BTFuzzer: a profile-based fuzzing framework for Bluetooth protocols

Min Jang[1,2], Yuna Hwang[2], Yonghwi Kwon[3], and Hyoungshick Kim[1]

[1] Sungkyunkwan University, Suwon, South Korea
{min.jang, hyoung}@skku.edu
[2] Samsung Electronics, Suwon, South Korea
{min.s.jang, yuna.hwang}@samsung.com
[3] University of Maryland, College Park, Maryland, USA
yongkwon@umd.edu

Abstract. Bluetooth vulnerabilities have become increasingly popular in recent years due to, in part, the remote exploitability of Bluetooth. Unfortunately, in practice, security analysts often rely on manual analysis to identify these vulnerabilities, which is challenging. Specifically, testing various workloads while maintaining reliable Bluetooth connections between devices requires complicated network configuration settings. This paper introduces BTFuzzer, a profile-based fuzzing framework for Bluetooth devices. BTFuzzer eliminates the need for complex network configurations by feeding Bluetooth packets directly into the target device's Bluetooth library without going through the Over-The-Air (OTA) transmissions. BTFuzzer carefully crafts test inputs based on protocol profiles and specifications to maximize code coverage efficiently. Our evaluation results show that BTFuzzer is highly effective. In particular, the framework has identified two security bugs in the latest Android versions (i.e., 10 and later): CVE-2020-27024 and a publicly unknown information leak vulnerability. The first is an out-of-bounds read vulnerability (CVE-2020-27024). The second vulnerability allows attackers to connect to a victim's device and leak sensitive data without the user's awareness, as the adversary is not shown in the list of connected Bluetooth devices.

Keywords: Bluetooth · Protocol · Fuzzing · Memory Corruption · Remote Code Execution.

1 Introduction

Recent Bluetooth vulnerabilities such as *BlueBorne* [16] have sparked interest in finding Bluetooth-related security bugs due to, in part, its broad impact across multiple platforms. For example, BlueBorne affects Bluetooth implementations across multiple platforms: Android, iOS, Windows, and Linux. As of September 2023, 720 CVEs have been registered as Bluetooth-related vulnerabilities [1], where they are remotely exploitable. For instance, CVE-2017-0781 is a vulnerability in the Android's BNEP service. It allows attackers to compromise Bluetooth devices [16, 25] remotely. Due to Bluetooth vulnerabilities' high and broad

security impact, security testing of the systems using Bluetooth is particularly important and critical.

Fuzzing is an automated testing approach that injects randomized inputs into a system under test to reveal vulnerabilities. For software testing, fuzzing has been successful over the years for various software systems, from OS kernels [6,7,15] to robotics systems [8–11]. However, unfortunately, fuzzing network protocols such as Bluetooth is still challenging. Specifically, the Bluetooth protocol is highly dependent on complex network configurations. Conducting various tests while preserving the same network configurations and states after each test requires non-trivial effort. In addition, practical challenges such as synchronization and delay of network communication further complicate the testing process. Worse, the root causes of many vulnerabilities stem from flaws in the Bluetooth chipset firmware rather than the software stack. Hence, various firmware implementations should be taken into consideration as well. Unfortunately, existing fuzzing approaches have difficulty thoroughly testing various layers of the system such as the Bluetooth protocol layer and the application layer. For example, many existing fuzzers generate test inputs targeting *device drivers*, which may not even reach the application layer, which may contains various potential vulnerabilities. In other words, existing techniques may underexplore a non-trivial amount of space for Bluetooth-related vulnerabilities.

This paper introduces BTFuzzer, a fuzzing framework that automatically identifies Bluetooth vulnerabilities. While there exist approaches for identifying Bluetooth security bugs [12, 13], they suffer from various challenges such as (1) obtaining and maintaining complex network configurations during the test and (2) crafting complex test inputs that can penetrate various software layers without violating the constraints from device drivers, network protocol, and applications. Our approach, BTFuzzer, addresses these challenges by creating an interface to inject Bluetooth packets into the library directly. It maximizes code coverage by carefully crafting specific test inputs (e.g., Bluetooth packets) with respect to the protocol specifications such as Hand-Free Profile (HFP), Human Interface Device (HID), and Bluetooth Radio Frequency Communication (RFCOMM). The framework encompasses key components for comprehensive Bluetooth protocol fuzzing, including a packet generator, crash collector, and coverage analyzer.

To demonstrate the effectiveness of BTFuzzer, we conducted experiments on Android using open-source software. BTFuzzer found two previously unknown vulnerabilities that are exploitable in most Android devices: (1) An out-of-bounds read vulnerability (CVE-2020-27024 [24]), affecting systems running Android version 10 or later and (2) an information leak vulnerability that allows attackers to connect to a victim's device and leak data without the user's awareness as it is not visible in the list of connected Bluetooth devices.

Organization. The remainder of the paper is organized as follows: Section 2 provides background on Bluetooth and fuzzing. Section 3 introduces our proposed fuzzing framework. Section 4 presents our experimental results. Section 5 discusses related work. Section 6 concludes the paper.

2 Background

This section outlines the structure of Bluetooth that is essential for understanding BTFuzzer. We also provide an overview of the Bluetooth stack, Bluetooth profiles, and a generic fuzzing environment for Bluetooth protocols.

2.1 Bluetooth components

Figure 1 illustrates a generic Bluetooth stack. Bluetooth packets move from the baseband to Logical Link Control and Adaptation Protocol (L2CAP) via the Host Controller Interface (HCI). L2CAP then routes these packets to the next appropriate stack for each channel. The HCI packet encapsulates data for the upper protocols and profiles, including L2CAP, and the path to the upper layer varies depending on the configuration of the HCI packet. If packets can be fed directly to the HCI, a security evaluation of the Bluetooth stack can be performed without the need for complex wireless configurations.

