# BloomFuzz: Unveiling Bluetooth L2CAP Vulnerabilities via State Cluster Fuzzing with Target-Oriented State Machines

Pyeongju Ahn, Yeonseok Jang, Seunghoon Woo(✉), and Heejo Lee(✉)

Korea University, Seoul, South Korea
{pingjuu,yeonseok,seunghoonwoo,heejo}@korea.ac.kr

**Abstract.** Bluetooth technologies are widely utilized across various devices. Despite the advantages, the lack of security in Bluetooth can pose critical threats. Existing approaches that rely solely on Bluetooth specification have failed to bridge the gap between documentation and implemented devices. Therefore, they struggle to (1) precisely generate state machines for target devices and (2) accurately track states during the fuzzing process, resulting in low fuzzing efficiency. In this paper, we propose BloomFuzz, a stateful fuzzer to discover vulnerabilities in Bluetooth Logical Link Control and Adaptation Protocol (L2CAP) layer. Utilizing the concept of the state cluster, which is a set of one or more states with similar attributes, BloomFuzz can generate a target-oriented state machine by pruning unimplemented states (missing states) and addressing states that are implemented but not introduced in the specification (hidden states). Furthermore, BloomFuzz enhances fuzzing efficiency by generating valid test packets for each cluster via cluster-based state machine tracking. When we applied BloomFuzz to real-world Bluetooth devices, we observed that BloomFuzz outperformed existing L2CAP fuzzers by (1) discovering 56 potential vulnerabilities (more than twice compared to existing fuzzers), (2) precisely generating a target-oriented state machine, (3) significantly reducing the probability of test packets being rejected (from 76% to 23%), and (4) producing nine times more valid malformed test packets. Our proposed approach can contribute to preventing threats within L2CAP, thereby rendering a secure Bluetooth environment.

**Keywords:** Bluetooth Security · L2CAP Security · Stateful Fuzzing

## 1 Introduction

Bluetooth is an integral aspect of our daily lives and facilitates seamless wireless communication between various devices. Owing to the proliferation of Bluetooth-enabled devices, addressing security issues has become increasingly important. This is because the current landscape underscores the urgent requirement for robust Bluetooth security measures, considering the potential risks posed by unauthorized access, data breaches, and malicious attacks [2,4,21,27].

To secure Bluetooth devices, one effective approach is leveraging fuzz testing (*i.e.*, fuzzing); because Bluetooth implementation operates based on *states*, stateful fuzzing (see Sect. 2.1) has been widely utilized. For this purpose, existing approaches first generate the Bluetooth state machine of the target device (*e.g.*, by examining the Bluetooth specification) and then perform fuzzing.

Unfortunately, performing effective fuzzing on Bluetooth devices is becoming challenging mainly owing to the following two aspects.

1) **Gap between specification and implementation.** Bluetooth devices do not strictly adhere to the states and transitions specified in the specification. In practice, the states and transitions specified in the specification may not be implemented (*i.e.*, missing states), or the states and transitions that are not mentioned in the specification may be implemented (*i.e.*, hidden states).
2) **Difficulty in state tracking.** To conduct efficient stateful fuzzing, it is essential to track the current state of the target device. However, when many packets are sent to the target device, state transitions occur nondeterministically, thus hindering the accuracy of state tracking.

Existing Bluetooth stateful fuzzing approaches (*e.g.*, [5,12,13]) failed to fully address the aforementioned challenges. For example, L2Fuzz [13] assumes that the states and transitions specified in the specification are implemented on the target device identically, compromising the efficiency of fuzzing (*e.g.*, sending test packets to unimplemented states). Moreover, existing approaches do not fully consider the current state of the target device, leading to an increased possibility of test packets being rejected (details are explained in Sect. 2.2).

To overcome these shortcomings, we propose BloomFuzz, a stateful fuzzer designed for Bluetooth host stacks. BloomFuzz targets the Logical Link Control and Adaptation Protocol (L2CAP) layer, which is the lowest layer in all Bluetooth devices and is thus a particularly sensitive area for security.

The core idea of BloomFuzz is to define and leverage a state *cluster* instead of individually considering each state. We define a cluster as a set of one or more states that share similar attributes, such as valid L2CAP commands (see Sect. 3.1). By using clusters, BloomFuzz addresses the two main challenges mentioned above and achieves effective Bluetooth L2CAP fuzzing.

To precisely generate a state machine for the target device, BloomFuzz initially considers the one presented in the Bluetooth specification. Subsequently, it (1) prunes missing states and (2) addresses hidden states. BloomFuzz identifies whether the specified states have been implemented on the target device by sending valid signaling packets to each state; if BloomFuzz cannot receive the corresponding response, then it considers the state as not implemented and removes it from the state machine. Thereafter, to address the hidden states, BloomFuzz utilizes a method called *packet recording.* After clustering the states in the state machine, we capture the normal communication packets between the target device and BloomFuzz to identify the structure of the internal state machines of each cluster. Consequently, by examining the entire set of clusters, BloomFuzz generates a target-oriented state machine for the target device (see Sect. 3.1).

Thereafter, BloomFuzz traverses a target-oriented state machine to perform fuzzing on the target device. Here, BloomFuzz overcomes the challenge of state tracking by traversing the state machine based on *clusters*. By designing clusters to ensure the occurrence of nondeterministic state transitions within a single cluster, BloomFuzz can identify the cluster where the target device resides, irrespective of the internal state structures within the cluster. Moreover, when mutating packets for testing, BloomFuzz utilizes AFL operators [29] and considers edge cases to generate more effective test packets (see Sect. 3.2). By transmitting the test packets based on the cluster in which the target device is located, BloomFuzz can perform efficient L2CAP fuzzing (see Sect. 3.3).

To evaluate BloomFuzz, we apply it to seven real-world devices utilizing Bluetooth (see Table 2). The result shows that BloomFuzz uncovers 56 potential vulnerabilities in six devices (see Sect. 4.2). Additionally, we compared BloomFuzz with the following three L2CAP stateful fuzzers: BSS [5], BFuzz [12], and L2Fuzz [13]. We observed that BloomFuzz outperformed existing approaches by (1) uncovering twice as many crashes, (2) generating the implemented state machine on the target device more precisely, (3) reducing the probability of test packets being rejected by the target device (from 76% to 23%), and (4) generating nine times more malformed testing packets (see Sect. 4).

This paper makes the following three main contributions.

