BLEDiff: Scalable and Property-Agnostic Noncompliance Checking for BLE Implementations

Imtiaz Karim*, Abdullah Al Ishtiaq‡, Syed Rafiul Hussain‡, and Elisa Bertino*
*Purdue University, ‡Pennsylvania State University
*{karim7, bertino}@purdue.edu, ‡{abdullah.ishtiaq, hussain1}@psu.edu,

***Abstract*—In this work, we develop an automated, scalable, property-agnostic, and black-box protocol noncompliance checking framework called BLEDiff that can analyze and uncover noncompliant behavior in the Bluetooth Low Energy (BLE) protocol implementations. To overcome the enormous manual effort of extracting BLE protocol reference behavioral abstraction and security properties from a large and complex BLE specification, BLEDiff takes advantage of having access to multiple BLE devices and leverages the concept of differential testing to automatically identify deviant noncompliant behavior. In this regard, BLEDiff first automatically extracts the protocol FSM of a BLE implementation using the active automata learning approach. To improve the scalability of active automata learning for the large and complex BLE protocol, BLEDiff explores the idea of using a divide and conquer approach. BLEDiff essentially divides the BLE protocol into multiple sub-protocols, identifies their dependencies and extracts the FSM of each sub-protocol separately, and finally composes them to create the large protocol FSM. These FSMs are then pair-wise tested to automatically identify diverse deviations. We evaluate BLEDiff with 25 different commercial devices and demonstrate it can uncover 13 different deviant behaviors with 10 exploitable attacks.**

***Index Terms*—Bluetooth Low Energy, Noncompliance checking, Implementation Security**

I. Introduction

Bluetooth Low Energy (BLE) has been the most widely used low-energy communication protocol for the last several years. With the recent impact of COVID-19, BLE devices have seen an unprecedented surge, with 7 billion device shipments expected in 2026 [1]. As these BLE devices are ubiquitous and support numerous services such as audio streaming, data transfer, location service, medical equipment, and many more, it is essential that the BLE devices are compliant with the protocol specifications to meet the security and privacy requirements recommended by the standard. Recent works, however, have shown several noncompliance instances of BLE devices with critical security and privacy consequences [2], [3], including bypassing key establishment procedure (*aka.,* pairing procedure) and accepting messages encrypted with the default key. Noncompliance checking of BLE implementations is, nonetheless, challenging due to a large protocol standard [4] (3000+ pages) written in natural language with underspecifications, ambiguities, and in some cases conflicting specifications [3]. Since manual identification of noncompliance protocol behavior in large and complex BLE implementations is error-prone and time-consuming, in the paper, *we aim to develop the first automated and plug & play noncompliance checker for BLE devices.*

Prior efforts [2], [3], [5]–[11] on analyzing the security and noncompliance of the BLE protocol have identified several implementation flaws. Although they show great promise, they have at least one of the following limitations. The approaches: (i) are completely manual and are not scalable for analyzing a large protocol such as BLE [3], [5], [6]; (ii) only analyze the specifications either manually [7], [8], [12] or using formal verification [3], [9] with the manually extracted abstract protocol model and security properties; (iii) use fuzzing [2], [13] through a hand crafted bug oracle or reference state machine; (iv) use reverse-engineering [10], [11], requiring heavy domain expertise and tedious manual effort, which are not directly portable to devices of other vendors and models. To improve the unsatisfactory state of affairs, in this paper, *we set out to design an automated, scalable, property-agnostic, and black-box protocol noncompliance checking framework called* BLEDiff *that can analyze and uncover noncompliant behavior in the BLE protocol stack implementations.* Performing noncompliance checking in a black-box fashion makes BLEDiff agnostic to the device's underlying embedded operating systems, peripherals, and programming languages, and thus enables it to cover a diverse set of BLE devices with different input/output capabilities, many of which were not studied before.

