SweynTooth: Unleashing Mayhem over Bluetooth Low Energy

Matheus E. Garbelini
SUTD

Chundong Wang*
ShanghaiTech University

Sudipta Chattopadhyay
SUTD

Sumei Sun
*Institute for Infocomm Research, A*Star*

Ernest Kurniawan
*Institute for Infocomm Research, A*Star*

Abstract

The Bluetooth Low Energy (BLE) is a promising short-range communication technology for Internet-of-Things (IoT) with reduced energy consumption. Vendors implement BLE protocols in their manufactured devices compliant to Bluetooth Core Specification. Recently, several vulnerabilities were discovered in the BLE protocol implementations of a few specific products via a manual approach. Considering the diversity and usage of BLE devices as well as the complexity of BLE protocols, we have developed a systematic and comprehensive testing framework, which, as an automated and general-purpose approach, can effectively fuzz any BLE protocol implementation. Our framework runs in a central device and tests a BLE device when the latter gets connected to the central as a peripheral. Our framework incorporates a state machine model of the suite of BLE protocols and monitors the peripheral's state through its responses. With the state machine and current state of the central, our framework either sends malformed packets or normal packets at a wrong time, or both, to the peripheral and awaits an expected response. Anomalous behaviours of the peripheral, e.g., a non-compliant response or unresponsiveness, indicate potential vulnerabilities in its BLE protocol implementation. To maximally expose such anomalies for a BLE device, our framework employs an optimization function to direct the fuzzing process. As of today, we have tested 12 devices from eight vendors and four IoT products, with a total of 11 new vulnerabilities discovered and 13 new Common Vulnerability Exposure (CVE) IDs assigned. We call such a bunch of vulnerabilities as SWEYNTOOTH, which highlights the efficacy of our framework.

1 Introduction

The Bluetooth Low Energy (BLE) is one of the key wireless communication technologies behind the massive progress of internet-of-things (IoT). Hence, vulnerabilities in the BLE protocol implementation may lead to concrete and serious aftermath. For instance, through reverse engineering on Broadcom's BLE System-on-Chip (SoC) devices, Mantz et al. [24] performed remote code execution in the device's functions with a malformed over-the-air packet. Similarly, BleedingBit [15], discovered in Texas Instruments BLE SoCs, allows adversaries to install a shellcode, which thereafter permits remote execution and authentication bypass upon receiving specific sequences of manipulated advertisement packets.

The preceding examples indicate that faulty BLE protocol implementations may exist in various IoT devices and potentially bring about chaotic consequences. In this paper, we propose a systematic and automated fuzzing framework that is able to discover vulnerabilities in the BLE protocol implementation of any device. Our framework neither requires access to the source code of an implementation nor changes a single line of code in a device's OS or firmware. In a nutshell, it runs in the user space of a customized BLE dongle (i.e., central) to test a BLE device (i.e., peripheral) during the process of establishing a connection between the two.

The essence of our framework is a fuzzer that systematically subjects the BLE implementation to adversarial conditions. However, it is non-trivial to develop a fuzzer to generate such adversarial conditions. Firstly, we construct a BLE state machine model from the Bluetooth Core Specification [36–38] to make valid BLE packets. This is essential, as a randomly generated, meaningless packet is likely to be rejected by any BLE implementation. Secondly, testing a BLE implementation with valid BLE packets is improbable to reveal flaws, because such compliant cases should have been covered in manufacturing tests [22, 41] as well as in Bluetooth stack certification [39]. Thus, our framework either sends malformed packets based on mutation, or normal packets at a wrong time or inappropriate state, or both, to a BLE peripheral. Through manipulating packets, our framework intends to bring on adverse *corner cases*. Thirdly, the complex structure of BLE packets (cf. Figure 1) and the versatile communication regulations necessitate a comprehensive and directed strategy for generating test cases of packets and

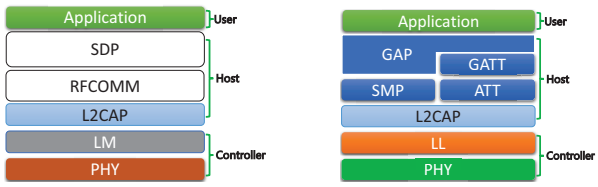
*This work was done when C. Wang worked at SUTD.

their timings. This aims to drive and stress non-compliant behaviours at the peripheral. To this end, our fuzzer mutates fields of a layer in the BLE stack and employs a particle swarm optimization (PSO) to heuristically refine the mutation probability distribution at both dimensions of each protocol's layers and each layer's fields. Finally, our framework validates any response from a peripheral on-the-fly according to a set of expected packets in each protocol state. This enables it to detect security issues beyond crashes, e.g., security bypass.

Our framework distinguishes itself from existing works [6, 15, 24] in view of being automated and comprehensive. Existing works require manual and tedious efforts, such as reverse engineering and attentive inspection of source code, to discover potential security flaws in the BLE implementation of specific devices [34]. By contrast, our framework is fully automated and embraces the capability to uncover more security issues than a manual approach. Concurrently, although a few scattered approaches have been presented in fuzzing Bluetooth devices [3, 9, 11, 18], they only cover a fraction of the Bluetooth stack. To the best of our knowledge, we compose the first comprehensive approach for BLE fuzzing that is not limited to one or several particular layers, e.g., L2CAP or ATT [3, 11], but fully controls the communication at the Link Layer (LL) as well as the interaction with the Secure Manager Protocol (SMP) for encrypted message exchanging. This, in turn, establishes the efficacy and viability of our framework in fuzzing arbitrary BLE protocol implementations.

The remainder of this paper is organized as follows. In particular, we present the following contributions.

- We present our fuzzing framework to discover implementation flaws for BLE protocols (Section 2).
- We present the optimization process embodied in our fuzzing framework to discover critical security vulnerabilities. We also discuss the systematic process of validating responses from BLE peripheral (Section 3).
- We discuss the implementation specific challenges in our approach and evaluate our fuzzing framework on several commodity BLE SoCs, including SoCs from NXP, Dialog, Texas Instruments, Microchip, ST Microelectronics and Cypress, among others. Our evaluation has revealed 11 unknown security vulnerabilities (nicknamed SWEYNTOOTH) and seven non-compliant behaviours. 13 new common vulnerability exposure (CVE) IDs are assigned and they potentially affect a few hundred types of IoT products. As all the vulnerable SoCs have passed the Bluetooth stack certification, our evaluation also clearly highlights the incompleteness of the certification process (Section 4).
- We evaluate the impact of new vulnerabilities, as discovered by our framework, on four IoT products (Section 4).
- We compare our framework with three other fuzzers and show that our framework is significantly more effective, in terms of finding security vulnerabilities in BLE implementations (Section 4).



(a) Classic (b) BLE

Figure 1: The Stacks of Bluetooth Classic and BLE

After discussing related work (Section 5), we conclude the paper and provide future directions (Section 6).

2 Overview of Our Framework

BLE is the successor of Bluetooth Classic to build a short-range wireless network with reduced energy consumption and improved usage capability. In this section, we first describe the BLE model used in our fuzzing and illustrate the challenges in developing a systematic fuzzing framework for BLE protocols with an example. Then we present an overview of our framework with its main components and workflow.

2.1 The Model of BLE Protocols

We aim to detect *implementation* flaws in BLE protocols defined in the Bluetooth Core Specification [36–38]. Particularly, we study the interactions on Attribute Protocol (ATT), Logical Link Control and Adaptation Protocol (L2CAP), Secure Manager Protocol (SMP), and Link Layer (LL), as shown in Figure 1. L2CAP and ATT are common to both Bluetooth Classic and BLE, while LL and SMP are exclusive to BLE.

Figure 2 illustrates the process of establishing the BLE connection between a central and a peripheral. Our fuzzer works during this process and it is guided by a BLE protocol model we have developed. A simplified representation of the model is presented in Figure 3. Initially, the peripheral periodically broadcasts `advertisements` to nearby devices and the central starts in the `scanning` state. The central scans for such `advertisements` and gets further information from the peripheral such as its name by sending a `scan request` (❶ in Figure 2). After receiving a `scan response` (❷ in Figure 2) from the peripheral, the central can choose to start a connection by sending a `connection request` (❸ in Figure 2) and proceeds to the `connection` state. On receiving an acknowledgment from the peripheral (❹ in Figure 2), the central proceeds to the `initial_setup` state (see Figure 3). As the `connection request` contains connection parameters relevant to the synchronization and communication timing between central and peripheral, after transiting to `initial_setup` state, the central requests information from the peripheral by sending `version request`, `feature request`, `length request` and `MTU length request` (❺ to ❽ in Figure 2) with the intention to know the peripheral's

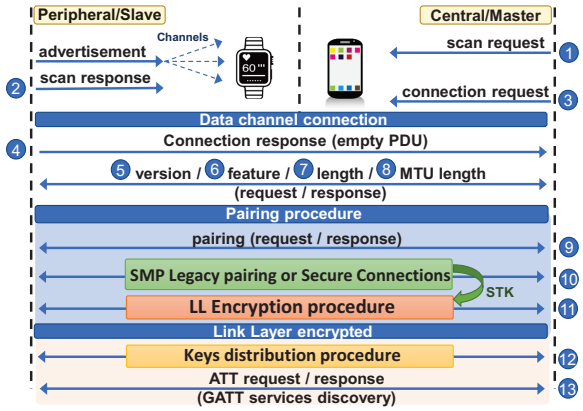


Figure 2: Message exchanges during BLE connection process

supported LL features and capabilities such as the maximum length of the packet it can send or receive. Likewise, the peripheral also gets the central's LL information during the same exchanges. Note that the preceding messages are not necessarily sequentially exchanged, because vendors are free to implement how the peripheral handles such messages. For instance, a peripheral may reply to `version` before `feature`. Similarly, the peripheral may choose to directly read some ATT atributes from the central and go to the `gatt_server` state or skip the state `length` before proceeding. To ensure compatibility with different implementations, we employ several transitions in the state `initial_setup` for the flexible message ordering, as shown at the upper-left of Figure 3.