- We, for the first time, highlighted the discrepancies in the L2CAP state machine between the Bluetooth specification and the implemented devices, and proposed a method to effectively address missing and hidden states based on the concept of state clusters.
- We propose a method to enhance fuzzing efficiency by precisely determining the current state of the target device within the state machine. The core technology involves tracking state machines based on clusters, resulting in a reduced probability of a test packet being rejected (from 76% to 23%).
- When we applied BloomFuzz to seven real-world Bluetooth devices, it detected 56 crashes, thereby outperforming existing L2CAP fuzzers. We have reported reproducible vulnerabilities to the respective vendors, and some of them were confirmed and will be patched. The source code BloomFuzz is available at https://github.com/pingjuu/BLOOMFUZZ/.

## 2 Motivation

### 2.1 Background

**Bluetooth L2CAP.** The Bluetooth protocol stack can be classified into the host and controller stacks. The L2CAP layer is situated at the bottom of the host stack. The L2CAP facilitates the transmission and reception of upper-layer data packets for higher-level protocols and applications. In Bluetooth communication, the L2CAP performs functions such as data transmission and channel control, including channel flow control, retransmission, and connection request.

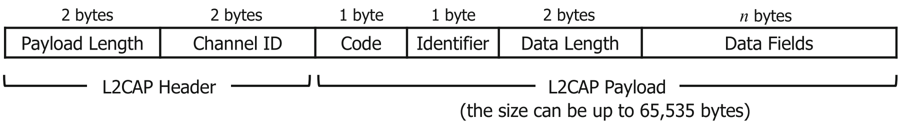


**Fig. 1.** L2CAP packet format.

**L2CAP Packet.** The L2CAP is packet-based but reflects a communication model based on channels. An L2CAP packet comprises a header and a payload (see Fig. 1). The L2CAP header indicates the type of L2CAP packet, which is categorized into (1) signaling and (2) data packets based on the `Channel ID`. Signaling packets are used to transmit commands for managing Bluetooth communications and comprise four fields: `Code`, `Identifier`, `Data Length`, and `Data Fields`. The `Code` and `Identifier` indicate the L2CAP command code and packet ID, respectively. The `Data Length` represents the length of a specific field, and the `Data Fields` vary depending on the L2CAP command.

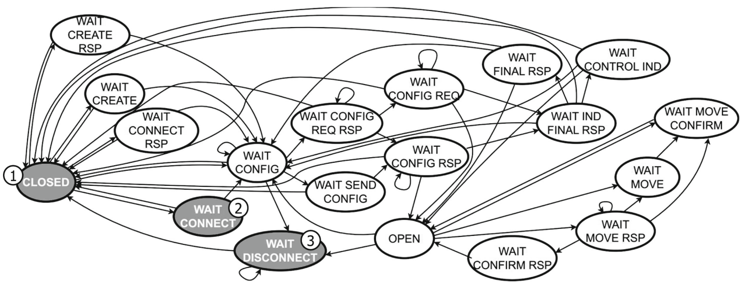
**Stateful Fuzzing.** The communication process of the L2CAP can be represented by *states*. Stateful fuzzing refers to a fuzzing technique designed to consider these states to detect potential threats. When a *peripheral* (*i.e.*, slave) receives an event from a *central* (*i.e.*, master), it performs an action and transitions from the current to the next state. Most events and actions correspond to L2CAP commands, with events triggered by the signaling packets. Because invalid events for a certain state are disregarded, stateful fuzzing in L2CAP enables efficient fuzzing by considering state transitions in the L2CAP.

## 2.2 Technical Challenges

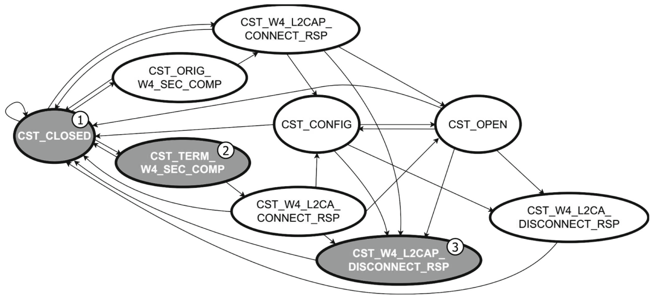
Stateful fuzzing is an effective method for detecting threats in the L2CAP. However, this task is not straightforward owing to the following two aspects.

**Difficulty in Precisely Generating a State Machine.** The first step in stateful fuzzing in L2CAP is to precisely construct a state machine for the target device. In this regard, existing approaches primarily (1) utilize the Bluetooth specification [13] or (2) dynamically capture communication [7]. However, precisely constructing state machines is becoming increasingly challenging.

The main issue is that real-world Bluetooth devices typically employ a state machine that is different from that mentioned in the specification. This is because developers modify the state machine mentioned in the specification primarily to ensure implementation efficiency. For example, Fig. 2 illustrates two different state machines: the state machine of the Bluetooth v5.2 specification (Fig. 2a) and the state machine of BlueDroid v12.1.0.r19 [1] (Fig. 2b), which was developed based on the Bluetooth v5.2 specification.



(a) State machine of Bluetooth 5.2 specification.



(b) State machine of BlueDroid v12.1.0.r19.

**Fig. 2.** Comparison of specification- and implementation-based state machines.

To clarify, we delineated the relationship between the states in the specification and those in the implementation. A state transition in the L2CAP (from the source state $v_i$ to the destination state $v_j$) is represented by $(v_i \xrightarrow{e,a} v_j)$, where $v_i$, $e$, $a$, and $v_j$ denote the source state, event, action, and destination state, respectively. We define three types of states.

- **Normal state.** This refers to a state defined in the specification and implemented in the Bluetooth devices (*e.g.*, ①, ②, and ③ in Fig. 2).
- **Missing state.** This refers to a state defined in the specification but not implemented in the Bluetooth device.
- **Hidden state.** This refers to a state not defined in the specification but implemented in the Bluetooth device, which can be accessed through transitions not defined in the specification. Consider the transition defined in the specification: $(v_i \xrightarrow{e,a} v_j)$. Even when an event and action other than $e$ and $a$ allow access to $v_j$ from $v_i$, $v_j$ is defined as the hidden state.

Existing approaches based on the specification [13,19] include the missing state in the state machine, thereby compromising the efficiency of fuzzing. In addition, they fail to consider hidden states, thus resulting in reduced fuzzing coverage. Meanwhile, existing approaches that aim to dynamically capture communication from the target device to build the state machine [7] fail to correctly address the hidden state. This is because they explore the states based on the

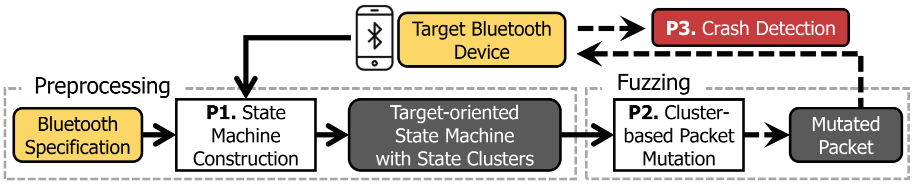

Target Bluetooth Device
P3. Crash Detection
Preprocessing
Bluetooth Specification
P1. State Machine Construction
Target-oriented State Machine with State Clusters
Fuzzing
P2. Cluster-based Packet Mutation
Mutated Packet

**Fig. 3.** High-level overview of BloomFuzz.

transitions in the specification, thereby hindering the effective generation of the state machine. Therefore, a method is required to precisely generate the state machine for the target device by addressing the missing and hidden states.

