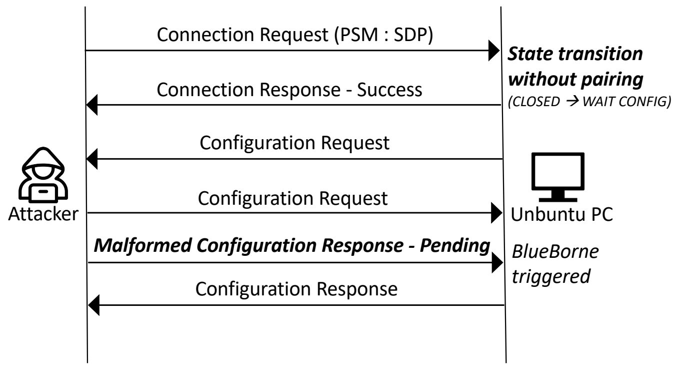

Connection Request (PSM : SDP)
Connection Response - Success
Configuration Request
Configuration Request
Malformed Configuration Response - Pending
Configuration Response
Attacker
Unbuntu PC
State transition without pairing
(CLOSED → WAIT CONFIG)
BlueBorne triggered

**Fig. 4:** Illustration of CVE-2017-1000251 attack flow.

## C. A Motivating Example

To suggest the need for an effective Bluetooth vulnerability detection technique, we introduce the CVE-2017-1000251 case (called BlueBorne [11]), which is a critical remote code execution vulnerability discovered in the Linux Bluetooth host stack (*i.e.*, BlueZ [20]). Figure 4 illustrates the attack flow of the BlueBorne.

First, the attacker forms an L2CAP connection using a service discovery protocol (SDP) port that does not require pairing. After the L2CAP connection is established, the victim device enters the configuration state. During the configuration process, the attacker sends a normal configuration request packet and a malformed configuration response packet to the victim device. Because these packets are valid in the configuration state, the victim's device accepts the packets without rejection, which leads to a fatal attack.

In the overall attack flow, we focused on the two main steps that are central to the BlueBorne attack scenario: entering the configuration state, and sending malformed packets. Existing Bluetooth fuzzing techniques: (1) do not fully consider the L2CAP states, and (2) do not generate valid malformed packets (for testing purposes) for each L2CAP state. Therefore, they easily fail to reach the specific state (e.g., configuration state), and even if they reach it, they send invalid packets that are rejected by the target device, which results in a failure to identify critical vulnerabilities including BlueBorne.

In this regard, a mature fuzzing technique is required to overcome the two aforementioned challenges (see Section II-B), that is, to generate malicious packets valid for each state while having high L2CAP state coverage.

## III. Methodology

In this section, we introduce the methodology of L2Fuzz, a stateful fuzzer for detecting Bluetooth L2CAP vulnerabilities.

## A. Overview

Figure 5 depicts the overall workflow of L2Fuzz, which comprises the following four phases: (1) target scanning, (2) state guiding, (3) core field mutating, and (4) vulnerability detecting. L2Fuzz first scans a target Bluetooth device and creates an L2CAP socket with its MAC address by connecting

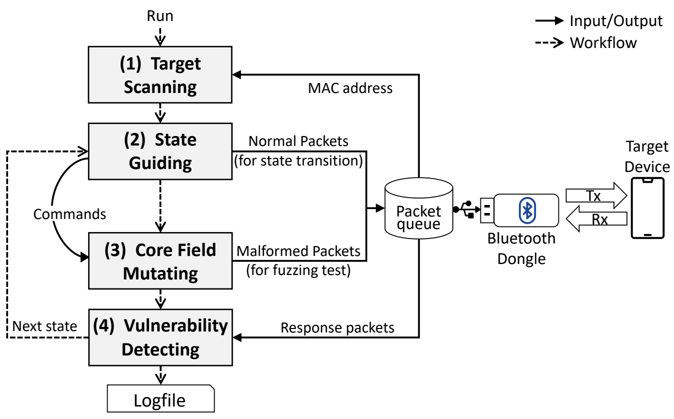


**Fig. 5:** High-level workflow of L2FUZZ.

a port that does not require pairing. L2FUZZ then traverses L2CAP states and generates malformed packets for testing purposes. For state transition, L2FUZZ uses predefined valid commands for each of the 19 L2CAP states; these valid commands were mapped based on the event and action of each state. To generate valid malformed packets, L2FUZZ segments an L2CAP packet format into parts to be mutated (*i.e.*, mutable fields) and parts to be maintained. By mutating only mutable fields while it does not change the maintained parts, L2FUZZ can generate valid malformed packets that are less likely to be rejected in the target device. Finally, L2FUZZ detects potential vulnerabilities by sending the generated malformed packets to the target device.

## B. Target scanning

L2FUZZ first scans the target device's meta-information, namely, the MAC address (for establishing L2CAP socket), device name, class of the device (*e.g.*, smartphone or laptop), and its Organizationally Unique Identifier (OUI). Thereafter, L2FUZZ scans the service ports of the target device to detect ports that do not require pairing. Because (1) attackers often exploit Bluetooth vulnerabilities without pairing, (*e.g.*, see Section II-C) (2) fuzzing after pairing is meaningless as this process is the same as remotely controlling the target device after gaining permission, and (3) for ports that require pairing, sending test packets without pairing causes the device to reject command packets without parsing any fields. Therefore, we decided that L2Fuzz should perform fuzz testing without pairing to counteract such external attacks.

**Potentially exploitable port scanning.** To find potentially exploitable ports that do not require pairing, L2FUZZ receives a list of supported service ports on the target device and attempts to connect to each service port. If all ports are identified as requiring pairing, L2FUZZ then attempts to connect to the SDP port, which does not require pairing and is supported by every Bluetooth device [21], [22] as an alternative. By communicating with the target device through the port that does not require pairing, L2FUZZ can detect L2CAP vulnerabilities that are not detected in a paired situation.

## C. State guiding

Next, L2FUZZ traverses each L2CAP state in the target device. To increase the state coverage, we first identify valid commands for each state, and use them in state transition.

**State classification.** Before identifying valid commands, we first clustered the 19 L2CAP states based on the *event* that each state receives, the *functions* that internally process the desired operation in each state, and the corresponding output *action*, which we referred to as a *job*. Consequently, we classified the 19 L2CAP states into the following seven jobs:

**TABLE I:** Jobs categorized based on events, functions, and actions of L2CAP states.

| Job | States |
|---|---|
| Closed | {CLOSED} |
| Connection | {WAIT CONNECT, WAIT CONNECT RSP} |
| Creation | {WAIT CREATE, WAIT CREATE RSP} |
| Configuration | {WAIT CONFIG, WAIT CONFIG RSP, WAIT CONFIG REQ, WAIT CONFIG REQ RSP, WAIT SEND CONFIG, WAIT IND FINAL RSP, WAIT FINAL RSP, WAIT CONTROL IND} |
| Disconnection | {WAIT DISCONNECT} |
| Move | {WAIT MOVE, WAIT MOVE RSP, WAIT MOVE CONFIRM, WAIT CONFIRM RSP} |
| Open | {OPEN} |

By identifying the commands used for each job, we included commands in the test packet, which are valid for the state of the target device; this significantly reduces the possibility that the packet will be rejected by the target device. Moreover, because valid commands can be shared between different states in a job, more diverse test packets can be generated; consequently, fuzzing coverage can be increased.