After the initial setup is done, the central proceeds to the `list_pri_services` state. Here it scans for peripheral's main services via the Generic Attribute Profile (GATT) Service Discovery procedure and stores their attributes in a local array. The central then proceeds to the state `pairing_req` and starts to establish an encrypted communication with the peripheral. The central sends a `pairing request` packet to the peripheral (9 in Figure 2), indicating the preferred pairing mode to be used in the next state. If the peripheral accepts the pairing mode proposed by the central, it replies to the central and both proceed to the `smp_pairing` state. As there are two pairing modes for them to choose, i.e., the Legacy pairing or Secure Connection (SC) pairing via SMP exchanges, they go through the pairing procedure from either the `legacy_pairing` or `sc_pairing` state, as shown at the middle-right of Figure 3. Once the pairing procedure is successful, the central derives a `sessionKey` from a Short Term Key (STK) received from `smp_pairing`, transits to the `ll_encryption` state and starts the challenge with the peripheral by sending an `encryption_req` (10 in Figure 2). With the peripheral's response, the central sends an encrypted `encryption_res` packet by using the obtained `sessionKey`. If the peripheral is able to correctly authenticate and decrypt the `encryption_res` from the central, it sends another encrypted `encryption_res` to the central, indicating that the connection is successfully encrypted.

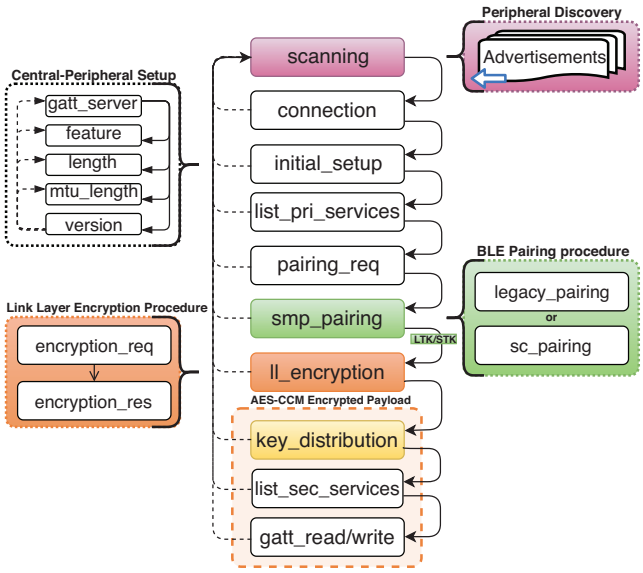


Figure 3: Simplified BLE Protocol Model

If `legacy_pairing` is used, the central and peripheral may optionally go through the `keys distribution procedure` (12 in Figure 2) to exchange a long term key (LTK).Otherwise, in `sc_pairing`, the LTK is the STK instead. The LTK can then be used by the central to avoid repeating the pairing process in subsequent connections and directly go to the step 11 in Figure 2. In the following stages, the central and peripheral exchange an LTK based on what has been negotiated in `pairing_req` and the central reads more services from the peripheral at the state `list_sec_services`.

After LL connection and pairing, the central discovers all the peripheral's available attributes (i.e., information) by performing the GATT Primary Service Discovery. This consists of sending and receiving a number of `ATT requests` and `ATT responses` (13 in Figure 2). so as to fetch predefined ATT attributes. In the next state `gatt_read/write`, we capture the read and write of locally stored ATT attributes at the `list_pri_services` and `list_sec_services` states. This step is to emulate writing malformed ATT attributes via our fuzzing methodology. Thus, the state `gatt_read/write` at the bottom of Figure 3 is not part of the BLE protocol specification. However, it is required to check the behaviour of a peripheral in the presence of malformed ATT attributes.

2.2 Problem Formulation with An Example

In this paper, we consider developing a systematic fuzzing framework that is 1) comprehensive with respect to all BLE stack layers, 2) directed as being with an optimization mechanism to maximally expose anomalies in BLE protocol implementations, and 3) applicable to fuzzing any product embracing BLE SoCs for wireless connectivity. Anomalous behaviours capture non-compliance against the Bluetooth Core Specification. To guarantee the comprehensiveness of cov-

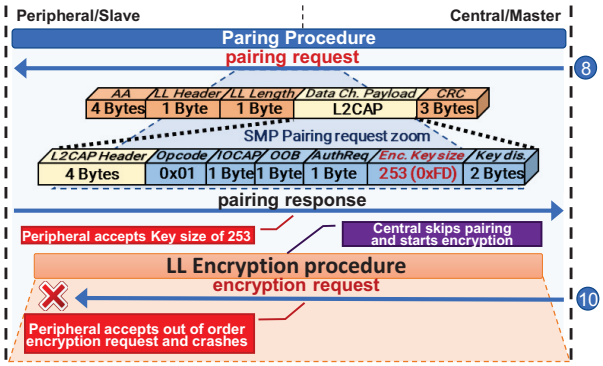


Figure 4: *Key Size Overflow* in Telink SoC (*CVE-2019-19196*)

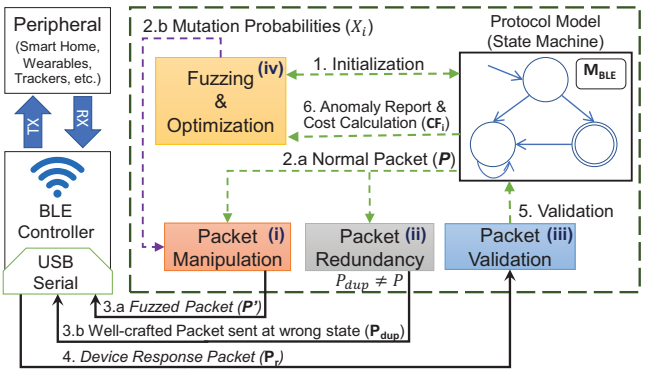


Figure 5: An Illustration of Fuzzing Architecture.

ering all protocol layers, we attentively study the Bluetooth Core Specification and incorporate an all-inclusive state machine model as presented in Section 2.1 at the central side. Thus, at the current state of the central, we monitor responses from the peripheral to check whether they are aligned with the Bluetooth Core Specification or not.

Technical Challenges: Section 2.1 indicates that devising a comprehensive state machine model itself is the first challenge due to the complexity of BLE connections. As shown in Figure 1, each of the BLE layers contains multiple fields that might be an exploitable factor. Furthermore, compared to Wi-Fi, BLE allows *move-back* and *move-forward* state transitions if a timeout event occurs and an expected response arrives, respectively. This also introduces the second challenge, i.e., the timing-critical constraints that must be accounted for fuzzing BLE SoCs. Thirdly, an online validation of peripheral responses is non-trivial at the central side. According to the Bluetooth Core Specification, at a given state, the central waits for two types of responses, i.e., normal responses and failure responses. The latter is a valid response, as a well-formed peripheral has the right-of-way to deny any illegal or unaligned request. Such a feature, again, does not exist in Wi-Fi protocols. Consequently, special care is demanded to distinguish expected and anomalous packets in the context of BLE communications. Last but not the least, uncovering vulnerabilities in BLE implementations requires systematically directing the fuzzing framework. In the following, we take an example vulnerability, i.e., *Key Size Overflow* (*CVE-2019-19196*) discovered by our framework, to illustrate how we resolve the aforementioned challenges.

Discovering* Key Size Overflow *Vulnerability: The *Key Size Overflow* vulnerability is caused only if the three following conditions are jointly satisfied: 1) `key_size` field of `SMP pairing request` is fuzzed, 2) the peripheral receives a certain packet in an inappropriate state, and 3) the peripheral may send a connection failure packet depending on the received fuzzed packet. The vulnerability is illustrated in Figure 4.