**Difficulty in Correctly Tracking States.** To achieve effective stateful fuzzing, it is essential to identify the current state in which the target device resides. This directly translates to the ability to generate valid test packets that are not rejected by the target device. To track the state, starting from the initial state of the state machine, the fuzzer sends the desired events to induce state transitions.

However, in environments where numerous packets are transmitted in a short period (*e.g.*, fuzzing), predicting state transitions becomes challenging, primarily owing to the difficulty of matching sent and received packets. Responses to earlier sent packets often arrive later than responses to subsequently sent packets. In addition, the target device may not provide a response, or non-deterministic transitions may occur (*e.g.*, zero-, single-, and multi-state transitions). Hence, identifying the state in which the target device resides is not an easy task.

## 3 Design of BloomFuzz

In this section, we describe the design of BloomFuzz, which is a stateful fuzzer that can effectively detect vulnerabilities in the Bluetooth L2CAP.

**Overview.** BloomFuzz detects crashes (*i.e.*, potential vulnerabilities) in the L2CAP via two main stages: *preprocessing* and *fuzzing*. It is organized into the following three phases: *target-oriented state machine construction (P1)*, *cluster-based packet mutation (P2)*, and *crash detection (P3)*. Figure 3 depicts the high-level workflow of BloomFuzz. The distinguishing concept of BloomFuzz is its consideration of state clusters, a set of states characterized by similar attributes.

In P1, BloomFuzz generates a target-oriented state machine on the target device (see Sect. 3.1). BloomFuzz initially generates a state machine from the specification. Subsequently, it verifies all states in the specification-based state machine to confirm its implementation on the target device and eliminates

**Table 1.** 19 states in the specification are classified by (1) L2CAP commands and (2) the role of a target device, which became 12 state clusters.

| Cluster IDX | States | Commands | *Role** |
|---|---|---|---|
| 1 | CLOSED | All commands | C/P |
| 2 | WAIT_CONNECT | L2CAP_Connect_Req/Rsp | P |
| 3 | WAIT_CONNECT_RSP | L2CAP_Connect_Req/Rsp | C |
| 4 | WAIT_CREATE | L2CAP_Create_Channel_Req/Rsp | P |
| 5 | WAIT_CREATE_RSP | L2CAP_Create_Channel_Req/Rsp | C |
| 6 | WAIT_CONFIG | L2CAP_Configuration_Req/Rsp | C/P |
| 7 | WAIT_SEND_CONFIG, WAIT_CONFIG_RSP, †WAIT_IND_FINAL_RSP, †WAIT_FINAL_RSP, †WAIT_CONTROL_IND | L2CAP_Configuration_Req/Rsp | P |
| 8 | WAIT_CONFIG_REQ, WAIT_CONFIG_REQ_RSP, †WAIT_IND_FINAL_RSP, †WAIT_FINAL_RSP, †WAIT_CONTROL_IND | L2CAP_Configuration_Req/Rsp | C |
| 9 | OPEN | All commands | C/P |
| 10 | WAIT_MOVE, WAIT_MOVE_CONFIRM | L2CAP_Move_Channel_Req/Rsp, L2CAP_Move_Channel_Confirmation_Req/Rsp | P |
| 11 | WAIT_CONFIRM_RSP, WAIT_MOVE_RSP | L2CAP_Move_Channel_Req/Rsp, L2CAP_Move_Channel_Confirmation_Req/Rsp | C |
| 12 | WAIT_DISCONNECT | L2CAP_Disconnection_Req/Rsp | C/P |

*The role of the target device (Central or Peripheral); †States belonging to Clusters #7 and #8.

missing states. BloomFuzz addresses hidden states by incorporating a packet recording method, thus resulting in a target-oriented state machine.

In P2, BloomFuzz traverses the state machine of the target device and generates a test packet for each state (see Sect. 3.2). Here, BloomFuzz identifies the cluster in which the target device is currently located and consequently generates well-crafted test packets with a low probability of being rejected by the target device. Additionally, it enhances the efficiency of packet mutation by (1) using modified AFL operators and (2) considering edge cases.

Finally, in P3, the generated test packets are used to scrutinize the target device and verify its safety (see Sect. 3.3).

## 3.1 State Machine Construction (P1)

Given a target device and the Bluetooth specification, BloomFuzz first generates a target-oriented state machine. Before providing detailed explanations, we introduce the core concept of BloomFuzz, *i.e.*, the *cluster* of states.

**State Cluster.** A cluster is a set of one or more states with similar attributes. When distinguishing clusters, BloomFuzz utilizes two features: (1) valid L2CAP commands and (2) the role of the target device. Table 1 summarizes the state clusters defined in this paper. The process of creating clusters was performed manually. While automation is possible, accuracy may decrease because unapproved commands may be delivered in states. By adopting the concept of a cluster, BloomFuzz can achieve two key effects: (1) the ability to correspond to hidden states and (2) the efficient generation of test packets with a low probability of rejection during packet mutation for fuzzing. Detailed explanations are provided in P1 and P2.

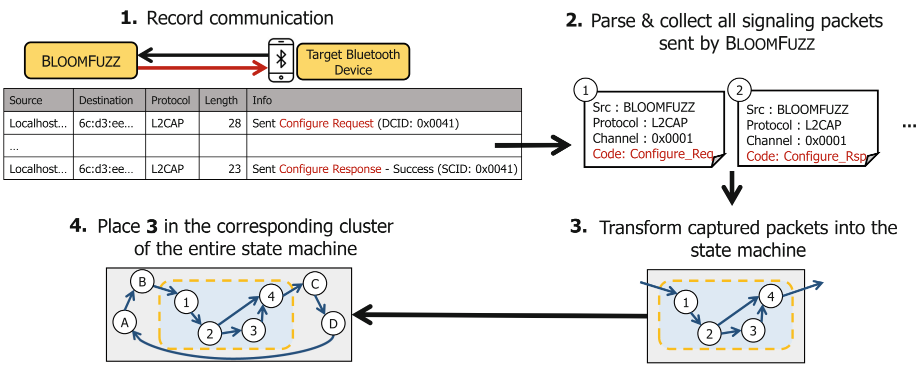
**Generating a Specification-Based State Machine.** Bluetooth devices are developed based on specifications. Therefore, we first construct a state machine based on the Bluetooth specification. This is accomplished through a manual analysis of the specification while considering all the states and transitions mentioned herein (see Fig. 2a). Because the specification clearly describes the states and transitions, the abovementioned task is not difficult [13].

