USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

# BLuEMan: A Stateful Simulation-based Fuzzing Framework for Open-Source RTOS Bluetooth Low Energy Protocol Stacks

Wei-Che Kao, Yen-Chia Chen, Yu-Sheng Lin, Yu-Cheng Yang, Chi-Yu Li, and Chun-Ying Huang, *National Yang Ming Chiao Tung University*

https://www.usenix.org/conference/usenixsecurity25/presentation/kao

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

# BLuEMan: A Stateful Simulation-based Fuzzing Framework for Open-Source RTOS Bluetooth Low Energy Protocol Stacks

Wei-Che Kao
*National Yang Ming Chiao Tung University*

Yen-Chia Chen
*National Yang Ming Chiao Tung University*

Yu-Sheng Lin
*National Yang Ming Chiao Tung University*

Yu-Cheng Yang
*National Yang Ming Chiao Tung University*

Chi-Yu Li
*National Yang Ming Chiao Tung University*

Chun-Ying Huang
*National Yang Ming Chiao Tung University*

## Abstract

Bluetooth Low Energy (BLE) is a dominant wireless communication technology widely used in low-power, short-range applications. Its broad adoption and inherent security vulnerabilities in certain implementations have prompted numerous efforts to identify flaws in BLE protocol stacks. Despite these efforts, many existing fuzz testing methods face substantial limitations in scalability and applicability. To address these challenges, we propose BLuEMan, a simulation-based fuzzing framework that integrates a Real-Time Operating System (RTOS) with a software-based physical layer simulator. BLuEMan executes the actual BLE protocol stack while simulating interactions between BLE targets. This design ensures scalability for rapid testing across various targets while maintaining high applicability to various platforms. Our evaluation demonstrates that BLuEMan achieves fuzzing rates up to 18.0 and 162.3 times faster than typical simulation-based and platform-based approaches, respectively. Moreover, BLuEMan has uncovered four new vulnerabilities in BLE protocol stacks, all of which have been reported and assigned CVEs. This approach provides valuable insights into efficient vulnerability discovery for BLE protocol stack developers.

## 1 Introduction

The rapid expansion of the Internet of Things (IoT) and wearable devices has driven a significant increase in demand for wireless communication technologies. In particular, most short-range communication for these battery-constrained devices relies on Bluetooth Low Energy (BLE), a low-power communication technology. According to market research from the Bluetooth Special Interest Group [9], annual shipments of Bluetooth devices have reached billions, with over 80% of them supporting the BLE protocol. This widespread adoption highlights the popularity and critical role of BLE in modern technological ecosystems.

The widespread adoption of BLE devices has brought significant attention to their security vulnerabilities. Research has uncovered vulnerabilities in Bluetooth protocol stacks across different versions, with some flaws enabling Denial-of-Service (DoS) attacks and code execution. Prominent examples include the Blueborne [5], BlueFrag [21], and Bleedingbit [4] vulnerabilities, which have been shown to allow malicious actors to exploit affected devices. These security issues highlight the critical need for effective detection methods within the Bluetooth protocol, driving numerous studies [15, 16, 18, 20, 23, 28, 31] to develop fuzz testing methodologies specifically tailored for Bluetooth stacks.

However, current fuzz testing solutions for BLE devices face limitations in scalability and applicability. Broadly, these solutions can be categorized into two approaches: platform-based approach and emulation-based. Platform-based approaches rely on specific hardware or dedicated OS environments to conduct fuzzing tests. While they offer high fidelity through direct interaction with actual platforms, they lack scalability for testing across target platforms. Emulation-based methods simulate the behavior of hardware or system environments, providing scalability for testing, but they often fall short in fully replicating the behavior of real targets and are typically restricted to specific systems or hardware chipsets.

To address the requirements for scalability and applicability in fuzz testing BLE protocol stacks, we propose BLuEMan, a stateful simulation-based full-stack fuzzer designed for open-source BLE stack implementations. BLuEMan leverages a Real-Time Operating System (RTOS) to execute the actual BLE protocol stack, ensuring both high fidelity and scalability. It incorporates a software-based physical layer simulator to simulate interactions between BLE targets. This framework supports various RTOS platforms with high applicability, including Zephyr [37], NimBLE [3], and BTstack [7], as well as all BLE stacks ported to these environments.

BLuEMan is also distinguished by its high compatibility, flexibility, traceability, and portability: (1) Compatibility: The framework compiles into a single ELF executable, allowing seamless integration with existing debugging, instrumentation, and state monitoring tools. (2) Flexibility: By introducing a stackable mutation architecture, BLuEMan can fuzz different

protocol layers and support customized protocols, making it highly adaptable. (3) Traceability: Using a novel Man-In-The-Middle (MITM) architecture, BLuEMan deploys a packet interceptor that facilitates comprehensive fuzz testing and enables fine-grained traceability. (4) Portability: BLuEMan leverages a Linux Foundation-backed open-source framework and supports BLE protocol stacks portable to the Nordic NRF52/53/54-series simulated boards with BLE hardware.

We evaluated BLuEMan by comparing it against prior notable Bluetooth fuzzers. BLuEMan demonstrated superior speed and flexibility, primarily due to its reliance on a physical-layer simulator. Specifically, it achieves fuzzing rates up to 18.0 and 162.3 times faster than typical simulation-based (BTFuzz [23]) and platform-based (SweynTooth [16]) approaches, respectively. In addition, BLuEMan's field-aware mutation method outperforms traditional AFL-based mutation in edge coverage, yielding improvements ranging from 6.14% to 256.49% on the standard BLE stack.

More importantly, BLuEMan identified four new vulnerabilities across different layers of the BLE protocol stack, all of which have been assigned CVEs. We dedicated approximately three months collaborating with the developers involved to validate the identified vulnerabilities, reproduce them on their platforms, and ensure they were properly patched. This work offers valuable insights into efficient vulnerability discovery for BLE protocol stack developers.

The remainder of this paper is organized as follows. Section 2 presents the necessary BLE background. Section 3 presents an overview of BLuEMan's objective, design, and real-world impact. Sections 4, 5, and 6 design, implement, and evaluate BLuEMan, respectively. Section 7 discusses the limitations and potential extensions. Section 8 presents related work, and Section 9 concludes the paper.

## 2 BLE Primer

BLE is a low-energy, short-range wireless communication technology introduced as part of the Bluetooth 4.0 core specification [8]. It supports four primary device modes. (1) Broadcaster: periodically broadcasts packets of data to nearby BLE observers without requiring a formal connection. (2) Observer: listens for and receives broadcast messages from other BLE devices. (3) Peripheral: sends advertising packets and awaits incoming connections from central devices, enabling periodic data exchange and facilitating interactive apps; (4) Central: acts as the main controller, responsible for discovering, connecting to, and managing peripheral devices; it structures data exchange with peripherals using the GATT (Generic Attribute Protocol). The broadcaster and observer modes are commonly applied in location-based services, asset tracking, and event management, while the peripheral and central modes are integral to wearables, medical devices, smart home technologies, and similar apps.

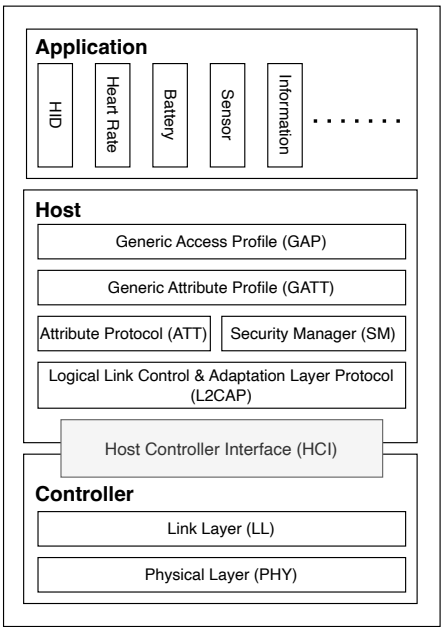


Figure 1: The BLE protocol stack.

**BLE Protocol Stack.** Figure 1 illustrates the BLE protocol stack, comprising the app, host, and controller layers from top to bottom. The app layer offers various services, such as HID (Human Interface Device), heart rate monitoring, and battery status reporting. The host manages high-level protocols, data exchange, and app logic, while the controller handles low-level, timing-critical operations at the radio and link layers. The host and controller communicate via the HCI (Host Controller Interface). Depending on the implementation, they can be located on separate chips (connected via a physical interface such as UART (Universal Asynchronous Receiver-Transmitter) or USB) or integrated on the same chip, using shared memory for interaction.

The host contains five major components: GAP (Generic Access Profile), GATT (Generic Attribute Profile), ATT (Attribute Protocol), L2CAP (Logical Link Control and Adaptation Layer Protocol), and SM (Security Manager). GAP defines device modes, manages device discovery, and handles the connection process. Once connected, GATT enables structured data exchange using the ATT for attribute-level operations. L2CAP provides data multiplexing for higher layers, ensuring efficient data transmission over the BLE link. SM ensures device security by managing authentication, encryption, and data integrity to secure BLE communication. On the other hand, the controller consists of the PHY (Physical) and LL (Link Layer) layers. The PHY layer operates on the 2.4 GHz ISM band and employs various modulation techniques, whereas the LL layer manages packet scheduling, role management, and timing parameters, such as connection intervals, advertising intervals, and window lengths.

**BLE Packet Type.** BLE packets are categorized into two

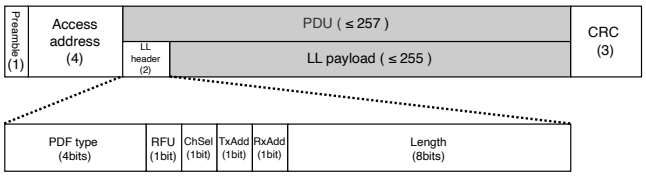



Figure 2: BLE advertising packet format.

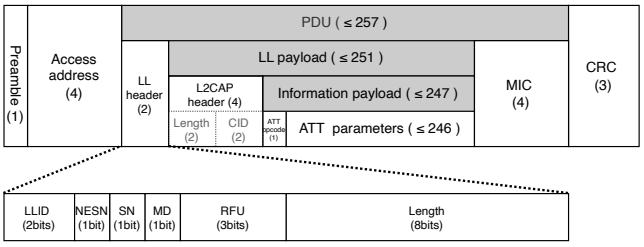



Figure 3: BLE data packet format: ATT as an example.

main types: advertising packets and data packets. Advertising packets are used during the advertising and scanning phases of BLE communication. They enable devices to announce presence and capabilities, broadcast information, and establish connections. Figure 2 illustrates the advertising packet format, which mainly includes a constant access address of `0x8E89BED6`, an advertising type specified in the PDU type field of the LL header, and the LL payload length field.

Data packets facilitate communication between a central device and a peripheral during an established connection. Figure 3 shows an example of an ATT data packet. In the LL header, the LLID field indicates the packet type (L2CAP or LL control), whereas the length field specifies the total length of the LL payload plus the MIC (Message Integrity Check). The LL payload comprises the L2CAP header and information payload. The CID (Channel ID) in the L2CAP header identifies the protocol of the information payload (e.g., `0x0004` for ATT). The information payload includes an ATT opcode (operation code), defining the operation type (e.g., request and response) and related parameters.

## 3 Overview

Current research on BLE stack fuzzing can be broadly classified into platform-based [15, 16, 28] and emulation-based [20, 23, 31] approaches. Platform-based approaches rely on specific hardware or dedicated OS environments as the foundation for conducting fuzzing tests. Examples of platform-based fuzzing include using FPGAs (Field-Programmable Gate Arrays) or specific IoT devices to identify vulnerabilities in embedded systems. The OS fuzzing examples primarily target RTOS to uncover software vulnerabilities. These approaches excel at providing high fidelity by closely simulating real-world scenarios and interacting directly with actual platforms.