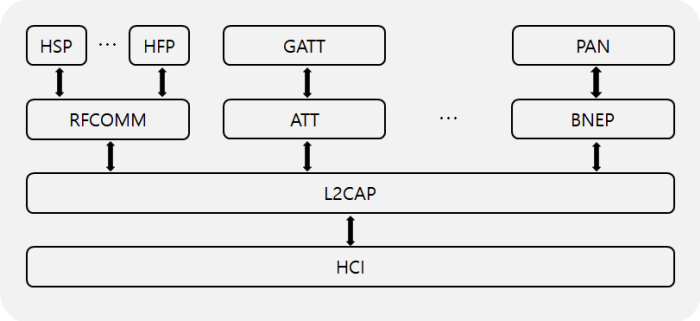


Fig. 1: Generic Bluetooth stack.

A Bluetooth profile is a protocol that aims to provide compatibility across various devices, allowing diverse Bluetooth devices to interact with each other. While the operation method may vary among devices, functions are implemented according to specific Bluetooth profiles, enabling communication between devices with different operating systems. Packet configurations differ for each profile and conform to the forms defined in their respective specifications [2]. Vulnerabilities may arise from improper profile implementations, making generating and transmitting packets tailored to each profile crucial for effective vulnerability discovery through fuzzing.

L2CAP operates based on the channel. A channel identifier (CID) [3] is the local name representing a logical channel endpoint on the device. When a Bluetooth device makes a connection, a channel is created and a CID is assigned. Communication with the device is possible through the assigned CID and channel. CID has a namespace designated according to its purpose. The CID namespace is 0x0000-0xFFFF. In the namespace, the null identifier (0x0000) is not

used, and the identifiers from 0x0001 to 0x003F are reserved for a specific L2CAP function, which is called fixed channels. Therefore, when connected to a generic Bluetooth device, CIDs are allocated within the range of 0x0040-0xFFFF, which are called dynamically allocated channels.

L2CAP's upper layers support various protocols. Radio Frequency Communications (RFCOMM) replaces the traditional wired RS232 serial port and shares characteristics with the TCP protocol. Currently, The Headset Profile (HSP) and Handsfree Profile (HFP) are the popular profiles that use it. The Generic Attribute Profile (GATT), or often referred to as GATT/ATT, outlines how to exchange data between BLE devices using services and characteristics. It represents the highest-level implementation of the Attribute protocol (ATT). Each attribute has a 128-bit UUID and ATT-defined attributes determine characteristics and services. The Bluetooth Network Encapsulation Protocol (BNEP) enables the transmission of common networking protocols over Bluetooth and offers functionalities similar to Ethernet's. Running on BNEP, the Personal Area Networking Profile (PAN) specifies how two or more Bluetooth-enabled devices can form an ad-hoc network and access a remote network via a network access point.

2.2 Generic fuzzing environment for Bluetooth protocols

The fuzz testing technique is widely employed to discover security vulnerabilities [4] automatically. A fuzzer can be specialized for a specific target (e.g., a particular protocol or class of applications) or designed for a generic purpose such as AFL. To conduct a successful vulnerability discovery, understanding the characteristics of various fuzzers and selecting the most suitable one based on the target and scope of the analysis is critical.

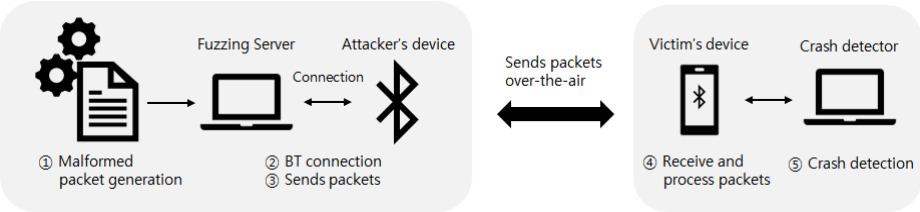


Fig. 2: Generic fuzzing environment for Bluetooth protocols.

Traditional Bluetooth fuzz testing requires two Bluetooth-capable devices: an attacker device that sends malformed packets and a victim device that processes the packets and potentially exposes vulnerabilities. The attacker device must maintain a state where it can send and receive packets. It must also implement a fuzzing engine with three functions: (1) Generating malformed packets (①), (2) Establishing a Bluetooth connection (②), and (3) Sending the malformed packets (③). The victim device must process packets (4) and detect crashes (⑤).

Setting up this environment is time-consuming and complex, as it essentially requires constructing the entire system, including the network environment.

The Bluetooth software stack processes packets sent over-the-air (OTA) via the Bluetooth firmware on the target device. In OTA-based fuzzing, whether specific packets reach the Bluetooth software stack may depend on the firmware configuration of the Bluetooth chipset. This environment is more suited for Bluetooth firmware code analysis and has limitations for Bluetooth software stack vulnerability analysis.

BTFuzzer simplifies the fuzz testing process by directly transmitting packets to the victim device, bypassing the wireless environment. This approach allows quicker fuzz testing and eliminates the need for the packets to go through the Bluetooth firmware before reaching the software stack. BTFuzzer proposes an automated method to identify logical errors within the Bluetooth software stack.

3 Proposed system

In this section, we explain how to fuzz the Bluetooth stack using the proposed fuzzing framework, BTFuzzer.

3.1 Overview

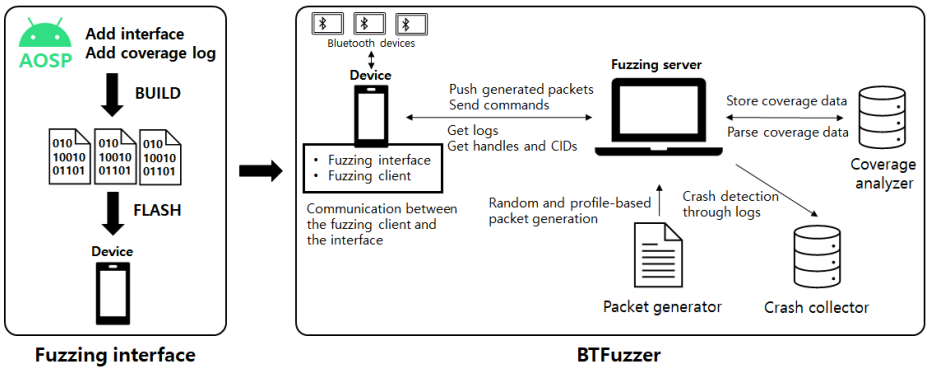


Fig. 3: Overview of BTFuzzer.