As an example of job classification, we introduce the process of "Connection job" identification. Table II summarizes the events and actions of the WAIT CONNECT state confirmed in the Bluetooth 5.2 specification document.

**TABLE II:** WAIT CONNECT state's events and actions.

| Event | Action | State transition? |
|---|---|---|
| Connect Req | Connect Rsp | WAIT CONFIG |
| Connect Rsp | Reject | No |
| Config Req | Reject | No |
| Config Rsp | Reject | No |
| Disconnect Rsp | Reject | No |
| Create Channel Req | Reject | No |
| Create Channel Rsp | Reject | No |
| Move Channel Req | Reject | No |
| Move Channel Rsp | Reject | No |
| Move Channel Confirm Req | Reject | No |
| Move Channel Confirm Rsp | Reject | No |

When the target device is in the WAIT CONNECT state, if we send a connection request (Connect Req) packet, the target device does not reject the packet because this packet is valid in the current state. After executing the function related to the connection, the target device sends a corresponding response packet (Connect Rsp) and changes its state to WAIT CONFIG. We confirmed that in the WAIT CONNECT RSP state, the target device performed almost similar operations (*i.e.*, events, functions, and actions); thus, we classified WAIT CONNECT and WAIT CONNECT RSP as the *connection job*.

One consideration was that Bluetooth devices did not always display the exact same operations as defined in the documentation. For example, some Android devices did not reject the "Connect Rsp" event even though the device was in the WAIT CONNECT state. This occurred because of the variety of implementations of the Bluetooth stack. Therefore, we set the boundaries of valid commands for each job slightly more generously to increase fuzzing effectiveness, knowing that several packets may be rejected.

Consequently, we map valid commands to each job based on the specification and various packet traces. The valid commands mapped for each job are shown in Table III.

**TABLE III:** Valid commands mapped for each job.

| Job | Valid commands |
|---|---|
| Closed | All commands |
| Connection | Connect Req/Rsp |
| Creation | Create Channel Req/Rsp |
| Configuration | Config Req/Rsp |
| Disconnection | Disconnect Req/Rsp |
| Move | Move Channel Req/Rsp, Move Channel Confirmation Req/Rsp |
| Open | All commands |

By clustering states into jobs and mapping valid commands for each job, L2FUZZ can cover most of the L2CAP states in the security validation of Bluetooth devices while decreasing the test packet rejection rate. This, in turn, renders L2FUZZ more likely to detect potential L2CAP vulnerabilities.

**State transition.** With the valid commands, L2FUZZ generates normal packets, and then sends the packets to the target device through the packet queue. After receiving the packets, the target device enters the corresponding state and sends a response packet. The packet queue parses the response packets and returns the state transition result. If L2FUZZ succeeds in the state transition, it obtains the valid commands for the target state and generates a valid malformed packet in next phase. When the fuzz testing of the target state is completed, the state transition is executed again to move to the next target state.

## *D. Core field mutating*

L2FUZZ then generates malformed packets that can lead to vulnerabilities in the entered L2CAP state of the target device. To increase the effectiveness of fuzzing, the generated packets should not be rejected by the target device.

To this end, we decided not to mutate fields that can easily be checked for anomalies. In particular, we segmented an L2CAP packet format into parts to be mutated (*i.e.*, mutable fields) and parts to be maintained by referring to VFUZZ [23].

**Field classification.** Let $L$ be the L2CAP packet. We segment $L$ into fixed ($F$), dependent ($D$), and mutable ($M$) fields as follows:

$$L = F \cup D \cup M$$

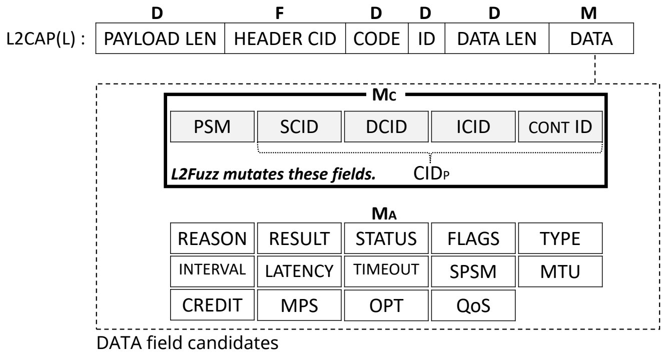


**Fig. 6:** Field classification for Bluetooth 5.2 L2CAP packet frame.

- $F$ is a set of *fixed fields*. The values are fixed.
- $D$ is a set of *dependent fields*. This values are determined by other values.
- $M$ is a set of *mutable fields*. The values are determined by devices or users.

We further classified $M$ into mutable *core* fields ($M_C$) and mutable *application* fields ($M_A$) to distinguish only core fields that can affect the core functions of the L2CAP:

$$M = M_C \cup M_A$$

- $M_C$ is a set of *mutable core fields*. The values determine the port and channel for Bluetooth network.
- $M_A$ is a set of *mutable application fields*. The values vary depending on commands and convey data to the target.

Consequently, an L2CAP packet $L$ can be conceptually segmented as follows: $L = F \cup D \cup M_C \cup M_A$. Based on this concept, we classified each field of the Bluetooth 5.2 L2CAP frame structure (see Figure 6) as follows.

- ***F*** **= {HEADER CID}**.
  - * This field is fixed because `0x0001` is used to manage the channel over the ACL-U logical links.
- ***D*** **= {PAYLOAD LEN, CODE, ID, DATA LEN}**.
  - * PAYLOAD LEN is determined by the length of the information payload (CODE, ID, DATA LEN, and DATA), CODE is determined by the valid command code (*i.e.*, determined in the state guiding phase), ID is dynamically assigned by the device, and DATA LEN is determined by the length of DATA.
- ***$M_C$*** **= {PSM, SCID, DCID, ICID, CONT ID}**.
  - * PSM is used for port settings. SCID, DCID, ICID, and CONT ID are responsible for setting the channel endpoint, and are also referred to as "Channel ID in Payload ($\text{CID}_\text{P}$)" in this paper.
- ***$M_A$*** **= {REASON, RESULT, STATUS, FLAGS, TYPE, INTERVAL, LATENCY, TIMEOUT, SPSM, MTU, CREDIT, MPS, OPT, QoS}**.
  - * Each value contains application data for commands, and is intended to deliver data without affecting port or channel management.