However, platform-based solutions face significant scalability challenges for two primary reasons. First, each target platform must be individually acquired, configured, and integrated into a testing environment, which can be both resource-intensive and complex. Second, extracting critical runtime state information from target platforms often requires extensive reverse engineering and the development of sophisticated tools, such as internalblue [26].

On the other hand, emulation-based methods utilize software to simulate the behavior of hardware or system environments. Such emulators are designed to provide controlled, flexible, and often faster testing without requiring actual hardware. For example, the QEMU emulator can replicate entire system architectures for running virtualized OS environments, where security vulnerabilities in specific apps and protocols can be tested and identified. These approaches are highly scalable, allowing rapid testing across multiple targets and offering flexibility to explore various configurations automatically within the emulator. Moreover, they often provide simpler access to runtime state information.

Nevertheless, emulation-based solutions are not without their limitations. The primary drawback is that emulated targets may not fully replicate the behavior of actual protocols and implementations, possibly overlooking some vulnerabilities. Moreover, their applicability is restricted to specific systems and hardware chipsets. For example, the approach by Huster et al. [20] can only test systems that support VirtIO [32], such as the Linux OS. Similarly, Frankenstein [31] is confined to particular chipsets, such as Cypress and Broadcom Bluetooth, and requires substantial effort to reconstruct runtime state information.

To address these limitations, we propose a simulation-based fuzzing framework, BLuEMan, for testing BLE protocol stacks. BLuEMan integrates the advantages of both platform-based and emulation-based approaches while tackling three major *challenges*: (1) generating high-quality seeds for effective fuzzing; (2) ensuring runtime scalability in terms of environment setup cost, coverage measurement, fuzzing speed, and crash verification; and (3) accommodating the increasing complexity of Bluetooth protocol development, which demands more sophisticated and resource-intensive fuzzing strategies. To overcome these challenges, BLuEMan introduces three *novel* methods: (1) employing an MITM architecture to generate high-quality seeds from real BLE apps; (2) utilizing the single ELF model in RTOSs to enable ultra-fast fuzzing with scalable performance; and (3) exploring protocol states via probabilistic mutation of packet sequences, eliminating the need to emulate complex BLE state machines.

Specifically, BLuEMan utilizes an RTOS to execute the actual BLE protocol stack, achieving both high fidelity and scalability. High fidelity is ensured by testing the real protocol implementation, while scalability is enabled by leveraging the flexibility of RTOS software simulator to facilitate rapid testing across various targets. Moreover, BLuEMan incorporates a software-based PHY simulator to simulate interactions at the physical layer between BLE targets, such as a BLE central

Table 1: Overview of the discovered vulnerabilities.

| Layer | CVE | Vulnerability Type | Elapsed Time |
|---|---|---|---|
| LL | CVE-2023-4424 | Buffer overflow | 5 m 24 s |
| ATT | CVE-2024-3077 | Integer underflow | 3372 m 15 s |
| SM | CVE-2024-3332 | Race condition | 1 m 10 s |
| LL | CVE-2024-4785 | Divide by zero | 1943 m 44 s |

and a BLE peripheral.

BLuEMan is aimed at testing open-source BLE protocol stacks. It currently supports a variety of RTOS platforms with integrated BLE stacks, including Zephyr [37], NimBLE [3], and BTstack [7]. BLuEMan's flexible design can be applied to any open-source BLE protocol stack that can be ported to these RTOS environments.

**Objectives and Design Components.** To meet the requirements of scalability and effective fuzz testing, BLuEMan is designed with the following objectives and the corresponding components:

- **Efficiency**: BLuEMan executes actual BLE protocol stacks without the overhead associated with interacting with actual platforms or processing within a simulation environment. This design inherently enables faster fuzzing than traditional methods.
- **Compatibility**: BLuEMan compiles the entire RTOS, BLE stack, and PHY simulator into a single ELF executable. This integration simplifies compatibility with existing development tools, such as debugging, instrumentation, and state monitoring.
- **Flexibility**: BLuEMan incorporates a stackable mutation architecture within its packet mutator. This architecture identifies protocol layers and performs mutations based on customizable fuzzing weights, offering flexibility for fuzzing different layers. Moreover, it is extensible, allowing support for custom protocols by implementing each as a plugin within the stackable architecture.
- **Traceability**: BLuEMan employs a novel MITM architecture, deploying a packet interceptor by patching the bridging interface at the PHY simulator. This approach not only enables comprehensive fuzz testing but also ensures fine-grained traceability for monitoring and analysis.
- **Portability**: The design of BLuEMan leverages a popular open-source framework. Furthermore, by using Nordic NRF52/53/54-series simulated boards with BLE hardware, our proposed approach is capable of assessing any BLE stack that can be ported to the hardware.

**Real-world Impact.** We discovered four new vulnerabilities using BLuEMan, all of which have been assigned CVEs, as detailed in Table 1. These vulnerabilities span multiple protocol layers, including LL, ATT, and SM, and involve various types of vulnerabilities: buffer overflow, integer underflow, race condition, and divide by zero. These findings highlight

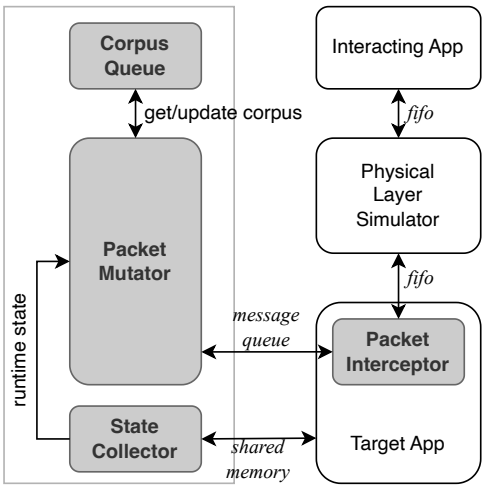


Figure 4: BLuEMan architecture.

the effectiveness of BLuEMan in identifying vulnerabilities across different layers without the overhead of managing complex state machines [15, 16, 23, 28]. Notably, all the identified CVEs can remotely cause DoS in the affected BLE protocol stacks, with three being exploitable without requiring pairing. This is primarily due to incorrect code behavior or memory corruption, such as buffer overflows in the BLE reception buffer. Typically, a device crash results in an automatic restart. However, in the worst-case scenario, the targeted device may require a manual hardware reset to resume normal operation, depending on the implementation of its hard fault handling mechanism [16]. All the identified CVEs have been reported to the respective developers and subsequently patched.

## 4 BLuEMan Design

In this section, we introduce the BLuEMan framework and design components. Figure 4 shows its architecture overview. We employ a *PHY layer simulator* for packet delivery between BLE devices for better fuzzing performance. All the BLE apps, the stack, and the PHY layer bridging interfaces are compiled as a single ELF executable. The gray boxes in the figure indicate the new components introduced in the framework:

- **Packet Interceptor**: Sits between the interacting app and the target app, intercepting packets sent to the target and forwarding packets from the target back to the interacting app.
- **State Collector**: Collects runtime information from the target app.
- **Corpus Queue**: Stores effective test cases.
- **Packet Mutator**: Generates fuzzing test cases by mutating received packets.

When two BLE devices' ELFs are ready, we invoke them

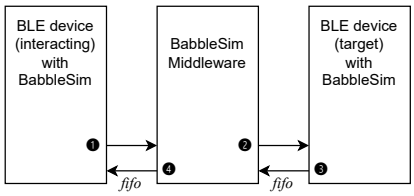
to communicate with each other. The packet interceptor intercepts a packet sent from the interacting app to the target app and forwards the packet for mutation. A mutated packet (test case) is fed to the target app to perform a fuzzing task. The mutation is performed on either the packet's payload or a selected test case from the corpus queue. Once the target app has processed a mutated packet, the state collector collects its runtime information, e.g., code coverage, and determines whether the test case is effective. The corpus queue is updated based on the state collector's judgment.

Our fuzzing framework features a novel design that performs mutations on payloads captured from interactions between apps. This approach achieves two key goals: (1) generating high-quality initial seeds and (2) enabling deeper exploration of protocol states. Prior research has shown that the quality of initial seeds significantly impacts fuzzing effectiveness [19, 20]. To this end, our design leverages interactions between different combinations of target and interacting apps to produce effective initial seeds and enhance testing coverage. An initial seed, also referred to as a test case or corpus throughout this paper, is composed of a sequence of BLE packets captured during communication between an interacting app and a target app. For example, in a heart-rate measurement app, interactions between the peripheral (interacting) and central (target) apps yield protocol messages at various layers, such as L2CAP and ATT, which are then used for fuzzing.

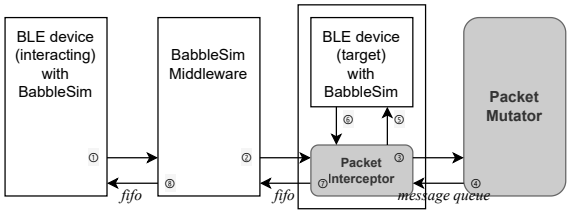
To support deeper state exploration, the interacting apps are implemented based on BLE protocol specifications, ensuring comprehensive coverage of BLE states. Mutations are applied to randomly selected packets and protocol layers within these captured interactions. By integrating stateful payloads generated from real protocol exchanges with a stateless fuzzer, BLuEMan forms a stateful fuzzing framework capable of targeting BLE protocol implementations at any state. To maintain robustness and simplify vulnerability detection, the fuzzed ELF executables are restarted in each fuzzing round, preventing the accumulation of program error states.

## 4.1 Packet Interceptor

The packet interceptor plays a crucial role in the proposed framework. We integrate our approach with a PHY simulator to capture packets exchanged between the involved apps. By using a simulator, we avoid the complexity of over-the-air packet capture and gain a streamlined mechanism for packet interception and manipulation. The packet interceptor is implemented by patching the bridging interface offered by the PHY simulator–specifically, by modifying the packet-receiving routine to invoke the interceptor before the packet is returned. The interceptor routine operates in a blocking manner: it forwards the intercepted packet to the packet mutator and waits for the mutated payload to be returned. This design grants the packet interceptor complete control over the data




(a) The typical packet flow.




(b) Work with the packet interceptor.

Figure 5: Packet flow for the PHY layer simulator.

passed from the PHY simulator to the mutator.

Figure 5 illustrates the packet flow with and without the packet interceptor. In the default setup (Figure 5a), a packet sent from the interacting app to the target app travels through the PHY simulator middleware and follows a direct return path to the sender, as indicated by the black circled steps 1 to 4. In contrast, with the packet interceptor enabled (Figure 5b), the packet is intercepted after transmission (steps 1 to 3), sent to the mutator, and then returned to the target app for processing (steps 4 and 5). The packet is then processed by the BLE stack integrated with the target app, and a response is generated and sent back to the sender (steps 6 to 8).

## 4.2 State Collector

The state collector is designed to provide sufficient information to guide the fuzzing process in accurately selecting more effective test cases from the corpus queue. Our framework achieves this by collecting runtime state information from the target BLE stack under test. A key advantage of using a PHY simulator is its ability to compile the entire RTOS, Bluetooth stack, and simulator into a single ELF executable, facilitating seamless integration with development tools such as debuggers, instrumentation utilities, and state monitors.

In our framework, we port AFL's edge code coverage mechanism into the state collector and perform target instrumentation using `afl-gcc`, thereby enabling edge coverage collection. To focus the fuzzing effort, runtime state collection is restricted to BLE-relevant components by patching the compiler to instrument only selected source code directories.