In brief, as a fuzzer, our framework mutates protocol layers and each layer's fields in a packet sent from the central to the peripheral under test. The mutation is based on probabilities assigned at both dimensions of layers and fields. It refines such probabilities via a cost function with a return value, say, the count of discovered anomalies, to direct the fuzzing process. Our framework identifies an anomaly by validating received responses. It discovers *Key Size Overflow* as follows. Initially, there is no information about the vulnerabilities. Therefore, the mutation probabilities are randomly assigned. Eventually, at the `paring_req` state, the fuzzer sends a `paring_request`, yet with fields other than `key_size` mutated. The peripheral sends a response `SM failure`, which is still deemed to be normal by the online validation of our fuzzer. Next, the fuzzer sends a malformed packet with mutated `key_size`. *Caveat Lector*: the peripheral of Telink Semiconductor unexpectedly replies with a valid `paring_response` for such a fuzzed, invalid request. Our framework legitimately catches this response as an anomaly. As a result, the mutation probability to fuzz the field `key_size` is increased. Thus, more malformed `pairing_request` packets with mutated `key_size` are sent to the peripheral. We note that our fuzzer also sends valid packets, but at an inappropriate state of the client. Eventually, the fuzzer sends an `encryption_request` to the peripheral immediately after the malformed `pairing_request` packets with mutated `key_size`. This crashes the peripheral, as detected due to the lack of any response from it.

To sum up, the `Key Size Overflow` presents an anomaly and a crash for BLE SoCs manufactured by Telink. During the fuzzing process, the scenario to send a malformed `paring_request` (with mutated `key_size`) followed by an `encryption_request` increases. This is because the response to these malformed packets are anomalous and such anomalous responses increase the value of the cost function (i.e., anomaly count). This, in turn, further increases the probability to fuzz `key_size` and indirectly, the likelihood of discovering the scenario causing the vulnerability.

2.3 High Level Workflow

System Architecture: Figure 5 illustrates the architecture of our fuzzer, which is composed of four main modules organized around the BLE model M_{BLE}: (i) the module of packet manipulation that mutates a packet, (ii) the module of packet

Algorithm 1 Main Steps of our fuzzer

```
1:  i ← 0                                              ▷ i captures fuzzing iteration
2:  ▷ generate BLE protocol model (cf. Figure 3)
3:  M_BLE ← Generate_Protocol_Model()
4:  ▷ wait to receive mutation probabilities from PSO
5:  X_i ←-- Particle_Swarm_Opt()
6:  ▷ initialize history of sent packets and redundant packets
7:  ℙ_hist ← ∅, P' ← ∅, P_dup ← ∅, P^h_dup ← ∅, S_0 ← ∅
8:  repeat
9:      Set central to be in scanning state
10:     ▷ assign expected layers
11:     For each S ∈ M_BLE, assign {expected(S), rejection(S)}
12:     repeat
13:         Wait for peripheral's packet
14:         Let the central receives packet P_r from the peripheral
15:         ▷ monitor states and checks anomalies
16:         (θ_anom, P_r) ← Run_Validation(S, P', P^h_dup, P_r)
17:         S_0 ← S; S ← Get_Current_State(M_BLE, P_r)
18:         ▷ exit the iteration on anomalies and no transition
19:         if θ_anom is false or S_0 = S then
20:             goto line 37
21:         end if
22:         ▷ generate a valid packet from the model
23:         P ← Get_Packet_from_Model(M_BLE, S)
24:         ▷ generate fuzzed packets from P via mutation
25:         P' ← Mutate_Packet(P, X_i)
26:         Send fuzzed packets P' to the peripheral
27:         P^h_dup ← P_dup
28:         Choose a packet P_dup ∈ ℙ_hist ∪ {∅} s.t. P_dup ≠ P
29:         Send redundant packet P_dup to the peripheral
30:         ▷ switch expected layers after fuzzing
31:         if P' ≠ P then
32:             expected(S) ← rejection(S)
33:         end if
34:         ℙ_hist ← ℙ_hist ∪ {P}
35:     until central does not reach the scanning state
36:     ▷ measure cost function value for X_i
37:     CF_i ← Measure_Cost_Function(X_i)
38:     ▷ send cost function value to PSO
39:     Particle_Swarm_Opt() ←-- CF_i
40:     ▷ wait to receive new mutation probabilities from PSO
41:     X_{i+1} ←-- Particle_Swarm_Opt()
42:     i ← i + 1
43: until timeout
```

redundancy that sends arbitrary packets of M_{BLE} to the peripheral at unaligned states (i.e., out of order) with the intention to trigger anomalies on the peripheral's protocol state machine, (iii) the module of packet validation that is responsible for checking the responses from the peripheral and detecting anomalies based on the current state of M_{BLE}, and (iv) the module of fuzzing & optimization that can direct the mutation of packets based on a cost function.

As shown by the arrows in Figure 5, the four modules of our fuzzer interact and collaborate with each other to attain the aim of discovering potential vulnerabilities in a peripheral device. Algorithm 1 illustrates the workflow of it.

Initialization: The fuzzer relies on the protocol model M_{BLE} to generate valid packets and a set of mutation probabilities X_i to probabilistically mutate such valid packets. At the initialization stage (Lines 3 to 5 in Algorithm 1), the fuzzer first loads the model M_{BLE} and receives initial mutation probabilities X_i from the optimization module (iv in Figure 5) by calling the `Particle_Swarm_Opt` function (Line 5). Next, the central is set to the `scanning` state and proceeds to wait for the peripheral's advertisement (Lines 9, 13 to 14). Once the central receives a packet P_r from the peripheral, the validation module (iii in Figure 5) checks whether P_r is expected or not via the `Run_Validation` function (Line 16). In short, the validation module decides the correctness of P_r based on a set of expected layers `expected(S)` or rejection layers `rejection(S)`, which are generated for every state $S \in M_{BLE}$ (Line 11) at startup. The validation is detailed in Section 3.2.

Fuzzing Iteration: If the validation does not detect any anomaly, P_r is fed to trigger the state transition in the model M_{BLE} by calling the `Get_Current_State` function (Line 17). `Get_Current_State` strictly follows the protocol model described in Section 2.1 and returns the new state S of M_{BLE}. Then at the state S, our framework generates a valid packet P (Line 23), which serves as an input to the manipulation and redundancy modules (i and ii in Figure 5). Starting with the packet manipulation via the `Mutate_Packet` function (Line 25), the contents of P are mutated according to the mutation probabilities X_i associated with the state S, resulting in a mutated packet P' (see Section 3 for details of mutation). Due to the probabilistic nature of X_i, the mutation yields either an incorrect packet such that $P' \neq P$ (i.e., malformed) or a mutated packet which doesn't differ from the original packet (i.e., $P' = P$). If a malformed packet P' is sent to the peripheral, the Bluetooth Core Specification allows the peripheral to respond with a packet that rejects P', i.e., one with a layer in the `rejection(S)`. Thus, the fuzzer perceives an anomaly if the response for a malformed P' is other than a legitimate packet with one of its layers in `rejection(S)`. To this end, the expected set of layers (`expected(S)`) for state S is set to the rejection layers for state S (`rejection(S)`) (Line 32).

The redundancy module (iii in Figure 5) keeps a history $\mathbb{P}_{hist}$ (initialized as $\emptyset$ at Line 7) of all the packets P generated by the model M_{BLE} (Line 34) and sends a redundant packet $P_{dup} \in \mathbb{P}_{hist}$ to the peripheral at random chance (Lines 28 to 29). The intention of this logic is to send out-of-order packets that may cause crash or anomalous behaviour onto the peripheral. However, using redundancy may trigger some ambiguous behaviour which is not necessarily an anomaly. For example, some BLE packets are not only tied to one single state and responses to them at a different state should not be flagged anomalous by the fuzzer. In Section 3.2, we present how the validation module resolves such challenges and avoids reporting false positives.

The fuzzing iteration finishes in one of three circum-

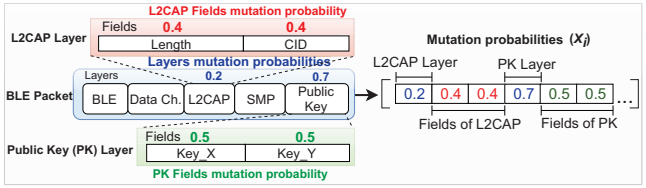


Figure 6: An illustration of our fuzzing. X_i shows the probability values for the packet `public_key` at state S.

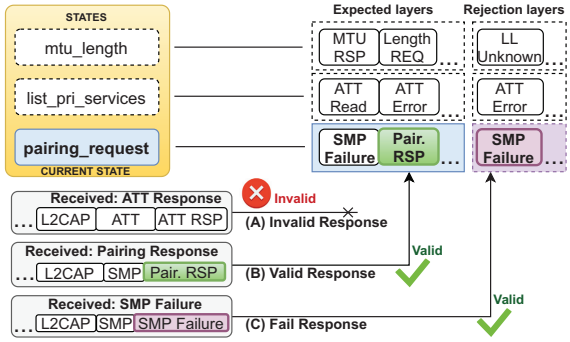


Figure 7: Packet dissection and validation during fuzzing

stances: 1) when the model M_{BLE} reaches the end state `gatt_read/write` (cf. Figure 3) and goes back to `scanning` state, 2) an anomaly is detected (Line 20), or 3) the fuzzer times out due to a crash in the peripheral (Line 13).