**TABLE IV:** Range of $M_C$ that can be used as malicious data.

| Fields ($M_C$) | Range (Hex) | | |
|---|---|---|---|
| PSM | 0100 - 01FF | 0300 - 03FF | 0500 - 05FF |
| | 0700 - 07FF | 0900 - 09FF | 0B00 - 0BFF |
| | 0D00 - 0DFF | All even values | |
| $\mathrm{CID_P}$ | 0040 - FFFF | | |

**Packet mutation.** To minimize packet rejection by the target device, L2FUZZ mutates only the $M_C$ field.

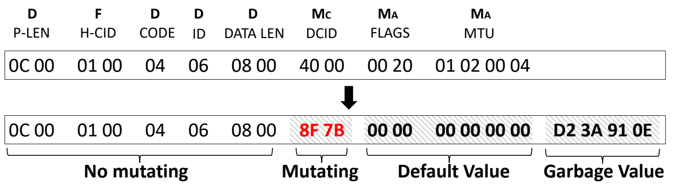
L2FUZZ does not mutate $F$ and $D$ to avoid possible packet rejection. If the target device receives a packet with $F$ or $D$ mutated, it will send a reject command response for "Command not understood". Furthermore, L2FUZZ maintains $M_A$ with its default values. This field is optional, thus, does not have a significant effect on the target device. Additionally, some of the fields can have up to 65,535 bytes of data, requiring a large amount of time to test various cases. Thus, L2FUZZ leaves these fields as default values.

In contrast, L2FUZZ mutates $M_C$ to generate various malformed packets. Specifically, different approaches are used for PSM and $\mathrm{CID_P}$ in the $M_C$ field. Regarding PSM (*i.e.*, port number), its normal range is defined in the Bluetooth specification document, and each device supports service ports within this range. Notably, the normal range has already been tested when scanning ports in the target scanning phase. Therefore, L2FUZZ considers values belonging to their abnormal range (see Table IV) and proceeds with a packet mutation. Next, in the case of $\mathrm{CID_P}$ (*i.e.*, SCID, DCID, ICID, and CONT ID), the value is dynamically assigned by the device during normal communication within the available range. If the target device receives an abnormal $\mathrm{CID_P}$ value, it sends a command rejection response for the reason "Invalid CID in request" Therefore, we decided to consider the normal range of $\mathrm{CID_P}$ (see Table IV) while ignoring dynamic allocation because, although the value is contained in the normal range, it can cause unexpected behavior on the target device due to ignoring dynamic allocation and putting different values.

Finally, L2FUZZ appends a garbage value to the tail of the packet, which increases the possibility of vulnerability detection when the packet is not rejected but parsed by the target device. Here, we considered garbage values that do not exceed the MTU size; if the garbage value exceeds the MTU size, the target device rejects the packet with the reason "Signaling MTU exceeded."

Figure 7 shows an example of mutating an L2CAP Config Req packet. L2FUZZ forcibly mutates the dynamically allocated DCID value (*i.e.*, "40 00") into "8F 7B", and adds a garbage value (*i.e.*, "D2 3A 91 0E") to the tail of the packet to generate a malformed packet.

Malformed packets generated in this manner are less likely to be rejected by the target device; thus, Bluetooth vulnerabilities can be detected more effectively than dumb mutation, which simply changes any or all fields of an L2CAP packet. The generated malformed packets are transmitted to the target device through the packet queue. The high-level algorithm of this phase is explained in Algorithm 1.



D P-LEN
F H-CID
D CODE
D ID
D DATA LEN
Mc DCID
MA FLAGS
MA MTU
0C 00 01 00 04 06 08 00 40 00 00 20 01 02 00 04
0C 00 01 00 04 06 08 00 8F 7B 00 00 00 00 00 00 D2 3A 91 0E
No mutating
Mutating
Default Value
Garbage Value

**Fig. 7:** Example of mutating L2CAP Config Req packet.

**Algorithm 1** Algorithm for core field mutating

```
    Input: $C$            // Valid commands (e.g., Connect Req)
    Output: $pkt$
    // Malformed packets to be stored in PACKETQUEUE
 1: procedure COREFIELDMUTATING($C$)
 2:     // $n$: the number of malformed packets to generate
 3:     for $i$ in range (0, len($C$)) do          // $i^{th}$ command
 4:         for $j$ in range (0, $n$) do
 5:             $pkt \leftarrow$ format($commands[i]$)
 6:             // Fields: $F$, $D$, $M_C$, and $M_A$
 7:             $pkt.F \leftarrow$ 0x0001
 8:             $pkt.D \leftarrow$ default  // Used without changes
 9:             if ($pkt.M_C$) then
10:                 if $pkt.M_C.PSM$ then
11:                     $pkt.M_C.PSM \leftarrow$ random($abnormal$)
12:                 end if
13:                 if $pkt.M_C.CID_P$ then
14:                     $pkt.M_T.CID_P \leftarrow$ random($normal$)
15:                 end if
16:             end if
17:             if $pkt.M_A$ then
18:                 $pkt.M_A \leftarrow$ default
19:                 // Used without changes
20:             end if
21:             $pkt$ = $pkt$.append($garbage$)
22:             PACKETQUEUE.append($pkt$)
23:         end for
24:     end for
25: end procedure
```

## E. Vulnerability detecting

Finally, L2FUZZ detects the L2CAP vulnerabilities in the target device. To this end, L2FUZZ checks (1) whether the packet received at the target device contains an error message, (2) whether the ping test was successfully performed, and (3) whether a crash dump was generated.

Because L2CAP is concerned with port and channel settings, if a vulnerability is found, we receive an error message related to the Bluetooth connection, which can be one of the following: *Connection Failed, Connection Aborted, Connection Reset, Connection Refused, and Timeout*. Of these, the *Connection Failed* error means that the target Bluetooth service has been shut down, which can lead to a DoS. The remaining errors indicate a target device crash and may induce a crash in the target device.

**TABLE V:** Summary of test devices used in the experiments.

| No. | Type | Vendor | Name | Year | Model | Chip | OS or FW | BT Stack | BT Ver. |
|---|---|---|---|---|---|---|---|---|---|
| D1 | Tablet PC | Google | Nexus 7 | 2013 | ASUS-1A005A | Snapdragon 600 | Android 6.0.1 | BlueDroid | 4.0 + LE |
| D2 | Smartphone | Google | Pixel 3 | 2018 | GA00464 | Snapdragon 845 | Android 11.0.1 | BlueDroid | 5.0 + LE |
| D3 | Smartphone | Samsung | Galaxy 7 | 2016 | SM-G930L | Exynos 8890 | Android 8.0.0 | BlueDroid | 4.2 |
| D4 | Smartphone | Apple | iPhone 6S | 2015 | A1688 | A9 | iOS 15.0.2 | iOS stack | 4.2 |
| D5 | Earphone | Apple | Airpods 1 gen | 2016 | A1523 | W1 | 6.8.8 | RTKit stack | 4.2 |
| D6 | Earphone | Samsung | Galaxy Buds+ | 2020 | SM-R175NZKATUR | BCM43015 | R175XXU0AUG1 | BTW | 5.0 + LE |
| D7 | Laptop | LG | Gram | 2019 | 15ZD990-VX50K | Intel wireless BT | Windows 10 | Windows stack | 5.0 |
| D8 | Laptop | LG | Gram | 2017 | 15ZD970-GX55K | Intel wireless BT | Ubuntu 18.04.4 | BlueZ | 5.0 |

Thereafter, L2FUZZ conducts a ping test. If the ping test fails, it is logged as a vulnerability according to the error message. L2FUZZ further checks whether there are any crash dumps or abnormalities. Finally, L2FUZZ stores the fuzzing results in a log file. When this step is over, the vulnerability assessment for the target state entered in the second phase (*i.e.*, state guiding) is finished. Hence, after this step, L2FUZZ returns to the second phase, and assess the next L2CAP state.