We propose a new fuzzing framework, BTFuzzer, which directly feeds packets into the target device, bypassing OTA. BTFuzzer generates packets and defines an interface for direct input into the device's HCI layer. Figure 3 provides an overview of the proposed system. This configuration allows direct access to the Bluetooth software stack for fuzz testing on profiles and protocols with independent specifications. We note that the framework is highly configurable, meaning that it can easily customized to support fuzz testing on diverse profiles and even other protocols of interest.

3.2 Fuzzing interface

We create a specialized fuzzing interface to feed packets directly into the device. In particular, based on our analysis of the Android Open Source Project (AOSP) Bluetooth stack, we implement our fuzzing interface in `libbluetooth.so`.[4]

The `hci_initialize` function within `hci_layer_android.cc` initializes the HCI and creates (1) a fuzzing interface thread and (2) a socket for communication with the fuzzing client. This client then feeds commands and packets from the fuzzing server into the interface through the socket.

HCI Handles and L2CAP CIDs are essential for generating valid Bluetooth packets. The interface receives and processes predefined commands from the client to obtain these values. The currently connected Handles and CIDs are saved, and the gathered Handles and CIDs are used for packet creation. Additionally, HCI packets fed into the interface are categorized into four types for processing: COMMAND, ACL, SCO, and EVENT. Figure 4 illustrates the architecture of the fuzzing interface within the AOSP device.

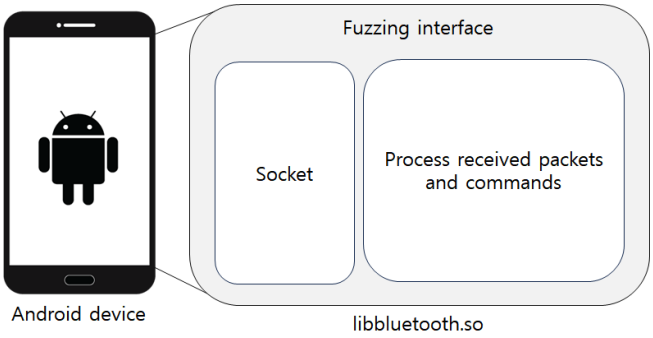


Fig. 4: Composition of fuzzing interface.

3.3 Fuzzing server

The fuzzing server consists of the following three modules:

– **Packet generator:** This module creates a large corpus of malformed packets by randomly injecting errors into valid packets. This addresses performance degradation when feeding individual packets to the Android device via ADB. This ensures that the fuzzing process covers a wide range of possible inputs. The corpus is transferred to the Android device using the `adb push` command.

[4] The exact location of the implementation is '`AOSP\system\bt\hci\src\hci_layer_android.cc:hci_initialize()`.'

– **Crash collector:** This module collects crashes that occur during fuzzing.
– **Coverage analyzer:** This module analyzes the coverage of the Bluetooth software stack during fuzzing.

HCI handles, and L2CAP CIDs are assigned when a Bluetooth device is connected. However, these values may change if the device is reconnected after a crash. This requires the packet generator to regenerate the packets. Additionally, the device's Bluetooth settings may change due to previous packets. To mitigate these issues, the fuzzing server initializes the Bluetooth stack before starting the fuzzing process. This ensures that HCI handles and L2CAP CIDs remain constant, allowing the use of pre-made packets even after a crash.

3.4 Fuzzing client

The fuzzing client is specialized for interaction with the fuzzing interface, implemented in the `libbluetooth.so` library. This client is an executable file that establishes a connection to the fuzzing interface's socket. It reads from the corpus file located at a predefined path and sequentially sends packets into the Bluetooth stack via this socket. Essentially, the fuzzing client is responsible for sending malformed packets to the Android device for testing.

Figure 5 illustrates the architecture of the fuzzing client, showcasing its various components and their interaction with the fuzzing interface. This helps to understand the role of the fuzzing client in the overall architecture of BTFuzzer, highlighting its critical role in injecting malformed packets into the system to identify vulnerabilities.

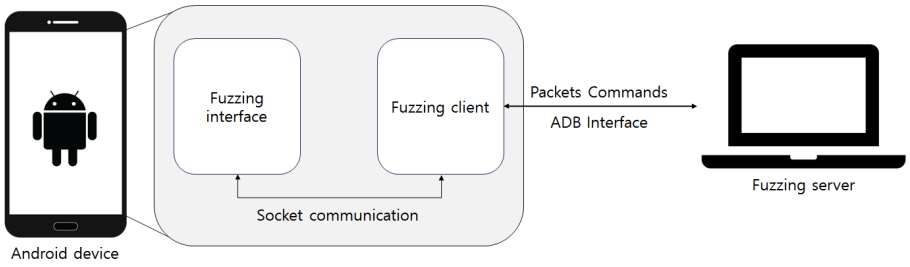


Fig. 5: Composition of the fuzzing client.

3.5 Packet generator

This paper focuses on fuzzing three key Bluetooth protocols commonly used in smartphones: RFCOMM, HFP, and HID. These were selected because they are essential for core smartphone functions and have significant security implications.

- RFCOMM is a simple, reliable data stream to which other applications can connect as if they were serial ports. It is one of the foundational profiles used in most Bluetooth devices, meaning that it is an essential test subject.
- HFP is crucial for enabling smartphone call functionalities. Given that calling is a core function of smartphones and a profile used daily by many users, any vulnerabilities in HFP could have significant security implications, such as the potential for eavesdropping.
- HID is related to input devices such as keyboard and mouse. Vulnerabilities in HID could allow an attacker to remotely control the victim's device, making it critical for security analysis.