Optimization: Once a fuzzing iteration finishes, the mutation probabilities X_i are refined by the optimization module (iv in Figure 5) via particle swarm optimization (Lines 37 to 41). The optimization uses the value of cost function CF_i obtained at the end of every fuzzing iteration (`Measure_Cost_Function`). The rationale of optimization is to guide the mutation probabilities X_i in such a fashion that the value of cost function CF_i is maximized. Specifically, the value of CF_i represents a metric that can direct X_i to fuzz packets that are more likely to optimize CF_i (e.g., the number of anomalies). Moreover, the refined mutation probabilities X_{i+1} are computed iteratively via `Particle_Swarm_Opt` and carried over to the next iteration (Line 41). This approach allows our fuzzer to be directed and facilitates the search for anomalies in the peripheral's protocol implementation.

3 Design of Fuzzer

3.1 Fuzzing and Optimization

The fuzzing effectiveness critically depends on the generation of malformed packets based on mutation. In the following, we discuss how such mutations are performed in detail.

Mutation: On receiving a generated packet from the protocol model, the fuzzing module evaluates it according to the set of mutation probabilities X_i. X_i represents the probabilities to mutate a packet along two dimensions: 1) the `layers`, which correspond to different protocols or packet types of a packet, and 2) the `fields`, each belonging to a layer in the packet. Figure 6 exemplifies the assignment of X_i over the layers and fields of a BLE packet. For instance, consider the Public Key layer to illustrate the use of X_i in generating a packet. The fields of `Key_X` and `Key_Y` can be mutated in an iteration only if the manipulation module randomly hits the layer probability chance (70%). Once a hit happens, the fuzzer needs to decide the set of fields in the layer to be mutated. To this end, the fuzzer iterates over each field within the layer and uses the individual mutation probability (50%) to mutate such fields. We note that all the fields of one layer shares the same mutation probability. This is to reduce the number of parameters during the iterative optimization (cf. Line 39 in Algorithm 1) without losing the efficacy significantly. When the mutation indeed occurs onto a field, the field value is changed via a randomly-chosen `Mutation Operator`.

Mutation Operators: The fuzzing module offers three `Mutation Operators`: 1) **Random bytes** that mutates the value of a packet's fields with random bytes, 2) **Zero filling** that clears the field value to zero, and 3) **Bit setting** that sets the most significant bit of a single-byte field value. The rationale of choosing such operators is to accelerate the search process for an anomaly. In practicality, `Zero filling` and `Bit setting` correlate to setting lower or higher values of a field value to manifest corner cases. These, in turn, are probable to trigger a buffer overflow or underflow in a peripheral's implementation that lacks comprehensive bound checks.

Optimizing Mutation Probabilities: In order to effectively discover anomalies (e.g., crashes or non-compliant behaviours against the Bluetooth Core Specification), our fuzzer employs a cost function to systematically guide the optimization process. The rationale behind such an approach is to measure a cost function value CF_i that informs how well a certain set of mutation probabilities X_i perform with respect to finding new anomalies. Therefore, the goal of the fuzzer is to maximize the discovery of potential anomalies by also maximizing the value of such a cost function. We use the *number of unique anomalies* discovered throughout the fuzzing session as the cost function. This is measured for each individual set of mutation probabilities X_i (cf. Line 37 in Algorithm 1).

The set of mutation probabilities X_i are refined while maximizing the cost function value on each fuzzing iteration by an optimization algorithm (cf. Line 41 in Algorithm 1). For the optimization, we apply the particle swarm optimization (PSO) due to its superior performance in the light of non-linear and stochastic behaviour shown in the protocol model [32]. Moreover, PSO has been successfully applied in a state-of-the-art software fuzzer [23]. The goal of PSO is to optimize the value of a chosen cost function via regulating the *position* of the *swarm of particles* (i.e., the population). In the context of our framework, the *position* is a probability value and each particle within the *swarm of particles* represents a different set of mutation probabilities X_i.

3.2 Packet Validation

The validation module detects responses that deviate from the Bluetooth Core Specification. It emphasizes on the correctness of a response in its *internal packet structure*, i.e., layers of the response, and the *correct reception order*, i.e., the response's arriving state. In particular, given a response packet received at state S, the validation module checks it among `Expected layers` or `Rejection layers` that are dedicated to state S in accordance with the protocol model M_{BLE}.

Validation Exemplified: Figure 7 shows three different cases where a packet from the peripheral arrives in response to a packet sent to the peripheral at state $S = \texttt{pairing\_request}$. The packet sent to the peripheral can either be a valid packet P or a mutated packet P'. In **case (A)**, on receiving the `ATT Response` due to a valid P, the validation module flags it as anomaly as none of the layers in the response is found in the `Expected layers` of state S. In **case (B)**, the response packet is deemed to be valid (i.e., `pairing_response`) since its layer is found in the `Expected layers`. On the other hand, after sending a malformed packet to the peripheral, our fuzzer only expects `Rejection layers` (Line 32 in Algorithm 1). In this sense, in **case (C)**, our fuzzer sends a mutated packet P' to the peripheral, and the response with `SMP Failure` is valid as a rejection of P', as $\texttt{SMP Failure} \in \mathit{rejection}(S)$.

Validation Procedure: More involved cases beyond Figure 7 exist. The validation module must correctly handle responses received due to legitimate, mutated, and/or redundant requests sent at both proper and improper states.

Algorithm 2 illustrates the function `Run_Validation` called in Algorithm 1. It validates if a response P_r is anomalous or not. The response P_r, received at state S, might be due to possible P' and P_{dup} sent in an arbitrary fuzzing iteration (Lines 1 to 5). At start, the validation module prepares the `Expected layers` in ε to be searched for P_r, as P_r might be a response to a non-empty P_{dup} (Line 6 to 10). We first compute the flag Ψ for state S. Ψ holds if the expected layers at S overlap with the expected layers of some other state S' in the protocol model M_{BLE} (Line 7). The flag Ψ does not hold for security-related states such as states involved in SMP pairing and Link Layer encryption, e.g., `smp_pairing` and `ll_encryption`. Specifically, these states (with Ψ false) do not accept any response except those aligned to their respective `Expected layers`. We then check whether a non-empty P_{dup} has been sent at any state M^p_{BLE} (Line 8). The set M^p_{BLE} is a subset of all BLE states (M_{BLE}). Specifically, response to a packet sent at a state $S' \in M^p_{BLE}$ is allowed to be received at any state where Ψ holds (i.e., states other than security-related ones). Thus, given a non-empty P_{dup} sent at a state of M^p_{BLE}, the validation module needs to extend ε if Ψ holds. This is accomplished by joining ε with `Expected layers` of the state P_{dup} belongs to (Lines 8 to 10). With the updated ε, the validation module sets a validity flag based on whether the layers of P_r are expected or not (Lines 11 to 12).

Algorithm 2 `Run_Validation Procedure`

1: **Input:** Current state S of BLE protocol model (cf. Figure 3)
2: **Input:** Packet P' sent from the current state S
3: **Input:** Packet P_{dup} sent at the immediately preceding state of S
4: **Input:** Packet P_r sent from BLE peripheral
5: **Output:** Absence of anomaly (*true* or *false*)
6: $\varepsilon \leftarrow \mathit{expected}(S)$
7: $\Psi \leftarrow \exists S' \in M_{BLE} \setminus \{S\}.\ (\varepsilon \cap \mathit{expected}(S')) \neq \emptyset$
8: **if** $\left(P_{dup} \neq \emptyset\right) \wedge \Psi \wedge \left(\mathit{state\_of}(P_{dup}) \in M^p_{BLE}\right)$ **then**
9: $\quad \varepsilon \leftarrow \varepsilon \cup \mathit{expected}(\mathit{state\_of}(P_{dup}))$
10: **end if**
11: ▷ Check if the received packet P_r is valid
12: is_valid $\leftarrow \exists l \in \mathit{layers\_of}(P_r)$ s.t. $l \in \varepsilon$
13: **if** $\left(P_{dup} \neq \emptyset\right) \wedge \left(\mathit{state\_of}(P_{dup}) \in M^o_{BLE}\right)$ **then**
14: $\quad M^p_{BLE} \leftarrow M^p_{BLE} \setminus \{\mathit{state\_of}(P_{dup})\}$
15: **end if**
16: $M^p_{BLE} \leftarrow \left(S \in M^o_{BLE}\right)\ ?\ \left(M^p_{BLE} \setminus \{S\}\right) : M^p_{BLE}$
17: ▷ Prevent redundant P_r from transiting the state machine
18: **if** $\left(P_{dup} \neq \emptyset\right)$ **and** (P' and P_{dup} have the same response) **then**
19: $\quad$ Wait for peripheral's response packet P_r
20: $\quad$ `Run_Validation`$(S, P', \emptyset, P_r)$
21: **end if**
22: **return** (is_valid, P_r)

The validation performs further acts before returning to Algorithm 1. Firstly, in M^p_{BLE} there is a subset, i.e., M^o_{BLE}. The response to the request sent at a state of M^o_{BLE} is allowed to be received in other states, but *only once*. One such state is the `Version` state. A normal peripheral responds to the `version request` only once irrespective of how many `version requests` it receives. Hence, if P_{dup} or P' belongs to some state $S' \in M^o_{BLE}$, then S' is removed from M^p_{BLE}. This ensures that future responses to P_{dup}, which belongs to state S', are classified as anomalies (Lines 13 to 16). Secondly, P_{dup} and P' may have the same response. In this case, we do not trigger a state transition until a response to P' is received (if any before the fuzzer times out). Specifically, after handling the response for P_{dup}, the validation module is recursively called with an empty P_{dup} (Lines 17 to 21). In the end, the anomaly flag and P_r are returned (Line 22).

Crash detection: There are two options to detect a crash or unresponsiveness of the peripheral. The intrusive option is applicable to BLE development boards that expose serial debug ports of their respective SoCs. We can use the debug information to detect a crash. For BLE products without such debug ports, we use a global timer and clear it on every packet response. If no response is received from the peripheral, the timer eventually overflows and a crash is signalized.

3.3 Non-compliant BLE Controller

Manipulation of the Link Layer is essential for fuzzing. However, the Core Specifications [37] undermines Link Layer (LL) manipulation from the host. Firstly, LL packets are heavily timing critical due to BLE frequency-hopping. The host

cannot send a packet in a precise time due to the high time variability of the OS scheduler. Secondly, the LL stack runs on a separate and closed source Bluetooth chipset, i.e. the controller. The chipset normally communicates with the host via the Host Controller Interface (HCI) protocol, which does not expose manual control over the LL stack.

To overcome the aforementioned challenges with a *practical* and *low cost* solution, we design a *non-compliant* BLE controller firmware that ignores standardised conventions such as HCI and abstracts away the timing and retransmission requirement between the central and the peripheral. This abstraction simplifies the BLE state machine and allows the host to manipulate all fields of the Link Layer packets.

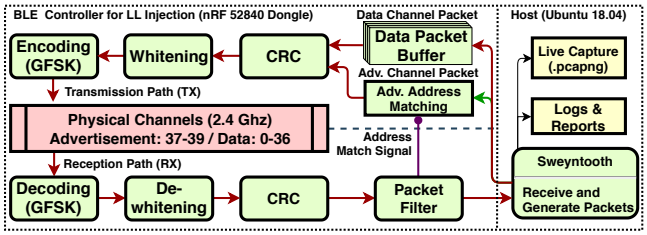


Figure 8: An Illustration of the transmission and reception path of a BLE Packet via the non-compliant BLE controller

Figure 8 details the internals of the non-compliant BLE controller depicted in the fuzzer architecture (cf. Figure 5). The controller reads packets from the host and transmits according to their radio channel type, which is inferred from the *access address* of the packet header. Data channel packets are buffered in the `Data Packet Buffer` and released for transmission after a time period defined by the connection interval. Concurrently, an advertisement packet is only transmitted to the peripheral after the controller receives an advertisement packet from the peripheral first. Upon reception, the `Packet Filter` checks the packet for the peripheral address and upon a match, the advertisement packet stored in the `Adv. Address Matching` is released and transmitted to the peripheral after the inter-frame spacing $\Delta_{IF} = 150us$. Other procedures such as `CRC` calculation, `whitening/dewhitening` and `encoding/decoding` are only necessary to ensure the correct encoding of the packet during over-the-air transmission and as such, do not expose fields for host side fuzzing.

4 Evaluation

Implementation: Our implementation efforts have been mainly spent on two parts: 1) the fuzzer, including the modules of fuzzing, validation and optimization, and 2) the non-compliant BLE controller that enables the over-the-air fuzzing. The fuzzer is written in Python 2.7 and C++ with a total number of 2,836 lines of code (LOC). In brief, our fuzzer extends the `Scapy` v2.4.3 [33] to recognize packet types, parse and validate a response from the peripheral. It also uses the BLESuite library [31] to handle the GATT Service Discovery. As to the fuzzing and optimization, our fuzzer leverages the PyGMO library and its Generational PSO implementation [10] with the optimizer following the common PyGMO structure and the default *pygmo.pso_gen* optimization parameters.

Table 1: Development Platforms used for evaluation

Silicon Vendor	Development Platform	BLE Ver.	Sample Code Name
Cypress (PSoC 6)	CY8CPROTO-63	5.0	Device_Information_Service
Cypress (PSoC 4)	CY5677	4.2	Device_Information_Service
Texas Instruments	LaunchXL-CC2640R2	5.0	project_zero
Texas Instruments	CC2540EMK-USB	4.1	simple_peripheral
Telink	TLSR8258 USB	5.0	8258_ble_sample
STMicroelectronics	NUCLEO-WB55	5.0	BLE_BloodPressure
STMicroelectroncis	STEVAL-IDB008V2	5.0	SlaveSec_A0
NXP	USB-KW41Z	4.2	heart_heart_rate_sensor_bm
Dialog	DA14681DEVKIT	4.2	ble_adv
Dialog	DA14580DEVKIT	4.1	ble_app_peripheral
Microchip	SAMB11 Xplained	4.1	blood_pressure_samb11
Nordic Semi.	nRF51 Dongle	5.0	ble_app_hrs
Nordic Semi.	nRF52840 Dongle	5.0	ble_app_gatts_c

The non-compliant BLE controller is written in C++ (1,096 LOC) within the nRF52840 dongle as the central device. It overcomes the isolation enforced by HCI (cf. Section 3.3).

Evaluation Setup: Table 1 shows the peripheral devices that we have tested. In each of these devices, the CPU is a microcontroller (SoC) that runs an undisclosed BLE stack implementation. IoT products using these devices only have access to interfaces for BLE communications provided by respective manufacturer-provided libraries. As a result, the device's BLE implementation runs alongside the product's main code, and a BLE implementation vulnerability may lead to catastrophic failure and insecurities into the product's functionalities. In other words, once BLE devices are found vulnerable, so are the IoT products relying on them.

We need to install a firmware in each brand new device to enable BLE connectivity. This is accomplished by compiling and programming a sample code provided by the device's corresponding SDK. Once a programmed device advertises itself as BLE peripheral, we can start our fuzzer to test it.

We answer the following research questions (RQs) through the evaluation of our fuzzer.

RQ1: How effective is our fuzzer in terms of generating error-prone inputs?

A summary of testing results is depicted by Table 2. The prefix **V** means a vulnerability while the prefix **A** means some anomalous behaviour that deviates from the legitimate behaviour defined by the Bluetooth Core Specification but is not a vulnerability. Overall, our fuzzer has discovered 11 new vulnerabilities and seven anomalous behaviours over all tested devices. The SoCs of particular vendors, e.g., Texas Instruments, NXP, Cypress and Dialog, have been used in many IoT products for Smart Home, wearables and gadget tracking. These vulnerabilities expose their respective SoCs to crashes, deadlocks or even a complete or partial bypass of pairing procedure. Hence the impact is significant. It's important to emphasize that all vulnerabilities have been automatically discovered by our fuzzer during the packet exchange, except for vulnerabilities classified as `Security Bypass`. After a `Security Bypass` is detected and classified as an anomaly

Table 2: Summary of new vulnerabilities and other anomalies found on the tested platforms. * indicates the case, which is not clear by the Bluetooth Core Specification [36–38]