After evaluating the vulnerabilities for all L2CAP states, the vulnerability detection process for the target device is finished, and L2FUZZ reports all the detected vulnerabilities.

## IV. EVALUATION

In this section, we evaluate L2FUZZ. Section IV-A introduces the experimental setup, including the experimental environment, target devices, and evaluation metrics. Section IV-B investigates the vulnerability detection results of L2FUZZ on real-world Bluetooth devices. We then compare L2FUZZ with existing Bluetooth fuzzing techniques using the two metrics, mutation efficiency (Section IV-C) and state coverage (Section IV-D), to demonstrate the effectiveness of L2FUZZ. Finally, Section IV-E introduces a case study of the denial of service vulnerability detected in Android Bluetooth stack.

### A. Experimental setup

**Experiment environment.** L2FUZZ was implemented in approximately 1,200 lines of Python code, excluding external libraries. In particular, we used the Scapy library (v2.4.4), an interactive packet manipulation program for mutating packets. We ran L2FUZZ on a virtual machine with Ubuntu 18.04.4 LTS, 8GB memory, Intel Core i5-7500 CPU @ 3.40GHz × 4, 50GB Disk and Billionton Bluetooth Class 1 dongle.

**Target devices.** We selected eight real-world test devices that can represent the general-purpose Bluetooth protocol stacks [24], including BlueZ (Linux), BlueDroid (Android), Apple BT stack, and Windows BT stack. Table V summarizes the target devices used in the experiments.

**Baseline fuzzers for comparison.** When evaluating the effectiveness of L2FUZZ, we compared the results of L2FUZZ with those of Defensics [2], BFuzz [3], and BSS [4]; other related techniques were excluded because they did not support L2CAP vulnerability detection or were not publicly available. We compared the mutation efficiency and state coverage of L2FUZZ to the baseline fuzzers using the test device D2 (*i.e.*, Google Pixel 3 smartphone, see Table V). We used D2 in the evaluation because D2 follows the Bluetooth standard with little customization as a "reference phone" selected by Google. Therefore, we expected that Bluetooth vulnerabilities would be most clearly tested in D2.

**Evaluation metrics.** Because most Bluetooth stacks, except for BlueZ and BlueDroid, are closed sources, Bluetooth fuzzers are close to blackbox fuzzers; evaluation metrics used in whitebox or greybox fuzzing [19], [25], such as source code coverage, are difficult to use for evaluation here. Thus, we suggest two metrics, which can be measured only with the packet trace, for evaluating Bluetooth fuzzers: *mutation efficiency* and *state coverage*, which can be measured even in an environment where the target device is a black-box:

- **Mutation efficiency.** This refers to the minimum percentage of malformed packets transmitted without rejection. To measure this metric, we calculated the *Malformed Packet Ratio (MP Ratio)* and the *Packet Rejection Ratio (PR Ratio)*, by capturing malformed and rejected packets through packet sniffing tools (*e.g.*, Wireshark [26]).

$$\mathit{MP\ Ratio} = \frac{\mathit{\#Transmitted\ Malformed\ Packets}}{\mathit{\#Transmitted\ Packets}}$$

$$\mathit{PR\ Ratio} = \frac{\mathit{\#Received\ Rejection\ Packets\ from\ Target}}{\mathit{\#Received\ Packets\ from\ Target}}$$

The mutation efficiency, which represents the ratio of malformed packets transmitted without rejection, is calculated as follows.

$$\mathit{Mutation\ efficiency} = \mathit{MP\ Ratio} * (1 - \mathit{PR\ Ratio})$$

- **State coverage.** This metric refers to the number of L2CAP states to be covered. Because vulnerabilities are highly likely to occur in the state transition process and the functions of each state, the more L2CAP states were covered, the higher the likelihood of detecting vulnerabilities. It can be measured by protocol reverse engineering tool (*e.g.*, PRETT [27]).

### B. Vulnerability detection results in real-world devices

We applied L2FUZZ to eight selected test devices for detecting unknown Bluetooth vulnerabilities. Owing to the characteristics of Bluetooth, wherein the device and fuzzing are terminated when a valid vulnerability is found, it is difficult to measure the number of detected vulnerabilities. Therefore, we measured (1) whether vulnerabilities were detected and (2) the elapsed time required to detect vulnerabilities.

In our experiments, we confirmed that L2FUZZ detected **five zero-day vulnerabilities**; the results are shown in Table VI.

**TABLE VI:** Vulnerability detection results of L2Fuzz.

| Device | Vuln? | Description | Elapsed Time | Reported to Vendors? |
|---|---|---|---|---|
| D1 | **Yes** | DoS | 1 m 32 s | Yes |
| D2 | **Yes** | DoS | 1 m 25 s | Yes |
| D3 | **Yes** | DoS | 7 m 11 s | Yes |
| D4 | No | N/A | N/A | N/A |
| D5 | **Yes** | Crash | 40 s | Yes |
| D6 | No | N/A | N/A | N/A |
| D7 | No | N/A | N/A | N/A |
| D8 | **Yes** | Crash | 2 h 40 m | Discussing |

* D1, D2, D3: A denial of service was triggered because of a null pointer dereference by malformed packets. The vendor became aware of this vulnerability. (see Section IV-E).

* D5: The device was unexpectedly terminated owing to the malformed packets. This has been patched by the vendor.

* D8: A crash dump was generated owing to a general protection failure by malformed packets. We are discussing this issue with the vendor.

L2Fuzz discovered DoS vulnerabilities in three Android devices (*i.e.*, D1, D2, and D3). The crash was triggered in the state of device, which allowed malicious commands with the value CID$_P$. Additionally, a tombstone file (*i.e.*, Android crash dump [28]) was generated in each device, resulting in Bluetooth termination for all devices (DoS triggered); details are explained in Section IV-E.

L2Fuzz further detected crashes in two devices (*i.e.*, D5 and D8), a wireless earphone, and a laptop. Regarding D5, a crash occurred in a state that allowed commands with a malicious PSM value, resulting in an abnormal phenomenon (*i.e.*, termination without any control). For D8, a crash dump file was created within the target device, and the Bluetooth communication content and general protection errors were recorded in the crash dump.

Notably, with the exception of D8, all vulnerabilities were detected within several minutes. It was infeasible to closely analyze the direct factors affecting performance, because the source code of all Bluetooth stacks were not publicly disclosed. Instead, we confirmed that the vulnerability was detected within one minute in D5 (supporting six service ports) while requiring more than two hours on D8 (supporting 13 service ports). Subsequently, we can infer that the elapsed time was determined based on the number of service ports provided and the logic complexity of Bluetooth applications. We responsibly reported all five detected vulnerabilities to the corresponding vendors.

Although L2Fuzz discovered five zero-day vulnerabilities, it failed to detect vulnerabilities in three devices: D4, D6, and D7, which used iOS, BTW, and Windows stack, respectively. Their Bluetooth stack is based on the Bluetooth specification document; however, they also have proprietary protocol layers and logic. They may have implemented an exception handling logic for malformed packets generated by L2Fuzz.

## C. Mutation efficiency measurement

Next, we measured the mutation efficiency of L2Fuzz and compared it with the three existing Bluetooth fuzzing techniques (*i.e.*, Defensics, BSS, and BFuzz). For a fair comparison, controlled experiments were required. Because each fuzzer sends a different number of packets per second, we measured the MP and PR Ratios of each fuzzer based on 100,000 sent packets. We ran the four fuzzers on Google Pixel 3 with Android 11 devices (*i.e.*, D2). Malformed and rejected packets were captured and analyzed using Wireshark.

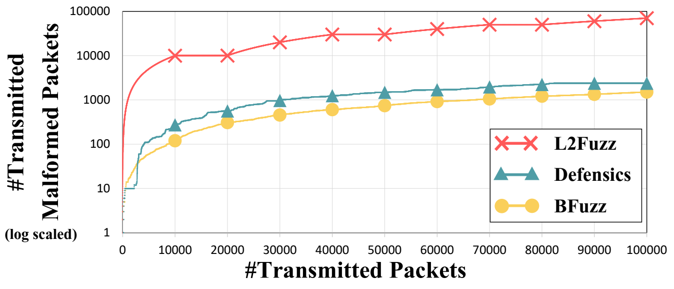


**Fig. 8:** MP Ratio measurement results for the four Bluetooth fuzzing techniques. BSS did not generate malformed packets, thus it is not displayed on the graph.

**MP Ratio measurement.** Figure 8 shows the MP Ratio measurement results for the four fuzzing techniques. Notably, L2Fuzz can generate up to 46 times more malformed packets than other techniques. L2Fuzz generated malformed packets accounting for an average of 33.48% during testing, and generated a total of 69,966 packets (*i.e.*, 69.96% *MP Ratio*). Coversely, Defensics generated malformed packets accounting for 1.40% on average and generated 2,380 packets in total (*i.e.*, 2.38% *MP Ratio*). BFuzz generated malformed packets accounting for 0.74% on average and generated a total of 1,506 packets (*i.e.*, 1.50% *MP Ratio*). Notably, the BSS did not generate any malformed packets (*i.e.*, 0% *MP Ratio*).

We confirmed that the MP Ratio values of the fuzzing techniques vary depending on the mutation strategy used for each technique. Particularly, the existing techniques performed packet mutation without considering the characteristics of the L2CAP packet fields. For examples, BFuzz mutated all fields of the packet except for the fixed fields, and BSS mutated only one field. Therefore, they failed to effectively generate malformed packets. However, the L2Fuzz approach, which generates malformed packets with core field mutating, showed much higher MP Ratio than others.

**PR Ratio measurement.** Figure 9 shows the measured PR Ratio values for the selected four fuzzing techniques. Based on 100,000 received packets from the target, BFuzz showed the highest packet rejection ratio (*i.e.*, 91.60% *PR Ratio*), followed by L2Fuzz (*i.e.*, 32.49% *PR Ratio*), and then Defensics (*i.e.*, 1.73% *PR Ratio*). Because BSS does not generate malicious packets, the PR Ratio was also 0%.

Similar to the MP Ratio cases, the PR Ratio values were different owing to the difference in the mutation strategy of each fuzzing technique. For example, BFuzz, which showed the highest PR Ratio, mutated the dependent fields ($D$), resulting in test packets rejected by the target device.

One important observation is that a lower PR Ratio does not always indicate that a fuzzer is efficient. In the experimental

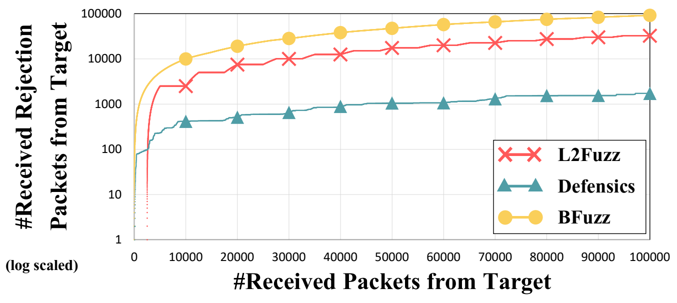


**Fig. 9:** PR Ratio measurement results for the four Bluetooth fuzzing techniques. BSS did not receive any rejection packets, thus it is not displayed on the graph.

results, we confirmed that the PR Ratio of Defensics was lower than that of L2FUZZ. This is mainly owing to following two reasons. First, Defensics exhibited a low rejection ratio because it hardly generated malformed packets. Furthermore, a Bluetooth application forms as many channels as the number of supported Bluetooth services. In L2FUZZ, some packets were rejected because L2FUZZ formed more channels than the maximum number in one L2CAP state. Because Defensics only tests one packet per state, there is less chance of being rejected. In summary, although Defensics showed a low PR Ratio, it hardly made malformed packets and did not sufficiently inspect each L2CAP state.

In contrast, we confirmed that L2FUZZ showed a relatively low PR Ratio while generating a sufficiently large number of malicious packets, with the help of core field mutating technique.

**TABLE VII:** Results of the mutation efficiency measurement.

| Fuzzer | MP Ratio | PR Ratio | Mutation efficiency |
|---|---|---|---|
| L2Fuzz | 69.96% | 32.49% | 47.22% |
| Defensics | 2.38% | 1.73% | 2.33% |
| BFuzz | 1.50% | 91.60% | 0.12% |
| BSS | 0% | 0% | 0% |

*MP Ratio = *Malformed Packet Ratio*
*PR Ratio = *Packet Rejection Ratio*
*Mutation efficiency = MP Ratio * (1 - PR Ratio)

**Mutation efficiency measurement.** We then calculated the mutation efficiency for each fuzzer using the measured MP and PR Ratios. Table VII presents the measurement results.