Identifying noncompliant behavior in a property-agnostic way, however, warrants capturing and representing the reference protocol behavior $\mathcal{B}_{ref}$ in a formal language and comparing it with a given BLE protocol implementation $\mathcal{B}_i$. Capturing BLE protocol's *reference behavioral abstraction* from large and complex Bluetooth specifications, riddled with ambiguities and underspecifications, requires a juggernaut manual effort which is often error-prone as well as incomplete. BLEDiff capitalizes on having access to multiple BLE devices and leverages the concept of *differential behavior*, in which if two implementations produce two different output sequences for the same input sequence, at least one of the implementations is noncompliant with respect to the specification, even though it is not clear which one. BLEDiff, therefore, uses *differential behavior*, also called *deviant behavior* analysis, as a proxy for identifying noncompliant behavior in a property-agnostic way without requiring any reference protocol behavior abstractions. Therefore, the underlying noncompliance checking problem that BLEDiff addresses can be reduced to the problem of identifying deviant behavior among multiple BLE implementations and can be further stated as follows: Given black-box access to multiple BLE implementations $(\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n)$, is the implementation $\mathcal{B}_i$ equivalent to $\mathcal{B}_j$ $(i \neq j)$, failure of which approximates that at least one of $\mathcal{B}_i$ and $\mathcal{B}_j$ deviates from the specification?

In this paper, for our automated and black-box compliance checker BLEDiff, we use a Finite State Machine (FSM) as the input-output protocol abstraction and use the FSM to identify diverse noncompliant behavior. For automatically extracting the protocol FSM of BLE implementations, BLEDiff relies on an active FSM learning approach. In FSM learning, the learner starts from a known initial state, sends a sequence of over-the-air protocol messages (queries) to the device-under-test, and, based on the responses to the queries, infers the FSM of the underlying protocol implementation. Although prior work has used automata learning in the context of testing various protocols [14]–[22], in most cases, automata learning has been shown to be viable for only a specific layer [22], or for specific procedures [14], [15] or within a limited scope [23]. This is primarily due to the scalability issues even when the protocols are less complex and smaller than BLE. As a consequence, active FSM learning often fail to learn security-critical interactions and to complete FSM exploration. In addition, automata learning has been challenging and a never tried technique for automated FSM extaction of BLE-like human-in-the-loop protocols where human intervention (e.g., entering a passphrase aka., pass keys or checking numeric values at devices) is essential.

To address the scalability challenge of FSM inference using automata learning, BLEDiff explores the idea of using a *divide and conquer* approach. At its core, BLEDiff divides the BLE protocol into multiple sub-protocols, identifies their dependencies and initial states, extracts the FSM of each sub-protocol separately, and finally composes them. The critical insights of dividing and merging/composing are the following: (i) input messages for one sub-protocol (e.g., *LenReq* message in BLE link layer) in most of the cases do not induce any changes to the state machine of other sub-protocols (e.g., pairing and bonding of SMP); and (ii) completion of one sub-protocol enables the execution of another one. For instance, the Security Manager Protocol (SMP), responsible for pairing and bonding of BLE devices, can be executed only when the underlying link layer connection establishment procedure is completed. To side-step human intervention during protocol runs, BLEDiff uses keystroke simulation to tackle all the possible human-in-the-loop association methods, such as passkey entry, numeric comparison, and out-of-band.

Once the FSMs have been extracted, the second part of BLEDiff is to devise an approach to identify noncompliant behavior. To resolve this, BLEDiff designs a property- and reference FSM-agnostic differential analysis in which it identifies *deviant behavior* as a proxy for identifying noncompliant behavior. In the context of BLE, deviant behavior is a sequence of inputs for which the two FSMs under analysis generate distinct output sequences when executed from the initial state of the protocol. BLEDiff, therefore, reduces the problem of deviant behavior identification to a model-checking problem with a safety property. BLEDiff composes two FSMs under analysis and identifies deviant behavior-inducing input sequences (i.e., traces) by following the counterexamples, i.e., violations of the safety property: *Two FSMs will always generate same outputs for the same inputs*. The automatic identification of *diverse* deviant inducing traces between two FSMs is, however, challenging because existing model-checking tools uncover only the first counterexample/deviation and then stop exploration. To address this, we design a *model-refinement-based* deviant behavior identification scheme in which BLEDiff, among two FSMs under comparison during a pairwise differential analysis, refines an FSM based on the output of other FSM where two outputs initially mismatched and runs the model checker again. This time the model checker finds a newer deviation with a higher depth. We run the model checker until we run out of deviations and both the FSMs are the same. The closest to our work is the *elimination-based* equivalence checker designed for 4G LTE protocol [21]. This approach, however, eliminates deviation-inducing transitions to further explore other deviant behavior, causing the FSMs to become disjoint and thus failing to find higher depth deviations.