**Pruning Missing States.** Merely utilizing the state machine specified in the specification results in a lower fuzzing efficiency (see Sect. 2.2). To generate a target-oriented state machine, BloomFuzz first prunes the missing states. This is accomplished by traversing the specification-based state machine and verifying whether the states specified in the specification are implemented in the target device. The detailed process of pruning missing states is as follows.

**1)** We first manually extract several paths that encompass all states in the state machine. BloomFuzz then traverses the specification-based state machine by following the extracted paths. Here, BloomFuzz first assumes that the state machine of the target device is consistent with that specified.
**2)** In each state, BloomFuzz sends a corresponding signaling packet to trigger an event that induces a state transition. An appropriate signaling packet for each state is specified in the specification.
**3)** If BloomFuzz receives the correct response (*i.e.*, action) from the sent signaling packet, then it determines that a corresponding state exists. Otherwise, BloomFuzz concludes that the state is not implemented on the target device (*i.e.*, the missing state) and removes it from the state machine.

Notably, tracking states in stateful fuzzing is challenging especially when multiple packets are sent to the target device in a short period (see Sect. 2.2). However, this issue does not affect here because the current stage is pre-fuzzing, thus BloomFuzz can track missing states by sending a single packet.

**Addressing Hidden States.** Next, BloomFuzz addresses hidden states by identifying internal state structures (including hidden states) for each cluster



| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| Localhost... | 6c:d3:ee... | L2CAP | 28 | Sent Configure Request (DCID: 0x0041) |
| ... | | | | |
| Localhost... | 6c:d3:ee... | L2CAP | 23 | Sent Configure Response - Success (SCID: 0x0041) |

**Fig. 4.** Internal state examination of clusters via packet recording.

without relying on the specification. Subsequently, the state machines of clusters are aggregated to generate a comprehensive target-oriented state machine.

To this end, BloomFuzz uses a technique called *packet recording*. Using a packet capturing tool (*e.g.*, `Wireshark`), we first captured the packets generated during normal communication. After examining these packets, we ascertained the structure of the states within the multistate cluster: (1) we parsed and obtained all signaling packets sent by BloomFuzz, (2) transformed the captured packets into the state machine, and (3) placed this state machine in the corresponding cluster of the entire state machine. To include potential hidden states, we utilized this approach regardless of the number of states within the cluster.

By utilizing this approach, BloomFuzz can track the states through which normal communication packets pass, irrespective of the specification. Thus, it is feasible to depict the states of the cluster while considering only the implemented states. This allows BloomFuzz to address hidden states within the cluster.

For example, Fig. 4 illustrates the process of packet recording. As shown, BloomFuzz sent packets using the `L2CAP_Configuration_Req/Rsp` codes. Because the role of the target device during the packet recording was peripheral, among the clusters mapped to the corresponding command (*i.e.*, Cluster #7 and #8), BloomFuzz can identify that this corresponds to Cluster #7 (see Table 1). Subsequently, by capturing the packets, BloomFuzz identifies the state machine of Cluster #7 and then integrates it into the entire state machine.

**Output of P1.** For the output of P1, BloomFuzz generates a target-oriented state machine for the target device. Unlike conventional specification-based state machines, target-oriented state machines do not include missing states as they would compromise the efficiency of fuzzing. Also, they incorporate hidden states within clusters, thereby they can be efficiently used in stateful fuzzing.

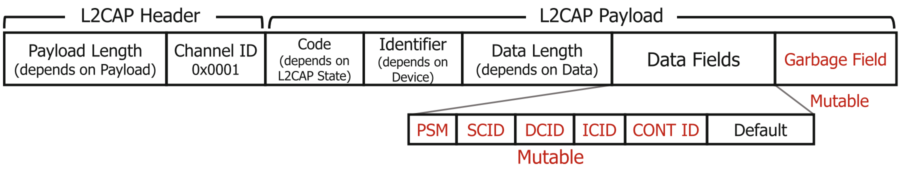

L2CAP Header
L2CAP Payload
Payload Length (depends on Payload)
Channel ID 0x0001
Code (depends on L2CAP State)
Identifier (depends on Device)
Data Length (depends on Data)
Data Fields
Garbage Field
Mutable
PSM
SCID
DCID
ICID
CONT ID
Default
Mutable

**Fig. 5.** Mutable field selection for packet mutation.

## 3.2 Cluster-Based Packet Mutation (P2)

Subsequently, based on a target-oriented state machine, BloomFuzz generates packets to test the target device.

**Packet Generation for Traversing the State Machine.** In P2, BloomFuzz traverses a target-oriented state machine and generates a valid test packet for each state. The first step is to generate normal packets that can transition to a state. This task is straightforward provided that we can determine the current state in which the target device resides. However, as mentioned in Sect. 2.2, discerning the current state of stateful fuzzing is difficult.

Hence, we consider transitions at the *cluster* level. Instead of considering the transitions for each state, BloomFuzz detects crashes (*i.e.*, potential vulnerabilities) in the target device by generating valid test packets and performing fuzzing within each cluster. Note that multiple states within a single cluster can recognize the same L2CAP command (see Table 1), resulting in a state transition accordingly. Thus, even if nondeterministic transitions occur for a single signaling packet, the cluster to which the target device belongs can be identified.

Based on the clusters in Table 1 and the corresponding L2CAP commands, BloomFuzz generates normal packets that can transition between clusters. As a result, BloomFuzz is prepared to traverse the target-oriented state machine.

**Field Classification.** While traversing each cluster, BloomFuzz generates packets to test the target device. If a packet with an invalid L2CAP command is generated for the current cluster, then it is disregarded by the target device. Therefore, BloomFuzz first generates valid packets for each cluster and performs mutations only in fields that do not affect the packet validity.

L2CAP packets are composed of fields, as illustrated in Fig. 5. The `Channel ID` must utilize `0x0001` to manage channels. If sensitive fields, such as `Length`-related fields or the `L2CAP command` field, are randomly altered, then the possibility of packet rejection by the target device increases significantly.

Hence, we focus on the `Data fields` of the L2CAP packet. Specifically, within the `Data fields`, certain portions can impact the channels and ports. For example, `PSM` is used for port setting, whereas `SCID` and `DCID` are used for channel configuration (see Fig. 5). BloomFuzz generates packets to test the target device by focusing on mutating particular areas (called *mutable* fields).

In addition, we include the garbage value at the end of the signaling packet. This value is selected randomly within a range that does not exceed the Maximum Transmission Unit (MTU) of the packet. BloomFuzz also mutates the garbage field to facilitate the discovery of crashes in the target device.