We confirmed that L2FUZZ was able to transmit the largest number of malformed packets without rejection; L2FUZZ showed a mutation efficiency of 47.22%. The mutation efficiency value of Defensics, which showed the lowest PR Ratio, is 2.33% because Defensics hardly generates malformed packets (*i.e.*, MP Ratio was significantly low). Further, the mutation efficiency of BFuzz, which produced few malicious packets and showed a high rejection ratio, was 0.12%, and the mutation efficiency of BSS, which failed to generate malicious packets, was 0%. Moreover, L2FUZZ transmitted 524.27 packets per second (pps), allowing more packets to be tested in a shorter time than Defensics (3.37 pps), BFuzz (454.54 pps), and BSS (1.95 pps).

From our experimental results, we confirmed that L2FUZZ outperformed existing Bluetooth fuzzing techniques in terms of generating more malformed packets that were less likely to be rejected by the target device.

## D. State coverage measurement

Next, we examined the number of L2CAP states that each fuzzing technique could cover; the more covered states, the more likely the fuzzing technique to detect a Bluetooth vulnerability (see Section III-C).

We investigated each fuzzer's state coverage by analyzing the packet trace captured using PRETT [27]. For fuzzers with fixed test times (*i.e.*, Defensics), we analyzed the packet traces at the end of the test. For the remaining fuzzer with no test time limit, packet traces were analyzed at the end of a single test cycle. The results are shown in Figure 10 and Figure 11.

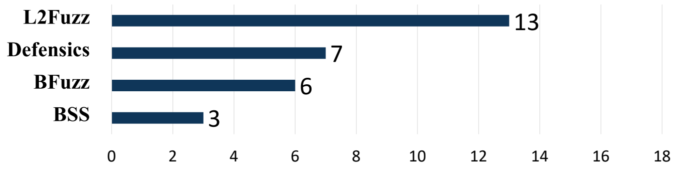


**Fig. 10:** L2CAP state coverage by different fuzzers.

**Results.** From our experiment, we confirmed that L2FUZZ can cover almost twice as many L2CAP states (13 states) than existing fuzzers (at most seven states). L2FUZZ improved the accuracy of state transition by mapping only valid commands to each state, which rendered it possible to cover more L2CAP states (see Section III-C). L2FUZZ could cover up to 13 states, including L2CAP states classified as *move* and *creation* jobs that were not covered by existing fuzzers. Conversely, the state coverage values of Defensics (*i.e.*, seven states), BFuzz (*i.e.*, six states), and BSS (*i.e.*, three states) were less than that of L2FUZZ because they did not leverage valid commands for each state and were less effective at checking the target's response. One reason is that the Bluetooth specification document they used was outdated (*i.e.*, Bluetooth core 2.1, published in 2007 [29]). This is not at technical limitation, but it indicates that L2FUZZ is more efficient for checking Bluetooth devices that reflect the latest specifications.

In summary, L2FUZZ showed far superior mutation efficiency and state coverage compared to existing fuzzers. This indicates that L2FUZZ can detect L2CAP vulnerabilities of Bluetooth devices more effectively in practice.

## E. Case study

We introduce a zero-day DoS vulnerability detected in Android Bluetooth devices (*i.e.*, D2, see Table V).

L2FUZZ connected to the D2 device's SDP port, and then performed state transition to the configuration states (*i.e.*, configuration job). Afterwards, when a malicious packet with a DCID value of 0x40 and garbage added was sent to the target device, we confirmed that a null pointer deference

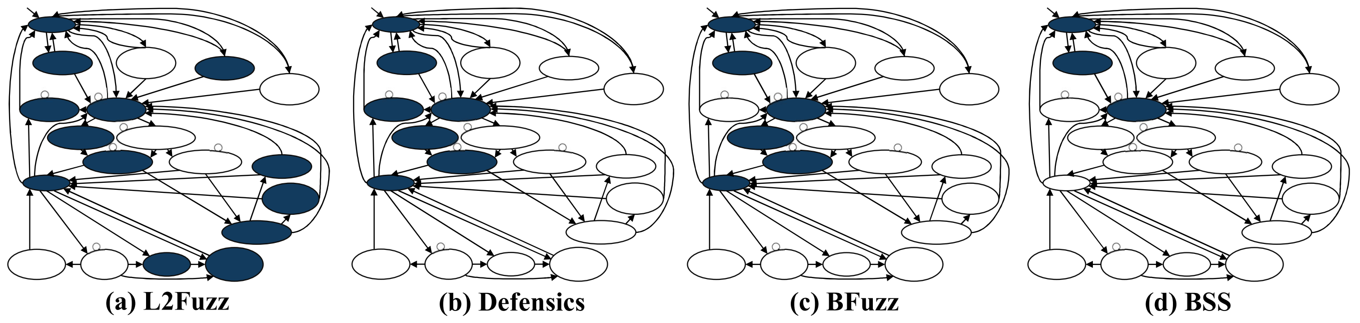
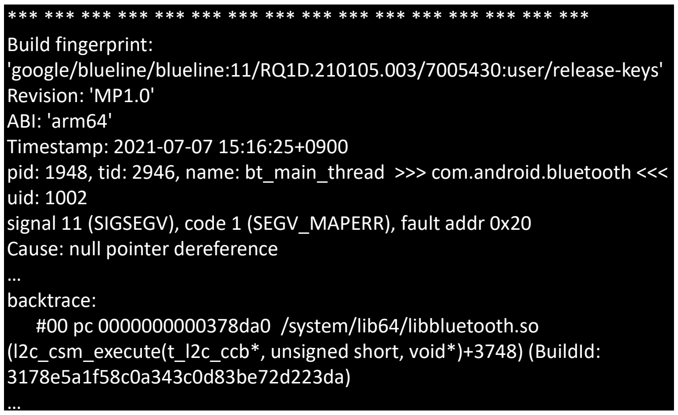


**Fig. 11:** Illustration of the state coverage for each fuzzer based on the L2CAP state machine (see Figure 2). Highlighted states represent testable L2CAP states in each fuzzer.



```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint:
'google/blueline/blueline:11/RQ1D.210105.003/7005430:user/release-keys'
Revision: 'MP1.0'
ABI: 'arm64'
Timestamp: 2021-07-07 15:16:25+0900
pid: 1948, tid: 2946, name: bt_main_thread  >>> com.android.bluetooth <<<
uid: 1002
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x20
Cause: null pointer dereference
...
backtrace:
    #00 pc 0000000000378da0  /system/lib64/libbluetooth.so
(l2c_csm_execute(t_l2c_ccb*, unsigned short, void*)+3748) (BuildId:
3178e5a1f58c0a343c0d83be72d223da)
...
```

**Fig. 12:** Tombstone of Google Pixel 3 with BlueDroid. Vulnerability occurs at $t\_l2c\_ccb*$ (L2CAP channel control block) that uses $\mathrm{CID_P}$.

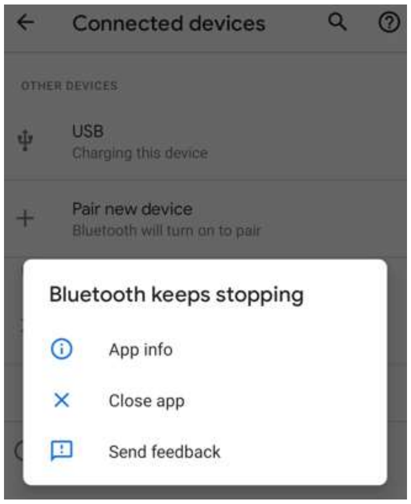


**Fig. 13:** Screenshot of crash message on Google Pixel 3. The device shows an error message and the Bluetooth is paralyzed.

was triggered in L2CAP layer of the target device. When a vulnerability occurs, the contents recorded in Tombstone [28], an Android crash dump file, are shown in Figure 12.

As a result of analyzing the root cause, the DCID and garbage values of the malformed packet influenced the channel control block of the L2CAP layer. In addition, the target device's display shows an error message about Bluetooth termination (see Figure 13). To use Bluetooth again, we had to reset the Bluetooth function. The L2FUZZ approach, which generates valid malformed packets while covering enough L2CAP states without pairing, could detect this zero-day vulnerability. Furthermore, DoS vulnerability was also detected on older versions of Android devices (*i.e.*, D1 and D3). It is noteworthy that in the case of D3 (Galaxy7), DoS was triggered by malformed Create-Channel-Request that only L2FUZZ supports, and was detected in the Wait-Create state, which only L2FUZZ covers.

We responsibly reported this vulnerability to the Android security team. After discussing the cause and symptoms of the vulnerability, they became aware of the vulnerability (Android ID 195112457). Additionally, we found a vulnerability that forced Apple's wireless earphone device (*i.e.*, D5, see Table V) to shut down. We reported this issue to the Apple security team, and they confirmed it and fixed the vulnerability [30].

## V. DISCUSSION

Here we discuss several considerations related to L2FUZZ and countermeasures towards detected vulnerabilities.

**Applicability to other protocols.** The methodology of L2FUZZ can be applied to other Bluetooth core protocols such as RFCOMM, SDP, and OBEX. Since these protocols also use their own state machines, we expect that the state guiding of L2FUZZ can lead users to test more states. Also, the packet format of these protocols can be divided into core fields and other fields, thus we can apply the core field mutating technique. Moreover, these protocols necessarily use L2CAP as they are on a higher layer than L2CAP (see Figure 1). This means that the generated L2FUZZ's malformed packets (for testing L2CAP vulnerabilities) can also be used directly during fuzz testing for the protocols. For these reasons, we determine that L2FUZZ can be applied to other Bluetooth protocols other than L2CAP; we leave this for future work.

**Countermeasures.** To counter detected Bluetooth vulnerabilities, vendors of Bluetooth devices are encouraged to patch any detected vulnerabilities by updating the functionality that leverages PSM and $\mathrm{CID_P}$ in the Bluetooth L2CAP layer. We also suggest encrypting each protocol as a fundamental solution. Existing Bluetooth security technology has relied heavily on pairing. However, this cannot address attacks that do not require pairing, such as Blueborne (see Section II-C). Therefore, we believe that developing encryption methods for each protocol can resolve exposure to more vulnerabilities.

**Limitations and future work.** First, although L2FUZZ effectively detected vulnerabilities in most cases, it was not capable of performing long-term fuzzing; when a fatal bug is triggered on the target device, it forcibly shuts down Bluetooth. Therefore, the tester must manually reset the device to perform another test. We will consider overcoming this issue by leveraging a virtual environment [31]. Second, L2FUZZ can detect vulnerabilities by analyzing the target's response packets; however, the root cause cannot be determined immediately. We intend to resolve this issue by considering the internal log hooking that analyzes the crash root cause, similar to ToothPicker [32]. Third, L2FUZZ cannot evaluate code coverage. Since Bluetooth devices are black-boxes and closed source, it is difficult to measure code coverage. We noted that Frankenstein [31] succeeded in measuring code coverage in a limited way using binaries even though it required complex tasks such as firmware emulation. We will try to apply Frankenstein's method to L2FUZZ. Finally, while L2FUZZ covers a considerable number of L2CAP states, there are still cases where it does not, *e.g.*, when L2FUZZ (as a master) connects with a target device that is a slave, there may be restrictions on the state that the target device can enter. We are considering leveraging techniques such as injecting applications that control state transitions of the test target.

**Responsible vulnerability disclosure.** We reported all detected vulnerabilities in our experiments to the vendors: Android, Apple, Samsung, and the Ubuntu BlueZ team. Among them, a crash found in Apple devices was patched by the vendors. The remaining vulnerabilities are currently under discussion. In addition, we have found several vulnerabilities in devices that are not mentioned in this paper. However, the information cannot be disclosed due to the vendor's rejection.

## VI. Related work

**Bluetooth fuzzing techniques.** Existing Bluetooth fuzzing techniques (1) are inefficient for testing various Bluetooth devices, (2) do not generate valid malformed packets, and (3) do not cover enough L2CAP states.

Sweyntooth [12], Frankenstein [31] and ToothPicker [32] attempted to detect Bluetooth vulnerabilities through fuzz testing. However, they did not focus on the Bluetooth BR/EDR host stack, which is a software commonly used in devices that provide Bluetooth services. In particular, Sweyntooth focused on Bluetooth Low Energy (BLE) protocol stack which is different from Bluetooth BR/EDR. Frankenstein focused on Bluetooth BR/EDR; however, it concentrates on the controller stack (firmware) that is different from the host stack (software). ToothPicker only focused on Apple's customized Bluetooth protocol stack, which is different from the common Bluetooth BR/EDR. Therefore, they are not suitable for detecting vulnerabilities in the commonly used BR/EDR host stack, which is the target of this paper.

There are several commercial Bluetooth fuzzers target BR/EDR host protocol stacks such as Bluetooth stack smasher (BSS) [4], BFuzz [3] and Defensics [2]. However, their test packets are not efficient in detecting vulnerabilities in Bluetooth devices (see Section IV). Regarding BSS, it simply mutates only one field of a packet, which is insufficient to trigger vulnerabilities in the latest Bluetooth devices. BFuzz mutates packets that have previously been determined to be vulnerable; however, because it mutates almost every field, it is easily rejected by the target device. In the case of Defensics, most of the test packets are normal packets (i.e., not malformed packets); thus, instead of yielding unexpected behaviors, it often results in normal communication.

**Other Bluetooth vulnerability detection techniques.** There are several approaches that attempted to detect Bluetooth vulnerabilities without using fuzz testing (*e.g.*, KNOB [5], BIAS [6] and BlueMirror [7]). However, they are inefficient to test various devices because the scope of the target is limited and complicated implementations are required (*e.g.*, they require link key sniffing, reverse engineering, and firmware patching). These tasks are difficult for the user to follow and implement; and also unsuitable for testing various Bluetooth devices.