The deviant traces are then analyzed based on two root causes: implementations deviate from specification [4] or the specification is ambiguous. These deviations are potential vulnerabilities and are grouped into exploitable attacks or potential interoperability issues.

Findings. To test the effectiveness of BLEDiff, we evaluate it with 25 devices from 9 different vendors. BLEDiff found a total of 13 unique deviations in the devices. Among them, 10 are exploitable attacks, and two are potential interoperability issues between different devices. After root cause analysis, eleven of them have been confirmed as deviations from the standard and two as standard being unclear or ambiguous. Among the attacks, two cause security bypass, two crash, and others cause denial-of-service attacks.

Contributions. In summary, the current paper makes the following contributions:

- We propose BLEDiff– an automated, scalable, property- and reference FSM-agnostic noncompliance checking framework that analyzes and uncovers vulnerabilities in BLE implementations based on automata learning and identifying deviant behavior.
- To the best of our knowledge, we are the first to utilize the idea of *dividing and conquering* the state space to address the scalability of automata learning in FSM extraction.
- We design a BLE checking module that automatically identifies deviations at higher depths of an FSM compared to the state-of-the-art.
- We implement and evaluate BLEDiff with 25 different devices and demonstrate it can uncover 13 different deviant behaviors with 10 exploitable attacks including 2 security bypass, 2 crash, and 7 denial-of-service attacks.

Responsible disclosure. We have responsibly disclosed the findings of our work to all the affected vendors and Bluetooth SIG, and are actively cooperating with them for mitigation. The bugs have been acknowledged by Google, Nordic Semiconductors, Huawei, Microchip, Samsung, and STM electronics and 9 CVEs have been assigned so far. Other vendors are still reviewing the vulnerabilities. The responsible disclosure's status can be tracked here: https://blediff.github.io/.

Open-source. To help vendors and foster future research, BLEDiff is open-sourced at: https://github.com/BLEDiff.

II. Background

In this section, we provide an overview of the BLE protocol, define finite state machines, and discuss high-level details of active automata learning.

A. Bluetooth Low Energy (BLE)

Bluetooth is a well-established standard for short-range communication over public radio frequency channels across a diverse range of devices, including mobile phones, IoT devices, computers, headphones, smart watches, etc. Unlike Bluetooth Classic, Bluetooth Low Energy (BLE) is more focused on the energy constraints of low-cost IoT devices.

BLE Protocol Stack. The BLE protocol stack is divided into two parts. At the lowest level, the BLE controller consists of the Physical Layer (PHY), which deals with transmission and reception of over-the-air packets, modulation, antenna switching, etc., and Link Layer (LL), which maintains connections at a logical level and encryption. Above that, the host includes Logical Link Control and Adaptation Protocol (L2CAP), Attribute Protocol (ATT), Generic Attribute Protocol (GATT), and Security Manager Protocol (SMP).The SMP defines all security-related procedures, such as pairing, bonding, and authentication.

BLE Procedures. BLE communication works in a central-peripheral system, where the peripheral device broadcasts advertisement indications to announce its presence, and the central initiates the connection. After connection, a few link layer optional packets are exchanged between the two devices to negotiate several connection parameters. After that, the pairing procedure takes place by exchanging *PairReq*/*PairResp*. These packets include different I/O capabilities (keyboard, display, no input, no output, out-of-band data availability).