**Packet Mutation.** Unlike existing approaches that randomly mutated packets (*e.g.*, [12,13]), we applied AFL operators [29] to mutable fields to achieve more diverse and efficient packet mutation. The AFL operator defines various changes with effective mutations, including random mutations.

However, using the predefined 11 AFL operators (*i.e.*, `bitflip`, `byteflip`, `arithmetic inc/dec`, `interesting values`, `user extras`, `auto extras`, `random bytes`, `delete bytes`, `insert bytes`, `overwrite bytes`, and `cross over`) in a black-box Bluetooth fuzzer presents challenges. We need to modify some of them to adapt to our specific target scenarios.

First, `interesting values` that required manual operations (i.e., the need to manually select the area of interest) for application to BloomFuzz, as well as `user extras` and `auto extras`, were excluded. Next, operators that can generate invalid L2CAP packets by adjusting the length of bytes are adjusted as follows.

- The `Insert byte` was redefined as an operator that selects the length of the garbage field to be used within the range of the MTU.
- The `Delete byte` was replaced with zero padding because randomly shortening the length of the field may increase the probability of packet rejection.
- The `Overwrite byte` and `Random byte` randomly set the values based on the size of the field to be mutated.

For more diverse packet mutations, BloomFuzz randomly selects and utilizes two different AFL operators simultaneously for packet mutation.

**Addressing Edge Cases.** Moreover, an edge operator was introduced to trigger underflow and overflow. It sets one of the values among the `minimum value`, `maximum value`, `minimum value`±1, and `maximum value`±1 of the mutable field. Using the modified AFL and edge operators, BloomFuzz modifies the mutable fields to generate various test packets.

**Output of P2.** For the output of P2, BloomFuzz generates mutated packets to test the target device. These packets are cluster-based, and their probability of being rejected by the target device is lower than that of randomly generated test packets. Additionally, BloomFuzz enhances the efficiency of packet mutations by leveraging the garbage field, AFL, and edge operators.

### 3.3 Crash Detection (P3)

Finally, BloomFuzz detects crashes by sending mutated packets to the target device. During the fuzzing process, if any of the following five errors

**Table 2.** The list of target devices used in our experiments.

| ID | Type | Vendor | Name | Model | Chip | *OS/FW*$^*$ | *BT*$^\dagger$ | *Y*$^\ddagger$ |
|---|---|---|---|---|---|---|---|---|
| D1 | Laptop | LG | Gram | 15ZD990-VX50K | Intel wireless BT | Windows 10 | 5.0 | 2019 |
| D2 | Laptop | LG | Gram | 15ZD970-GX55K | Intel wireless BT | Ubuntu 18.04.4 | 5.0 | 2017 |
| D3 | Phone | Google | Pixel7 | GVU6C | Google Tensor G2 | Android 14 | 5.2 | 2022 |
| D4 | Phone | Google | Pixel3 | GA00464 | Snapdragon 845 | Android 12 | 5.0 | 2018 |
| D5 | Tablet | Samsung | Galaxy Tab S6 Lite | SM-P610 | Snapdragon 855 | Android 12 | 5.0 | 2019 |
| D6 | Earphone | Samsung | Galaxy Buds+ | SM-R175 | BCM43015 | R175XXU0AUK1 | 5.0 | 2020 |
| D7 | Earphone | Xiaomi | Redmi Buds 3 Pro | TWSEJ01ZM | QCC3040 | 1.0.9.9 | 5.2 | 2021 |

*OS/FW: Operating system or firmware; $^\dagger$BT: Bluetooth version; $^\ddagger$Y: Released year.

occur – `ConnectionResetError`, `ConnectionRefusedError`, `Connection-AbortedError`, `TimeoutError`, and `OS Error` – BloomFuzz concludes that the packet has unintentionally affected the target device. Upon receiving one of these error messages, BloomFuzz requests an L2CAP echo from the target device using `l2ping` to verify its continued functionality. If the target device fails to respond to the echo request, BloomFuzz determines that a crash has occurred. Subsequently, BloomFuzz logs the sent packet, current fuzzer state, and error type.

## 4 Evaluation

In this section, we evaluate BloomFuzz. Section 4.1 introduces the experimental setup. Section 4.2 presents an analysis of the effectiveness of BloomFuzz in discovering crashes from real-world Bluetooth devices. Section 4.3 presents the effectiveness of BloomFuzz in state machine generation. Section 4.4 investigates how BloomFuzz performs state tracking and packet mutation effectively.

### 4.1 Experimental Setup

**Implementation.** BloomFuzz comprises two modules: a preprocessor and fuzzer. Literally, these modules perform preprocessing (*e.g.*, target-oriented state machine generation) and fuzzing. BloomFuzz is written in approximately 3,000 lines of Python code, excluding external libraries (*e.g.*, `Scapy`).

**Experimental Environment.** We executed BloomFuzz on an Ubuntu 20.04 LTS machine equipped with 16 GB memory, Intel Core i5-7500 CPU @ 3.30 GHz, and 64 GB SSD. To enable communication with the target device, a Cambridge Silicon Radio Bluetooth Classic dongle was connected to a Linux machine.

**Target Devices.** For the experiments, we targeted widely used Bluetooth devices in the real world and selected devices with various Bluetooth versions manufactured by different companies (*i.e.*, LG, Google, Samsung, and Xiaomi). Table 2 summarizes the real-world Bluetooth devices used in our experiments.

**Comparison Targets.** To evaluate the efficiency of BloomFuzz, we compared its results with those of existing approaches that aim to fuzz the Bluetooth L2CAP. Specifically, we selected the following three approaches: BSS [5], BFuzz [12], and L2Fuzz [13]. Existing approaches that do not target the L2CAP layer or focus primarily on the controller stack were excluded.

**Evaluation Metrics.** Because most Bluetooth fuzzing is conducted using a black-box approach, acquiring the source code or internal log information of the target devices is challenging. Hence, we define the following four main evaluation criteria to investigate the effectiveness of BloomFuzz.

1) **Crash detection efficiency.** This indicates the number of crashes detected by each fuzzer when sending two million test packets to the target device.
2) **State machine generation accuracy.** This indicates the accuracy with which the fuzzer identifies the state machine of the target device. We used the following two specific metrics: *accuracy of pruning missing states ($A_m$)* and *accuracy of addressing implemented states ($A_i$)*.

$$A_m = \frac{\#\text{Identified Missing States}}{\#\text{Total Missing States}}$$

$$A_i = \frac{\#\text{Addressed Hidden States} + \#\text{Identified Normal States}}{\#\text{Total Hidden and Normal States}}$$

3) **State Tracking Efficiency.** This indicates the efficiency with which the fuzzer tracks the state machine of the target device during fuzzing. If the states are tracked effectively, then the number of test packets rejected by the target device decreases. Subsequently, we consider the *acceptance ratio of test packets ($A_t$).*

$$A_t = 1 - \left(\frac{\#\text{Rejected Packets}}{\#\text{Total Sent Packets}}\right)$$

4) **Mutation efficiency.** This indicates the efficiency of a fuzzer in performing packet mutations. We define it as the probability that a test packet sent when accepted by the target device is malformed. *Mutation efficiency ($M_e$)* is evaluated as follows.

$$M_e = \left(\frac{\#\text{Malformed Packets}}{\#\text{Total Sent Packets}}\right) \times A_t$$

### 4.2 Experiment on Crash Detection

**Methodology.** We applied BloomFuzz and three selected comparison targets (BSS, BFuzz, and L2Fuzz) to the seven target devices to assess the effectiveness of the fuzzers in crash detection. Here, we evaluate the effectiveness of the four fuzzers by examining the number of discovered crashes (*i.e.*, potential vulnerabilities) when sending two million test packets to the target device.

**Table 3.** Crash detection results of each fuzzer. BloomFuzz was able to discover the highest number of crashes, except in the cases of D5 and D7.

| Target | #Detected crashes in each fuzzer | | | |
|---|---|---|---|---|
| | BloomFuzz | L2Fuzz | BFuzz | BSS |
| D1 | **17** | 0 | 3 | 0 |
| D2 | **6** | 0 | 0 | 0 |
| D3 | **8** | 0 | 8 | 0 |
| D4 | **1** | 0 | 0 | 0 |
| D5 | 0 | 0 | **12** | 0 |
| D6 | **14** | 4 | 0 | 0 |
| D7 | 10 | **26** | 0 | 0 |
| Total | **56** | 30 | 23 | 0 |

**Results.** Table 3 summarizes the experimental results. Notably, we confirmed that BloomFuzz discovered the highest number of crashes across five (D1 - D4, and D6) out of seven devices.

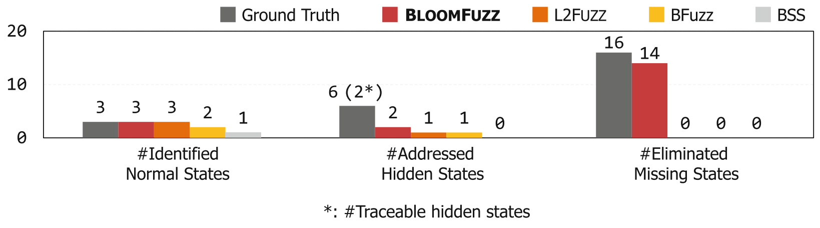
In the case of D5, only BFuzz could detect crashes. We observed that most of the packets leading to crashes had values assigned to the `Code field` of the L2CAP packet that were not defined in the specification. Thus, BloomFuzz, which did not consider this field mutable, failed to detect crashes. However, BFuzz has a high probability of generating test packets that are rejected by the target device owing to randomly mutating all fields (see Sect. 4.4). In the case of D7, L2Fuzz was able to detect the highest number of crashes. When L2Fuzz discovered a crash, it tended to focus intensively on that state, generating many similar crash-inducing packets. Consequently, it could trigger many crashes in D7. Last, BSS did not trigger any crashes across all devices.

We reported two vulnerabilities that were reproducible among the detected crashes to the respective vendors, *i.e.*, Galaxy Buds+ (D6) of Samsung and Redmi Buds 3 Pro (D7) of Xiaomi (see Appendix A). Xiaomi confirmed that the crash is indeed a vulnerability. Samsung confirmed and provided the opinion that vulnerabilities can be prevented by configuring internal device options.[1]

### 4.3 Effectiveness of State Machine Generation

**Methodology.** We evaluated how precisely fuzzers generate state machines by comparing the state machine implemented on the target device with the one generated by each fuzzer. Here, we considered the Pixel 3 (D4 in Table 2) running Android 12 to determine the ground truth of the implemented state machine. This is because, unlike other operating systems or firmware, the source code of Android 12 is publicly available, hence we can examine the code and identify the

[1] Since the vulnerabilities have not been patched yet, detailed explanations are omitted. We plan to introduce details after the completion of the patching process.



| Fuzzers | $A_m$ | $A_i$ | $A_i$ (traceable) |
|---|---|---|---|
| BloomFuzz | **14/16 (88%)** | **5/9 (56%)** | **5/5 (100%)** |
| L2Fuzz | 0/16 (0%) | 4/9 (44%) | 4/5 (80%) |
| BFuzz | 0/16 (0%) | 3/9 (33%) | 3/5 (60%) |
| BSS | 0/16 (0%) | 1/9 (11%) | 1/5 (20%) |

**Fig. 6.** Illustration of state machine generation effectiveness.

implemented state machine. We verified that the state machine of D4 comprises nine states (three normal and six hidden states), with 16 missing states.

**Results.** Figure 6 shows the experimental results. First, only BloomFuzz and L2Fuzz could identify all the implemented normal states. In addition, BloomFuzz successfully addressed two hidden states. Among the total six hidden states discovered on the target device, only these two were traceable: three hidden states were accessible only when the target device was a central, and the remaining one could only be entered through the controller stack.

Unlike existing fuzzers, BloomFuzz eliminated the detected 14 missing states (88%). In fact, BloomFuzz recognized all missing states and excluded them from the generation of the target-oriented state machine, but during the fuzzing process, BloomFuzz attempted to access these two missing states unintentionally (*e.g.*, internal operation of the Controller stack). Therefore, we determined that the two missing states were not completely removed by BloomFuzz. Nonetheless, BloomFuzz can generate a target-oriented state machine very close to the implemented state machine by using the concept of clusters.

## 4.4 Efficiency of State Tracking and Packet Mutation

**Methodology.** Finally, we evaluated the fuzzers used in the experiments to assess how efficiently they (1) track the state machine of the target device and (2) perform packet mutation. Each fuzzer sent two million packets to the target device; because the count verified in `Wireshark` differed from the values provided by the fuzzers, the number of transmitted packets observed in the results could be either less or more than two million. Note that the completeness of the state machine directly affects the evaluation of state tracking and packet mutation effectiveness. Hence, to illustrate the correlation between these factors, we focus on the same D4 device that was previously employed in Sect. 4.3.