**General vulnerability detection techniques.** In addition, various approaches attempted to detect general vulnerabilities in a given codebase (*e.g.*, [33]–[35]). Although these techniques can detect Bluetooth vulnerabilities, however, they can only be applied in an environment where the source codes of Bluetooth devices are available.

## VII. Conclusion

Security vulnerabilities in Bluetooth can pose a serious threat in the daily lives of people. In response, we present L2FUZZ, a stateful fuzzer for detecting Bluetooth L2CAP vulnerabilities. By generating malformed packets (for testing purposes) that are less likely to be rejected by the target devices, L2FUZZ can detect potential vulnerabilities in Bluetooth devices more effectively than existing Bluetooth fuzzers. With L2FUZZ, developers can prevent risks in the Bluetooth host stack, which can increase the reliability of Bluetooth devices. The source code of L2FUZZ is available at https://github.com/haramel/L2Fuzz and will be publicly serviced at https://iotcube.net as a part of BFuzz.

## Acknowledgment

We appreciate the anonymous reviewers and our shepherd for their helpful comments. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01697 Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security, No.2019-0-01343 Regional Strategic Industry Convergence Security Core Talent Training Business, and No.IITP-2022-2020-0-01819 ICT Creative Consilience program).

## References

[1] Bluetooth SIG. (2021) Bluetooth market update. [Online]. Available: https://www.bluetooth.com/ko-kr/bluetooth-resources/2021-bmu/

[2] Synopsys. Defensics Fuzz Testing. [Online]. Available: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html

[3] Kim, Seulbae and Woo, Seunghoon and Lee, Heejo and Oh, Hakjoo, "Poster: Iotcube: an automated analysis platform for finding security vulnerabilities," in *Proceedings of the 38th IEEE Symposium on Poster presented at Security and Privacy*, 2017.

[4] Pierre Betouin. (2006, May) [Infratech - release] version 0.6 de Bluetooth Stack Smasher. [Online]. Available: http://www.secuobs.com/news/05022006-bluetooth10.shtml

[5] Antonioli, Daniele and Tippenhauer, Nils Ole and Rasmussen, Kasper B, "The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1047–1061.

[6] Antonioli, Daniele and Tippenhauer, Nils Ole and Rasmussen, Kasper, "BIAS: bluetooth impersonation attacks," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 549–562.

[7] Claverie, Tristan and Esteves, José Lopes, "BlueMirror: Reflections on Bluetooth Pairing and Provisioning Protocols," 2021.

[8] Bluetooth SIG. (2019, December) Bluetooth Core Specification 5.2. [Online]. Available: https://www.bluetooth.com/ko-kr/specifications/bluetooth-core-specification/

[9] Hua, Yang and Zou, Yuexian, "Analysis of the packet transferring in L2CAP layer of Bluetooth v2. x+ EDR," in *2008 International Conference on Information and Automation*. IEEE, 2008, pp. 753–758.

[10] Sharan, Ketan and Sharma, Neel and Sharda, Vangmayee and Arora, Neha, "Air Mouse Using Bluetooth Technolgy," in *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE, 2018, pp. 499–503.

[11] Seri, Ben and Vishnepolsky, Gregory. (2017, November) BlueBorne: The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks. [Online]. Available: https://www.armis.com/research/blueborne/

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

[13] Satam, Shalaka Chittaranjan, "Bluetooth Anomaly Based Intrusion Detection System," Ph.D. dissertation, The University of Arizona, 2017.

[14] Prabadevi, B and Jeyanthi, N, "Distributed Denial of service Attacks and its effects on Cloud Environment-a Survey," in *The 2014 International Symposium on Networks, Computers and Communications*. IEEE, 2014, pp. 1–5.

[15] Dunning, John, "Taming the blue beast: A survey of bluetooth based threats," *IEEE Security & Privacy*, vol. 8, no. 2, pp. 20–27, 2010.

[16] Patel, Chandni M and Borisagar, APVH, "Survey on taxonomy of ddos attacks with impact and mitigation techniques," *International Journal of Engineering Research and Technology*, vol. 1, no. 9, 2012.

[17] Yang, Yi and Jiang, HT and McLaughlin, Kieran and Gao, L and Yuan, YB and Huang, W and Sezer, Sakir, "Cybersecurity test-bed for IEC 61850 based smart substations," in *2015 IEEE Power & Energy Society General Meeting*. IEEE, 2015, pp. 1–5.

[18] Torres, George and Pesavento, Davide and Shi, Junxiao and Benmohamed, Lotfi, "NFDFuzz: A Stateful Structure-Aware Fuzzer for Named Data Networking," in *Proceedings of the 7th ACM Conference on Information-Centric Networking*, 2020, pp. 169–171.

[19] Michal Zalewski. (2017, April) American fuzzy lop. [Online]. Available: https://github.com/google/AFL

[20] Bluetooth SIGBluetooth SIG". (2021) Bluez : Official Linux Bluetooth protocol stack. [Online]. Available: http://www.bluez.org/

[21] Shafranovich, Yakov, "Bluetooth data exchange between android phones without pairing," *arXiv preprint arXiv:1507.00650*, 2015.

[22] Jung, Youngman and Shin, Junbum and Jang, Yeongjin, "BlueMaster: Bypassing and Fixing Bluetooth-based Proximity Authentication."

[23] Carlos Kayembe Nkuba, Seulbae Kim and Sven Dietrich and Heejo Lee, "Riding the IoT Wave With VFuzz: Discovering Security Flaws in Smart Homes," *IEEE Access*, vol. 10, pp. 1775–1789, 2021.

[24] Wikipedia. (2021) Bluetooth Stack. [Online]. Available: https://en.wikipedia.org/wiki/Bluetooth_stack

[25] Muench, Marius and Stijohann, Jan and Kargl, Frank and Francillon, Aurélien and Balzarotti, Davide, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices." in *NDSS*, 2018.

[26] Lamping, Ulf and Warnicke, Ed, "Wireshark user's guide," *Interface*, vol. 4, no. 6, p. 1, 2004.

[27] Lee, Choongin and Bae, Jeonghan and Lee, Heejo, "PRETT: Protocol Reverse Engineering Using Binary Tokens and Network Traces," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2018, pp. 141–155.

[28] Google Android Security team. (2021, January) Debugging Native Android Platform Code. [Online]. Available: https://source.android.com/devices/tech/debug

[29] Bluetooth SIG. (2007, July) Bluetooth Core Specification 2.1 + EDR. [Online]. Available: https://www.bluetooth.com/ko-kr/specifications/specs/cs-core-specification-2-1edr/

[30] Apple. (2021, December) Apple Security Update. https://support.apple.com/HT212975, https://support.apple.com/HT212976, https://support.apple.com/HT212978, and https://support.apple.com/HT212980.

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

[32] Heinze, Dennis and Classen, Jiska and Hollick, Matthias, "ToothPicker: Apple Picking in the iOS Bluetooth Stack," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[33] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[34] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi, "Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1165–1182.

[35] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 860–872.