Vulnerabilities / Inconsistencies		Platform(s)	Model state(s)	Impact Type	Compliance Violated
V1 - Link Layer Length Overflow V2 - Link Layer LLID Deadlock	(CVE-2019-16336, CVE-2019-17519) (CVE-2019-17061, CVE-2019-17060)	CY8CPROTO-063 CY5677 USB-KW41Z	*initial_setup*	Crash Crash, Deadlock	[Vol 1] Part E, Section 2.7
V3 - Silent Buffer Overflow	(CVE-2019-17518)	DA14681 DEVKIT-B	*smp_pairing*	Crash	[Vol 1] Part E, Section 2.7
V4 - Truncated L2CAP Packet	(CVE-2019-17517)	DA14580 DEVKIT-B	*list_pri_services*	Crash	[Vol 1] Part E, Section 2.7
V5 - Unexpected Public Key V6 - DHCheck Skipping	(CVE-2019-17520) (CVE-2020-13593)	LaunchXL-CC2640R2	smp_pairing	Crash Security Bypass	[Vol 1] Part E, Section 2.7 [Vol 3] Part H, Section 2.3.5.6.5
V7 - Invalid connection request	(CVE-2019-19193)	CC2540EMK-USB	*connection*	Deadlock	N.A
V8 - Sequential ATT message	(CVE-2019-19192)	NUCLEO-WB55 STEVAL-IDB008V2	*gatt_read/write*	Crash	[Vol 1] Part E, Section 2.7
V9 - Invalid L2CAP fragment	(CVE-2019-19195)	SAMB11 Xplained	*list_pri_services* *gatt_read_write*	Crash	[Vol 1] Part E, Section 2.7
V10 - Key size overflow V11 - Zero LTK installation	(CVE-2019-19196) (CVE-2019-19194)	TLSR8258 USB	*pairing_req* sc_pairing	Crash Security Bypass	[Vol 3] Part H, Section 3.5.1 [Vol 3] Part H, Section 2.4.4
A1 - Unexpected encryption start response*		SAMB11 Xplained TLSR8258 USB USB-KW41Z	*pairing_request* *smp_pairing*	Non-specified	N.A
A2 - Accept non-zero EDIV and Rand during Secure Connection pairing		LaunchXL-CC2640R2 NUCLEO-WB55 STEVAL-IDB008V2 CY5677	*sc_pairing* *ll_encryption*	Non-Compliance	[Vol 3] Part H, Section 2.4.4.1
A3 - Responds to VERSION_IND more than once			**many**	Non-Compliance	[Vol 6] Part B, Section 5.1.5
A4 - Responds to data channel PDUs during encryption procedure		TLSR8258 USB	*ll_encryption*	Non-Compliance	[Vol 6] Part B, Section 5.1.3.1
A5 - Sends unknown LL control PDU opcode			*smp_pairing*	Non-Compliance	[Vol 6] Part B, Section 2.4.2
A6 - Accepts malformed CONNECT_IND		CC2540EMK-USB	*connection*	Non-Compliance	[Vol 6] Part B, Section 2.3.3.1
A7 - Accepts CONNECT_IND with hopIncrement less than 5		**All tested devices**	*connection*	Non-Compliance	[Vol 6] Part B, Section 2.3.3.1

Table 3: Vulnerabilities and SDK versions of the affected SoCs. * indicates vendors that reported other affected SoCs.

Silicon Vendor	BLE SoC	SDK Ver.	Vuln. / Anomalies
BLE Version 5.0			
Cypress (PSoC 6)	CYBLE-416045	2.10	V1,V2 / A7
Texas Instruments	CC2640R2	2.2.3	V5,V6 / A1,A7
Telink*	TLSR8258	3.4.0	V10,V11 / A3-A5,A7
STMicroelectronics	WB55	1.3.0	V8 / A2,A7
STMicroelectroncis	BlueNRG-2	3.1.0	V8 / A2,A7
Nordic Semi.	nRF51422	11.0.0	A7
Nordic Semi.	nRF52840	15.3.0	A7
BLE Version 4.2			
Cypress (PSoC 4)	CYBL11573	3.60	V1,V2 / A7
NXP	KW41Z	2.2.1	V1,V2 / A1,A7
Dialog*	DA14680	1.0.14.X	V3 / A7
BLE Version 4.1			
Texas Instruments	CC2540	1.5.0	V7 / A6,A7
Dialog*	DA14580	5.0.4	V4 / A7
Microchip	ATSAMB11	6.2	V8 / A2,A7

by our fuzzer, a manual check is required to classify it as a security issue. We note that twelve CVEs have been assigned, but at the time of writing this paper, the details of vulnerabilities V1-V11 were publicly undisclosed for confidentiality. Moreover, we followed responsible disclosures and notified all vendors 90 days in advance for them to provide corresponding patches. At the time of writing, all vendors except STMicroelectronics and Microchip have released their patches. Table 3 highlights the SoCs and the SDK versions where these vulnerabilities were first discovered.

For each anomaly, Table 2 also outlines the specific section of the Bluetooth Core Specification being violated. To summarize, the results signalize that the current status of BLE security demands more attention not only onto the design of protocols, but also onto the implementation phases. Specifically, the two critical security bypass vulnerabilities (V6 and V11) are caused due to the lack of handling corner cases in the Bluetooth Core Specification, causing misinterpretations and implementation flaws. A detailed description of the vulnerabilities is shown in the supplemental material.

Table 4: A Summary of Evaluation Time for Each Device. The connection interval is fixed to 20*ms* for all devices.

Platform	Iterations	Total Time	1st Crash	1st Anomaly	Model Coverage
CY8CPROTO-63	1000	1 h. 06 min.	1 min.	<1 min.	27 (50.0%)
CY5677	1000	2 h. 27 min.	<1 min.	8 min.	29 (53.7%)
USB-KW41Z	1000	1 h. 30 min.	<1 min.	2 min.	24 (44.4%)
DA14681DEVKIT	1000	1 h. 16 min.	10 min.	6 min.	30 (55.5%)
DA14580DEVKIT	1000	2 h. 7 min.	5 min.	1 min.	32 (59.3%)
CC2640R2 Devkit	1000	1 h. 57 min.	4 min.	1 min.	31 (57.40%)
CC2540 Devkit	1000	1 h. 37 min.	2 min.	19 min.	34 (62.96%)
Nucleo-WB55	1000	1 h. 45 min.	<1 min.	2 min.	26 (48.15%)
BlueNRG-2	1000	1 h. 14 min.	<1 min.	9 min.	30 (55.55%)
ATSAMB11	1000	2 h. 39 min.	2 min	10 min.	33 (61.1%)
TLSR8258	1000	1 h. 56 min.	5 min.	<1 min.	36 (66.67%)

RQ2: How efficient is our fuzzer?

When our fuzzer exchanges packets with the peripheral, the efficiency in finding anomalies mainly depends on two factors, i.e., the connection interval and the peripheral's capabilities. While the first factor can be initiated by the central, the peripheral decides whether to accept the value of the connection interval proposed by the central. The connection interval is the time between consecutive messages and thus controls the frequency of messages exchanged between central and peripheral. It is negotiated at the `connection` state. A short connection interval naturally leads to an efficient fuzzing process. During the fuzzing process, the connection interval is fixed to a value that is acceptable to all tested devices. Table 4 shows the overall time taken by our fuzzer to complete 1,000 iterations with a connection interval of 20*ms*. Due to the diverse capabilities of devices, the message-processing time varies significantly even with the same connection interval.

For instance, the CY5677 device is much slower in the pairing procedure, resulting in the longest evaluation time.

The time required to find the first vulnerability in a peripheral's implementation depends on its features. As shown by the rightmost two columns of Table 4, most of the first crash or other anomaly have been discovered within 10 minutes. As a result, our fuzzer is opportune to ascertain a vulnerable implementation of BLE device.

Finally, the last column of Table 4 holds the number of different valid transitions traversed in our BLE state machine (cf. Figure 3) after 1000 iterations. Specifically, the BLE model employs a total of 54 valid transitions. Overall, each peripheral traverses the model differently and does not trigger all possible valid transitions in our BLE model. This is because states *initial_setup*, *list_pri_services* and *list_sec_services* allow multiple transitions and peripheral implementations differ in terms of the exact packet sequence accepted in such states. This results in peripherals missing some transitions employed in the BLE model. As per coverage efficiency, the fuzzer takes more time to fully explore unstable peripherals. This is the case for peripherals impacted by vulnerabilities triggered in states with multiple transitions (V1, V2 and V8). For example, peripherals from Cypress, NXP and STMicroelectronics exhibit a slightly lower coverage value for 1000 iterations.

RQ3: How do the different design choices contribute to the effectiveness of our fuzzer?

To answer this question, we disable two components of our fuzzer to make two variants, respectively. Firstly, we only keep the redundancy module active without packet mutation or optimization. This means packets are sent at a wrong state to the peripheral. Secondly, our fuzzer solely relies on the mutation module without optimization. In this sense, we mutate valid packets from the protocol model M_{BLE} according to a random set of mutation probabilities X_i that is not refined after each iteration. The two variants are referred to as "Redundancy" and "Mutation", respectively.

Figure 9 illustrates the number of anomalies with respect to fuzzing iteration for each relevant BLE SoC. The "Evolution" represents the results achieved by our fuzzer with the optimization, which serves as a reference to compare against the two variants. In all cases, "Evolution" results in finding all anomalies due to the collaborative contributions among all fuzzing components, while the two variants miss some anomalies (cf. Figure 9). This is expected and shows that certain vulnerabilities can only be triggered by either redundancy, mutation or a combination thereof. For example, the vulnerability `Key Size Overflow` (V10, cf. Section 2) associated with Telink TLSR8258, requires that the mutation and redundancy complement during the fuzzing process to trigger it. That explains the superior effectiveness of "Evolution" in Figure 9(b). Also in Figure 9, "Mutation" cannot achieve as many anomalies as "Redundancy". This is because many anomalies indicated for "Redundancy" are due to the fact that A3 to A5 are triggered upon the peripheral receiving redundant packets in the BLE connection, but not by "Mutation" through sending malformed packets.