**Table 4.** Measurement results of packet acceptance ratio and mutation efficiency.

| Fuzzers | #Total Sent Pkts | #Rejected Pkts | #Malformed Pkts | $A_t$ | $M_e$ |
|---|---|---|---|---|---|
| BloomFuzz | 1,459,515 | **341,858** | **923,468** | **77%** | **49%** |
| L2Fuzz | 926,768 | 511,070 | 585,616 | 45% | 28% |
| BFuzz | **2,002,862** | 1,457,943 | 99,745 | 27% | 1% |
| BSS | 1,202,518 | 908,986 | 389,763 | 24% | 8% |

**Results.** Table 4 summarizes the measurement results. First, both BSS and BFuzz exhibited low packet acceptance ratios (*i.e.*, less than 30%) as they failed to (1) address normal and hidden states (see Sect. 4.3) and (2) track the current state of the target device. Although L2Fuzz addressed this to some extent, it hardly achieved a packet acceptance ratio of 45%. In contrast, BloomFuzz, which addresses hidden states using clusters, demonstrated a significantly higher ratio of 77%, thereby outperforming other fuzzers.

Considering mutation efficiency, BFuzz and BSS based on simple randomness showed very low mutation efficiency (less than 10%). L2Fuzz, by focusing only on mutable L2CAP packet fields, achieved a moderate level of efficiency (28%). In contrast, BloomFuzz not only focused on mutable fields but also (1) utilized AFL operators, (2) considered edge cases, and even (3) accounted for garbage fields. As a result, it significantly increased the efficiency of packet mutation compared to existing L2CAP fuzzers, by showing the mutation efficiency of 49%.

## 5 Discussion

**Limitations.** BloomFuzz makes several assumptions that limit its application. First, when removing missing states, if unintended transitions are included, missing states may not be eliminated. We plan to identify and remove transitions that were not intended by the fuzzer. Next, our target-oriented state machine is constructed based on the specification. Unless updated to reflect newly added L2CAP commands, BloomFuzz cannot cover them. This can be addressed by defining signaling packets containing new L2CAP commands as new state transitions and incorporating them into the state machine.

Some limitations commonly exist with Bluetooth fuzzers. Most Bluetooth fuzzers can only cover states that can be reached when the target device operates as a peripheral. We are considering methods to change the role of the target device during fuzzing. Finally, while there are reported crashes that have been reproduced, not every detected crash may always be reproducible. We will strive for the root cause analysis and reproduction of all discovered crashes.

**Applicability of BloomFuzz.** BloomFuzz was implemented based on Bluetooth Specification v5.2, but can be used regardless of the target Bluetooth version, because Bluetooth v5.2 encompasses all states present in each version. Furthermore, the idea of BloomFuzz's state machine generation can be applied to protocols defining state machines in the specification. Thus, we anticipate that enhanced stateful fuzzing will be achievable through the generation of a more effective state machine in various protocols.

## 6 Related Works

**Bluetooth Fuzzing.** Several approaches have been devised to identify Bluetooth vulnerabilities through fuzzing. Defensics [20] aims to uncover Bluetooth vulnerabilities but requires a pairing process. BSS [5] and BFuzz [12] can be used to detect Bluetooth vulnerabilities, but they are limited to generating valid malformed packets because they fail to effectively consider the Bluetooth states. L2Fuzz [13] identifies L2CAP vulnerabilities via core-field mutation. However, it (1) constructs the state machine based solely on the Bluetooth specification and (2) assumes that the state of the target device is the same as that of the fuzzer. Therefore, the effectiveness of fuzzing is compromised (see Sect. 4).

Several approaches (*e.g.*, [7,8,15]) aim to discover Bluetooth vulnerabilities through fuzzing but their main target is a different layer (*i.e.*, not the L2CAP). As these approaches were developed by focusing on their respective target layers or stacks, applying them to detect vulnerabilities in L2CAP is challenging.

**General Bluetooth Vulnerability Discovery.** Several approaches have been devised to discover Bluetooth vulnerabilities without fuzzing (*e.g.*, [3,6,16,17, 27]). However, they require considerable manual operations, such as firmware patching, reverse engineering, or manual examination of Bluetooth specifications. Therefore, their application to diverse Bluetooth devices is limited.

**Other Protocol Vulnerability Detection with State Machine Inference.** Several approaches have been suggested to uncover vulnerabilities in diverse protocols by inferring the state machine of the target device (*e.g.*, [9,14,18]). However, these techniques do not target the Bluetooth protocol, and applying them directly to L2CAP would require extensive modification of the original technology (*e.g.*, modify existing technology to align with the Bluetooth stack).

**General Vulnerability Detection Techniques.** General vulnerability detection approaches, including source code-based (*e.g.*, [11,22–24,26]) and binary-based (*e.g.*, [10,25,28]) approaches, can be applied to Bluetooth software. However, due to most Bluetooth devices being black boxes, direct application of source code-based methods is challenging. Binary-based approaches face difficulties in discovering Bluetooth vulnerabilities while specifically targeting Bluetooth functionality, making them unsuitable for our goals.

## 7 Conclusion

Vulnerabilities present in Bluetooth devices could potentially pose a threat to our daily lives. In response, we proposed BloomFuzz, a stateful fuzzer for the Bluetooth L2CAP. By leveraging the concept of state clusters, BloomFuzz infers the state machine implemented in the target device with high accuracy and enhances fuzzing efficiency. BloomFuzz exhibited significantly higher fuzzing efficiency compared to existing L2CAP fuzzers, detecting 56 potential vulnerabilities in real-world Bluetooth devices. With BloomFuzz, the security of Bluetooth devices can be strengthened, consequently, rendering a secure wireless ecosystem.

**Acknowledgment.** We appreciate the anonymous reviewers for their valuable comments to improve the quality of the paper. Additionally, we appreciate Haram Park and Choongin Lee for their valuable comments. This work was supported by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-00277, Development of SBOM Technologies for Securing Software Supply Chains, No.2022-0-01198, Convergence Security Core Talent Training Business (Korea University), and IITP-2024-2020-0-01819, ICT Creative Consilience program).

## A Discovered Crashes

BloomFuzz could discover 56 potential vulnerabilities (see Table 5). Among them, two potential vulnerabilities (in D6 and D7; see Table 2) were reported and confirmed by each vendor. Next, the eight potential vulnerabilities were patched by the vendors while we were analyzing the root causes. Eighteen crashes occurred intermittently, while the remaining 28 crashes are still under analysis. We will report to the vendor as soon as we complete the analysis.

**Table 5.** Classification results of discovered potential vulnerabilities.

| Classification | Number | Devices |
|---|---|---|
| Confirmed | 2 | D6, D7 |
| Already patched | 8 | D3 |
| Intermittently | 18 | D2, D4 |
| Under investigation | 28 | D1, D5, D6, D7 |

## B Efficiency in Addressing Missing and Hidden States

Figure 7 shows state machine generation effectiveness and packet acceptance ratio. The $A_t$ demonstrates the effectiveness of missing state pruning (see Sect. 4.1). The better the missing state is removed, the higher the probability that the packet will not be rejected. Note that BloomFuzz exhibits the highest $A_t$. Additionally, $A_i$ indicates how well missing and hidden states are handled. While we cannot directly determine whether vulnerabilities were found in the hidden state, we can indirectly infer that by effectively managing missing and hidden states. As a result, BloomFuzz can discover more crashes than other fuzzers.

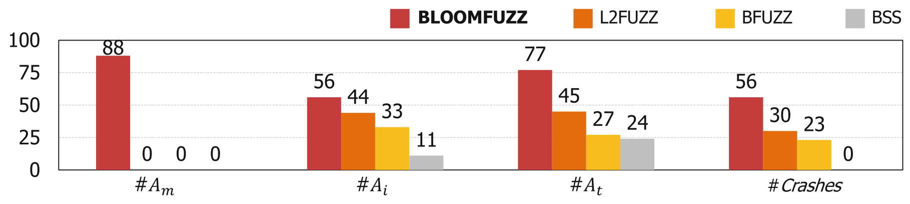


**Fig. 7.** State machine generation effectiveness and packet acceptance ratio.

## References

1. Android Build Coastguard Worker, BlueDroid_12.1.0_r19 (2023). https://android.googlesource.com/platform/system/bt/+/refs/tags/android-platform-12.1.0_r19. Accessed 4 Jan 2024
2. Antonioli, D., Tippenhauer, N.O., Rasmussen, K.: BIAS: bluetooth impersonation attacks. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 549–562 (2020)
3. Antonioli, D., Tippenhauer, N.O., Rasmussen, K.: Key negotiation downgrade attacks on bluetooth and bluetooth low energy. ACM Trans. Priv. Secur. (TOPS) **23**(3), 1–28 (2020)
4. Antonioli, D., Tippenhauer, N.O., Rasmussen, K.B.: The KNOB is broken: exploiting low entropy in the encryption key negotiation of bluetooth BR/EDR. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 1047–1061 (2019)
5. Betouin, P.: [Infratech - vulnérabilité] Nouvelle version 0.8 de Bluetooth Stack Smasher (2015). http://www.secuobs.com/news/15022006-bss_0_8.shtml. Accessed 4 Jan 2024
6. Claverie, T., Esteves, J.L.: BlueMirror: reflections on bluetooth pairing and provisioning protocols. In: 2021 IEEE Security and Privacy Workshops (SPW), pp. 339–351 (2021)
7. Garbelini, M.E., Bedi, V., Chattopadhyay, S., Sun, S., Kurniawan, E.: BRAKTOOTH: causing havoc on bluetooth link manager via directed fuzzing. In: 31st USENIX Security Symposium (USENIX Security 2022), pp. 1025–1042 (2022)
8. Garbelini, M.E., Wang, C., Chattopadhyay, S., Sumei, S., Kurniawan, E.: SweynTooth: unleashing mayhem over bluetooth low energy. In: 2020 USENIX Annual Technical Conference (USENIX ATC 2020), pp. 911–925 (2020)
9. Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K.: Pulsar: stateful black-box fuzzing of proprietary network protocols. In: Thuraisingham, B., Wang, X.F., Yegneswaran, V. (eds.) SecureComm 2015. LNICST, vol. 164, pp. 330–347. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28865-9_18
10. Han, H., Kyea, J., Jin, Y., Kang, J., Pak, B., Yun, I.: QueryX: symbolic query on decompiled code for finding bugs in COTS binaries. In: 2023 IEEE Symposium on Security and Privacy (SP), pp. 3279–312795 (2023)
11. Kim, S., Woo, S., Lee, H., Oh, H.: VUDDY: a scalable approach for vulnerable code clone discovery. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 595–614 (2017)
12. Kim, S., Woo, S., Lee, H., Oh, H.: 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)

13. Park, H., Nkuba, C.K., Woo, S., Lee, H.: L2Fuzz: discovering bluetooth L2CAP vulnerabilities using stateful fuzz testing. In: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 343–354 (2022)
14. Rasoamanana, A.T., Levillain, O., Debar, H.: Towards a systematic and automatic use of state machine inference to uncover security flaws and fingerprint TLS stacks. In: Atluri, V., Di Pietro, R., Jensen, C.D., Meng, W. (eds.) ESORICS 202. LNCS, vol. 13556, pp. 637–657. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17143-7_31
15. Ruge, J., Classen, J., Gringoli, F., Hollick, M.: Frankenstein: advanced wireless fuzzing to exploit new bluetooth escalation targets. In: 29th USENIX Security Symposium (USENIX Security 2020), pp. 19–36
16. Seri, B., Vishnepolsky, G., Zusman, D.: BLEEDINGBIT: the hidden attack surface within BLE chips (2019)
17. Seri, B., Vishnepolsky, G.: BlueBorne: The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks (2017). https://www.armis.com/research/blueborne/. Accessed 3 Jan 2024
18. Shu, Z., Yan, G.: IoTInfer: automated blackbox fuzz testing of IoT network protocols guided by finite state machine inference. IEEE Internet Things J. **9**(22), 22737–22751 (2022)
19. SIG, B.: Bluetooth Core Specification 5.2 (2019). https://www.bluetooth.com/specifications/specs/
20. Synopsys: Defensics Fuzz Testing. https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html. Accessed 4 Jan 2024
21. von Tschirschnitz, M., Peuckert, L., Franzen, F., Grossklags, J.: Method confusion attack on bluetooth pairing. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 1332–1347 (2021)
22. Woo, S., Choi, E., Lee, H., Oh, H.: V1SCAN: discovering 1-day vulnerabilities in reused C/C++ open-source software components using code classification techniques. In: 32nd USENIX Security Symposium (USENIX Security 2023), pp. 6541–6556 (2023)
23. Woo, S., Hong, H., Choi, E., Lee, H.: MOVERY: a precise approach for modified vulnerable code clone discovery from modified open-source software components. In: 31st USENIX Security Symposium (USENIX Security 2022), pp. 3037–3053 (2022)
24. Woo, S., Park, S., Kim, S., Lee, H., Oh, H.: CENTRIS: a precise and scalable approach for identifying modified open-source software reuse. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 860–872 (2021)
25. Wu, J., et al.: OSSFP: precise and scalable C/C++ third-party library detection using fingerprinting functions. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 270–282
26. Xiao, Y., et al.: MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In: 29th USENIX Security Symposium (2020), pp. 1165–1182 (2020)
27. Xu, F., Diao, W., Li, Z., Chen, J., Zhang, K.: BadBluetooth: breaking android security mechanisms via malicious bluetooth peripherals. In: NDSS (2019)
28. Yuan, Z., et al.: B2SFinder: detecting open-source software reuse in COTS software. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1038–1049 (2019)
29. Zalewski, M.: American fuzzy lop (2021). https://github.com/google/AFL. Accessed 3 Jan 2024