Based on these capabilities BLE has four types of association methods: *just works*, *numeric comparison*, *passkey entry*, and *Out-Of-Band (OOB)*. Just works association is appropriate when at least one of the devices does not have any display (output) or keyboard (input) and assumes the Temporary Key (TK) as 0 while pairing. In numeric comparison, each end is shown a six-digit number for comparison, and users are prompted to enter "yes" or "no." OOB association is possible when an out-of-band mechanism (e.g., NFC) can be used to discover or exchange cryptographic numbers for pairing. Finally, in the passkey entry association model, the user is shown a six-digit number on one device and is required to input the same number on the other.

Furthermore, two types of pairing may be supported–*legacy pairing* and *Secure Connections (SC)*. In secure connections, instead of a Short Term Key (STK) as the legacy pairing, a Long Term Key (LTK) is generated. From version 4.2, all the devices support both legacy and secure connections pairing. After the pairing procedure, the link layer encryption

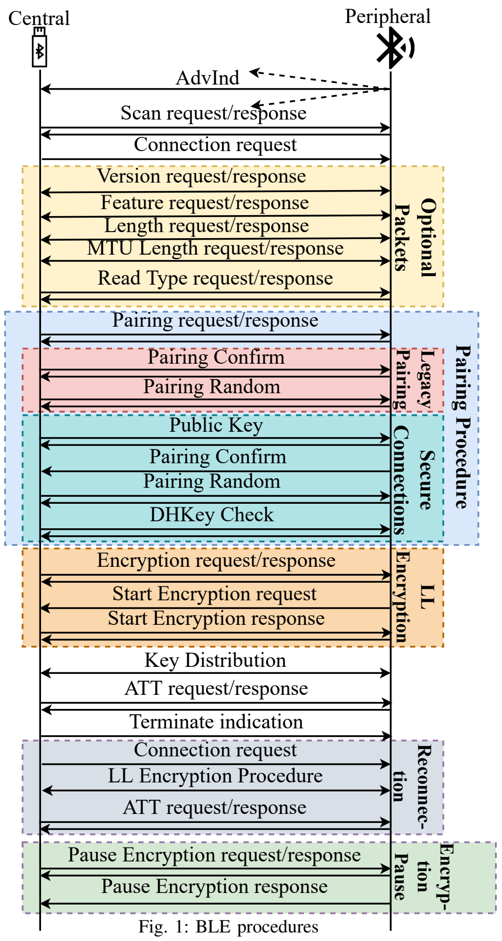

Fig. 1: BLE procedures

procedure is performed with a three-way handshake. At this point, the connection is encrypted, and the key distribution procedure takes place, which establishes encryption information, identification, address, and signing information. Other than these procedures, a BLE device terminates a connection using *TerminateInd* or when a device goes out of range. It can also reconnect with a paired and bonded device by sending *ConReq* and enable encryption by repeating the link layer encryption procedure. The details of these procedures are shown in Figure 1, where packet sequences and directions are available as well.

B. Finite State Machine (FSM)

For BLEDiff, we define a finite state machine $(\mathcal{M})$ as a 6-tuple $(\mathcal{S}, \mathcal{S}_0, \Psi, \Sigma, \Lambda, \Omega)$, where $\mathcal{S}$ is a finite set of states, $\mathcal{S}_0 \in \mathcal{S}$ is the initial state of the FSM. Σ and Λ are the sets of input and output alphabets, respectively, which represent the set of possible input and output messages. The transition relation $(\Psi : \mathcal{S} \times \Sigma \rightarrow \mathcal{S})$ maps the pair of the current state and an input symbol to the corresponding next state, and the output relationship $(\Omega : \mathcal{S} \times \Sigma \rightarrow \Lambda)$ maps the pair of a current state and an input symbol to the corresponding output symbol.

C. Active Automata Learning

Automata learning is the process of learning the behavior of a system from a set of execution traces. