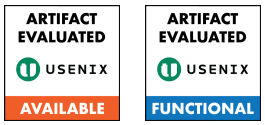


FuzzBT: Holistic-State-Guided Fuzzing for Bluetooth Host Stack in Kernels

Sungwoo Kim, *Purdue University*
Hui Peng, *Google, Inc.*
Imtiaz Karim, *The University of Texas at Dallas*
Ruoyu Wu, *Purdue University*
Jianliang Wu, *Simon Fraser University*
Elisa Bertino, *Purdue University*
Mathias Payer, *EPFL*
Dave (Jing) Tian, *Purdue University*

Abstract

Bluetooth is both pervasive and vulnerable, yet fuzzing Bluetooth is challenging. While research on Bluetooth fuzzing has advanced to emulate Bluetooth devices and generate effective inputs for controllers, the host stack has been overlooked. The host stack is responsible for issuing commands to controllers, providing API abstractions for user applications, establishing logical links for asynchronous connections, and multiplexing channels. Thus, a systematic approach to identifying vulnerabilities in a Bluetooth host stack is required, but still lacking.

The primary challenges in testing the Bluetooth host stack are (1) configuration diversity and (2) statefulness. The host stack can be configured with over 3,000 options, each of which may introduce configuration-specific bugs. Also, the host stack state is inherently complex because multiple protocols comprise it. To address the aforementioned challenges, we design FuzzBT, a state-guided fuzzer that adopts (1) configuration iteration and (2) stack-level state exploration. Specifically, we iterate over configurations across fuzzing campaigns and explore each configuration's unique logic using configuration-aware seeds extracted from the source code via concolic execution. For stack-level state, we aggregate the states of individual protocols via compiler instrumentation. We applied FuzzBT to two Bluetooth host stack implementations, the Linux and Zephyr kernels, and identified 18 previously unknown bugs with 9 CVEs.

1 Introduction

Bluetooth vulnerabilities directly affect users by exposing smart locks, microphones, and medical devices to potential attackers. In 2017, Armis announced the BlueBorne [33] attack, reporting that 8.2 billion Android devices were exposed to the remote code execution attack [7], allowing attackers to breach sensitive information and spread malware. Similarly, in 2020, Google discovered BleedingTooth [26], announcing that Linux machines, including embedded devices using the Linux kernels, are vulnerable to remote code execution with kernel privilege. Likewise, Bluetooth attacks [2–6, 9, 14, 17, 22, 33–36, 38–40, 43, 45–48, 50–52] have demonstrated that vulnerabilities in a Bluetooth host and controller result in severe security and privacy repercussions. Quantitatively, a recent CVE database query on Bluetooth shows 1,307 CVEs.[1], underscoring the scale of this threat.

Testing Bluetooth implementations is thus critical but challenging. Several works [16, 18, 23, 30] have advanced emulation-based Bluetooth fuzzing, removing the dependency on physical devices. Stateful Bluetooth fuzzers [12, 13, 28] integrated state-awareness techniques into Bluetooth bug discovery, aiming for higher state coverage to discover more bugs. Specifically, response-code-based methods [12, 13] use response codes to infer states of black-box Bluetooth implementations. Specification-based works [12, 13, 28, 49] extracted a state machine from the specification. Profiling-based works [1, 24] collect well-formed Bluetooth packets from normal communications to build high-quality corpora.

Unfortunately, existing work has mainly focused on the controller stack. Above the controllers, the Bluetooth host stack manages communication between user applications and the controller, handling protocol processing, connection establishment, and data multiplexing. It implements multiple layers, corresponding to the TCP/UDP stack in the Internet, and higher-level profiles, each with complex state machines and asynchronous event handling. Although several works targeted a host stack protocol, they are limited to fuzzing one protocol [1, 28] (L2CAP) and focusing on improving virtual devices [12, 13, 16] rather than considering the host stack components. Thus, we argue that *systematic fuzzing of the Bluetooth host stack remains insufficient.*

We propose FuzzBT, a Bluetooth host-stack fuzzer designed to be aware of host-stack properties. FuzzBT (1) explores the host stack configurations and (2) is guided by holistic host stack states.

Host stack configurations. Bluetooth configurations are designed to support diverse use cases, such as low energy con-

[1] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=bluetooth, retrieved at Jun. 05. 2026

sumption, TCP- and UDP-like retransmission policy, and Bluetooth over WiFi support. Once a configuration is fixed, a fuzzing input cannot reach the code of a different configuration. Therefore, it is necessary to cover all configurations in order to achieve comprehensive coverage of the host stack.

We test all configurations by iterating over all the host stack configurations during a fuzzing campaign. First, we extract possible configurations from the source code. Then we generate programs that can configure the host stack according to the extracted configurations.

Each configuration supports different options and thus has different logic. Hence, reusing the same fuzzing corpus across different configurations is inefficient. Our ablation study in Section 7.3 also supported that reusing the same corpus in a different configuration decreased code coverage. Instead, we generate seeds for each configuration through concolic execution. These seeds can pass the configuration-specific checks, enabling fuzzers to reach the core logic more efficiently.

Host stack state. The Bluetooth host stack comprises multiple host-layer protocols. Therefore, the state of a single protocol is insufficient to represent the host stack. To represent the host stack state, FuzzBT concatenates multiple protocol states. FuzzBT first collects each protocol's state by inserting probes at state transitions via compiler instrumentation, and then concatenates them within the fuzzer to obtain the stack-level state.

We applied FuzzBT to the Linux and Zephyr kernels and compared it with two state-of-the-art open-source Bluetooth fuzzing solutions [16, 41]. Using FuzzBT, we discovered 18 previously unknown bugs with 9 CVEs. Compared to a baseline fuzzer that does not consider configurations or the host stack state, FuzzBT improved bug discovery by 20% and code coverage by 8.2% on average. FuzzBT also achieved 31% higher code coverage compared to an existing host stack fuzzer.

In summary, we make three contributions as follows:

- We introduce Bluetooth configurations as a novel fuzzing factor affecting bug detection. Our approach first showed that bugs often rely on specific configurations.
- We design FuzzBT, a state-guided fuzzer for the Bluetooth host stack that incorporates host-stack configurations and stack-level states.
- We applied FuzzBT to two open-source kernels, Linux and Zephyr, and found 18 previously unknown bugs, including 9 CVEs.

  As part of our commitment to open science, we released the source code of FuzzBT at https://doi.org/10.5281/zenodo.20172790.

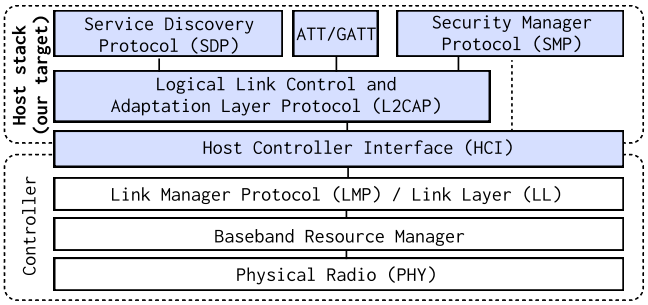


Figure 1: A minimum stack to establish a Bluetooth connection. The blue protocols indicate the core protocols.

2 Background

The Bluetooth stack consists of a controller and host stack, as shown in Figure 1. Both stacks run on separate CPUs on high-end devices, e.g., smartphones and desktops, due to their different functionalities and locations [45]. Still, both stacks can operate on a single chip in embedded devices like headsets. Either way, a unified Host Controller Interface (HCI) connects the two software stacks.

Controller stack. The controller stack directly communicates with the Bluetooth radio, which handles Bluetooth transmission and reception. It provides an interface for remote procedure calls to the host stack. The procedures are responsible for processing *commands* from the host and returning the results as *events*. Followed by those commands, the controller performs (1) device discovery, (2) link service: asynchronous connection-less (ACL), synchronous (SCO), and isochronous (ISO), (3) link control: connect and disconnect, or (4) secure pairing. The controller stack is composed of a physical radio (PHY), a baseband resource manager, a Link Manager Protocol (LMP) for Basic Rate/Enhanced Data Rate (BR/EDR), and a Link Layer (LL) for Bluetooth Low Energy (BLE).

Host controller interface. The HCI layer transports data between a controller and a host stack. HCI commands, events, and data packets pass through this layer without being decoded. A carrier could be UART, USB, or Secure Digital (SD).

Host stack (our target). The host stack actively commands controllers and establishes synchronous or asynchronous connections at a software level, providing abstracted functions to users. The host includes Bluetooth's core protocols: HCI, Logical Link Control and Adaptation Protocol (L2CAP), Service Discovery Protocol (SDP), Attribute Protocol (ATT), Generic Attribute Profile (GATT), and Security Manager Protocol (SMP). The design goal of L2CAP is to multiplex an ACL link with Protocol Service Multiplexer (PSM), while other core protocols accomplish interoperability by providing essential communication functionality, such as service discovery, authentication, and encryption. In addition to the core protocols, the Bluetooth host stack can provide application-level proto-

Listing 1 Motivating example (CVE-2024-36968)

```
1  /* HCI layer (lower) */
2  u8 hci_cc_le_read_buffer_size()
3  {
4    /* cross-protocol state propagation */
5    hdev->le_mtu = __le16_to_cpu(rp->le_mtu);
6  }
7
8  /* L2CAP layer (higher) */
9  struct l2cap_conn *l2cap_conn_add()
10 {
11   conn->mtu = hdev->le_mtu;
12 }
13
14 void l2cap_le_flowctl_init()
15 {
16   chan->mps = min(chan->imtu, conn->mtu - L2CAP_HDR_SIZE);
17   /* error: div-by-zero. */
18   chan->rx_credits =
19     (chan->imtu / chan->mps) + 1;
20 }
```

cols such as Radio Frequency Communication (RFCOMM), Human Interface Device Protocol (HIDP), and Bluetooth Network Encapsulation Protocol (BNEP). The Bluetooth host stack implementation resides in the kernel and user space.

3 Motivation

Previous research [4, 6, 33, 34, 43] has shown that attackers can exploit Bluetooth vulnerabilities and remotely execute arbitrary code, spreading malware and leaking critical private information. Linux, Windows, macOS, and Zephyr run some Bluetooth host stack protocols in kernel mode, thereby exposing kernel privileges to attackers.

Existing Bluetooth fuzzing frameworks have primarily focused on the controller stack [12, 13, 15], device virtualization [16,30], or a single protocol [1,28], thereby imposing fundamental limitations in testing the host stack. We summarize these fundamental limitations of existing fuzzing solutions as follows.

(1) Fixing the host stack configuration. The Bluetooth host stack is layered with multiple stateful protocols, each of which is *configurable*. For example, L2CAP supports Streaming Mode (SM) and Enhanced Retransmission Mode (ERTM), similar to UDP and TCP. From a software testing perspective, configurations are variants of a target state machine that increase complexity.

If a fuzzer targets a single configuration, some code remains unreachable because Bluetooth packets take different code paths depending on the configuration. For example, if a fuzzer only supports L2CAP SM, its inputs will never reach the L2CAP ERTM code, let alone trigger ERTM state transitions. Moreover, a host stack implementation can introduce more configurations beyond the specification. For instance, the Linux kernel Bluetooth L2CAP implementation has 1,800 different configuration combinations. This includes 10 specification configurations (i.e., one flushable bit and five modes) and 180 implementation-defined configurations (i.e., three socket types, one defer setup bit, one power bit, and three channel policies). So far, existing works [12, 13, 15, 18, 28, 30] have only tested a single configuration since they used preconfigured Bluetooth applications, resulting in limited code coverage and bug detection for the rest of the 1,799 configurations.

(2) Targeting a single protocol. Testing a single protocol inevitably leaves some code unreachable from the whole stack perspective. Simply put, an HCI fuzzer cannot cover bugs that require both HCI and L2CAP packets. For example, Listing 1 contains a bug involving two protocols, HCI and L2CAP. To trigger this, a fuzzer should test HCI and L2CAP simultaneously, followed by a sequence of HCI and L2CAP packets. Specifically, an HCI packet sets `hdev->le_mtu` to `L2CAP_HDR_SIZE (4)` at Line 5. Later, an L2CAP packet propagates this value at Line 11 and Line 16, setting `chan->mps` to zero. Eventually, Line 19 tries to divide by `chan->mps`, which is zero, causing a division-by-zero bug. As shown, fuzzers should send out two packets towards distinct layers to discover this bug. Unfortunately, existing fuzzers are limited to targeting a single protocol and thus fail to trigger bugs that require awareness of the entire stack.

(3) Ignoring implementation-specifics. Implementation-specific features ensure flexibility across diverse hardware, allow vendor optimizations, and maintain backward compatibility. For instance, the Linux kernel adopted Microsoft's vendor extension, allowing users to optionally enable extended HCI command support for compatible controllers. Thus, the specification alone cannot serve as a comprehensive basis for Bluetooth implementations, as it does not capture implementation specifics. This may result in missing CVE-2024-36012 [2], which is a use-after-free bug in the Microsoft extension. To detect this, a fuzzer should be aware of the Microsoft extension and sensitive to the relevant state transitions, which are not documented in the specification. So far, existing works [12, 13, 15, 28, 30, 49] have focused on only specification-defined state, leaving implementation-specific vulnerability detection uncovered.

4 Challenges and Solutions

Motivated by the three limitations above, we aim to cover the entire host stack by tackling the following three challenges.

C1: Covering multiple configurations. Effectively covering all configurations is challenging because each configuration has unique logic. To reach the core of each configuration, fuzzing inputs must satisfy configuration-specific checks. Because these checks differ across configurations, simply reusing a corpus from other configurations may be ineffective and can even slow exploration.

To solve this, we generate high-quality seeds prior to

[2] Found by FuzzBT.

fuzzing that are aware of each configuration through concolic execution. The corpus contains concrete inputs that satisfy the basic checks for each configuration, helping fuzzers reach the core logic.

C2: Testing multiple protocols. Compared to single-protocol testing, host stack testing requires testing multiple protocols. Thus, the fuzzer should be guided by multiple protocols to send packets of multiple protocols.

We signal the fuzzer if at least one protocol state changes. To accomplish this, a kernel concatenates multiple protocol states and send it to the fuzzer by monitoring state transitions using compiler instrumentation. Maximizing this concatenated state, the fuzzer generates packets across multiple protocols and explores the stack space comprehensively.

C3: Covering implementation-specific states. Implementation specifics introduce undocumented states that are difficult to identify and, consequently, challenging to cover. We cannot use the specification to identify implementation-specific states, as they fall outside its scope by definition.

We identify implementation-specific state variables from the source code using common coding patterns. These state variables are then exposed to the fuzzer, allowing it to cover and be guided by them.

5 Design

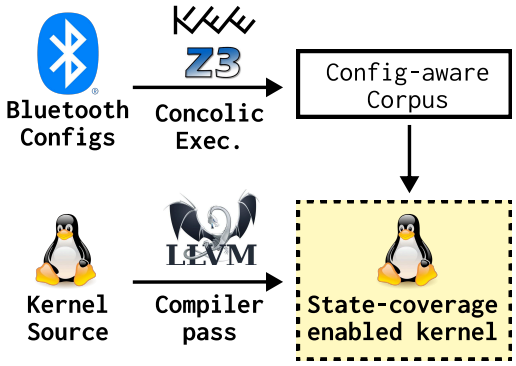


Figure 2: A design overview of FuzzBT.

Overview. Figure 2 describes the overall design of FuzzBT. We first extract configurations and their tailored seeds (Section 5.1) that help reach the core logic (**C1**). Then we compile the target kernel with state-coverage probes (Section 5.2), resulting in a state-coverage enabled kernel. We automatically insert probes throughout all protocols in the host stack (**C2** and **C3**). Inside the probe, we encode state transitions to signal fuzzers about the new state (Section 5.3).

Threat model. We assume that the attacker directly targets the Bluetooth host stack by compromised HCI controller. Thus, it may send any HCI frames, including arbitrary device information, inconsistent resource responses, and noncompliant responses. We also assume the attacker cannot install and run any arbitrary user-level application that uses Bluetooth in the victim's machine. Our victim has a running Bluetooth device in discoverable mode, thereby being ready to accept Bluetooth packets.

Terms. We define the following terms to avoid confusion. `BT_state_typ` is the type of a state variable of a Bluetooth protocol, implemented in a programming language, e.g., `struct l2cap_chan` in C. If a certain variable's type is `BT_state_typ`, then, by definition, the variable is a Bluetooth protocol state variable. `BT_state_var` is the state variable of a Bluetooth protocol. Thus, `BT_state_var`'s type is always `BT_state_typ`. `BT_state_tran` is a state transition in a Bluetooth protocol. We define `BT_state_tran` by having `BT_state_var` a new value.

5.1 Cover Multiple Configurations

The goal is to enable fuzzers to cover multiple configurations using a high-quality corpus (**C1**). Fortunately, Bluetooth supports a manageable number of configurations (3,000) that can be enumerated. Therefore, we enumerate configurations after each fuzzing campaign, e.g., after one minute. To achieve this, we extract configurations from the source code to construct a configuration list. We then generate a high-quality corpus for each configuration using concolic execution.

Configuration enumeration. The kernel exposes system calls that provide an interface for configuring Bluetooth communication in the user space. So, the system call implementation on the kernel side should describe all available configurations. Leveraging this, we statically extract and enumerate available configurations from the kernel source code. First, we identify system calls related to Bluetooth configuration. For example, the Linux kernel provides `setsockopt()` and `ioctl()` for Bluetooth configuration, in compliance with the POSIX standard. Next, we collect possible arguments for these system calls from `#define` directives or `enum` definitions in header files. As a result, we get a list of available configurations that are ready to enumerate.

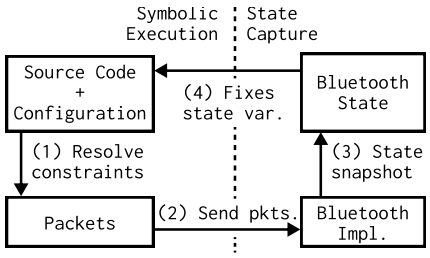


Figure 3: Configuration-aware inputs extractions in a loop.

Configuration-aware inputs extraction. Figure 3 describes an algorithm for corpus extraction through concolic execution. At step (1), we resolve the symbolic constraints of packets. Since the packet is the fuzzing input, we assign constant

Listing 2 Concolic execution example

```
1  void l2cap_data_channel(struct sk_buff* skb, ...)
2  {
3    /* concretize symbolic variables */
4    chan->mode = L2CAP_MODE_LE_FLOWCTL; // configuration
5    chan->state = BT_CONNECTED;
6    chan->imtu = 26111;
7    /* concretize symbolic variables done */
8
9    if (chan->state != BT_CONNECTED)
10     return;
11
12   switch (chan->mode) {
13   case L2CAP_MODE_LE_FLOWCTL:
14     uint16_t sdu_len = get_unaligned_le16(skb->data);
15     if (sdu_len > chan->imtu) {
16     /** Covering here requires LE_FLOWCTL configuration.
17      * Concolic execution generates packets covering here.
18      */
19   // ...
20 }
```

values to other symbolic variables (e.g., a variable that holds a configuration setting) to ensure that only the packet remains symbolic. To attain such constants, we run a kernel aside and extract the values. Next, we send these packets to the Bluetooth stack to trigger state transitions at step (2). We then capture the resulting states at step (3) and reassign constants to the symbolic variables based on the updated state at step (4). Returning to step (1), we generate new packets. The routine terminates either when no new states are discovered or when constraint solving fails due to time running out.

Concrete example. Listing 2 shows an example of the corpus extraction. We symbolize packets, i.e., `skb->data`, and assign configurations and state variables (Line 4-Line 6). Resolving symbolic constraints, we attain packets reaching configuration-specific code, Line 16, and potentially leading to state transitions. Repeatedly performing this within various configurations and states, we collect a set of configuration-aware inputs that we use as seeds later.

5.2 Bluetooth State Coverage Probe

In the fuzzing phase, we guide inputs for higher host stack state coverage (**C2**), including implementation-specific states (**C3**). For this, we present Bluetooth State Coverage(BTSCov), a diff-based state-collection design that monitors and captures Bluetooth state transitions for the host stack.

We want to insert BTSCov probes at Bluetooth state transitions, i.e., `BT_state_tran`. Thus, we first need to identify the locations of these state transitions, which essentially correspond to memory write operations. To identify such memory write operations, we need to answer two key questions. First, which variables qualify as `BT_state_var`? Second, which write operations target a `BT_state_var`? In the following section, we describe our design for addressing these questions.

State variables. We adopt a heuristic to identify `BT_state_var` from other variables. Our observations are as follows: (1) A Bluetooth connection and `BT_state_var` should have the same lifecycle. So, `BT_state_var` should be newly assigned for a new connection. Otherwise, it represents another kernel state rather than a Bluetooth connection. This should cover both specification-defined and implementation-specific state variables. (2) A Bluetooth connection may have multiple `BT_state_var`. To organize multiple `BT_state_var`, the most common convention is to use structures. As shown in Table 1, `BT_state_var` in real-world implementations are packed into a single structure and allocated once per connection. Guided by those observations, FuzzBT treats those structures as `BT_state_typ`. Thus, we consider all fields in the structures as `BT_state_var`. This can lead to both false positives and negatives, which we will discuss in Section 8.

Layer	State variable types	
	Linux	Zephyr
HCI	struct hci_dev struct hci_conn struct hci_chan	struct bt_conn
L2CAP	struct l2cap_conn struct l2cap_chan	struct bt_l2cap_br_chan struct bt_l2cap_le_chan
RFCOMM	struct rfcomm_session struct rfcomm_dlc	struct bt_rfcomm_dlc

Table 1: Bluetooth state variable types in Linux and Zephyr.

State transitions. We identify `BT_state_tran` from other memory access using `BT_state_typ`. First, we recover the type information for the memory write address. If the recovered type is `BT_state_typ`, the operation is `BT_state_tran`. We employ backward slicing and debugging information tracing to identify the type of a target memory address. Figure 4a provides an example of backward slicing. The goal is to identify that the store instruction, `store i16 %0, ptr %state`, is a `BT_state_tran`. We slice out other instructions backward until we find the type information. Eventually, we find that `%state` is a field of `l2cap_chan`, meaning that the store instruction was a state transition of `l2cap_chan::state`. Thus, this operation is `BT_state_tran`. Debug information tracing follows a similar process. As shown in Figure 4b, we trace debug information of the store instruction until we find the type information. In `!3`, we find a symbol, "l2cap_chan" with a variable name "chan", indicating that the store instruction writes a value into `l2cap_chan::chan`, which is a `BT_state_tran`.

Unfortunately, the static analyses cannot cover direct memory access and path explosion. For example, `(*(int*)0xffff) = 42` could be `BT_state_tran`, if `0xffff` is an address of `BT_state_var`. Nevertheless, we cannot perform backward slicing on this, as no instruction exists backward. In this case, we stop static analysis and leave it for runtime.

At runtime, we use memory allocation information. Figure 5 demonstrates how this runtime checking works. Firstly, we manage addresses of `BT_state_var` by hooking mem-

Listing 3 State transition encoding

```
1  struct btscov_unit {
2    uint64_t data;  // New state to be assigned.
3    /* Below are metadata */
4    uint8_t conn_id[16];  // connection identifier
5    uint8_t proto;    // protocol identifier
6    uint16_t state_var;     // state variable identifier
7  } __packed;
```

ory allocation functions. Secondly, whenever a memory access happens, we check that the target address is in the address table. If so, we consider and record the operation as a `BT_state_tran`.

```
call void @llvm.dbg.value(metadata ptr %state, metadata !1, ...)
; /* ... (sliced out) ... */
%state = getelementptr inbounds %struct.l2cap_chan, ptr %chan, ...
; /* ... (sliced out) ... */
store i16 %0, ptr %state, align 2
```

(a) Backward slicing considers that `%state` belongs to `l2cap_chan`, which is a state variable. Thus, *store %state* is a state transition.

```
call void @llvm.dbg.value(metadata ptr %state, metadata !1, ...)
; /* ... (sliced out) ... */
%state = getelementptr inbounds %struct.l2cap_chan, ptr %chan, ...
; /* ... (sliced out) ... */
store i16 %0, ptr %state, align 2

!1 = !DILocalVariable(name: "chan", ..., type: !2)
!2 = !DIDerivedType(..., basetype: !3, size: 64, offset: 16)
!3 = distinct !DICompositeType(..., name: "l2cap_chan");
```

(b) Debug information detects that `%state` belongs to `l2cap_chan`, which is a state variable. Thus, *store %state* is a state transition.

Figure 4: Technical details for compile-time checking. We use backward slicing and debug information to identify whether a given memory access corresponds to a state transition.

5.3 Bluetooth Host Stack State

This section describes the internal behavior of BTSCov probes introduced in Section 5.2. The probes record and encode `BT_state_tran` for later fuzzing. Importantly, the encoding must be compatible with multiple protocols in the Bluetooth host stack (**C2**). In the following, we present the design for protocol-compatible encoding across layers.

State coverage unit. `BT_state_tran` is a memory operation, which does not exceed 8 bytes in size. Thus, we need at least 8 bytes to represent a single state transition, covering all 1- to 8-byte writes. Listing 3 shows our design of a `BT_state_tran` encoding. Similar to conventional edge coverage, which uses an 8-byte address as feedback, our fuzzer uses a 27-byte chunk as state transition feedback. This unit comprises 8 bytes of new state data with 19 bytes of metadata.

This design is orthogonal to protocols and implementations, applicable to all Bluetooth protocols. For instance, an L2CAP channel's MTU value assignment of 42 is encoded as follows:

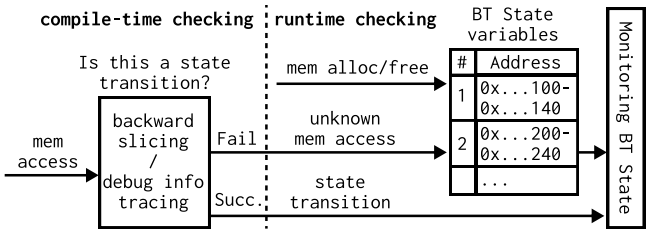


Figure 5: We consider all memory write operations as potential `BT_state_tran` and filter the genuine `BT_state_tran` using both compile-time and runtime analysis.

Listing 4 Concrete example of state coverage instrumentation

```
1  u8 hci_cc_le_read_buffer_size()
2  {
3    /** Compile-time instrumentation:
4     * The actual instrumentation is done at the LLVM IR.
5     * For convenience, we provide the equivalent C code.
6     */
7    // instrumentation starts
8    void* state_var = hdev;
9    void* state_var_type = HCI_DEV;
10   uint64_t offset = offsetof(struct hdev, le_mtu);
11   uint64_t rvalue = __le16_to_cpu(rp->le_mtu);
12   __bts_trace(state_var, state_var_type, offset, rvalue);
13   // instrumentation ends
14   hdev->le_mtu = __le16_to_cpu(rp->le_mtu);
15 }
```

```
// l2cap_chan->mtu = 42 is encoded as below:
struct btscov_unit unit = {
  .data = 42,
  .proto = L2CAP_CHAN,
  .state_var = offsetof(struct l2cap_chan, mtu),
  .conn_id[16] = concat(src, dst, psm),
}
```

To represent HCI state transitions, adjusting `proto`, `conn_id`, and `state_var`, is enough.

Per-connection coverage. During a fuzzing campaign, multiple Bluetooth connections can co-exist. Thus, state transitions can occur across multiple Bluetooth connections in parallel, making coverage data unreliable by mixing data from different connections. A BTSCov unit distinguishes each Bluetooth connection, providing per-connection coverage support via the `conn_id` field. We derive a unique connection identifier from each protocol layer. For example, the collection of source, destination, and PSM (similar to port) functions as a unique identifier for an L2CAP connection.

Host stack state. The host stack state is the concatenation of all protocol states. For each state transition, a kernel appends the 27-byte coverage chunk into a file, e.g., DebugFS. Later on, a fuzzer reads the file and concatenates all coverage units and detects a new state transition.

Concrete example. Listing 4 shows the results of the state coverage pass. The goal is to guide `rp->le_mtu` to have a new value as much as possible. To facilitate this, FuzzBT monitors `rp->le_mtu` and gives feedback if it has a new value. Specifically, the pass inserts a `__bts_trace()` hook (Line 8-Line 12) before modifying `rp->le_mtu`. Inside the

Layer	State	Packet	Depth			
			Max	Med	Mean	Stdev
HCI	7	2,371	6	2	2.47	1.41
SCO	-	-	-	-	-	-
L2CAP	422	30,765	22	11	10.81	4.77
RFCOMM	157	4,332	11	5	5.38	1.72

Table 2: Number of states and packets generated by concolic execution. Depth is the minimum number of packets required to reach a state from the initial state without a cycle. SCO is not stateful.

hook, FuzzBT collects state coverage and stimulates a fuzzer to modify `rp->le_mtu` with a new value repeatedly.

6 Implementation

Harness. To use BTSCov for fuzzing, we modified Syzkaller to accept BTSCov feedback. We implemented BTSCov to have the same interface as kCov, thereby allowing Syzkaller to easily use BTSCov with small modifications. We implemented BTSCov in a Linux kernel module in C with 800 LoC. We implemented a compiler pass for BTSCov in C++, using LLVM, with 286 LoC. We enabled this compiler pass only for `net/bluetooth` to prevent noise.

Fuzzing entry. The entry point for fuzzing is the entry function of the HCI layer. The fuzzer sends both BR/EDR and BLE packets to this entry point, where the function multiplexes the packets.

Concolic execution against kernels. The entry point of concolic execution is the same for each protocol entry point, e.g., `l2cap_recv_acldata()` for the Linux kernel's L2CAP implementation. We added 16 KLEE API calls into the source code to symbolize a socket buffer's header, data, and length. However, the path explosion could occur, and the kernel code contains inline assemblies that a common symbolic execution engine does not yet support. To address these, we created C-compatible functions for inline assembly and set a timeout for constraint solving to avoid path explosion. We limited the packet length to 12 to expect constraint solving could be completed within the practical time limit.

Table 2 shows the number of states and packets extracted by concolic execution, which constitute the configuration-aware corpus. It does not represent a complete set of states and packets because we could not completely resolve symbolic constraints. However, we confirmed that every state transition is reproducible. Thus, the corpus is sound. In our implementation, we extracted 30,765 packets with 422 unique states in the L2CAP layer. Also, the maximum depth is 22, meaning that we extracted a deep state that requires 22 packets to reach from an initial state without a cycle.

FuzzBT for Zephyr. At the time of writing, Syzkaller does not support Zephyr. We implemented a harness for Zephyr that exposes its Bluetooth host stack entry point to Syzkaller. Also, we fully emulate Zephyr using `native_sim`. We implemented BTSCov collection as a Zephyr kernel module 189 LoC in C. The LLVM pass is identical to FuzzBT for Linux.

7 Evaluation

We evaluate FuzzBT to answer the following research questions:

- RQ1: Is FuzzBT effective in finding previously unknown bugs?
- RQ2: Which feature improves fuzzing the most?
- RQ3: How does FuzzBT compare to existing fuzzers?

The corresponding evaluation metrics are as follows:

- Number of bugs that were previously unknown (RQ1)
- Bug and coverage comparison in feature ablation study (RQ2)
- Bug and coverage comparison with existing fuzzers (RQ3)

7.1 Experimental Setup

We ran experiments on an AMD EPYC 7773X 64-core processor with 256 logical cores at 2.20GHz to 3.50GHz.

Finding previously unknown bugs. We evaluated FuzzBT on a long-term support kernel that has actively received security updates. We excluded the latest branch of the Bluetooth subsystem as a target, since its experimental features are error-prone and can artificially inflate FuzzBT's bug-finding results. We also evaluated FuzzBT for one Zephyr release. In the end, we evaluated FuzzBT on the following branches:

- Linux longterm release kernel (5.15, 6.1, 6.6)
- Zephyr 4.0.0

We enabled KASAN, lockdep, UBSAN, hung task detection, and work queue stalls in the Linux kernel, and ASAN in Zephyr.

Feature ablation study. We conducted a feature ablation study to evaluate each feature's contribution. We disabled FuzzBT's all features in the baseline, and enabled each one as described in Table 3. We enabled edge coverage for the baseline as a feedback mechanism. The target Linux kernel version is 6.0 to ensure consistency for comparison with VirtFuzz in Section 7.4.

Comparison with existing fuzzers. Directly comparing FuzzBT with all fuzzers may be infeasible or unfair. Table 4 summarizes FuzzBT with existing Bluetooth fuzzing research, showing each fuzzer's target protocols and supported radio types. Frankenstein [30], SweynTooth [13], BrakTooth [12],

Label	Edge Coverage	State Coverage	Configuration Iteration	Configuration-aware Corpus
Baseline	O			
FuzzBT-btscov	O	O		
FuzzBT-config	O		O	
FuzzBT-corpus	O			O
FuzzBT	O	O	O	O

Table 3: Feature ablation study settings

BSFuzzer [49], and BLuEMan [18]'s entry point is the controller layer, while FuzzBT directly conveys input to the host stack. Also, SweynTooth and BrakTooth are not yet publicly available, and only proof-of-concept attacks with a controller part have been released. L2Fuzz [28] requires a physical dongle, whereas FuzzBT fully emulates a kernel and Bluetooth device with QEMU and VHCI driver.

Name	Radio Type	Protocols
SweynTooth [13]	LE	Host/Controller
BLuEMan [18]	LE	Host/Controller
BSFuzzer [49]	LE	Controller(Link Layer)
BrakTooth [12]	BR/EDR	Host/Controller
L2Fuzz [28]	BR/EDR	Host(L2CAP)
Frankenstein [30]	BR/EDR/LE	Controller
VirtFuzz [16]	BR/EDR/LE	Host
FuzzBT (Ours)	BR/EDR/LE	Host

Table 4: Summary of existing Bluetooth fuzzing research

We compared FuzzBT with VirtFuzz [16] and Syzkaller, which also targets the host stack via full emulation with QEMU. Both FuzzBT and VirtFuzz use a virtualized Bluetooth device in the Linux kernel. However, VirtFuzz uses libAFL, whereas FuzzBT uses Syzkaller. Also, VirtFuzz uses VirtIO, and FuzzBT uses a VHCI driver for Bluetooth device emulation. Thus, comparing coverage is not trivial. To handle this, we instrumented the same location (net/bluetooth) with the same number of probes. We also use the same coverage-collection implementation, KCOV, to ensure the collection overhead is equal. Thus, the location and total number of coverage instrumentation are the same, and the overhead is the same, allowing a fair comparison of coverage. We configured Syzkaller to enable only the dedicated system call for Bluetooth (`syz_emit_vhci`), to prevent fuzzing inputs from other system calls. By doing so, Syzkaller, FuzzBT, and VirtFuzz have a unified entry point into the Bluetooth host stack, the HCI layer.

VirtFuzz and FuzzBT do not use a uniformly distributed seed. VirtFuzz uses its pre-recorded Bluetooth packets as seeds, and FuzzBT extracts configuration-aware seeds from source code. We experimented with and without seeds for both fuzzers to determine the effectiveness of the seeds.

We set the Linux version to the latest supported by VirtFuzz, the kernel version 6.0 with Debian Stretch. We ran fuzzers in QEMU with 2GB of memory and 1 virtual CPU. In sum, our experimental setup for comparison is as follows:

- Syzkaller with uniformly random seed
- VirtFuzz with uniformly random seed
- VirtFuzz with VirtFuzz's seed
- FuzzBT with uniformly random seed
- FuzzBT with FuzzBT's seed

Note that Syzkaller with FuzzBT's seed is equivalent to FuzzBT-corpus, which is evaluated in Section 7.3.

Statistical significance. Due to the probabilistic nature of fuzzing, we executed fuzzers 20 times and 50 hours each to ensure statistical significance [31]. We applied the Mann–Whitney U test, bootstrapping estimation, and Vargha and Delaney A_{12} test to assess the significance of differences in the collected data. We adopt a significance level of 0.05 ($\alpha = 0.05$).

7.2 Bug Discovery

Target	Type	Protocol	Configuration	CVE
Linux	use-after-free	L2CAP	Basic	
	use-after-free	L2CAP	Basic	
	use-after-free	L2CAP	Basic	
	use-after-free	L2CAP	LE	CVE-2023-40283
	use-after-free	L2CAP	LE	
	use-after-free	L2CAP	N/S	CVE-2024-36013
	use-after-free	HCI	LE	
	use-after-free	HCI	MSFT	CVE-2024-36012
	use-after-free	A2MP	N/S	
	out-of-bound	HCI	AMP	CVE-2023-28866
	null-ptr-deref	L2CAP	Basic	
	null-ptr-deref	HCI	LE	CVE-2024-36011
	null-ptr-deref	HCI	N/S	CVE-2024-50255
	int overflow	L2CAP	LE (Flow control)	CVE-2024-36968
	int overflow	L2CAP	Basic	CVE-2022-45934
	div-by-zero	L2CAP	LE (Flow control)	CVE-2024-36968
	div-by-zero	HCI	High Speed (HS)	CVE-2024-38620
Zephyr	buffer overflow	HCI	N/S	

Table 5: Bugs found by FuzzBT. All bugs are previously unknown. All bugs are manually confirmed and fixed.

Bug triage. We did not count a simple crash as a bug. Instead, we consider the crash report is a bug if the maintainers fix the corresponding code. Otherwise, we consider the findings as simple crashes that do not require fixing. Also, we counted only previously unknown bugs. If FuzzBT found an already reported bug, which occasionally happens, we did not report it. A total of nine CVE numbers are assigned. CVE-2024-36968 is assigned to two bugs because Linux issues a CVE per a patch, and a single patch can fix both bugs. In sum, FuzzBT discovered 18 previously unknown bugs that were confirmed and fixed, resulting in 9 CVEs.

We juxtaposed protocols, configurations, and types with bugs in Table 5. We determined the vulnerable protocol by triggering input packets and corresponding bug fixes. We also manually checked whether a configuration is required to

Type	HCI		A2MP	L2CAP	
	Config S.	N/S	N/S	Config S.	N/S
use-after-free	2	0	1	5	1
out-of-bound read	1	1	0	0	0
null-ptr-deref	1	1	0	1	0
int overflow	0	0	0	2	0
div-by-zero	1	0	0	1	0
total	5	2	1	9	1

Table 6: Class of vulnerability discovered by FuzzBT. Configuration-specific vulnerabilities are labeled Config S., while those not specific to configurations are labeled N/S. We count buffer overflow as out-of-bound access.

trigger the bug-introducing functions. If a specific configuration is required to trigger the functions, we consider the bugs configuration-specific. If not, the functions are common and not specific to configurations. In this case, we marked N/S. For bug types, we followed the bug oracle's classification.

Table 6 further categorizes the bugs by bug types, protocols, and configuration dependency. We find that 14 bugs trigger invalid memory accesses that could be exploited for privilege escalation. Also, 5 bugs in HCI and 9 bugs in L2CAP are configuration-specific and are likely to remain hidden because Bluetooth packets alone are insufficient to trigger bugs without the required configurations. Regarding this, we detailed the analysis for CVE-2023-28866 and CVE-2024-36012 as follows.

CVE-2023-28866 was introduced in Oct. 2021 and remained unknown for 1.5 years until Mar. 2023. However, the patch for CVE-2023-28866 was trivial, adding one line, as follows:

```
@ struct hci_xxx amp_xxx[] = {
        ..., // function pointers
+       {}
 };
```

As shown, the missing indicator at the end of the array triggered an out-of-bounds access. Since the array element is a function pointer, this vulnerability modified the program counter, leading to a general protection fault.

Despite the patch's simplicity, the bug persisted for 1.5 years because triggering the bug depended on a configuration. To trigger this bug, the HCI configuration should be set to Alternative MAC/PHY (AMP) mode, which was originally developed for Bluetooth over WiFi. Without enabling AMP mode, the host stack does not execute AMP functions, including the vulnerable ones. Thus, the vulnerable AMP functions were unreachable in the first place, regardless of how fuzzers make Bluetooth packets.

CVE-2024-36012 was introduced in Jan. 2021, discovered in May 2024. Triggering this bug requires enabling Microsoft HCI extension support in the kernel configuration. Although it is developed by Microsoft, Linux has also adopted it, and users can optionally enable it. The default setting is disabled as it is not a part of the specification. Thus, the specification-guided approach and Bluetooth packets alone are insufficient to trigger this bug. Similarly, 14 bugs in Table 6 cannot be triggered without a specific configuration and have therefore remained unknown.

RQ1: FuzzBT discovered 18 previously unknown bugs with 9 CVEs.

7.3 Feature Ablation Study

To answer RQ2, we ran an ablation on FuzzBT, enabling one feature at a time. We evaluate each feature's performance by unique bug count, function coverage, and code coverage. We formulate three alternative hypotheses:

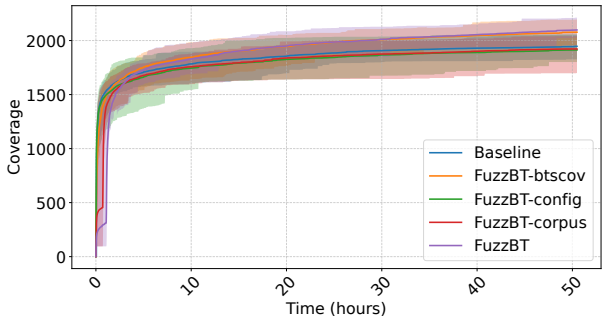
- H_1 : FuzzBT-config < FuzzBT-btscov
- H_2 : FuzzBT-corpus < FuzzBT-btscov
- H_3 : FuzzBT-config < FuzzBT-corpus

The corresponding null hypotheses($H_{0,n}$) are the inverses of the alternative hypotheses.

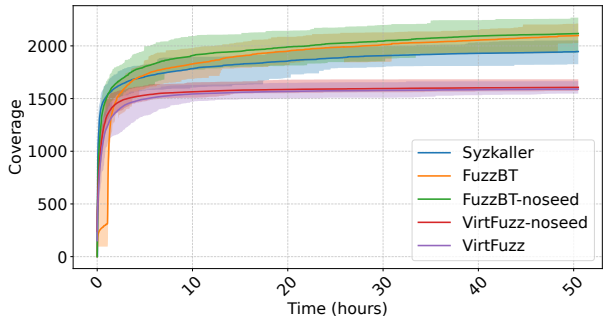
Unique bug comparison. Table 7a supports that FuzzBT-btscov reports the highest average number of bugs. However, it also exhibits the largest standard deviation, indicating that its performance varies more substantially across runs. Meanwhile, FuzzBT-config exhibits the lowest average number of discovered bugs, even lower than the baseline, as expected. As discussed in Section 4 **C1**, simply iterating over configurations may degrade fuzzing performance, since each configuration introduces distinct execution logic. Thus, BTSCov appears to be the most significant factor in unique bug findings.

Figure 7a further evidences this. We merged the bug reports from 20 fuzzing runs, deduplicated them, and drew an overlap diagram. Compared to the baseline, FuzzBT-btscov discovered 11 bugs exclusively, and FuzzBT-corpus and FuzzBT-config discovered 3 and 6 bugs exclusively.

Table 8a shows that BTSCov and configuration-aware corpus contributed more to unique bug findings than configuration iteration. Comparing FuzzBT-btscov with FuzzBT-config (H_1), the observed U statistic and A_{12} are 326.5 and 0.8163, meaning that in 81% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-config. Additionally, when comparing FuzzBT-corpus with FuzzBT-config (H_3), the observed U statistic and A_{12} are 119.5 and 0.2988, respectively. This indicates that in 29% of pairwise comparisons, FuzzBT-config achieves higher coverage than FuzzBT-corpus. Conversely, FuzzBT-corpus outperforms FuzzBT-config in 71% of the comparisons. Also, both results are statistically significant ($p < \alpha = 0.05$), and thus we reject the null hypotheses $H_{0,1}$ and $H_{0,3}$. However, the difference between FuzzBT-corpus and FuzzBT-btscov (H_2) is not statistically significant (p=0.13), so we do not reject $H_{0,2}$.



(a) Feature ablation settings (Table 3)



(b) Comparison with existing fuzzers

Figure 6: Code coverage over 50 hours. The solid line indicates the average. The shaded area represents the minimum and maximum values. Exact numbers are provided in Table 7.

Function coverage comparison. We converted to function coverage using `addr2line`. We mark a function as covered if any part of it is executed. Thus, a covered function may still contain uncovered portions.

Table 7a supports that BTSCov is the most influential feature to function coverage, showing the most number of covered functions on average. Conversely, FuzzBT-config and FuzzBT-corpus showed less function coverage than the baseline. As expected, simply iterating over configurations can destabilize fuzzing performance, since different configurations have distinct state machines. This leads to the highest observed standard deviation. A fuzzer may occasionally cover more functions than the baseline if the random seed fits to the configuration with a fortune. In this case, the maximum function coverage(376) can grow, but it typically covers fewer functions. Furthermore, the configuration-aware corpus has less impact because FuzzBT-config does not iterate over configurations. Consequently, most of the corpus is unused, resulting in lower function coverage.

Figure 7b describes the function coverage overlaps after 50 hours. Compared to the baseline, FuzzBT-btscov covered 16 additional functions, FuzzBT-corpus covered 3 additional functions, and FuzzBT-config covered 10 additional functions. In the overlap, BTSCov contributed the most to function coverage.

Table 8a shows that BTSCov is the most contributing factor to function coverage. Comparing FuzzBT-btscov with FuzzBT-config (H_1), the observed U statistic and A_{12} are 357 and 0.8925, meaning that in 89% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-config. Also, comparing FuzzBT-btscov with FuzzBT-corpus (H_2), the observed U statistic and A_{12} are 349 and 0.8725, meaning that in 87% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-corpus. Moreover, both results are statistically significant ($p < \alpha = 0.05$), and thus we reject the null hypotheses $H_{0,1}$ and $H_{0,2}$. However, the difference between FuzzBT-config and FuzzBT-corpus (H_3) is not statistically significant (p=0.5606), so we do not reject $H_{0,3}$.

Code coverage comparison. Figure 6a compares the coverage growth with the baseline fuzzer and each feature. FuzzBT and FuzzBT-corpus show a slow increase in the beginning, indicating the overhead of loading corpus. We summarized the numbers in Table 7a. On average, FuzzBT and FuzzBT-btscov showed increased coverage compared to the baseline. Other features, FuzzBT-config and FuzzBT-corpus, showed lower code coverage. Simply iterating over configurations without a corpus resulted in degraded fuzzing performance. Additionally, the configuration-aware corpus was largely unused when the configuration is fixed, further reducing fuzzing effectiveness.

Table 8a supports that BTSCov is the most contributing factor to code coverage. Comparing FuzzBT-btscov with FuzzBT-config (H_1), the observed U statistic and A_{12} are 366 and 0.085, meaning that in 91.5% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-config. Also, comparing FuzzBT-btscov with FuzzBT-corpus (H_2), the observed U statistic and A_{12} are 370 and 0.925, meaning that in 92.5% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-corpus. Moreover, both results are statistically significant ($p < \alpha = 0.05$), and thus we reject the null hypotheses $H_{0,1}$ and $H_{0,2}$. However, the difference between FuzzBT-config and FuzzBT-corpus (H_3) is not statistically significant (p=0.33), so we do not reject $H_{0,3}$.

RQ2: BTSCov contributed the most to overall fuzzing performance. Also, **reusing the seed across different configurations degraded fuzzing performance** because the fuzzer tries inputs that are nonsensical to other configurations.

7.4 Comparison with Existing Fuzzers

To answer RQ3, we compared FuzzBT with Syzkaller and VirtFuzz. The comparison metrics are unique bug count, function coverage, and code coverage. For statistical support, we formulate six alternative hypotheses:

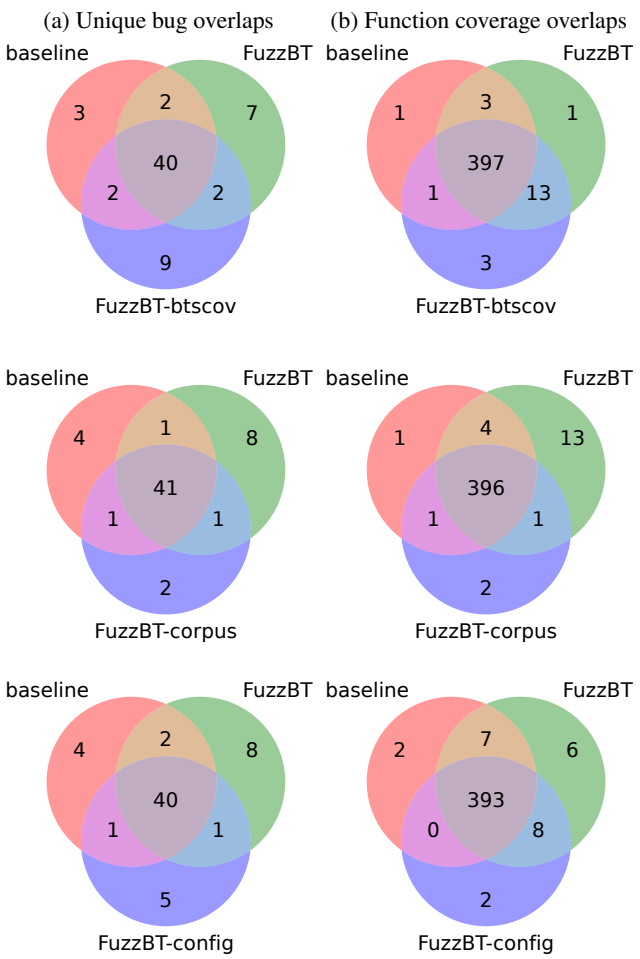


Figure 7: Overlaps of unique bugs and function coverage after 50 hours. We merged and deduplicated the 20 experiment results. Only Bluetooth functions are counted.

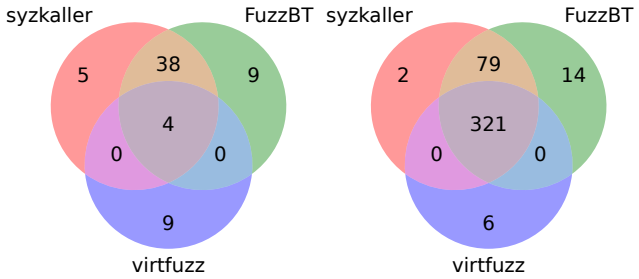
- H_1 : Syzkaller < FuzzBT-noseed
- H_2 : VirtFuzz-noseed < FuzzBT-noseed
- H_3 : VirtFuzz < FuzzBT-noseed
- H_4 : Syzkaller < FuzzBT
- H_5 : VirtFuzz-noseed < FuzzBT
- H_6 : VirtFuzz < FuzzBT

The corresponding null hypotheses($H_{0,n}$) are the inverses of the alternative hypotheses. We processed the data and tested the hypotheses in the same manner as described in Section 7.3.

Unique bug comparison. Table 7b supports that FuzzBT discovers the highest average number of bugs, regardless of whether high-quality or uniform seeds are used. Meanwhile, the uniform seed (FuzzBT-noseed) exhibits a higher standard deviation than FuzzBT, suggesting that seeding stabilizes the number of discovered bugs.

As shown in Figure 8a, FuzzBT discovered a total of 51 bugs, identifying more bugs than both Syzkaller and VirtFuzz. However, 42 of the 51 bugs were also identified by other fuzzers, leaving 9 exclusive bugs, matching the number of exclusive bugs found by VirtFuzz.

Table 8 shows that FuzzBT found more bugs than existing fuzzers. Comparing FuzzBT-noseed with Syzkaller (H_1), the observed U statistic and A_{12} are 279 and 0.6975, meaning that in 69% of pairwise comparisons, FuzzBT-noseed has higher coverage than Syzkaller. With a high-quality seed (H_4), the observed U statistic and A_{12} increase to 315 and 0.7875, respectively. This indicates that in 78% of pairwise comparisons, FuzzBT achieves higher coverage than Syzkaller. Also, the observed U statistic and A_{12} for VirtFuzz (H_2, H_3, H_5, H_6) reached its max, indicating the clear difference between FuzzBT and VirtFuzz. Six results are statistically significant ($p < \alpha = 0.05$), and thus we reject all null hypotheses.



(a) Unique crash overlap (b) Function coverage overlap

Figure 8: Overlaps of unique bugs and function coverage after 50 hours. We merged and deduplicated the 20 experiment results. Only Bluetooth functions are counted.

Function coverage comparison. Table 7b supports that FuzzBT covered the most functions compared to Syzkaller and VirtFuzz. On average, FuzzBT covered 371 functions, Syzkaller covered 353 functions, and VirtFuzz covered 297 functions. Without the configuration-aware corpus, FuzzBT-noseed still achieved higher function coverage than the others, but it also exhibited the largest standard deviation, indicating destabilized fuzzing results.

Figure 7b describes the function coverage overlaps after 50 hours. FuzzBT covered the most functions compared to existing works, both in total and uniquely. Compared to the Syzkaller and VirtFuzz, FuzzBT covered 14 and 93 more functions. Also, FuzzBT uniquely covered 14 functions while Syzkaller covers 2, and VirtFuzz covers 6.

Table 8 evident that FuzzBT showed the clear distinction in function coverage. Comparing FuzzBT-noseed with Syzkaller (H_1), the observed U statistic and A_{12} are 276.5 and 0.6913, meaning that in 69% of pairwise comparisons, FuzzBT-noseed achieved higher coverage. Comparison be-

tween FuzzBT with Syzkaller (H_4) provides the U statistic and A_{12} with 387 and 0.9675, meaning that in 96% of pairwise comparisons, FuzzBT has higher coverage than Syzkaller. Similarly, comparison with VirtFuzz (H_2, H_3, H_5, and H_6) showed the clear distinction. All results are statistically significant ($p < \alpha = 0.05$), and thus we reject all null hypotheses.

Code coverage comparison. Figure 6b compares the coverage growth with FuzzBT, Syzkaller, and VirtFuzz. Although FuzzBT has an initial overhead for loading the corpus, after 50 hours, FuzzBT reached more code. As shown in Table 7b, FuzzBT's lower 2.5% results exceed the maximum coverage achieved by Syzkaller and VirtFuzz.

Comparing FuzzBT-noseed with Syzkaller (H_1), the observed U-statistic and A_{12} are 392.5 and 0.9812, indicating that in the 98.1% of pairwise comparisons, FuzzBT-noseed has higher coverage than Syzkaller. The difference becomes greater for FuzzBT (H_4), meaning that FuzzBT achieves significantly higher coverage than Syzkaller. Against VirtFuzz-noseed and VirtFuzz (H_2, H_3, H_5, and H_6), the observed U statistic and A_{12} reached their maximum values (400 and 1.0), indicating a clear separation between FuzzBT and VirtFuzz, regardless of seed. Based on the p-value, we reject all null hypotheses.

RQ3: Compared to Syzkaller and VirtFuzz, FuzzBT found 20% and 146% more bugs and achieved 8.1% and 25% higher code coverage.

8 Discussion and Limitations

False positives in BTSCov. Our heuristic treats all fields in certain structures as state variables, thereby potentially overestimating the number of state variables. For example, pointer fields, which often store dynamically allocated memory addresses from memory allocators, are often mistakenly treated as state variables. In the Linux kernel, the false positive rate is 27%(129/484). We manually excluded these variables.

False negatives in BTSCov. Our heuristic considers only structures, potentially missing state variables defined outside these structures. We manually compared the specifications with two Bluetooth implementations, Linux and Zephyr. In our analysis, we did not observe any false negatives.

Real-world attack scenario. As a proof of concept, we validated CVE-2024-36968 in a real-world setting. As illustrated in Figure 9, we confirmed that the attacker's packets trigger a divide-by-zero error in the victim, resulting in a Bluetooth denial-of-service attack. Notably, the attacker requires a compromised HCI device attached to the victim. The victim kernel uses the default configuration, so no special kernel build is required to trigger this vulnerability. This is not a zero-click exploit. The victim must either approve a link request or actively connect to the attacker's Bluetooth device.

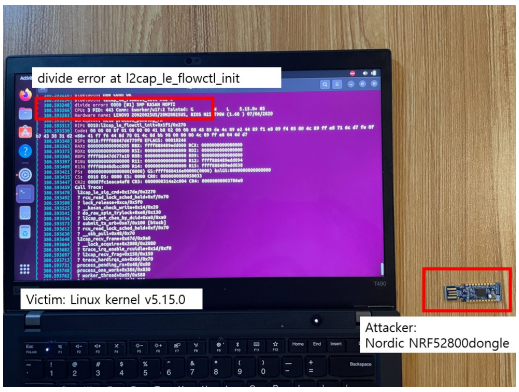


Figure 9: In the real world, CVE-2024-36968 allows a denial of service by causing division by zero. This attack does not require any particular changes in the victim kernel.

9 Related Work

Bluetooth security. Many approaches have been proposed to secure Bluetooth technology. Frankenstein [30], BSFuzzer [49], and FirmXRay [42] focused on the Bluetooth controller stack, while FuzzBT focuses on the host stack. VirtIO [16] proposed a device virtualization for Bluetooth host stack testing with profiling, without consideration of statefulness. BrakTooth [12] and Sweyntooth [13] statefully fuzzed the greybox Bluetooth implementation, referring to the state machine based on the specification, whereas FuzzBT focuses on implementations. The following works are also Bluetooth-related but solve different research problems. BLuEMan [18] increased the fuzzing throughput by compacting the whole Bluetooth stack into an executable. BLEDiff [19] proposed a non-compliance checking using differential testing. Wu et al. [44] identified a design flaw in the specification itself through formal verification. Compared to these, FuzzBT employs Bluetooth configurations in Bluetooth vulnerability identification.

General stateful fuzzing. Existing generic stateful fuzzers, such as SGFuzz [10], MendelFuzz [54], and AFLLive [11], typically focus on a single protocol, whereas FuzzBT is designed to handle a Bluetooth host stack. AFLNet [24] infers the protocol state from response codes, which provides less state information than directly monitoring state variables. Similar to IJON [8], StateFuzz [53], and SGFuzz [10], FuzzBT instruments state variables in source code. Profiling-based extraction methods, such as StateAFL [25] and MoonShine [27], require online pre-execution analysis to identify state variables or generate high-quality seeds, whereas FuzzBT's state variable identification and seed extractions are offline. NSFuzz [29] and Nyx-Net [32] take a snapshot for state collection. In contrast, FuzzBT first considers the configurations of the Bluetooth host stack.

Symbolic execution for kernel fuzzing. Since Driller [37] proved that symbolic execution could guide fuzzers to higher

code coverage, many works [10, 20, 21] have applied symbolic execution to kernel fuzzing. HFL [20] uses symbolic execution to extract a proper ordering of system calls. StateFuzz [53] leverages symbolic constraints to detect state transitions. FuzzUSB [21] leveraged offline symbolic execution to generate inputs that deterministically reach specific program points without consideration of state variables. In a similar fashion, FuzzBT performs symbolic execution to effectively resolve the Bluetooth configuration-specific input constraints.

10 Conclusion

While vulnerabilities within the Bluetooth host stack have caused critical security issues, existing Bluetooth fuzzing frameworks have mainly focused on the Bluetooth controller stack and the host stack has remained unexplored. In this paper, we present FuzzBT, a Bluetooth fuzzing framework for the host stack that explores configuration and stack state. We highlighted that the host stack has multiple configurations that substantially affect bug detection. Moreover, FuzzBT is guided by the overall host stack state rather than by a single protocol state, enabling more comprehensive coverage of the host stack. The aforementioned methodology was effective, as FuzzBT found 18 previously unknown bugs with 9 CVEs in the Linux and Zephyr kernels, achieving 8.2% higher code coverage according to the feature ablation study. We opened FuzzBT at https://doi.org/10.5281/zenodo.20172790.

Acknowledgments

The authors thank Luiz Augusto von Dentz and the Linux maintainers for their professional discussions and responses to bug reports. The authors also thank the anonymous reviewers of the paper and the artifact for their comments and suggestions, which helped improve the paper's quality. This work is supported in part by the National Science Foundation (NSF) under Award Number CNS-2145744, the Defense Advanced Research Projects Agency (DARPA) under contract number HR00112590042, the National Center for Transportation Cyber Security and Resiliency (TraCR) under Award Number 2589-211-2026310, SNSF 200021-236559, and the University of Texas System Rising STARs Award (No. 40071109). Any opinions, findings, recommendations, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

A Appendix

Ethical Considerations

Stakeholders. The key stakeholders in this research include Bluetooth developers and users, as well as the broader public who may be indirectly affected by the misuse of Bluetooth technology. This includes, but is not limited to, users of smartphones, laptops, and desktops. The release of FuzzBT aims to assist researchers and developers in identifying and mitigating Bluetooth vulnerabilities, thereby enhancing system security and contributing to a safer, more resilient network.

Potential impact. We recognize that FuzzBT could potentially be exploited by malicious actors to discover new vulnerabilities, which may facilitate harmful exploits. Although our goal is to enhance software security, the tools and techniques we develop carry an inherent risk of misuse. This risk is particularly pronounced given that FuzzBT could be adapted to target smartphones, potentially generating outcomes that breach ethical constraints, such as denial-of-service attacks or privacy violations.

Mitigations. While the core components of FuzzBT will be open-sourced, specific proof-of-concept exploits used for successful attacks will not be released in order to prevent misuse. This approach strikes a balance between transparency and research utility, while minimizing the risk of abuse. Furthermore, we will actively monitor community feedback and adjust dissemination practices as necessary to mitigate potential misuse. Our fuzzing tool is designed to identify security vulnerabilities in software. Failure to disclose such vulnerabilities responsibly could result in significant security risks, including unauthorized access or exploitation by malicious actors. To address this, we have established a responsible disclosure process. When vulnerabilities are discovered, we promptly notify the affected software vendors, allowing them sufficient time to patch the issues before any public disclosure. This process ensures that our research enhances security while minimizing risks to users.

Decision to both do and publish the research. While conducting and publishing research involves both benefits and risks, we believe the benefits outweigh the potential drawbacks. By identifying and addressing vulnerabilities before they can be exploited in the wild, FuzzBT proactively enhances security and reliability, safeguarding users' data and privacy. Our research is designed to minimize harm and make a positive contribution to the field of study. We consider our work to have both an ethically and socially beneficial overall impact. Accordingly, our ethical decisions prioritize the protection of Bluetooth users, aiming to identify vulnerabilities and enhance security.

Open Science

We are releasing our source code to advance open science and foster transparency and collaboration in research. FuzzBT is available at https://doi.org/10.5281/zenodo.20172790.

Metric	Label	mean			med	stdev	min	max
		arith	2.50%	97.50%				
Unique bug count	Baseline	19.04	17.3	20.7	19	3.9	11	25
	FuzzBT-btscov	23.21	21.2	25.15	22	4.36	17	31
	FuzzBT-config	18.02	16.55	19.6	18	3.5	12	26
	FuzzBT-corpus	21.07	19	23.05	21	4.24	16	28
	FuzzBT	22.78	22	23.8	23	2.06	19	29
Function coverage	Baseline	353.75	349.55	358	356	9.47	329	367
	FuzzBT-btscov	369.29	364.7	373.6	369	10.41	345	388
	FuzzBT-config	349.79	344.29	355.2	350	11.95	319	376
	FuzzBT-corpus	351.23	345.6	356.5	354	11.92	328	372
	FuzzBT	371.95	368.9	375.35	371	7.33	362	387
Code coverage	Baseline	1949.58	1917.98	1980.8	1960	50.93	1866	2035
	FuzzBT-btscov	2097.29	2069.14	2126.16	2117	68.55	1956	2195
	FuzzBT-config	1930.71	1895.25	1967.61	1902	83.7	1817	2088
	FuzzBT-corpus	1943.09	1898.69	1983.61	1972	96.21	1712	2063
	FuzzBT	2108.28	2077.5	2135.93	2104	46.5	2028	2179

(a) Feature ablation settings (Table 3). Figure 7 further provides the overlaps for bugs and function coverage.

Metric	Label	mean			med	stdev	min	max
		arith	2.50%	97.50%				
Unique bug count	Syzkaller	19.04	17.3	20.7	19	3.9	11	25
	VirtFuzz-noseed	9.46	8.9	10.05	9	1.24	7	12
	VirtFuzz	9.26	8.75	9.8	9	1.22	7	12
	FuzzBT-noseed	22.42	20.15	24.6	22	5.2	13	32
	FuzzBT	22.78	22	23.8	23	2.06	19	29
Function coverage	Syzkaller	353.75	349.55	358	356	9.47	329	367
	VirtFuzz-noseed	299.81	297.85	302.45	299	5.27	294	319
	VirtFuzz	297.24	295	299.8	296	5.49	291	316
	FuzzBT-noseed	360.84	351.85	368.8	368	19.29	314	388
	FuzzBT	371.95	368.9	375.35	371	7.33	362	387
Code coverage	Syzkaller	1949.58	1917.98	1980.8	1960	50.93	1866	2035
	VirtFuzz-noseed	1607.94	1600.3	1615.4	1606	12.16	1592	1628
	VirtFuzz	1604.92	1596.1	1615.3	1601	15.29	1579	1637
	FuzzBT-noseed	2090.65	2037.28	2147.94	2097	92.86	1935	2221
	FuzzBT	2108.28	2077.5	2135.93	2104	46.5	2028	2179

(b) Comparison with existing fuzzers. Figure 8 further provides the overlaps for bugs and function coverage.

Table 7: Fuzzing results statistics. Arith denotes the arithmetic mean. We performed 1,000 bootstrap resamples to estimate the mean coverage and its 95% confidence interval (2.5%-97.5% percentiles).

Metric	x	y	U-test	p-value	A_{12}
Unique bug count	FuzzBT-config	FuzzBT-btscov	326.5	0.0006	0.8163
	FuzzBT-corpus	FuzzBT-btscov	256	0.1321	0.64
	FuzzBT-corpus	FuzzBT-config	119.5	0.0298	0.2988
Function coverage	FuzzBT-config	FuzzBT-btscov	357	0	0.8925
	FuzzBT-corpus	FuzzBT-btscov	349	0.0001	0.8725
	FuzzBT-corpus	FuzzBT-config	178	0.5606	0.445
Code coverage	FuzzBT-config	FuzzBT-btscov	366	0	0.915
	FuzzBT-corpus	FuzzBT-btscov	370	0	0.925
	FuzzBT-corpus	FuzzBT-config	163.5	0.3300	0.4088

(a) Feature ablation settings (Table 3)

Metric	x	y	U-test	p-value	A_{12}
Unique bug count	Syzkaller	FuzzBT-noseed	279	0.0331	0.6975
	VirtFuzz-noseed	FuzzBT-noseed	400	0	1
	VirtFuzz	FuzzBT-noseed	400	0	1
	Syzkaller	FuzzBT	315	0.0018	0.7875
	VirtFuzz-noseed	FuzzBT	400	0	1
	VirtFuzz	FuzzBT	400	0	1
Function Coverage	Syzkaller	FuzzBT-noseed	276.5	0.0396	0.6913
	VirtFuzz-noseed	FuzzBT-noseed	398	0	0.995
	VirtFuzz	FuzzBT-noseed	399	0	0.9975
	Syzkaller	FuzzBT	387	0	0.9675
	VirtFuzz-noseed	FuzzBT	400	0	1
	VirtFuzz	FuzzBT	400	0	1
Code Coverage	Syzkaller	FuzzBT-noseed	373.5	0	0.0663
	VirtFuzz-noseed	FuzzBT-noseed	400	0	0
	VirtFuzz	FuzzBT-noseed	400	0	0
	Syzkaller	FuzzBT	392.5	0	0.9812
	VirtFuzz-noseed	FuzzBT	400	0	1
	VirtFuzz	FuzzBT	400	0	1

(b) Comparison with existing fuzzers

Table 8: Fuzzer comparison with Mann–Whitney U and Vargha-Delaney A_{12}. The sample size is 20. The value of U-test and A_{12} is higher if $x < y$.