To generate test cases for these profiles, we have implemented two different types of packet generation techniques: mutation-based and profile-based.

First, the mutation-based packet generator takes existing valid Bluetooth packets and modifies them in various ways to create malformed packets. These malformed packets are then used to test how well the Bluetooth stack can handle unexpected or non-standard data.

Second, the profile-based packet generator creates packets according to the specifications of the target Bluetooth profiles (RFCOMM, HFP, and HID). By adhering closely to the specifications, we can test for vulnerabilities caused by wrong implementations of the protocols.

By combining the two different packet generation techniques, BTFuzzer aims to achieve a comprehensive set of test cases that can thoroughly evaluate the robustness and security of Bluetooth implementations in Android devices.

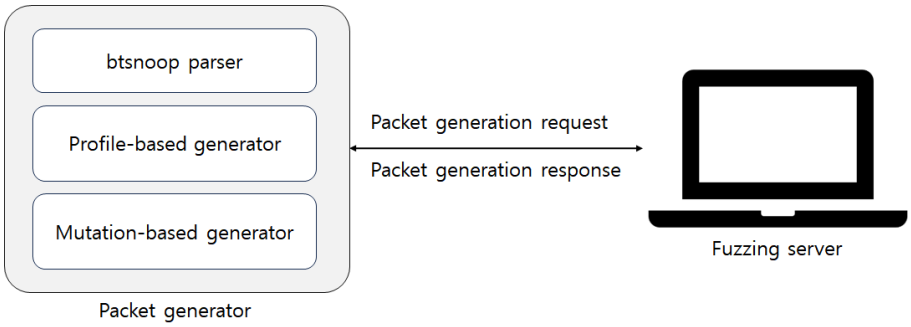


Fig. 6: Composition of the packet generator.

Mutation-based packet generation. Mutation-based packet generation creates new packets through mutation, using packets transmitted and received between devices to enhance code coverage. Base packets are obtained from Android Bluetooth snoop logs [18]. Bluetooth HCI Snoop is specified in RFC 1761 [17]. A simple script was developed to parse these Snoop logs into a mutational hex

format. Pyradamsa is used to mutate the parsed packets. A base packet is selected for mutation. A packet must be generated with a matching HCI Handle and L2CAP CID to facilitate normal communication and data processing. In mutation-based generation, packets are created using two methods. The first method sequentially writes and mutates the entire set of recorded packets. The second method randomly selects a packet for mutation.

Profile-based packet generation. Profile-based packet generation produces packets tailored for specific Bluetooth profiles and protocols. Target profiles and protocols were selected, and their specifications were analyzed. We examined the specifications for three items: HFP [19], HID [20], and RFCOMM [21]. Payloads for each item are generated using Python's random library. Like in mutation-based generation, the HCI and L2CAP portions, excluding the payload, utilize the allocated HCI Handle and L2CAP CID. Packets, including the generated payload, are generated with matching HCI and L2CAP lengths.

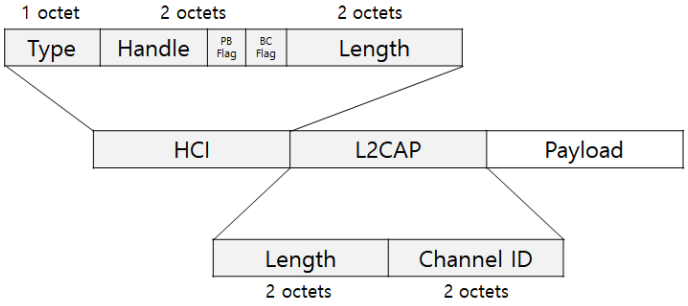


Fig. 7: Structure of HCI and L2CAP packets.

Figure 7 illustrates the basic structure of HCI and L2CAP packets. The type field in HCI packets consists of one octet and classifies COMMAND, ACL, SCO, and EVENT types. The handle field, comprising two octets, holds connection information between devices. The Length field, also of two octets, specifies the total length of the HCI packet. If the length field value does not match the packet length, Android Bluetooth HCI will immediately abort the connection. Therefore, it is crucial to calculate and set the correct length and handle values when generating a packet. Detailed specifications for HID, HFP, and RFCOMM, along with their implementation in BTFuzzer, are outlined below.

Figure 8 depicts the packet structure of HID. The Header field contains HID Header information in one octet. Only HANDSHAKE, HID_CONTROL, and DATA Message types are used for packet generation. These types facilitate data transmission from HID to the smartphone. The payload part consists of randomly generated data, varying in size from 0x00 to 0xFF.

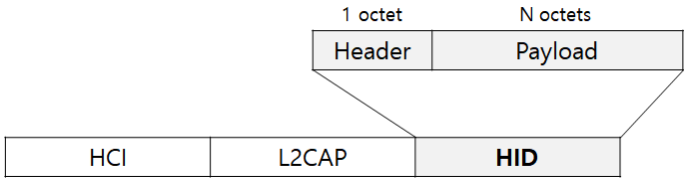


Fig. 8: Packet structure of HID.

Figure 9 shows the packet structure of RFCOMM. The Address field, consisting of one octet, contains the DLCI (Data Link Connection Identifier) or the connection information for RFCOMM. To transmit data correctly, this address value must be set accurately, which can be retrieved from Bluetooth logs. The Control field is one octet and includes frame type and poll/final bit information. Depending on the payload size, the Length field consists of one or two octets. If the payload size exceeds 127 bytes, two octets are used. The Payload field is filled with random values, and its size determines the Length field. Finally, the FCS field, comprised of one octet, is used for CRC (Cyclic Redundancy Check). It is calculated based on predefined CRC table values, Address, and Control fields.

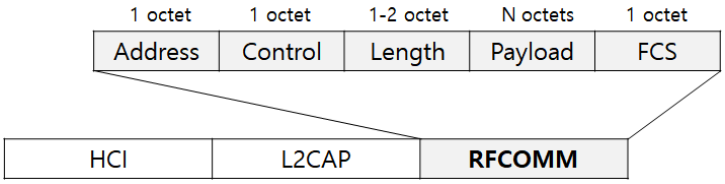


Fig. 9: Packet structure of RFCOMM.