Beyond edge coverage, researchers can customize the framework to collect other types of coverage by modifying the coverage collection module. Moreover, sanitizers like UBSan (UndefinedBehaviorSanitizer), MSan (MemorySanitizer),

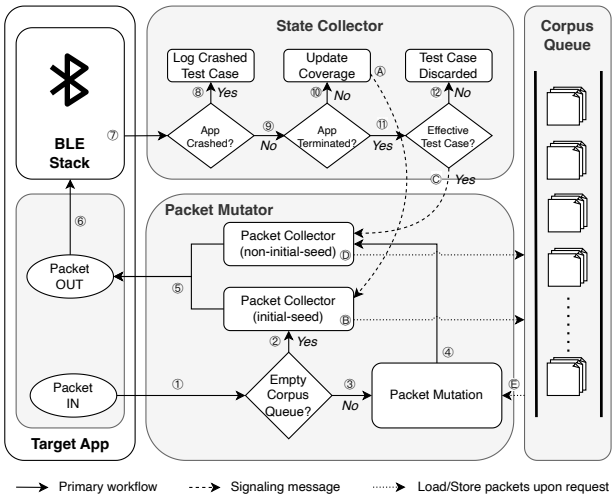



Figure 6: The packet-driven workflow of the packet mutator.

and ASan (AddressSanitizer) can be enabled to improve the detection of undefined behavior and memory corruption errors.

## 4.3 Corpus Queue

The corpus queue stores candidate test cases for fuzzing. Initially empty, it is populated with effective test cases as the fuzzing process progresses. Each entry in the corpus queue represents a sequence of multiple BLE packets collected in the same fuzzing round. Thanks to the packet interceptor, valid BLE packets exchanged between the interacting and target apps are captured early in the fuzzing process and added as initial candidates. Subsequent test cases are derived from these initial entries to further expand the corpus.

Only test cases that do not crash the target app are retained in the corpus queue. If a test case causes a crash, it is separately logged and preserved for further analysis and validation.

## 4.4 Packet Mutator

The packet mutator is the most critical component of the framework. We propose a modular, packet-driven workflow and a stackable mutation architecture to enable an efficient and flexible fuzzing process. The modular workflow ensures streamlined execution, while the stackable architecture allows weighted fuzzing based on identified protocol layers. This process is illustrated in Figure 6. It is triggered upon receiving an input packet ①. Depending on whether the corpus queue is empty, the mutator either collects an initial seed ② or performs packet mutation ③. All packets processed by the mutator are duplicated and stored in a packet collector, which is later used to maintain the fuzzing corpus. The output packet ⑤ is then sent to the target app ⑥ to fuzz the BLE stack. After the packet is processed, the target app's runtime state is recorded by the state collector. The test case used may

**Algorithm 1** The Packet Mutation Algorithm

**Input:** $pkt$: Input packet (original).
**Input:** $w_x$: Weight for not performing mutation.
**Input:** $L$: List of available protocols. Each element is in the form of $\{R, w, M\}$, where $R$ is the recognization function, $w$ is the weight for mutation, and $M$ is the mutation function.
**Output:** $pkt'$: Output packet (mutated).

L' $\leftarrow \varnothing$ ▷ The array containing the detected protocols.
**for** $p \in L$ **do**
  **if** $p.R(pkt)$ is **true then**
    $p.w \leftarrow$ UpdateWeight$(p)$
    add $p$ to $L'$
  **end if**
**end for**
$n \leftarrow$ number of elements in $L'$
add $\{\varnothing, w_x, \varnothing\}$ to $L'$
W $\leftarrow \sum p.w \; \forall p \in L'$
$m \leftarrow -1$ ▷ Mutation index
$pr \leftarrow$ draw a probability from $[0, 1]$
$w_L \leftarrow 0$
**for** $i = 0$ **to** $(n-1)$ **do**
  $w_U \leftarrow w_L + L'[i].w/W$
  **if** $w_L \leq pr < w_U$ **then**
    $m \leftarrow i$ ▷ Assign mutation index
  **end if**
  $w_L \leftarrow w_U$
**end for**
**if** m $> -1$ **then**
  $pkt' \leftarrow L'[m].M(pkt)$
**else**
  $pkt' \leftarrow pkt$
**end if**
**return** $pkt'$

be logged ⑧, used to update coverage ⑩, or discarded ⑫, depending on whether the runtime state indicates a crash, termination, or effective behavior (i.e., increased coverage).

During the initial seed collection process, the state collector updates coverage information for each processed packet in the packet collector Ⓐ. This process halts if no new coverage is observed in the most recent $n$ packets. For non-initial-seed packets, the coverage state is only reported when the target app terminates normally at the end of a fuzzing round Ⓑ. In both cases–either when initial seed collection halts or a fuzzing round ends–the packets stored in the packet collector are treated as a complete test case and added to the corpus queue (Ⓒ and Ⓓ).

To enhance fuzzing flexibility, we introduce a stackable mutation architecture capable of recognizing protocol layers and applying mutations based on preferred fuzzing weights. Each supported protocol is implemented as a plugin, which includes: a protocol recognition function ($R_i$), a mutation weight

($w_i$), and a protocol mutation function ($M_i$). The recognition module identifies the presence of a specific protocol header and payload in the packet. If the protocol is detected, a fuzzing probability is sampled to determine whether its header or payload should be mutated. The corresponding protocol mutation module then performs the actual mutation.

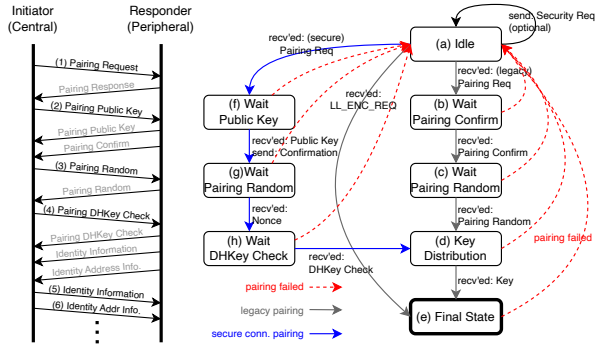
The stackable mutation architecture allows researchers to design and implement fuzzers for unsupported or customized protocols. In the case of BLE fuzzing, we provide implementations for the LL, L2CAP, SMP, and ATT protocols. Notably, each protocol fuzzer can be configured to target the protocol header, payload, or both. In general, we apply field-aware mutations to protocol headers to ensure that mutated packets can pass the basic validation checks regulated by the protocol specifications.

Specifically, BLuEMan incorporates mutation operations from traditional fuzzers, such as the havoc operations from AFL, and enhances their effectiveness through three lightweight mutation constraints: (1) restricting bitwise mutations to the Link Layer, (2) keeping the upper-layer protocol fields (e.g., the LLID field in the LL protocol) unchanged, and (3) adjusting length fields in response to changes in payload size. The traditional mutators promote testing diversity, while the added constraints help maintain protocol correctness.

The packet mutation algorithm is detailed in Algorithm 1. Given an input packet $pkt$, a weight $w_x$ representing the probability of skipping mutation, and a list of available protocols $L$, the algorithm returns either a mutated packet or the original input packet, based on the assigned weights and a sampled probability. Each protocol in the list $L$ is represented as a tuple $\{R, w, M\}$, where $R$ is the protocol recognition function, $w$ is the mutation weight, and $M$ is the mutation function.

The algorithm starts by determining which protocols are present in the input packet (lines 1-7), storing all detected protocols in a separate array, $L'$. To simplify implementation, a dummy protocol $\{\varnothing, w_x, \varnothing\}$ is appended to the end of $L'$ (line 9). The total mutation weight $W$ is then calculated by summing the weights of all protocols in $L'$ (line 10). Next, the algorithm samples a probability from the range $[0,1]$, determines which protocol to mutate, and stores the selection in an index variable $m$ (lines 11-20). While line 12 draws a uniform random probability, the actual likelihood of selecting each protocol is influenced by the UpdateWeight function, which adjusts weights based on the runtime coverage of each protocol layer. Finally, the algorithm invokes the mutation function $L'[m].M$ on the input packet $pkt$. If no protocol is selected, the original input packet is returned unchanged.

It is important to note that the fuzzing speed of the packet-driven workflow largely depends on the interaction frequency between the selected interacting app and the target app. To maintain progress, if no interaction is detected within a specified timeout threshold (2 s by default in this study), both apps are terminated and restarted for a new fuzzing round. For further details on fuzzing speed, please refer to Section 6.1.




(a) Packet Sequence. (b) Simplified State Machine.

Figure 7: Example of SM pairing: packet sequence and state machine mapping.

## 4.5 Packet-driven State Machine Traversal

One key novelty of BLuEMan is its ability to explore protocol states without emulating BLE state machines. It takes a packet-driven approach to state machine traversal by associating packet sequences with state transitions. To illustrate this, we use the SM pairing process as an example.

Figure 7a shows the LE Secure Connections Pairing packet sequence using the Just Works method, which is naturally generated by the `sm_pairing` apps during interactions between the central and peripheral apps. In contrast, Figure 7b presents a simplified peripheral state machine for LE pairing using the same method. Starting from the idle state, the peripheral transitions through states (a), (f)–(h), and (d)–(e), guided by the packet sequence (1)–(6) produced within BLuEMan. Our mutation strategy probabilistically selects different packets and protocol layers from these live interactions to mutate, enabling exploration of any state reachable by the BLE apps under test.

A compelling example of the effectiveness of this approach is the discovery of CVE-2024-3332 (see Appendix B.3). Although the root cause lies in a race condition between the HCI and SM pairing protocol implementations, the vulnerability is only triggered when the SM protocol reaches the "Key Distribution" state, highlighting the practical value of our method.

# 5 Implementation

Our proposed framework is integrated with a selected RTOS and a PHY simulator. Unless otherwise specified, we adopt the Zephyr RTOS, a general-purpose RTOS maintained by the Linux Foundation, which supports over 750 boards. Among the available open-source simulators, BabbleSim [6] and RootCanal [17] are two widely used options. Based on the completeness of their BLE implementations, we choose BabbleSim as the PHY simulator to capture packet exchanges between apps. Both BabbleSim and Zephyr are patched to sup-

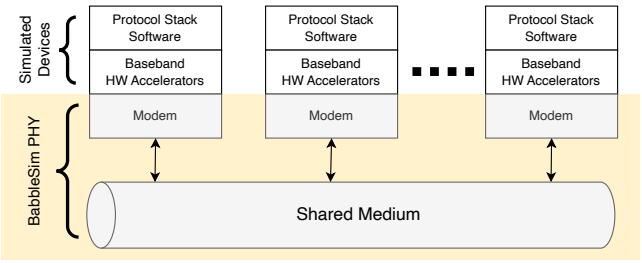
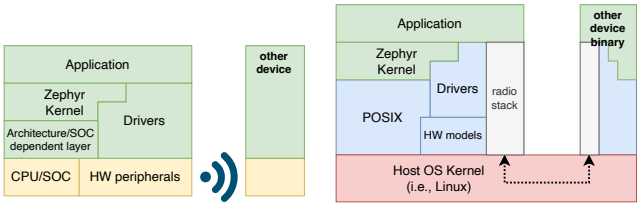



Figure 8: BabbleSim PHY: shared medium and device interface.




(a) With physical hardware. (b) with POSIX API.

Figure 9: Zephyr architecture comparison [36].

port the BLuEMan framework. BLuEMan also incorporates instrumentation, coverage collection, and mutation operators adapted from the AFL fuzzer.