Table 5: A Comparison among Testing Tools: *Handcrafted* means tests can be manually configured, whereas a *Test Database* contains a corpus of tests for validation

Comparison			Crashes / Anomalies	
Tools	**Supported Layer(s)**	**Fuzzing Strategy**	**WB55, BlueNRG-2**	**Others**
Stack Smasher	L2CAP	Random	0 / 0	0 / 0
BLEFuzz	ATT	Random / Handcrafted	1 / 0	0 / 0
bfuzz (IotCube)	L2CAP	Random / Test database	1 / 0	0 / 0
Our Fuzzer	LL / L2CAP / SMP / ATT	Evolutionary	1 / 2	10 / 7

RQ4: How effective is our fuzzer compared to existing BLE fuzzing tools?

We compare the competitiveness of our fuzzer by evaluating it against publicly available tools, including `Stack Smasher`, `BLEFuzz`, and `bfuzz` that most closely match the objective of our fuzzer. We note that handcrafted efforts were required to apply these tools. Firstly, `bfuzz` and `Stack Smasher` demand modifications so that they can send malformed packets through our BLE controller. Secondly, both `bfuzz` and `Stack Smasher` were primarily developed for Bluetooth Classic implementations supporting only a few protocols like L2CAP and ATT. Therefore, they also require adjustments for fuzzing L2CAP and ATT layers in BLE implementations. Finally, `BLEFuzz` is the only tool that supports fuzzing BLE implementations. Table 5 summarizes the comparison between our fuzzer and the three chosen competitors.

For a fair comparison, we run our fuzzer and all the competitors for the same duration ($\approx$ three hours). As shown in Table 5, `WB55` and `BlueNRG-2` are the only two SoCs for which the competitors discover crashes (third column in Table 5). Specifically, `BLEFuzz` and `bfuzz` discovered only V8. For all other SoCs (cf. the "Others" column in Table 5), none of the competitors found either vulnerabilities or other anomalies. In a nutshell, our fuzzer significantly outperforms all competitors, as exemplified in Table 5. The reason is twofold. Firstly, our fuzzer comprehensively models the BLE stack, e.g., it includes modeling and fuzzing SMP and LL protocols, which are not handled by other fuzzers. Secondly, none of the competitors employ an optimization to refine mutation probabilities or send redundant packets. As shown by Figure 9, these features are critical for fuzzing effectiveness.

It is worthwhile to mention that a comparison with the aforementioned tools requires the usage of our non-compliant BLE controller (cf. Section 3.3). This approach is justifiable, *as currently there is no accessible BLE fuzzing alternative with the same level of control and flexibility as provided by our non-compliant BLE controller*. Finally, our comparison did not include traditional fuzzers such as AFL [44] due to their reliance in code coverage. Such a metric is often difficult to obtain in the context of over-the-air-fuzzing, as commercial BLE stacks are undisclosed. Furthermore, traditional fuzzers (e.g. AFL) lack the capability to generate a specific sequence of messages with strict timing constraints. To extend traditional fuzzers with such capabilities requires significant changes to the underlying fuzzing engine. Nevertheless, we

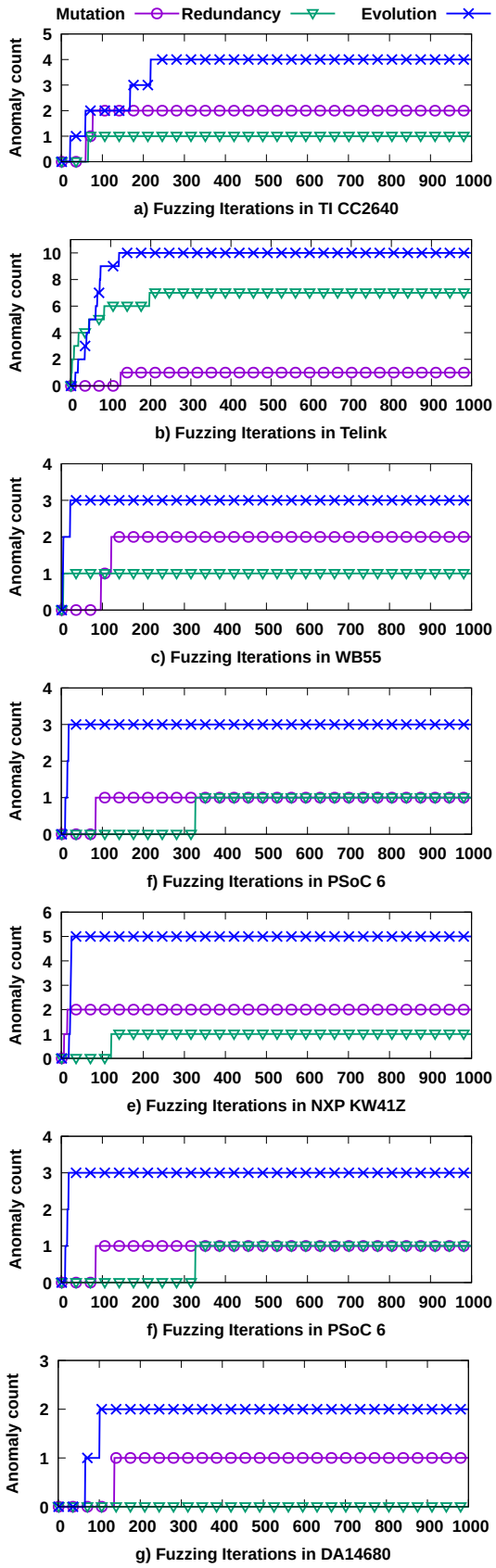


Figure 9: Fuzzing effectiveness w.r.t. design components

Table 6: Products verified to be vulnerable

Product	Category	BLE SoC	Vulnerability	Impact
Eve Energy August Smart Lock	Smart Home Smart Home	DA14680	V3	Crash
Fitbit Inspire	Wearables	CY8C68237	V1,V2	Crash
CubiTag	Gadget Tracking	CC2640R2	V5	Deadlock
eGeeTouch TSA Lock	Security	CC2540	V7	Deadlock

envision that even a loose adaptation of traditional fuzzers would yield results similar to Table 5, as anomalies other than crashes cannot be detected out of the box.

Case Studies on IoT Products: The exploitation of SweynTooth vulnerabilities, as summarized in Table 2, offers dangerous attack vectors against many IoT products. An investigation of certified products on the Bluetooth Listing site [40] reveals that SweynTooth is likely to affect ≈480 IoT products using the vulnerable SoCs from Table 3. These products are mainly applied in Smart Home, Fitness, Entertainment and Consumer Electronics. To raise awareness of the threats and risks of potentially vulnerable products available on the market, we performed attacks on some representative IoT products that use the affected SoCs and recorded our observations. Some salient features of these products are outlined in Table 6. In Table 6, we also indicate the BLE SoC used by each product and the vulnerabilities discovered in these SoCs by our fuzzer. We choose these products for their prevalence in the relevant application domains, e.g., Smart Home.

To exploit SweynTooth on an IoT product, we launch an attack code that captures the exact sequence of packet exchanges in the respective SweynTooth vulnerability. One such example is an attack code for vulnerability `V5` (found in `CC2640R2`) on `CubiTag`. Next, we describe, for each chosen IoT product, the impact of the launched attack code.

When attacking `Fitbit Inspire`, the smartwatch freezes its screen and immediately restarts when the *Link Layer Overflow* (V1) is attempted. By contrast, *LLID Deadlock* stops Fitbit advertisements for several seconds before the smartwatch abruptly restarts. Similarly, when *Silent Buffer Overflow* is exploited on both `Eve Energy` and `August Smart Lock`, users can immediately experience their smart things being restarted (e.g., via a beep sound in the smart lock and switching off the light attached to the `Eve Energy` plug). This is especially crucial for Eve System products, as the company relies almost entirely on the vulnerable DA14680. As for `CubiTag`, the attack exploiting *Public Key Crash* (V5) immediately stops the tracker to advertise and puts it in deadlock. Only a manual restart by opening CubiTag (e.g., via a screwdriver) and re-attaching its battery brings CubiTag back to a working state. Finally, when the *Invalid connection request* (V7) is exploited on `eGeeTouch TSA Lock`, it hangs and the user needs to manually press the *power on* button for further interaction.

5 Related Work

Security is critical for IoT devices [7]. Existing Bluetooth vulnerabilities, such as Blueborne [34], BleedingBit [15] and

KNOB [1], allow unauthorized remote access or remote code execution. They mostly require tedious manual effort (e.g., reverse engineering and inspecting code) and careful inspection of the protocol standard. By contrast, we provide a systematic and automated approach to discover BLE implementation flaws in any BLE device.

Existing works based on static analysis or verification technologies [14, 25, 27, 42] either suffer from false positives or are incapable to generate concrete packet sequences to trigger communication in real devices. An existing test generation approach targeting network protocol implementations [30] require access to the implementation code. Although a recent work `packetdrill` [5] provides a testing framework of the entire TCP/UDP/IP network stack, it lacks support for automated test packet generation. Similarly, Jero et al. [16] devised a technique to search a reduced state-space for suitable attack injection in stateful protocol implementations, but does not employ a comprehensive and directed approach for fuzzing packets. Furthermore, our validation strategy, being employed directly at the central, differs from passive wireless validation [35] that requires a sniffer. Finally, none of the aforementioned works set foot in Bluetooth.

Directed fuzzing is a prevalent software testing strategy [4, 17, 19, 21, 29, 43], yet faces significant challenges in the context of over-the-air fuzzing. Firstly, vulnerabilities in wireless protocol implementation often appear with a sequence of packets being injected even with strict time constraints. Traditional stateless fuzzers such as AFL [44] are mostly suitable for single input leading to vulnerabilities. Secondly, most of the commercial wireless protocol stacks are undisclosed. Thus, it is often not possible to have a greybox (e.g. based on code coverage) or whitebox approach (e.g. based on symbolic execution) for wireless security testing. Thirdly, wireless protocols often exhibit stochastic behaviour, packet drops and packet retransmissions due to the inherent nature of the wireless medium. This introduces additional complexity in security testing, especially in terms of distinguishing normal and abnormal behaviour. Fourthly, wireless protocol stacks often impose isolation between link layer and host layer protocols. A comprehensive security testing should break such isolation to find zero day vulnerabilities. Finally, detecting critical security issues in a wireless implementation, such as security bypass, requires significant changes to the underlying vulnerability detection logic of traditional fuzzers.

Emulation-based fuzzing [13] can obtain coverage information directly from the firmware and is faster than over-the-air fuzzing [26]. Nonetheless, such approaches require extensive reverse engineering of the firmware (if accessible at all) for a substantial number of wireless devices. For example, Frankenstein [20] is an emulation-based fuzzing approach that works with only specific Cypress/Broadcom firmware and demands significant engineering effort to handle other devices.

Previous works in Bluetooth fuzzing [3, 9, 18] support only L2CAP and ATT layers and do not employ test optimization for fuzzing effectiveness. InternalBlue [24] investigates the lower level of Bluetooth implementation and allows BLE packet sniffing and injection. However, InternalBlue can work only after the peripheral is connected and the number of accessible fields in a packet is limited. Our fuzzing framework, by contrast, allows packets injection, fuzzing and sniffing directly from the host and during the BLE connection process.

Our work is orthogonal to several works on network protocol testing [2, 12, 28] that target text structured protocols e.g. `ftp` and `http`, yet they ignore wireless protocols including BLE. A recent work [8] targets the discovery of memory corruptions in IoT devices by fuzzing the mobile app through which the device is accessible. Our work neither intends to fuzz the application layer nor relies on the availability of a mobile app. Moreover, by design of our validation component, our fuzzer can discover security vulnerabilities beyond memory corruptions e.g. security bypass.

In summary, our work is the first comprehensive approach to systematically and automatically fuzz arbitrary BLE protocol implementations. Also, this is accomplished without changing anything in the OS/firmware of tested device.

6 Conclusion

This paper presents a systematic and automated framework for fuzzing arbitrary BLE implementations. This is engineered with the aim to discover implementation behaviours that deviate from Bluetooth Core Specification. The efficacy of this framework is exemplified via the discovery of 11 new security vulnerabilities, named SWEYNTOOTH, across seven BLE SoCs. Moreover, we exploit several SWEYNTOOTH vulnerabilities on popular IoT products used as wearable, smart home products and logistic tracking, among others. This further shows the danger and criticality of SWEYNTOOTH vulnerabilities, potentially affecting a few hundred types of IoT products. Our fuzzer shares the limitation of any framework based on testing. This means, our fuzzer does not *guarantee* the security of a BLE device even if it fails to discover any anomalous behaviour.

SWEYNTOOTH highlights concrete flaws in the BLE stack certification process. We hope that our work provides an opportunity for further research in the area and initiates technologies to harden and secure current and next-generation wireless protocol implementations. For reproducibility and research, the fuzzer source code is available upon request to sweyntooth@gmail.com. All exploits are publicly available in the following URL:

https://github.com/Matheus-Garbelini/sweyntooth_bluetooth_low_energy_attacks

Acknowledgement: We thank the anonymous reviewers and our shepherd Kevin Butler for their insightful comments. This work is partially supported by Keysight Technologies grant no. RTKS171003.

References

[1] 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, Santa Clara, CA, August 2019. USENIX Association.

[2] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. SNOOZE: Toward a stateful network protocol fuzzer. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security*, pages 343–358, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] Pierre Betouin. Bluetooth stack smasher version 0.6. http://www.secuobs.com/news/05022006-bluetooth10.shtml, May 2006.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[5] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 213–218, San Jose, CA, 2013. USENIX.

[6] Damien Cauquil. Btlejuice: The Bluetooth smart MITM framework. DEFCON 24, 2016. https://github.com/DigitalSecurity/btlejuice.

[7] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated IoT safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 147–158, Boston, MA, July 2018. USENIX Association.

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA*, February 2018.

[9] Hou-Fu Cheng and Yu-Qing Zhang. Bluetooth OBEX vulnerability discovery technique based on fuzzing. *Computer Engineering*, 34(19):151–153, 2008.

[10] Pagmo development team. Pagmo & Pygmo. https://esa.github.io/pagmo2/, 2019.

[11] Gianluigi Me. Exploiting buffer overflows over Bluetooth: the BluePass tool. In *Second IFIP International Conference on Wireless and Optical Communications Networks, 2005. WOCN 2005.*, pages 66–70, March 2005.

[12] Serge Gorbunov and Arnold Rosenbloom. AutoFuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.

[13] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, Michael Grace, et al. PARTEMU: Enabling dynamic analysis of real-world trustzone software using emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[14] Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 627–638, June 2017.

[15] Armis Inc. Bleedingbit vulnerability. https://armis.com/bleedingbit/, 2018.

[16] Samuel Jero, Hyojeong Lee, and Cristina Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, June 2015.

[17] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening pandora's box through atfuzzer: dynamic analysis of at interface for android smartphones. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 529–543, 2019.

[18] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Poster: Iotcube: an automated analysis platform for finding security vulnerabilities. In *Symposium on Poster presented at Security and Privacy (SP)*. IEEE, 2017.

[19] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. CAB-Fuzz: Practical concolic testing techniques for COTS operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 689–701, Santa Clara, CA, July 2017. USENIX Association.

[20] Secure Mobile Networking Lab. Broadcom and Cypress firmware emulation for fuzzing and further full-stack debugging. https://github.com/seemoo-lab/frankenstein, 2020.

[21] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[22] LitePoint. Practical manufacturing testing of bluetooth® wireless devices. https://mcs-testequipment.com/resources/Datasheets_Downloads/Litepoint/Practical-Testing-of-Bluetooth-Devices_WhitePaper.pdf, 2012.

[23] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[24] 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*, MobiSys '19, pages 79–90, New York, NY, USA, 2019. ACM.

[25] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[26] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[27] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, page 12, USA, 2004. USENIX Association.

[28] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz, April 2017.

[29] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 543–553, New York, NY, USA, 2016. Association for Computing Machinery.

[30] JaeSeung Song ; Cristian Cadar ; Peter Pietzuch. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, July 2014.

[31] NCC Group Plc. BLESuite libirary. https://github.com/nccgroup/BLESuite, 2019.

[32] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, Jun 2007.

[33] Rohith Raj S, Rohith R, Minal Moharir, and Shobha G. SCAPY- a powerful interactive packet manipulation program. In *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, pages 1–5, Dec 2018.

[34] Ben Seri and Alon Livne. Exploiting blueborne in Linux-based IoT devices. https://go.armis.com/hubfs/ExploitingBlueBorneLinuxBasedIoTDevices.pdf, 2019. Armis, Inc.

[35] Jinghao Shi, Shuvendu K Lahiri, Ranveer Chandra, and Geoffrey Challen. Wireless protocol validation under uncertainty. *Formal methods in system design*, 53(1):33–53, 2018.

[36] Bluetooth SIG. Bluetooth Core Specification v4.0, June 2010. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[37] Bluetooth SIG. Bluetooth Core Specification v4.2, December 2014. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[38] Bluetooth SIG. Bluetooth Core Specification v5.0, December 2016. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[39] Bluetooth SIG. Bluetooth certification guideline: Qualify your product. https://www.bluetooth.com/develop-with-bluetooth/qualification-listing/, 2019.

[40] Bluetooth SIG. View previously qualified designs and declared products, January 2020. https://launchstudio.bluetooth.com/Listings/Search.

[41] Agilent Technologies. Bluetooth® manufacturing test: A guide to getting started. https://testunlimited.com/pdf/an/5988-5412EN.pdf, 2006. Application Note 1333-4.

[42] Octavian Udrea, Cristian Lumezanu, and Jeffrey S Foster. Rule-based static analysis of network protocol implementations. *Information and Computation*, 206(2-4):130–157, 2008.

[43] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 724–735. IEEE Press, 2019.

[44] Michal Zalewski. American fuzzy lop. https://github.com/google/AFL, April 2017.