Figure 10 presents the packet structure of HFP. The payload field is the only variable part based on AT Commands from the RFCOMM packet structure. We extracted a list of usable AT Commands from Android Bluetooth code and configured the system to randomly generate payloads for each AT Command.

3.6 Crash collector

When a crash occurs during fuzzing, the crash collector gathers and stores relevant information. On Android devices, Signals 6 and 11 automatically generate tombstone files. The crash collector checks whether a tombstone file is created during fuzzing. If created, it collects the tombstone file from the Android device. The generated corpus, handle, and CID information are stored to facilitate crash reproduction. Figure 11 illustrates the components of the crash collector.

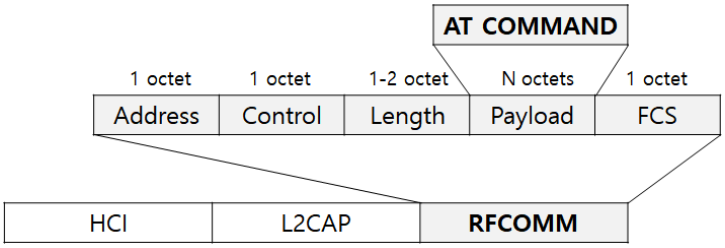


Fig. 10: Packet structure of HFP.

AT Command List		
AT+VGS	AT+VGM	AT+CCWA
AT+CHLD	AT+CHUP	AT+CIND
AT+CLIP	AT+CMER	AT+VTS
AT+BINP	AT+BLDN	AT+BVRA
AT+BRSF	AT+NREC	AT+CNUM
AT+BTRH	AT+CLCC	AT+COPS
AT+CMEE	AT+BIA	AT+CBC
AT+BCC	AT+BCS	AT+BIND
AT+BIEV	AT+BAC	

Table 1: List of AT Commands used for packet generation.

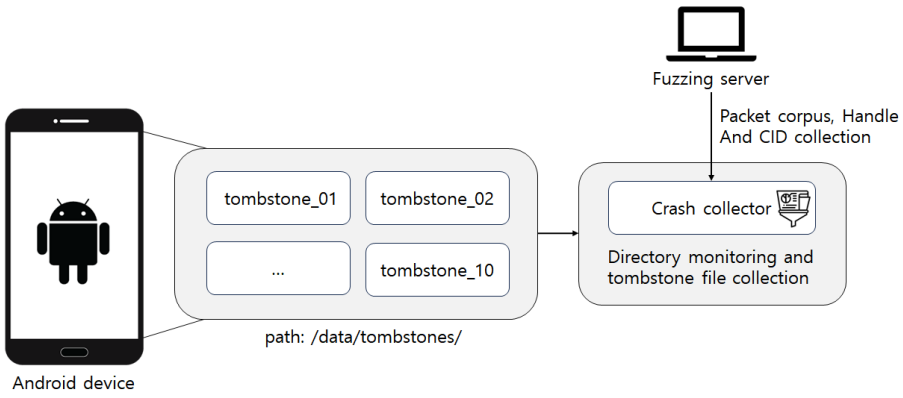


Fig. 11: Composition of the crash collector.

3.7 Coverage analyzer

To measure the coverage of the code, the coverage analyzer inserts log codes into all AOSP Bluetooth stack files. To avoid duplicates, the log format is set as `FUZZ_COVERAGE _FileName_Count`. For automated log insertion, we developed a Python script. Once log code insertion is complete, the number of logs added to

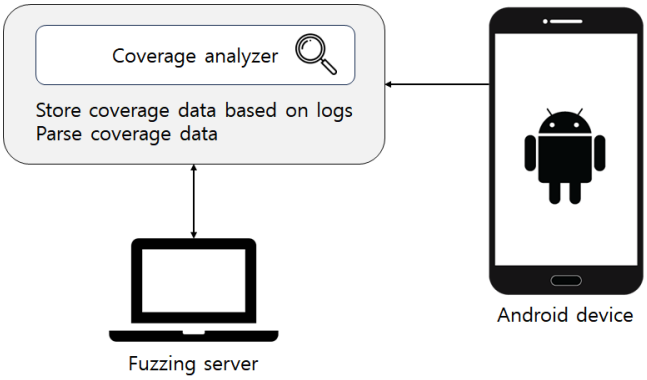


Fig. 12: Composition of the coverage analyzer.

each file and the total log count are recorded. By comparing the number of output logs during fuzzing with the total number of logs, we can assess the extent of code execution. Logs are inserted to identify most branching statements, allowing efficient code coverage measurement for `libbluetooth.so`. Figure 12 illustrates the structure of the coverage analyzer.

4 Evaluation

BTFuzzer was tested on a Pixel 3a device running Android 10. During the evaluation, it was paired with a Galaxy Watch, Galaxy Buds, a Bluetooth keyboard, and a Bluetooth mouse. Fuzzing was conducted after analyzing the packets obtained during basic interactions between the Pixel 3a and each Bluetooth device. After that, random packets were generated for fuzzing. The profiles evaluated were RFCOMM, HID, and HFP. To assess BTFuzzer's effectiveness, we applied it to the binary code before patching the vulnerability known as BlueFrag (CVE-2020-0022) [22, 23], one of the most critical Android Bluetooth vulnerabilities of 2020.