In this section, we briefly introduce BabbleSim and discuss the necessary integration steps. BLuEMan is publicly available at https://doi.org/10.5281/zenodo.15601101 and https://github.com/zoolab-org/blueman.artifact.

**BabbleSim.** BabbleSim is a simulator designed for shared medium networks at the physical layer, where the shared medium represents wireless communication in the 2.4GHz ISM band. The BabbleSim PHY, illustrated in Figure 8, is divided into two main components: (1) the shared medium and (2) the simulated modems that use BabbleSim's library functions. Programs with these simulated modems are referred to as simulated devices. The propagation in the simulated medium closely replicates real-world wireless communication in the 2.4GHz ISM band, with BabbleSim's library enabling simulated devices to communicate wirelessly through the shared medium. Multiple simulated devices can connect to a single simulated medium.

BabbelSim supports compiling device and app codes into ELF executables, allowing it to run on Linux systems. This feature provides developers with flexibility and ease of use, enabling direct simulation and testing of wireless communication protocols within a Linux environment. BabbleSim allows developers to harness the powerful capabilities of the Linux platform for efficient development and testing.

Through BabbleSim's modular library, developers can implement communication protocols for the 2.4GHz ISM band, such as Bluetooth Low Energy, Thread, and 6LoWPAN. A notable project, *ext_nRF_hw_models*, involves developing software-simulated peripherals based on nRF5-series hardware models, including wireless protocols such as Bluetooth Low Energy and IEEE 802.15.4.

We integrate our packet interceptor with the radio stack of the BabbleSim simulator. It is done by invoking the packet interceptor function from within the `start_Rx` routine before the `start_Rx` routine returns a packet. The implementation of the packet interceptor contains 168 lines of C codes, and the patch contains only a single line function call to the BabbleSim routine.

**Integration.** We further adopt the PHY simulator in the targeted RTOS and BLE protocol stack implementation. Specifically, we evaluate our approach using two RTOS platforms, Zephyr [37] (supported by the Linux Foundation) and Mynewt [3] (supported by the Apache Foundation), along with their built-in BLE protocol stacks. Other BLE protocol stacks, such as BTstack [7], that can be adapted to run on either Zephyr or Mynewt can also be evaluated using our framework.

We use Zephyr as an example to illustrate the integration of our framework with an RTOS. Figure 9 compares the Zephyr architecture on hardware versus a PHY simulator, i.e., BabbleSim. As shown in Figures 9a and 9b, Zephyr provides a POSIX-compatible implementation to replace architecture- and SoC[1]-dependent layers and CPU/SoC hardware. For instance, architecture-specific halt instructions are implemented using the POSIX API `exit` function. Additionally, Zephyr allows the driver to interface with the modem API (the radio stack in Figure 9b) provided by the BabbleSim project. By substituting hardware peripherals with software-simulated radios, each Bluetooth device/app is treated as a standalone app running in a typical OS environment, suitable for fuzzing.

## 6 Evaluation

In this section, we evaluate BLuEMan from multiple perspectives and summarize the results in the form of research questions (RQ). We begin by describing the experimental setting and our comparison with existing Bluetooth fuzzers.

**Experimental Setting.** Experiments are conducted on a server equipped with an Intel(R) Xeon(R) Gold 5118 CPU and 48 GB RAM, running Debian 12 Linux OS. We select several pairs of interacting and target apps, as summarized in Table 2. Two key points are worth noting. First, the selected BLE protocol stacks were developed by three independent teams, demonstrating the portability of BLuEMan. The corresponding source code can be found in the `samples/bluetooth` directory of the Zephyr project, the `babblesim/targets` directory of the NimBLE project, and the `example` directory of the BTstack project. Second, some BLE apps may require

[1] SoC: System on Chip

Table 2: BLE app pairs used for experiments.

| Stack | Name (App #1) | (App #2) |
|---|---|---|
| BTstack | Battery Query (gatt_battery_query) | (gatt_counter) |
| | SM Pairing (sm_pairing_central) | (sm_pairing_peripheral) |
| | LE Credit (le_credit_based_flow_control_mode_client) | (le_credit_based_flow_control_mode_server) |
| NimBLE | GATT (blecent) | (bleprph) |
| Zephyr | GATT Write (central_gatt_write) | (peripheral_gatt_write) |
| | OTS (central_otc)† | (peripheral_ots) |
| | Heart Rate (central_hr) | (peripheral) |
| | ISO Broadcast (iso_broadcast) | (iso_receive)§ |
| | BLE Mesh (mesh)† | (mesh_provisioner)†§ |

†: Replace hardware-dependent functions with mock ones.
§: Only application #2 in this pair is fuzzed.

physical hardware (e.g., a button) to activate specific features; for such cases, we patch the apps to enable those features programmatically using mock functions, removing the need for physical components.

**Comparison with Existing Bluetooth Fuzzers.** We summarize key distinctions by comparing BLuEMan with prior notable Bluetooth fuzzers in Table 3. An experimental comparison with BrakTooth is presented in Section 6.5. Direct comparisons with other solutions are not feasible due to differences in target protocols or the unavailability of the required platform or hardware, as presented below.

Platform-based approaches, including SweynTooth [16], BrakTooth [15], and L2FUZZ [28], typically use a dongle to interact with target devices. To estimate fuzzing progress, they develop custom state machines that track the hardware state by mapping received messages from the target device to corresponding protocol states. For experimental evaluation, SweynTooth is not publicly available–only proof-of-concept attacks have been released–and L2FUZZ is incompatible with the BLE devices we target. However, BrakTooth can be executed using the ESP32-WROVER-KIT platform, so it is chosen for the experimental comparison.

Emulation-based fuzzers include Frankenstein [31] and VirtFuzz [20], whereas BTFuzz [23] is a simulation-based fuzzer like BLuEMan. Frankenstein uses QEMU user-mode as its primary emulation engine, targeting Broadcom and Cypress Bluetooth firmware; however, the source code required by BLuEMan is unavailable. VirtFuzz employs VirtIO to fuzz Bluetooth protocols above the HCI layer, which differs from BLuEMan's protocol entry point. Moreover, VirtIO requires a guest OS to implement specialized drivers for their interaction, which are typically unavailable on most RTOSs. BTFuzz, as a simulation-based approach, utilizes a state machine to monitor the fuzzing state of the target protocol stack. However, its HCI-based interface limits the fuzzing scope to host protocols.

Table 3: Summary of Bluetooth fuzzing research works.

| Name | Radio Type<br>Approach | Protocols<br>Description | Exploration Strategy |
|---|---|---|---|
| BLuEMan (Ours) | LE<br>Simulation-based | Host/Controller<br>Compile firmware into an ELF executable. | Code coverage (edge) |
| SweynTooth [16] | LE<br>Platform-based | Host/Controller<br>Use a dongle to interact with BLE devices. | Customized state machine |
| BTFuzz [23] | LE<br>Simulation-based | Host<br>Attach customized HCI interface to BLE host. | Customized state machine |
| BrakTooth [15] | BR/EDR<br>Platform-based | Host/Controller<br>Use a dongle to interact with BR/EDR devices. | Customized state machine |
| L2FUZZ [28] | BR/EDR<br>Platform-based | L2CAP<br>Use a dongle to interact with BR/EDR devices. | Customized state machine |
| Frankenstein [31] | BR/EDR/LE<br>Emulation-based | Controller<br>Emulate firmware using QEMU user-mode. | Code coverage (basic block) |
| VirtFuzz [20] | BR/EDR/LE<br>Emulation-based | Host<br>Full system emulation using QEMU/VirtIO. | Code coverage (edge) |

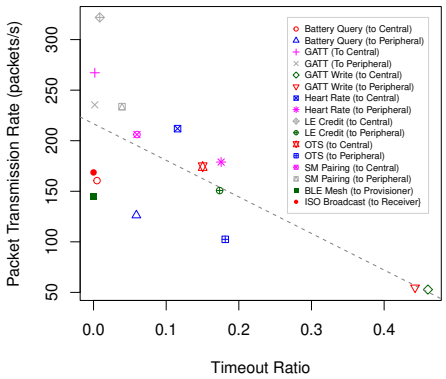



Figure 10: Scatter plot for the packet transmission rate and the corresponding timeout ratio for each app pair.

Note that, regarding initial seeds, Frankenstein starts with a null byte; BLuEMan, BrakTooth, and VirtFuzz use captured Bluetooth packets; while SweynTooth, BTFuzz, and L2FUZZ generate seeds from self-simulated protocol state machines.

## 6.1 RQ1: Fuzzing Speed

We discuss the fuzzing speed of our proposed approach due to the use of the packet-driven design. With this design, the fuzzer can only perform mutations when a packet passes through the packet interceptor and the mutator. As a result, the fuzzing speed is primarily determined by the packet transmission rate between the client and the server. We set a default timeout threshold of 2 seconds to detect whether a fuzzing process initiated by an interacting app is stalled. If a stall is detected, the involved apps are restarted. We run the selected BLE app pairs to perform the measurement. Each app pair is run for 24 hours.

Figure 10 presents the scatter plot of the packet transmission rate and the corresponding timeout ratio for each app pair.

Table 4: Packet transmission rates for BLE works.

| Name | Approach | Rate (packets/min) |
|---|---|---|
| SweynTooth [16] | Platform-based | 119 |
| BTFuzz [23] | Simulation-based | 1,073 |
| BLuEMan (Ours) | Simulation-based | 19,315 |

Each app is represented by a different symbol in the figure. The y-axis shows the average packet transmission rate measured during the experiments. At the same time, the x-axis represents the timeout ratio, defined as the number of stalls detected divided by the total number of invocations collected in the same experiments. The Pearson correlation coefficient for the measured average packet transmission rate and the timeout ratio is -0.74, indicating that the number of timeouts has a strong negative impact on the packet transmission rate.

App stalls do not always occur when invoking an app pair. We attempt to examine the root cause of app stalls. One primary reason is unhandled simulated hardware interrupts observed in a few apps. For example, we observed an unexpectedly lower transmission rate for the “GATT Write" app pair. Our in-depth investigation revealed that the behavior of building and running the app on a native host machine (e.g., an x86-based server) differs significantly from that on actual hardware. The stalls observed in the “GATT Write" app pair were caused by the packet interceptor not being triggered as expected by simulated hardware interrupts. The root cause of this issue is that the apps implement infinite loops to process BLE protocol messages, which prevents them from handling simulated hardware interrupts raised by the BabbleSim simulator. Following the official guidelines from the Zephyr project, we resolved this issue by inserting an additional `k_cpu_idle` function call within the infinite loops to yield the CPU. The recommended fix mitigates the issues and enables the PHY simulator bridging interface to handle hardware interrupts.

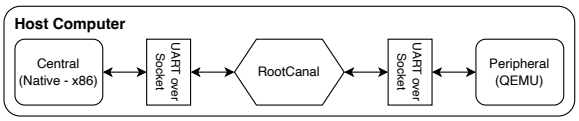
We compare the packet transmission rate of the BLE fuzzing research works reported from SweynTooth [16] and BTFuzz [23], as shown in Table 4. The numbers show that our proposed approach efficiently fuzzes the targeted BLE implementations. Note that BTFuzz does not explicitly describe its runtime setup; however, its PTY[2]-based interface design implies the need for an OS kernel running on physical or virtual hardware. While BTFuzz is not publicly available, we use it as the reference for emulation-based performance benchmarks.

## 6.2 RQ2: Performance of the Radio Media

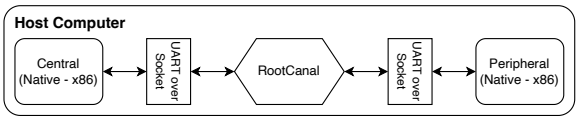
To have a fair comparison of the radio media performance for fuzzing, we evaluate the performance of selected BLE stack implementation by measuring the transmission performance over different radio media, including platform-based media, emulation-based media, and our proposed simulation-based media. The summary of the three setups is as follows.

[2]pseudo terminal.

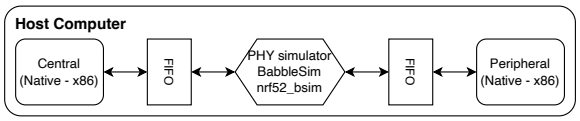

(a) Platform-based measurement over the air (nRF52840).




(b) Emulation-based measurement using the QEMU-based emulator (RootCanal).




(c) Simulation-based media using the Controller emulator (RootCanal).




(d) Simulation-based media using the PHY simulator (BabbleSim).

Figure 11: Architectural overview of the evaluated media.

- **Platform-based media.** We deploy our selected RTOS firmware and the target BLE stack to an nRF52840 dongle and perform the performance measurement over the air.
- **Emulation-based media.** An RTOS firmware and the target BLE stacks are run using QEMU user-mode emulation. The measurement inputs are sent to the target via the RootCanal framework. It is worth noting that only protocol layers above the HCI interface can be controlled due to the RootCanal design.
- **Simulation-based media.** Our proposed approach leverages an RTOS and a PHY simulator to perform performance measurements. The targets are compiled as ELF binary executables running on a server machine.

We use customized GATT Write central and peripheral apps originated from the Zephyr project to perform the benchmark. The architectural overview of the evaluated scenarios is shown in Figure 11. The host computer is a laptop with an AMD Ryzen HX 370 CPU and a MediaTek MT7922 Bluetooth chip running a Debian 12 Linux operating system. We measure the elapsed setup and transmission round-trip time on the central app. Each measurement iteration begins with the pairing of the two involved apps and ends with the first receipt of the response message for a GATT write request. We

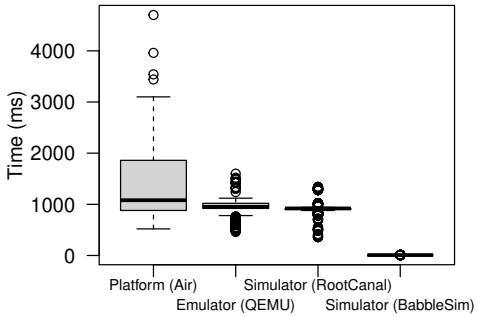



Figure 12: Measured performance for different radio media.

perform 1,000 iterations and summarize the results in Figure 12. The figure shows that the BabbleSim PHY simulator provides the most stable and efficient medium for transmitting Bluetooth packets, being approximately 100 times faster than other media. Surprisingly, the RootCanal controller simulator performs similarly whether using native code or an emulator. The elapsed time for pairing and packet transmission round-trip over the air is similar to that of the emulator. However, the signal strength fluctuates over time, leading to less stable performance.

## 6.3 RQ3: Fuzzing Coverage

We measure the fuzzing coverage for the evaluated apps to demonstrate the effectiveness of our proposed approach. Each selected app pair is fuzzed three times, with each experiment lasting 24 hours. We evaluate three different mutation strategies: field-aware mutation, AFL-only mutation and random mutation. For the field-aware mutation, the packet header is mutated within the bounds defined by the corresponding protocol specification, with selected fields set to either valid or invalid values. The payload, on the other hand, is mutated using random byte-level changes. For AFL-only mutation, we directly apply havoc mutators from AFL. To avoid immediate rejection, any packet exceeding the current connection's maximum allowed LL payload size is truncated. In the random mutation strategy, random byte mutations are applied to both the header and payload of the entire LL packet.

In the experiments, all app pairs swap the roles of the interacting and target apps, except for the BLE Mesh and ISO Broadcast apps. For the BLE Mesh app, we evaluate only the provisioner because it is more complex and is responsible for initializing and configuring the entire mesh network, while the provisionee only plays a passive role. For the ISO Broadcast app, we only test the receiver because the ISO receiver is a BLE observer that cannot send packets. Its sole function is to receive packets, making it unsuitable as an interacting app.

Figure 13 shows the coverage measured over time for all the evaluated apps. Due to space limitations, only a limited number of test cases are presented in the figure. Readers may refer

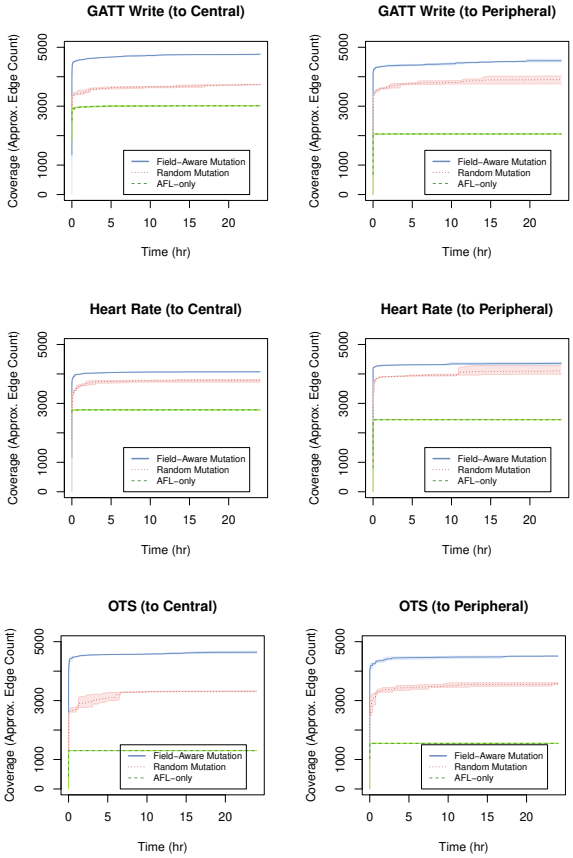



Figure 13: Coverage for the selected evaluated apps.

to Figure 17 in Appendix A for the rest of the plots. Among the 16 test cases, those using field-aware mutations demonstrated significant improvements in edge coverage compared to both random and the AFL-only mutators, with gains of 4.48%-40.05% and 6.14%–256.49%, respectively, excluding the BLE Mesh case.

Notably, in the BLE Mesh case, the three methods achieved comparable edge coverage, with differences of less than 5.64%. This is because, unlike the standard BLE stack, BLE Mesh is based on a separate specification developed on top of BLE and is not part of the core BLE specification. The advertising bearer used in our tests is built on the LL layer of the BLE stack, with additional layers specific to BLE Mesh's Network, Transport, and Access layers, but without the L2CAP, SM, and ATT layers found in the standard BLE stack. Since the field-aware mutator handles only the standard LL, L2CAP, SM, and ATT layers of the BLE stack, it does not offer any advantages for the BLE Mesh protocol layers above the LL layer.

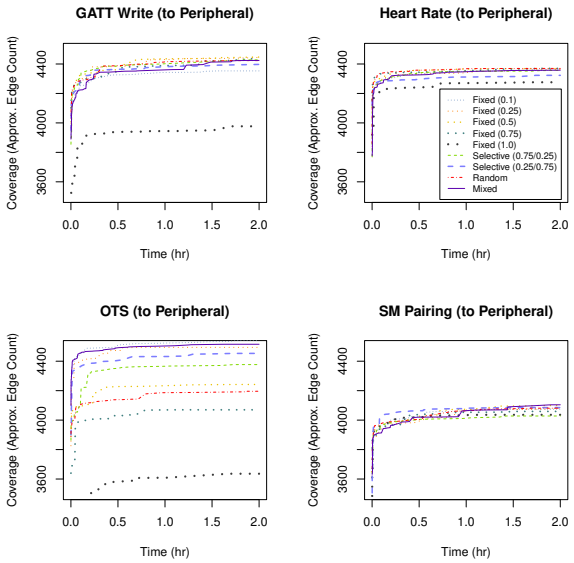



Figure 14: Coverage for different packet selection strategies.

## 6.4 RQ4: Packet Selection Strategy

We experimentally evaluate the impact of different packet selection strategies on coverage. For each selected packet sequence, packets are chosen for mutation using one of the following four strategies:

- **Fixed Probability**: A fixed mutation probability is uniformly applied to all packets in a sequence. We test several values, including 0.1, 0.25, 0.5, 0.75, and 1.0. A probability of 1.0 indicates all packets in the sequence are mutated.
- **Selective Probability**: A transition point within the packet sequence is identified to switch mutation probabilities. We use coverage information from initial corpus collection to locate the point with the most significant change in coverage. Two probability pairs are evaluated: (0.25, 0.75) and (0.75, 0.25), where the first value is applied before the transition point and the second value after.
- **Random Probability**: For each mutation, a probability is randomly chosen from the range [0.1, 1.0].
- **Mixed Strategy**: This strategy iteratively applies one of the above three methods. Each selected method is used for $N$ packet sequences before switching to the next; in our implementation, $N = 2000$.

We evaluate these strategies across four peripheral apps–GATT Write, Heart Rate, OTS, and SM Pairing–with results shown in Figure 14. The Fixed Probability strategy with a value of 1.0 generally performs poorly in most cases, as excessive mutation often prevents packets from passing basic protocol state validation, limiting deeper exploration of protocol states. In contrast, the other strategies show strengths depending on the app. Based on these observations, we recommend the Mixed strategy to achieve balanced and effective fuzzing performance.

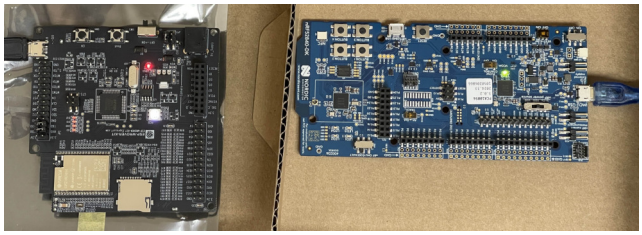
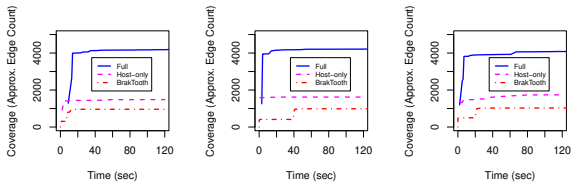


Figure 15: Experimental setup for running the BrakTooth tool. (Left) ESP-WROVER-KIT running BrakTooth firmware. (Right) nRF52840 kit running the Zephyr BLE stack and apps.




(a) GATT Write. (b) Heart Rate. (c) OTS.

Figure 16: Coverage and speed comparison between BLuEMan and BrakTooth. Legend: (Full) BLuEMan with full (host+controller) coverage collection; (Host-only) BLuEMan with BLE host-only coverage collection; (BrakTooth) BrakTooth coverage, which evaluates only BLE host protocols.

## 6.5 RQ5: Comparison with BrakTooth

We use the ESP32-WROVER-KIT and the nRF52840 development kit to run the BrakTooth firmware and the Zephyr BLE stack (supported by BLuEMan), respectively, as shown in Figure 15. Both devices are connected to the same PC for experiment execution. To minimize potential signal interference, the devices are enclosed in a shielding box.

For BrakTooth, we use the `bthost_fuzzer` tool from its repository, which supports fuzzing BLE peripheral devices. It automatically scans nearby devices and sends GATT commands to those it recognizes. It is limited to peripheral devices, as central devices do not respond to Bluetooth scanning.

We select three Zephyr peripheral apps–`gatt_write`, `heart_rate`, and `ots`–as targets for fuzzing by BrakTooth and BLuEMan. Each app is fuzzed for 5 hours due to hardware stability. During fuzzing, we collect runtime coverage from apps instrumented using a patched version of `AFL-gcc`.

It is observed that coverage reported by BrakTooth stops increasing after around 40 seconds, although each app is fuzzed for several hours. Thus, we plot only the first 120 seconds of coverage progression, as shown in Figure 16. It shows that our simulation-based method, combined with the MITM architecture, explores significantly more states in the target BLE host compared to the platform-based approach, BrakTooth.

### 6.6 RQ6: Real-World Vulnerability

This subsection discusses new vulnerabilities our proposed approach recognizes, as illustrated in Table 1. The recognized vulnerabilities span across different layers, i.e., LL, ATT, and SM. It demonstrates that our framework can effectively uncover vulnerabilities across various layers through its stateful exploration design. All the reported CVEs can remotely cause DoS in the affected BLE protocol stacks, with three being exploitable without requiring pairing (CVE-2023-4424, CVE-2024-3077, and CVE-2024-4785). Note that all the vulnerabilities discussed in this study have been reported to the corresponding developers and are fixed.

Below, we provide a brief introduction to each CVE, with full details presented in Appendix B due to space constraints.

**CVE-2023-4424.** This vulnerability can be triggered in Zephyr's LL implementation when processing BLE advertising packets. The flawed implementation incorrectly assumes that an advertising packet contains at least 6 bytes for the broadcast address, followed by advertising data. If the actual data length is less than 6 bytes, this assumption leads to an integer underflow. As a result, an attacker can broadcast malformed packets, enabling DoS attacks against any vulnerable BLE device that receives them. Affected devices must be rebooted to restore the BLE stack. The CVE was assigned a CVSS score of 8.8 (high) by the NVD, as it can be exploited remotely with low complexity, no privileges required, and no user interaction.

**CVE-2024-3077.** This vulnerability can be triggered in Zephyr's ATT layer implementation, causing a BLE device to crash due to an integer underflow issue. Specifically, this issue arises when handling an unexpected response to the ATT Find Information Request. An attacker can spoof or control a GATT server and respond with a valid protocol message of zero length, leading to out-of-bounds memory access. This type of vulnerability was found in multiple locations, affecting a total of five functions related to ATT response handling. The fix involves adding checks to ensure that the length of the ATT response meets the minimum size required for a valid Find Information Response.

**CVE-2024-3332.** This vulnerability occurs in Zephyr's SM layer and is rooted in improper synchronization within the HCI driver. Specifically, the Host HCI driver processes a connection termination HCI command before completing prior SM-related HCI commands. As a result, an SM command may attempt to access a shared resource that has already been cleared. Since this issue stems from synchronization errors, it is not confined to the SM layer; other code segments that rely on shared resources may also be affected. Exploitation requires a specific execution order, making the vulnerability more difficult to trigger. An attacker would need to repeatedly attempt the exploit to increase the likelihood of success.

**CVE-2024-4785.** This vulnerability occurs in Zephyr's LL implementation. Under normal conditions, once a central and peripheral device establishes a link-layer connection, they can negotiate connection parameters using the `LL_CONNECTION_PARAM_REQ` packet, which can be initiated by either device. In contrast, the `LL_CONNECTION_UPDATE_IND` packet can only be sent from the central to the peripheral. This vulnerability arises because no validation is performed before using parameters in the `LL_CONNECTION_UPDATE_IND` packet, leading to a potential divide-by-zero error. To exploit this flaw, an attacker can send a malicious `LL_CONNECTION_UPDATE_IND` packet to the victim after establishing their connection. The patch addresses this issue by checking whether received parameters, such as the connection interval, meet the minimum required threshold.

## 7 Limitation and Extension

**Need for Source Code.** The BLuEMan framework is coupled with a supported RTOS and a PHY simulator, and requires access to the source code of the BLE implementation. It can support various open-source BLE implementations, including BTstack [7], NimBLE [3], and Zephyr-native [37], by porting them to compatible RTOSs such as Zephyr and Mynewt.

**Inaccuracy of Hardware Simulations.** Using a PHY simulator to simulate the hardware may introduce inaccuracies in behavior compared to physical hardware. One such inaccuracy involves the handling of hardware interrupts. In the simulated environment, the kernel thread may delay hardware interrupts until it returns from certain waiting states, such as an API call or a CPU yield operation, e.g., `k_cpu_idle`.

Additionally, the sequential execution of software and hardware tasks in the simulated environment can result in false negatives and missing vulnerabilities that depend on specific timing or interactions between software and hardware. Physical hardware, with its more randomized execution sequences, is less prone to such omissions. Therefore, while PHY simulation provides a valuable testing tool, it has limitations that can impact the accuracy of results, particularly in timing and interaction-dependent scenarios.

**BLuEMan Scalability.** While this work mainly uses the Zephyr RTOS for executing BLuEMan, the framework is also compatible with other popular RTOSs, such as FreeRTOS [14] and ThreadX [12]. Firmware for these systems can be compiled into a single ELF executable, enabling it to run efficiently as native code on a host system running the fuzzer. Some manual effort is required to port BabbleSim to this native runtime. It is important to note that our approach focuses on evaluating the BLE stack rather than the RTOS itself. While FreeRTOS and ThreadX do not provide built-in Bluetooth stacks like Zephyr, they are compatible with some open-source Bluetooth stacks such as BTstack and NimBLE, both of which have been evaluated in the work.

## 8 Related Work

We study related work in Bluetooth vulnerability, BLE-related fuzzer, and general protocol fuzzer directions.

**Bluetooth vulnerability.** Numerous security vulnerabilities have been discovered in existing Bluetooth implementations. Specifically, BlueBorne [5] reveals vulnerabilities in several Bluetooth protocols, including L2CAP, BNEP (Bluetooth Network Encapsulation Protocol), and SDP (Service Discovery Protocol), whereas the others identify vulnerabilities from the L2CAP implementation of the Android Bluetooth stack [21], BlueZ in the Linux kernel [1], and the Texas Instrument's BLE implementation [4]. BLUFFS [2] presents six novel attacks to break the future and forward secrecy of Bluetooth sessions, whereas BLESA [35] discovers a vulnerability in the device reconnection procedure at the LL layer. In this paper, we focus on a full-stack fuzzing framework against BLE vulnerabilities.

**BLE Emulation-based fuzzers.** BLE emulation-based fuzzers can be classified into BLE host fuzzers and full-stack fuzzers. *BLE host fuzzers:* BTFuzz [23] adopts black-box fuzzing and communicates with the BlueZ BLE host on QEMU via an emulated serial device, while guiding fuzz testing using a customized state machine. VirtFuzz [20] collects edge coverage to guide the fuzz testing and communicates with the Linux Bluetooth host on QEMU through a customized VirtIO [27, 32] device. Although it is a powerful framework for fuzzing wireless implementations, most RTOS platforms do not support VirtIO Bluetooth, and applying it to RTOS can introduce significant overhead. *BLE full-stack fuzzers:* Frankenstein [31] operates by dumping memory contents and registering values at runtime, through patching the memory of physical Bluetooth controllers. The dumped firmware is then linked into an ELF file to execute in QEMU user mode. This approach provides faster execution speeds and flexibility for firmware modification. However, it requires extensive reverse engineering and is prone to hardware discrepancies across platforms, further increasing complexity and cost.

**BLE Platform-based Fuzzers.** Platform-based fuzzers can also be classified into BLE host fuzzers and full-stack fuzzers. *BLE host fuzzers:* L2Fuzz [28] uses black-box fuzzing based on Bluetooth v5.2, creating an L2CAP state machine where each state has predefined legitimate packets to support state transitions. ToothPicker [18] targets the iOS Bluetooth stack using FRIDA for dynamic instrumentation and radamsa for mutation on iOS devices. *Full-stack fuzzers:* BrakTooth [15] and SweynTooth [16] employ black-box fuzz approaches to perform platform-based BLE full-stack fuzzing. BrakTooth focuses on testing Bluetooth BR/EDR, while SweynTooth specializes in testing Bluetooth LE. Both studies design a state machine targeting the entire stack rather than specific layers. However, these platform-based solutions are all limited by physical environments, restricting the speed of fuzz testing.

**BLE GATT Service Fuzzers.** GATT is an essential component of many IoT devices, defining how devices exchange data through services and characteristics, so its security and stability are important [33, 38]. In the BLE security testing framework [30], the fuzzer is built using Go and interacts with Bluetooth devices through GATT packets. The fuzzer adjusts the parameter size based on the protocol specification, while adopting a random approach to generate parameters. However, the triggered failures are not deterministic, making it challenging to identify root causes.

**Protocol Fuzzers.** Protocol fuzzing has evolved through various implementations. Grammar-based fuzzing [10, 25, 34] relies on hard-coded or user-defined grammar specifications to generate test cases, which define the structure and field types of packets. With advancements, stateful protocol fuzzing techniques [11, 13, 22, 29] have emerged, aiming to learn state models of network protocols. Some tools have further enhanced these methods, such as TCP-Fuzz [39], which introduces a dependency-based strategy to handle dependencies between system calls and packets and uses a transition-guided fuzzing approach to improve state transition coverage. Additionally, TCP-Fuzz employs a differential checker to compare the outputs of multiple TCP stacks to detect semantic bugs. BLEEM [24] is a packet-sequence-oriented black-box fuzzer at the sequence level and uses a non-invasive feedback mechanism to track system states dynamically. FuzzUSB [22] is a fuzzer specifically for the USB gadget stack, addressing the limitations of prior USB host stack fuzzing tools. This framework adopts a multi-channel input mutation strategy, which considers the multi-phase nature of USB communication, allowing mutation testing across different states.

## 9 Conclusion

In this work, we introduced BLuEMan, a simulation-based fuzzing framework for evaluating the security of BLE protocol stack implementations. By combining an RTOS with a software-based PHY simulator, BLuEMan provides an efficient and scalable platform for protocol fuzzing. It features a novel MITM architecture for automated seed collection and a packet-driven workflow that streamlines state management. The framework supports comprehensive fuzzing across the BLE stack. Experimental results show that BLuEMan is effective in identifying a wide range of vulnerabilities–including buffer overflows, integer overflows, memory errors, and race conditions. Notably, it contributed to the discovery of four critical CVEs, all of which were responsibly disclosed and patched. These results highlight the practicality and effectiveness of BLuEMan in advancing secure development practices for BLE protocol stacks.

## Acknowledgment

We thank the anonymous reviewers and the shepherd for their valuable and insightful comments for improving this paper. The works presented in this study are supported, in part, by the National Science and Technology Council and Taiwan Academic Cybersecurity Center (TACC) at NYCU under the grants NSTC-113-2221-E-A49-186-MY3, 113-2634-F-A49-001-MBK, 112-2628-E-A49-016-MY3, and 114-2218-E-A49-017. One of the authors, Wei-Che Kao, has been affiliated with DEVCORE since graduation. We thank the company for supporting their employee's ongoing academic research involvement and sponsoring his travel expenses.

## Ethics Considerations

In our vulnerability discovery efforts, we prioritized ethical practices to benefit stakeholders and minimize harm. A key stakeholder was the Zephyr Project of the Linux Foundation, a community-driven open-source initiative. Upon identifying vulnerabilities, we promptly notified the Zephyr team, collaborated on solutions, and allowed sufficient time for fixes. All issues were reported as CVEs and resolved before public disclosure, minimizing disruption while maintaining transparency.

Our decisions balanced risks to Zephyr's developers and users with the need for swift action. We ensured sensitive information remained protected and prioritized solutions that upheld security and privacy. Recognizing that perspectives on ethics can vary, we engaged openly with the Zephyr team to address concerns and strengthen the project's security framework. This collaborative approach ensured effective resolutions and long-term improvements for the community.