References

[1] Pyeongju Ahn, Yeonseok Jang, Seunghoon Woo, and Heejo Lee. Bloomfuzz: Unveiling bluetooth l2cap vulnerabilities via state cluster fuzzing with target-oriented state machines. In European Symposium on Research in Computer Security, pages 110–129. Springer, 2024.

[2] Mingrui Ai, Kaiping Xue, Bo Luo, Lutong Chen, Nenghai Yu, Qibin Sun, and Feng Wu. Blacktooth: breaking through the defense of bluetooth in silence. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pages 55–68, 2022.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Nearby threats: Reversing, analyzing, and attacking google's' nearby connections' on android. 2019.

[4] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Bias: Bluetooth impersonation attacks. In 2020 IEEE Symposium on Security and Privacy (SP), pages 549–562, 2020.

[5] Daniele Antonioli, Nils Ole Tippenhauer, Kasper Rasmussen, and Mathias Payer. Blurtooth: Exploiting cross-transport key derivation in bluetooth classic and bluetooth low energy. In Proceedings of the 2022 ACM on Asia conference on computer and communications security, pages 196–207, 2022.

[6] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B Rasmussen. The {KNOB} is broken: Exploiting low entropy in the encryption key negotiation of bluetooth {BR/EDR}. In 28th USENIX Security Symposium (USENIX Security 19), pages 1047–1061, 2019.

[7] Armis. Blueborne cyber threat impacts amazon echo and google home, 2017.

[8] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1597–1612, 2020.

[9] Pierre Ayoub, Romain Cayre, Aurélien Francillon, and Clémentine Maurice. Bluescream: Screaming channels on bluetooth low energy. In 2024 Annual Computer Security Applications Conference (ACSAC), pages 636–649. IEEE, 2024.

[10] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In 31st USENIX Security Symposium (USENIX Security 22), pages 3255–3272, 2022.

[11] Octavio Galland and Marcel Böhme. Invivo fuzzing by amplifying actual executions. In Proceedings of the 47th International Conference on Software Engineering (ICSE'25), 2025.

[12] 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, 2022.

[13] 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, 2020.

[14] Hadi Givehchian, Nishant Bhaskar, Eliana Rodriguez Herrera, Héctor Rodrigo López Soto, Christian Dameff, Dinesh Bharadia, and Aaron Schulman. Evaluating physical-layer ble location tracking attacks on mobile devices. In 2022 IEEE symposium on security and privacy (SP), pages 1690–1704. IEEE, 2022.

[15] Dennis Heinze, Jiska Classen, and Matthias Hollick. Toothpicker: Apple picking in the ios bluetooth stack. In Yuval Yarom and Sarah Zennou, editors, 14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020. USENIX Association, 2020.

[16] 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 24–24. IEEE Computer Society, 2023.

[17] Mohit Kumar Jangid, Yue Zhang, and Zhiqiang Lin. Extrapolating formal analysis to uncover attacks in bluetooth passkey entry pairing. In NDSS, 2023.

[18] Wei-Che Kao, Yen-Chia Chen, Yu-Sheng Lin, Yu-Cheng Yang, Chi-Yu Li, and Chun-Ying Huang. Blueman: A stateful simulation-based fuzzing framework for open-source rtos bluetooth low energy protocol stacks. August 2025.

[19] Imtiaz Karim, Abdullah Al Ishtiaq, Syed Rafiul Hussain, and Elisa Bertino. Blediff: Scalable and property-agnostic noncompliance checking for ble implementations. In 2023 IEEE Symposium on Security and Privacy (SP), pages 3209–3227, 2023.

[20] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In NDSS, 2020.

[21] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In 2022 IEEE Symposium on Security and Privacy (SP), pages 2212–2229. IEEE, 2022.

[22] Norbert Ludant, Tien D Vo-Huu, Sashank Narain, and Guevara Noubir. Linking bluetooth le & classic and implications for privacy-preserving bluetooth-based protocols. In 2021 IEEE Symposium on Security and Privacy (SP), pages 1318–1331. IEEE, 2021.

[23] 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, 2019.

[24] Ruijie Meng, Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet five years later: On coverage-guided protocol fuzzing. IEEE Transactions on Software Engineering, 2025.

[25] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. Empirical Software Engineering, 27(7):191, 2022.

[26] Andy Nguyen. Bleedingtooth: Linux bluetooth zero-click remote code execution, 2020.

[27] Shankara Pailoor, Andrew Aday, and Suman Jana. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In 27th USENIX Security Symposium (USENIX Security 18), pages 729–743, 2018.

[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] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. ACM Transactions on Software Engineering and Methodology, 32(6):1–26, 2023.

[30] 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, 2020.

[31] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. In 2024 IEEE Symposium on Security and Privacy (SP), pages 1974–1993. IEEE, 2024.

[32] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In Proceedings of the seventeenth european conference on computer systems, pages 166–180, 2022.

[33] Ben Seri and Gregory Vishnepolsky. Blueborne: Unveiling zero day vulnerabilities and security flaws in modern bluetooth stacks., 2017.

[34] Ben Seri and Gregory Vishnepolsky. Bleedingbit: The hidden attack surface within ble chips., 2018.

[35] Min Shi, Jing Chen, Kun He, Haoran Zhao, Meng Jia, and Ruiying Du. Formal analysis and patching of {BLE-SC} pairing. In 32nd USENIX Security Symposium (USENIX Security 23), pages 37–52, 2023.

[36] Pallavi Sivakumaran, Chaoshun Zuo, Zhiqiang Lin, and Jorge Blasco. Uncovering vulnerabilities of bluetooth low energy iot from companion mobile apps with ble-guuide. In Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, pages 1004–1015, 2023.

[37] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In NDSS, volume 16, pages 1–16, 2016.

[38] Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick. Disrupting continuity of apple's wireless ecosystem security: New tracking,{DoS}, and {MitM} attacks on {iOS} and {macOS} through bluetooth low energy,{AWDL}, and {Wi-Fi}. In 30th USENIX security symposium (USENIX Security 21), pages 3917–3934, 2021.

[39] Tyler Tucker, Hunter Searle, Kevin Butler, and Patrick Traynor. Blue's clues: Practical discovery of non-discoverable bluetooth devices. In 2023 IEEE Symposium on Security and Privacy (SP), pages 3098–3112. IEEE, 2023.

[40] Maximilian Von Tschirschnitz, Ludwig Peuckert, Fabian Franzen, and Jens Grossklags. Method confusion attack on bluetooth pairing. In 2021 IEEE symposium on security and privacy (SP), pages 1332–1347. IEEE, 2021.

[41] D. Vyukov. Syzkaller, 2015.

[42] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 167–180, 2020.

[43] 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 WOOT@ USENIX Security Symposium, 2020.

[44] Jianliang Wu, Patrick Traynor, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. Finding traceability attacks in the bluetooth low energy specification and its implementations. In 33rd USENIX Security Symposium (USENIX Security 24), pages 4499–4516, 2024.

[45] Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave Tian, and Antonio Bianchi. Sok: The long journey of exploiting and defending the legacy of king harald bluetooth. In 2024 IEEE Symposium on Security and Privacy (SP), pages 23–23. IEEE Computer Society, 2023.

[46] Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. Formal model-driven discovery of bluetooth protocol design vulnerabilities. In 2022 IEEE Symposium on Security and Privacy (SP), pages 2285–2303. IEEE, 2022.

[47] Xinyi Xie, Kun Jiang, Rui Dai, Jun Lu, Lihui Wang, Qing Li, and Jun Yu. Access your tesla without your awareness: Compromising keyless entry system of model 3. In NDSS, 2023.

[48] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals. In NDSS, 2019.

[49] Ting Yang126, Yue Qin, Lan Zhang, Zhiyuan Fu, Junfan Chen, Jice Wang, Shangru Zhao, Qi Li78, Ruidong Li, He Wang, et al. Bsfuzzer: Context-aware semantic fuzzing for ble logic flaw detection.

[50] Tingfeng Yu, James Henderson, Alwen Tiu, and Thomas Haines. Security and privacy analysis of samsung's {Crowd-Sourced} bluetooth location tracking system. In 33rd USENIX Security Symposium (USENIX Security 24), pages 5449–5466, 2024.

[51] Yue Zhang and Zhiqiang Lin. When good becomes evil: Tracking bluetooth low energy devices via allowlist-based side channel and its countermeasure. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pages 3181–3194, 2022.

[52] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking secure pairing of bluetooth low energy using downgrade attacks. In 29th USENIX Security Symposium (USENIX Security 20), pages 37–54, 2020.

[53] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. {StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing. In 31st USENIX Security Symposium (USENIX Security 22), pages 3273–3289, 2022.

[54] Han Zheng, Flavio Toffalini, Marcel Böhme, and Mathias Payer. Mendelfuzz: The return of the deterministic stage. Proceedings of the ACM on Software Engineering, 2(FSE):44–64, 2025.