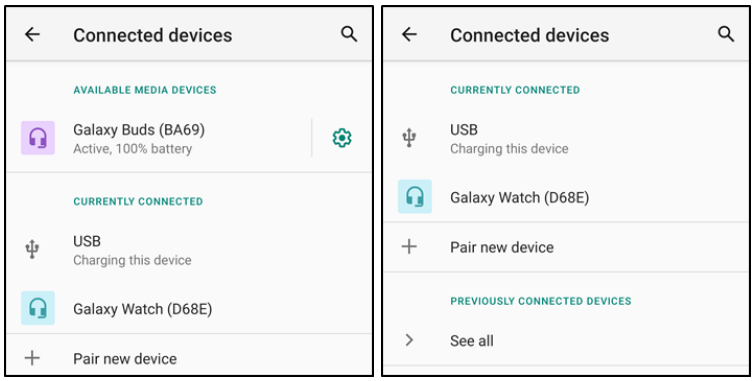
BTFuzzer discovered two vulnerabilities that could affect most Android devices, including the latest version. One was reported to the Google Android Security Team and recognized as a new vulnerability under the identifier `A-182388143`. The other was reported as `A-182164132` but was marked as a duplicate of `A-162327732`, which has been assigned CVE-2020-27024 [24]. The BlueFrag vulnerability, for which the patch had been removed, was also detected.

The code coverage of BTFuzzer was assessed using the coverage analyzer. When delivering packets generated specifically for a particular profile, it was observed that the code coverage corresponding to that profile increased significantly. This observation validates the effectiveness of profile-based fuzz testing.

4.1 Hiding the list of malicious Bluetooth devices

We discovered a new vulnerability in the Bluetooth stack of Android devices. This vulnerability allows attackers to manipulate the list of Bluetooth-connected devices on a victim's device. The vulnerability, which is assigned to the identifier `A-182388143`. It was discovered by using RFCOMM profile-based fuzz testing with BTFuzzer. The Google Android Security Team has confirmed it as a security vulnerability.

The vulnerability can be exploited on most Android devices, including the latest version. An attacker could use this flaw to hide a malicious Bluetooth device connected to the user's device, making it undetectable to the user. Consequently, the attacker could access contacts and SMS messages or intercept calls without the user noticing the attacker's activities. The vulnerability can be exploited by sending just one malicious packet to the user's device.



(a) `Before the attack` (b) `After the attack`

Fig. 13: Result of the attack that exploited the `A-182388143` vulnerability on Google Pixel 3a. The connected devices list is shown before (a) and after (b) the attack. Galaxy Buds are initially displayed in the connected list before the attack but not in the connected list after the attack. This is because the attacker was able to remove Galaxy Buds from the list by exploiting the vulnerability.

As shown in Figure 13, we can see the Bluetooth device in the connected list before the attack is performed. However, after the attack is performed, the Bluetooth device is not visible in the connected device list even though the device can still maintain the connection with the victim device. This vulnerability was discovered while fuzzing RFCOMM. It was possible to trigger the vulnerability through a specific packet generated by BTFuzzer's Profile-based. This attack can hide the device by sending only one simple packet. We received a 2,000 USD reward from the Google Android Security Team for reporting this vulnerability. However, this vulnerability has not been patched yet and detailed information cannot be disclosed to prevent malicious exploitation.

4.2 Buffer overflow vulnerabilities

CVE-2020-0022. To demonstrate BTFuzzer's effectiveness, we conducted fuzzing tests on a binary containing the BlueFrag vulnerability, a significant Android Bluetooth vulnerability from 2020. Our goal is to evaluate whether BTFuzzer can find a known vulnerability effectively. Just less than 5 minutes, BTFuzzer detected the CVE-2020-0022 vulnerability. Figure 14 displays the crash log for this vulnerability, triggered by BTFuzzer.

```
pid: 7221, tid: 10924, name: bt_hci_thread  >>> com.android.bluetooth <<<
uid: 1002
signal 11 (SIGSEGV), code 2 (SEGV_ACCERR), fault addr 0x6ff73ffff0

backtrace:
     #00 memcpy+104
     #01 reassemble_and_dispatch(BT_HDR*) [clone .cfi]+948
```

Fig. 14: CVE-2020-0022 crash log.

A-182164132. The out-of-bounds vulnerability was discovered through the BTFuzzer, and the vulnerability was reported to `A-182164132`. However, it was already reported as a vulnerability with A-162327732. This vulnerability has been assigned CVE-2020-27024. CVE-2020-27024 is a vulnerability that can cause out-of-bounds read due to a missing boundary check in `smp_br_state_machine_event()` of `smp_br_main.cc`, Figure 15 shows the CVE-2020-27024 vulnerability crash log triggered via BTFuzzer. The vulnerability (i.e., related to the missing boundary check) is mitigated through Bounds Sanitizer, which is supported from Android 10. However, it can be still exploited in the previous Android versions or customized/specialized Android systems forked from the previous Android versions. This vulnerability can be attacked when the connection handle is 0x02. Figure 16 shows packets that can reproduce CVE-2020-27024. Sending these two packets could trigger the CVE-2020-27024 vulnerability.

4.3 Coverage

Code coverage was measured using a log-based approach, in which 31,997 logs were instrumented into the Android Bluetooth-related code. Fuzzing was carried out for 24 hours for each of the three methods used to generate packets: mutation-based, profile-based, and RFCOMM, HFP, and HID. The code coverage was then measured after each fuzzing run.

Figure 17 shows the change in code coverage over time during fuzzing. Figure 17(a) shows the total coverage for the 24 hours, which reveals an initial rapid increase followed by a slower growth rate. Figure 17(b) focuses on the first 10

```
pid: 3753, tid: 3802, name: bt_main_thread  >>> com.android.bluetooth <<<
uid: 1002
signal 6 (SIGABRT), code -1 (SI_QUEUE), fault addr --------
Abort message: 'ubsan: out-of-bounds'

backtrace:
     #00 abort+160
     #01 abort_with_message(char const*)+20
     #02 __ubsan_handle_out_of_bounds_minimal_abort+24
     #03 smp_br_state_machine_event(tSMP_CB*, unsigned char, tSMP_INT_DATA*)+1212
```