## Open Science

This research complies with the open science policy required by the USENIX Security conference. Our codes, scripts, and sample applications are available at https://doi.org/10.5281/zenodo.15601101 and https://github.com/zoolab-org/blueman.artifact.

## References

[1] Andy Nguyen. BleedingTooth: Linux Bluetooth Zero-Click Remote Code Execution. https://google.github.io/security-research/pocs/linux/bleedingtooth/writeup.html.

[2] Daniele Antonioli. BLUFFS: Bluetooth Forward and Future Secrecy Attacks and Defenses. In *ACM conference on Computer and Communications Security (CCS)*, November 2023.

[3] Apache Mynewt. NimBLE, 2024. https://github.com/apache/mynewt-nimble.

[4] Armis. Bleedingbit: Exposes Enterprise Access Points and Unmanaged Devices to Undetectable Chip Level Attack. https://www.armis.com/research/bleedingbit/.

[5] Armis. BlueBorne: The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks. https://www.armis.com/research/blueborne/.

[6] BabbleSim. BabbleSim, 2024. https://babblesim.github.io/.

[7] bluekitchen. BTstack, 2024. https://github.com/bluekitchen/btstack.

[8] Bluetooth SIG. Bluetooth® Core Specification 5.4, 2023. https://www.bluetooth.com/specifications/specs/core-specification-5-4/.

[9] Bluetooth SIG. 2023 Bluetooth® Market Update, 2024. https://www.bluetooth.com/2023-market-update/.

[10] boofuzz. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz.

[11] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206. USENIX Association, August 2015.

[12] Eclipse ThreadX. ThreadX Repository, 2025. https://github.com/eclipse-threadx/threadx.

[13] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540. USENIX Association, August 2020.

[14] FreeRTOS. FreeRTOS Repository, 2025. https://github.com/FreeRTOS/FreeRTOS.

[15] Matheus E. Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. BrakTooth: Causing havoc on bluetooth link manager via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1025–1042, Boston, MA, August 2022. USENIX Association.

[16] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. SweynTooth: Unleashing mayhem over bluetooth low energy. In *2020 USENIX Annual Technical Conference*

*(USENIX ATC 20)*, pages 911–925. USENIX Association, July 2020.

[17] Google Inc. RootCanal Repository, 2024. https://github.com/google/rootcanal.

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

[19] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 230–243, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Sönke Huster, Matthias Hollick, and Jiska Classen. To boldly go where no fuzzer has gone before: Finding bugs in Linux' wireless stacks through VirtIO devices. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4629–4645. IEEE, 2024.

[21] Jan Ruge. BlueFrag: CVE-2020-0022 an Android 8.0-9.0 Bluetooth Zero-Click RCE. https://insinuator.net/2020/04/cve-2020-0022-an-android-8-0-9-0-bluetooth-zero-click-rce-bluefrag/.

[22] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin R. B. Butler, Antonio Bianchi, and Dave Jing Tian. FuzzUSB: Hybrid stateful fuzzing of USB gadget stacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, July 2022.

[23] Jin Lei, Yongjun Wang, Xu Zhou, and Ke Yan. BTFuzz: Accurately fuzzing bluetooth host with a simulated non-compliant controller. In *2022 IEEE 4th International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*, pages 1195–1201, 2022.

[24] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: Packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4481–4498, Anaheim, CA, August 2023. USENIX Association.

[25] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. Polar: Function code aware fuzz testing of ICS protocol. *ACM Trans. Embed. Comput. Syst.*, 18(5s), October 2019.

[26] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - bluetooth binary patching and experimentation framework. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 79–90. Association for Computing Machinery, 2019.

[27] Tsirkin Michael S. and Huck Cornelia, 2023. https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html.

[28] Haram Park, Carlos Kayembe Nkuba, Seunghoon Woo, and Heejo Lee. L2Fuzz: Discovering bluetooth l2cap vulnerabilities using stateful fuzz testing. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 343–354, 2022.

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

[30] Apala Ray, Vipin Raj, Manuel Oriol, Aurelien Monot, and Sebastian Obermeier. Bluetooth low energy devices security testing framework. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018.

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

[32] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[33] SecuRing Sławomir Jasek. GATTacking bluetooth smart devices, 2016. Black Hat USA.

[34] Andreas Walz and Axel Sikora. Exploiting dissent: Towards fuzzing-based differential black-box testing of TLS implementations. *IEEE Transactions on Dependable and Secure Computing*, 17(2):278–291, 2020.

[35] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave (Jing) Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. BLESA: Spoofing attacks against reconnections in bluetooth low energy. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[36] Zephyr Project. Zephyr - The POSIX architecture, 2024. https://docs.zephyrproject.org/latest/boards/native/doc/arch_soc.html.

[37] Zephyr Project. Zephyr RTOS Repository, 2024. https://github.com/zephyrproject-rtos/zephyr.

[38] Yue Zhang, Jian Weng, Zhen Ling, Bryan Pearson, and Xinwen Fu. BLESS: A ble application security scanning framework. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. IEEE, July 2020.

[39] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021.

# A Coverage Measurement Result

Due to space limitations, the plots of the measured coverage for the selected applications not presented in Figure 13 are shown in Figure 17.

# B Details of Recognized CVEs

This section discusses new vulnerabilities our proposed approach recognizes, as illustrated in Table 1. The recognized vulnerabilities span across different layers, i.e., LL, ATT, and SM. It demonstrates that our framework can effectively uncover vulnerabilities across various layers through its stateful exploration design. As mentioned in Section 3, all the reported CVEs can remotely cause DoS in the affected BLE protocol stacks, with three being exploitable without requiring pairing (CVE-2023-4424, CVE-2024-3077, and CVE-2024-4785). Notably, a single packet cannot trigger the vulnerabilities discovered by BLuEMan in the SM layers. It requires a sequence of packets to guide the program into a vulnerable state, which is further discussed in Section B.3. Note that all the vulnerabilities discussed in this study have been reported to the corresponding developers and are fixed. The recognized CVEs are summarized in the following subsections.

## B.1 CVE-2023-4424

This vulnerability can be triggered in Zephyr's Link Layer (LL) implementation when processing BLE advertising packets. The vulnerable implementation incorrectly assumes that a BLE advertising packet is at least 6 bytes in length (the broadcasting address), followed by the advertising data. As a result, it calculates the advertising data length using the logic shown in Figure 18, which causes an integer underflow if the actual data length is less than 6 bytes. This flaw allows an attacker to broadcast malformed packets, enabling DoS attacks against all vulnerable BLE devices that receive advertising packets. A victim device has to be rebooted to recover the state of the BLE stack. The CVE received a CVSS score of 8.8 (high) from the NVD, as it can be exploited remotely with low complexity, no privileges required, and no user interaction.

## B.2 CVE-2024-3077

This vulnerability can be triggered in Zephyr's Attribute Protocol (ATT) layer implementation when an unexpected ATT response packet is received from a malicious BLE device. It crashes a BLE victim device due to an integer underflow issue. Specifically, this vulnerability happens when an unexpected response to the ATT Find Information Request is handled. An attacker can spoof or control a GATT server, respond to an ATT Find Information Request using a valid protocol message of length zero, and cause out-of-bound memory access to crash the interacting device.

Figure 19 compares the typical packet flow with the exploitation packet flow of CVE-2024-3077. The typical flow, shown in Figure 19a, starts with discovering the device and requesting GATT service information from the GATT server. Once the setup is complete, the client sends an ATT Find Information Request to the server and waits for the corresponding ATT Find Information Response. Zephyr implements this feature by registering a callback function, `gatt_find_info_rsp`, to handle any packets returned from the server after sending an ATT Find Information Request.

Although the lower layer implementation in Zephyr ensures that only valid ATT messages are handled by the callback function, some valid ATT messages, such as the ATT Execute Write Response having a zero-length payload, can still bypass the check. As a result, the `length-1` operation in the `gatt_find_info_rsp` function inadvertently sets the `length` variable to a large value ($2^{16}-1$) when the payload length of the response is zero. It then causes a crash due to out-of-bounds access to inaccessible memory regions, as shown in Figure 19b.

We have reported these issues to the developers, and all of them have been patched. Figure 23 in Appendix C.1 shows the patch for this vulnerability. The fix involves adding a check to ensure that the length of the ATT response meets the minimum size required for a Find Information Response. This type of vulnerability was found in multiple locations, with a total of five functions related to ATT response handling being affected.

## B.3 CVE-2024-3332

This vulnerability occurs in Zephyr at the Security Manager (SM) layer, with the root cause being improper synchronization in the HCI driver. Specifically, the Host HCI driver processes the connection termination HCI command before completing the HCI commands related to the SM layer. This causes the previous SM-related command to access a shared resource that has already been cleared. Since this issue stems from improper synchronization, it is not limited to the SM layer; other code segments that use shared resources may also be affected. The vulnerability requires a specific execution order to trigger, making exploitation more difficult. Attackers

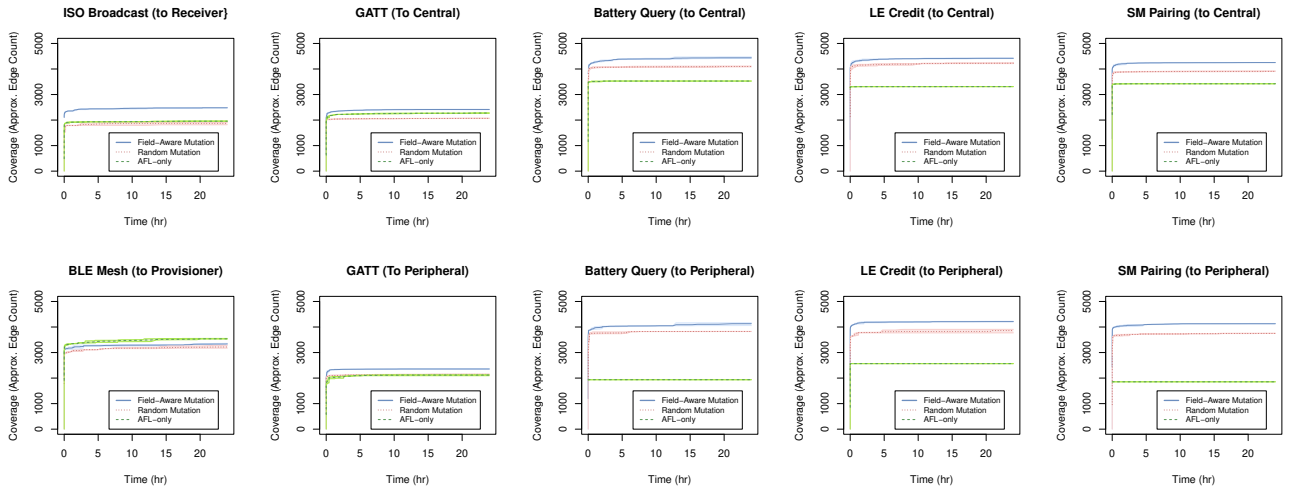



Figure 17: Measured coverage for the selected evaluated applications (the rest).