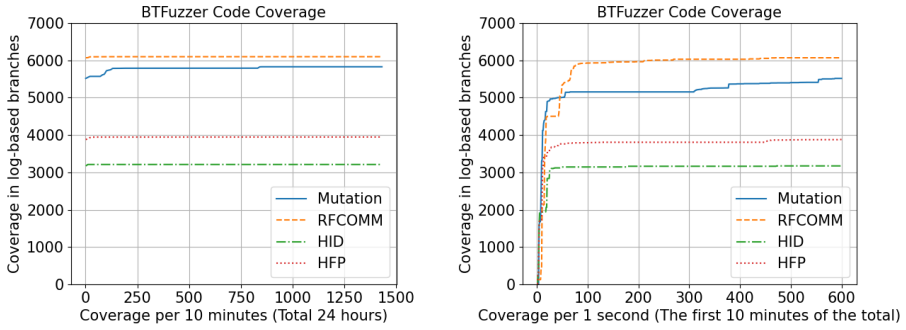
Fig. 15: CVE-2020-27024 crash log.

```
[Packet 1]
02022069006500060000befbb0057fe410e98007b22726573f3a0819e756c74223a22737563
63657373222c22726561736f6e223a302c226d73674964223a226d757369632d7175657565
6368616e6765642d696e64222c22636f756e74223a302c226c697374223a5b5d7dce249a

[Packet 2]
0202201500110007000bff1702000503c01af001c08007c07a86
```

Fig. 16: CVE-2020-27024 trigger packets.

minutes of this period, demonstrating a similar trend: an initial swift rise in coverage that eventually plateaus.



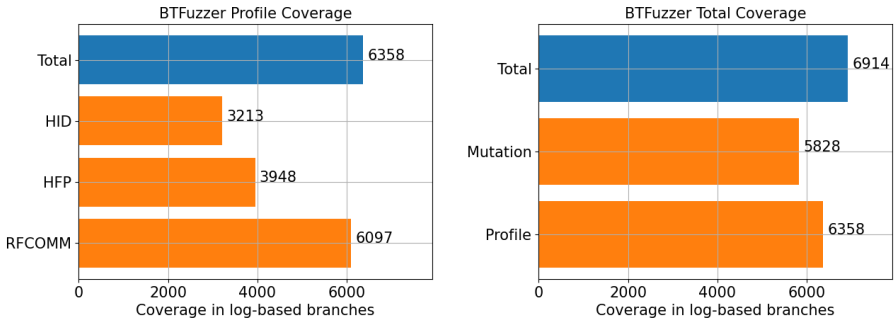
(a) `Total 24 hours` (b) `First 10 minutes`

Fig. 17: Code coverage changes over time. (a) represents the 24-hour coverage for mutation, RFCOMM, HID, and HFP methods. (b) represents the coverage changes during the first 10 minutes of the 24-hour period.

Figure 18(a) and (b) present the coverage results of mutation-based and profile-based (HFP, RFCOMM, HID) fuzzing, respectively. Figure 18(a) illustrates the outcomes of profile-based fuzzing, where "Total" denotes the combined

log results for HID, HFP, and RFCOMM, exceeding the individual log count for RFCOMM, the highest among them. Each method executed distinct code segments. Figure 18(b) contrasts mutation-based and profile-based fuzzing. Tests conducted on the same three types of Bluetooth devices (Galaxy Watch, Galaxy Buds, and a Bluetooth keyboard and mouse) showed that profile-based fuzzing achieved more code coverage than mutation-based fuzzing. Although profile-based fuzzing offers more code coverage, it requires understanding the profile and creating a packet structure code that aligns with the profile. Conversely, mutation-based fuzzing, while achieving less code coverage than profile-based fuzzing, allows fuzzing without profile comprehension. More importantly, each method executed different code segments, indicating that the two methods are complementary and could maximize fuzzing code coverage when combined.

Out of the 31,997 logs instrumented, 6,914 were recorded, representing approximately 21.6% of the total code coverage. Enhanced results are expected with further profile/protocol testing.



(a) `Profile-based coverage` (b) `Total coverage`

Fig. 18: Coverage results for 31,997 instrumented logs. (a) represents the coverage of profile-based fuzzing, and (b) compares mutation-based and profile-based methods.

4.4 Summary of evaluation results

BTFuzzer is an effective tool for finding vulnerabilities in Android Bluetooth stacks. It found two vulnerabilities in the Pixel 3a, one of which was a new vulnerability that allowed attackers to hide a Bluetooth device in the list of Bluetooth-connected devices on a victim's device. BTFuzzer also detected the CVE-2020-0022 vulnerability, a significant Android Bluetooth vulnerability from 2020, in less than 5 minutes.

Our evaluation also shows that BTFuzzer's profile-based fuzzing is more effective than mutation-based fuzzing at achieving more code coverage. However,

each approach targeted different code segments, meaning that they are complementary. We believe that combining both techniques could maximize fuzzing code coverage.

5 Related work

Research on Bluetooth security is diverse, covering topics such as attacks via malicious devices, vulnerabilities in protocol implementations, and methodologies for vulnerability analysis, including active fuzzing studies.

One approach focuses on exploiting the Bluetooth function by taking control of Bluetooth communication authority. Xu et al. [5] describe an attack that leverages a device's inherent trust in an already-connected Bluetooth device. This research suggests that devices better manage Bluetooth function authority, pairing conditions, and the intent of paired devices. A more straightforward method of identifying vulnerabilities is to analyze Bluetooth protocol implementations. A notable example is BlueBorne [16], published by ARMIS Lab in 2017, which examined Bluetooth specifications and identified vulnerabilities and logical errors. However, auditing the code for the entire Bluetooth specification and its various profiles is challenging.

Another technique to consider is fuzzing. Mantz et al. [13] introduced a versatile framework for finding vulnerabilities in Bluetooth firmware. Ruge et al. [12] proposed an advanced, firmware emulation-based fuzzing framework for undisclosed Bluetooth implementations and firmware. However, these studies focus on chipset firmware-level security evaluation, not the Bluetooth software stack. Heinze et al. [14] recently suggested a fuzzing approach targeting specific L2CAP Channels in Apple's private Bluetooth stack.