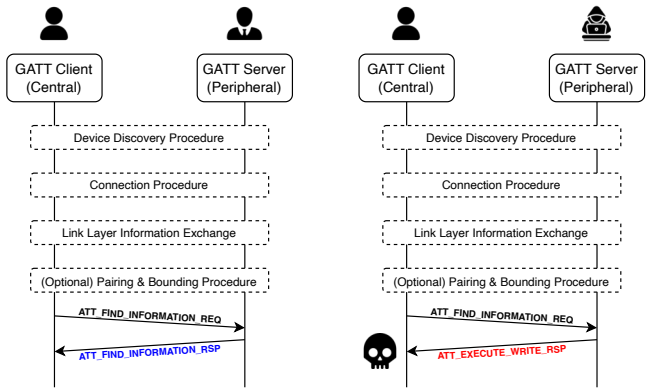
```
1 ...
2 if (adv->type != PDU_ADV_TYPE_DIRECT_IND) {
3    data_len = (adv->len - BDADDR_SIZE);
4 }
5 ...
6 memcpy(&adv_info->data[0], &adv->adv_ind.data[0], data_len);
7 ...
```

Figure 18: The root cause of CVE-2023-4424.

need to repeatedly attempt to exploit the issue for a chance to trigger it.

One example to illustrate this vulnerability is shown in Figure 20. During a pairing process, a victim BLE device creates an asynchronous task to generate the Diffie-Hellman (DH) key upon receipt of a pairing public key. Before the asynchronous task completes, an attacker can send a pairing DH-key check with an incorrect Ea value and then issue an `LL_TERMINATE_IND` to terminate the SM connection. When the asynchronous task is complete, the DH key check verification fails due to the incorrect Ea received from the attacker. The victim then attempts to return a pairing failed message to the central. However, since the SM connection state has been cleared due to previous termination, subsequent access to the SM connection state leads to access to a released memory pointer, causing a memory access error.

We have reported these issues to the developers, and all of them have been patched. Figure 24 in Appendix C.2 shows the patch to this vulnerability. Because the race condition is caused by the simultaneous handling of HCI commands, a function call to the original `bt_recv` function implemented in the HCI layer is protected with a lock to prevent current threads from accessing the function and leading to the race condition.




(a) Typical packet flow. (b) Exploitation packet flow.

Figure 19: CVE-2024-3077: Packet flow comparison.

## B.4 CVE-2024-4785

This vulnerability occurs in the link layer (LL) of Zephyr. Figure 21 shows a typical packet flow of link layer connection establishment. Under normal circumstances, once a central and peripheral device establishes a link layer connection, they can negotiate connection parameters via the `LL_CONNECTION_PARAM_REQ` packet. This packet can be initiated by either the central or the peripheral device, while the `LL_CONNECTION_UPDATE_IND` can only be sent from the central to the peripheral.

This vulnerability is triggered because no checks are performed before using the parameters in the `LL_CONNECTION_UPDATE_IND` packet, leading to divide-by-zero errors. Specifically, the `interval` value received from an attacker is used directly in division operations, as shown in Figure 22. It only affects BLE peripheral devices, as only a central device can return an `LL_CONNECTION_UPDATE_IND`

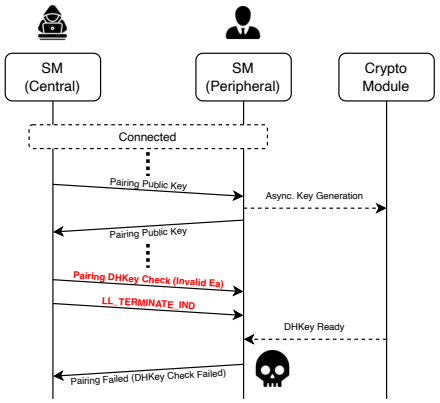



Figure 20: Attack scenario for demonstrating CVE-2024-3332.

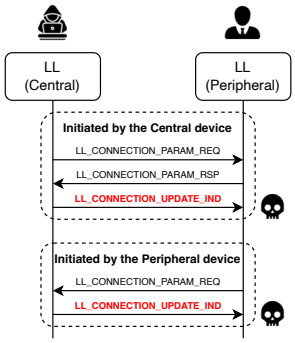



Figure 21: Attack scenario for demonstrating CVE-2024-4785.

packet to a peripheral device. To exploit the vulnerability, an attacker (central) first establishes a link layer connection with the victim (peripheral). After establishing the connection, the attacker can send a malicious `LL_CONNECTION_UPDATE_IND` packet to trigger a divide-by-zero error and crash the victim.

This vulnerability affects multiple locations in the link layer source codes. One patch is to create a new sanitizer function called `cu_check_conn_ind_parameters` to check the validness of fields (`interval`, `latency`, `timeout`) in a received packet, as shown in Figure 25. Another patch snippet for this vulnerability is shown in Figure 26. In general, the patches check whether a received parameter, such as the `interval` parameter in the packet, exceeds a minimal value (`BT_HCI_LE_INTERVAL_MIN`, which is 6). If the parameter does not meet this threshold, the patch adds one to the received interval to prevent possible subsequent divide-by-zero errors.

# C Patch to Recognized CVEs

## C.1 CVE-2024-3077

Figure 23 shows the patch for CVE-2024-3077. The fix

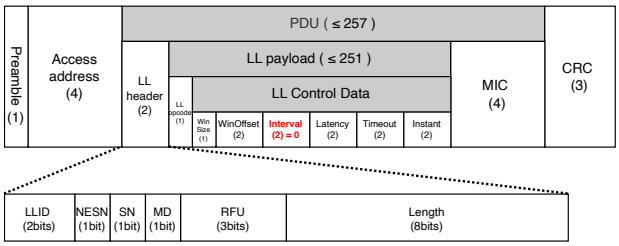



Figure 22: CVE-2024-4785: Attacking the LL control message.

```
diff --git a/subsys/bluetooth/host/gatt.c
  ↪ b/subsys/bluetooth/host/gatt.c
index 69d43de569..6744d77210 100644
--- a/subsys/bluetooth/host/gatt.c
+++ b/subsys/bluetooth/host/gatt.c
@@ -4365,7 +4422,7 @@ static void
  ↪ gatt_find_info_rsp(struct bt_conn *conn, int err,
         const void *pdu, uint16_t length,
         void *user_data)
 {
-  const struct bt_att_find_info_rsp *rsp = pdu;
+  const struct bt_att_find_info_rsp *rsp;
   struct bt_gatt_discover_params *params = user_data;
   uint16_t handle = 0U;
   uint16_t len;
@@ -4387,6 +4444,13 @@ static void
  ↪ gatt_find_info_rsp(struct bt_conn *conn, int err,
     goto done;
   }

+  if (length < sizeof(*rsp)) {
+    LOG_WRN("Parse err");
+    goto done;
+  }
+
+  rsp = pdu;
+
   /* Data can be either in UUID16 or UUID128 */
   switch (rsp->format) {
   case BT_ATT_INFO_16:
```

Figure 23: Patch for CVE-2024-3077.

involves adding a check to ensure that the length of the ATT response meets the minimum size required for a Find Information Response. This type of vulnerability was found in multiple locations, with a total of five functions related to ATT response handling being affected.

## C.2 CVE-2024-3332

Figure 24 in Appendix C.2 shows the patch to CVE-2024-3332. Because the race condition is caused by the simultaneous handling of HCI commands, a function call to the original `bt_recv` function implemented in the HCI layer is protected with a lock to prevent current threads from accessing the function and leading to the race condition.

## C.3 CVE-2024-4785

CVE-2024-4785 can be used to attack the parser of link layer messages. One patch is to create a new sanitizer function

```
diff --git a/subsys/bluetooth/host/hci_core.c
↪ b/subsys/bluetooth/host/hci_core.c
index 35ba75701b4..5b515bd929c 100644
--- a/subsys/bluetooth/host/hci_core.c
+++ b/subsys/bluetooth/host/hci_core.c
@@ -3900,7 +3900,7 @@ static void rx_queue_put(struct
↪ net_buf *buf)
        }
 }

-int bt_recv(struct net_buf *buf)
+static int bt_recv_unsafe(struct net_buf *buf)
 {
        bt_monitor_send(bt_monitor_opcode(buf), buf->data,
        ↪ buf->len);

@@ -3939,6 +3939,17 @@ int bt_recv(struct net_buf
↪ *buf)
        }
 }

+int bt_recv(struct net_buf *buf)
+{
+       int err;
+
+       k_sched_lock();
+       err = bt_recv_unsafe(buf);
+       k_sched_unlock();
+
+       return err;
+}
+
 int bt_hci_driver_register(const struct bt_hci_driver *drv)
 {
        if (bt_dev.drv) {
```

Figure 24: Patch for CVE-2024-3332.

```
diff --git a/subsys/..../ll_sw/ull_llcp_conn_upd.c
↪ b/subsys/..../ll_sw/ull_llcp_conn_upd.c
index eb1f692e55f..db0d2711748 100644
--- a/subsys/..../ll_sw/ull_llcp_conn_upd.c
+++ b/subsys/..../ll_sw/ull_llcp_conn_upd.c
@@ -196,6 +196,22 @@ static bool
↪ cu_check_conn_parameters(struct ll_conn *conn,
↪ struct proc_ctx *ctx)
 }
 #endif /* CONFIG_BT_CTLR_CONN_PARAM_REQ */

+static bool cu_check_conn_ind_parameters(
+            struct ll_conn *conn, struct proc_ctx *ctx)
+{
+  const uint16_t interval_max =
+            ctx->data.cu.interval_max; /* unit 1.25ms */
+  const uint16_t timeout = ctx->data.cu.timeout;
+                                    /* unit 10ms */
+  const uint16_t latency = ctx->data.cu.latency;
+
+  /* Valid conn_update_ind parameters */
+  return (interval_max >= CONN_INTERVAL_MIN(conn)) &&
+      (interval_max <= CONN_UPDATE_CONN_INTV_4SEC) &&
+      (latency <= CONN_UPDATE_LATENCY_MAX) &&
+      (timeout >= CONN_UPDATE_TIMEOUT_100MS) &&
+      (timeout <= CONN_UPDATE_TIMEOUT_32SEC) &&
+      ((timeout * 4U) > /* *4U ... */
+       ((latency + 1U) * interval_max));
+}
...
```

Figure 25: The sanitizer function for CVE-2024-4785.

called `cu_check_conn_ind_parameters` to check the validness of fields (`interval`, `latency`, `timeout`) in a received packet, as shown in Figure 25. Another patch snippet for this vulnerability is shown in Figure 26. In general, the patches check whether a received parameter, such as the `interval` parameter in the packet, exceeds a minimal value (`BT_HCI_LE_INTERVAL_MIN`, which is 6). If the parameter does not meet this threshold, the patch adds one to the received interval to prevent possible subsequent divide-by-zero errors.

```
diff --git
↪ a/subsys/bluetooth/controller/ll_sw/ull_conn.c
↪ b/subsys/bluetooth/controller/ll_sw/ull_conn.c
index c9ebeb9c274..93a610fa705 100644
--- a/subsys/bluetooth/controller/ll_sw/ull_conn.c
+++ b/subsys/bluetooth/controller/ll_sw/ull_conn.c
...
@@ -2186,17 +2217,22 @@ void
↪ ull_conn_update_parameters(struct ll_conn *conn,
↪ uint8_t is_cu_proc, uint8_t win_size,
    uint32_t win_offset_us, uint16_t interval,
    uint16_t latency, uint16_t timeout, uint16_t instant)
...
+   if (interval >= BT_HCI_LE_INTERVAL_MIN) {
+      uint16_t max_tx_time;
+      uint16_t max_rx_time;
+      uint32_t slot_us;
...
+   } else {
+      conn_interval_new = interval + 1U;
+      conn_interval_unit_new = CONN_LOW_LAT_INT_UNIT_US;
...
+   }
+
+   conn_interval_us = conn_interval_new
+                      * conn_interval_unit_new;
+   periodic_us = conn_interval_us;
+
+   conn_interval_old_us = conn_interval_old
+                          * conn_interval_unit_old;
+   latency_upd = conn_interval_old_us / conn_interval_us;
...
```

Figure 26: Patch snippet for CVE-2024-4785.