We propose a new approach: a profile-based fuzzing framework for the Bluetooth stack. This framework facilitates creating and fuzzing packets for each Bluetooth profile, enabling comprehensive coverage of various protocols and profiles within the Bluetooth stack.

6 Conclusions

As Bluetooth technology becomes ubiquitous and its applications span multiple devices and functionalities, vulnerabilities in Bluetooth technology have become high-impact security risks. Despite ongoing research to enhance Bluetooth security, new vulnerabilities continue to be discovered and exploited, demanding a systematic approach to search for vulnerabilities effectively.

We introduce BTFuzzer, a scalable, profile-based fuzzing framework for Bluetooth devices. BTFuzzer implements in-device packet transmission, eliminating the need for complex environment setup. It generates packets according to specific Bluetooth profiles to maximize code coverage. BTFuzzer has identified a new vulnerability that allows an attacker's Bluetooth device to remain concealed while connected to a victim's device. Additionally, BTFuzzer has demonstrated its efficacy by detecting previously disclosed Bluetooth vulnerabilities.

BTFuzzer is a generic approach and not limited to Android. It is highly configurable, meaning that it can be easily configured to support other operating systems and protocols. Our preliminary results indicate that BTFuzzer is compatible with Linux Bluez, making it a viable tool for evaluating vulnerabilities in the Linux Bluetooth software stack. Further experimentation with the multitude of Bluetooth profiles will enhance code coverage and enable the discovery of additional vulnerabilities. We plan to expand our research to other operating systems, Bluetooth profiles, and other wireless technologies such as NFC, Wi-Fi, and Zigbee to improve wireless network security.

Acknowledgements We thank our anonymous shepherd and reviewers for their valuable feedback and insights. Hyoungshick Kim is the corresponding author. This work was supported by Institute for Information & communication Technology Planning & Evaluation grant funded by the Korea government (No.2018-0-00532, Development of High-Assurance (>=EAL6) Secure Microkernel (50%), No.2022-0-00495 (30%), and No.2022-0-01199 (20%)).

References

1. MITRE, `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=bluetooth`. Accessed on September. 15, 2023
2. Bluetooth Specifications, `https://www.bluetooth.com/specifications/specs/`. Accessed on September. 15, 2023
3. BLUETOOTH CORE SPECIFICATION Version 5.2, page 1026-1028, `https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc\_id=478726`. Accessed on September. 15, 2023
4. Valentin Jean Marie Manès and HyungSeok Han and Choongwoo Han and Sang Kil Cha and Manuel Egele and Edward J. Schwartz and Maverick Woo.: The Art, Science, and Engineering of Fuzzing: A Survey. IEEE Transactions on Software Engineering (2019)
5. Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen and Kehuan Zhang.: BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals. Network and Distributed System Security Symposium (NDSS 2019)
6. Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu and Taesoo Kim.: Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. ACM Symposium on Operating Systems Principles (SOSP 2019)
7. Sergej Schumilo, Cornelius Aschermann, and Robert Gawlik, Ruhr-Universität Bochum; Sebastian Schinzel, Münster University of Applied Sciences; Thorsten Holz, Ruhr-Universität Bochum.: kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. USENIX Security Symposium (USENIX Security '17)
8. Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z. Berkay Celik, and Dongyan Xu.: PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. Network and Distributed System Security Symposium (NDSS 2021)
9. Taegyu Kim, Purdue University; Chung Hwan Kim and Junghwan Rhee, NEC Laboratories America; Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu, Purdue University.: RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles through Control-Guided Testing. USENIX Security Symposium (USENIX Security '19)

10. Seulbae Kim, and Taesoo Kim.: RoboFuzz: fuzzing robotic systems over robot operating system (ROS) for finding correctness bugs. ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)
11. Chijung Jung, Ali Ahad, Yuseok Jeon and Yonghwi Kwon.: SWARMFLAWFINDER: Discovering and Exploiting Logic Flaws of Swarm Algorithms. IEEE Symposium on Security and Privacy (SP 2022)
12. Jan Ruge, Jiska Classen, Francesco Gringoli and Matthias Hollick.: Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. USENIX Security Symposium (USENIX Security '20)
13. Dennis Mantz, Jiska Classen, Matthias Schulz and Matthias Hollick.: InternalBlue – Bluetooth Binary Patching and Experimentation Framework. ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2019)
14. Dennis Heinze, Matthias Hollick and Jiska Classen.: ToothPicker: Apple Picking in the iOS Bluetooth Stack. USENIX Workshop on Offensive Technologies (WOOT '20)
15. syzkaller, `https://github.com/google/syzkaller`. Accessed on September. 15, 2023
16. BlueBorne, `https://www.armis.com/blueborne/`. Accessed on September. 15, 2023
17. RFC 1761, `https://tools.ietf.org/html/rfc1761`. Accessed on September. 15, 2023
18. Android Bluetooth Verifying and Debugging, `https://source.android.com/devices/bluetooth/verifying_debugging#debugging-with-logs`. Accessed on September. 15, 2023
19. Hands-Free Profile 1.8, `https://www.bluetooth.com/specifications/specs/hands-free-profile-1-8/`. Accessed on September. 15, 2023
20. Human Interface Device Profile 1.1.1, `https://www.bluetooth.com/specifications/specs/human-interface-device-profile-1-1-1/`. Accessed on September. 15, 2023
21. RFCOMM 1.2, `https://www.bluetooth.com/specifications/specs/rfcomm-1-2/`. Accessed on September. 15, 2023
22. BlueFrag, `https://insinuator.net/2020/04/cve-2020-0022-an-android-8-0-9-0-bluetooth-zero-click-rce-bluefrag/`. Accessed on September. 15, 2023
23. CVE-2020-0022, `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0022`. Accessed on September. 15, 2023
24. CVE-2020-27024, `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-27024`. Accessed on September. 15, 2023
25. CVE-2017-0781, `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0781`. Accessed on September. 15, 2023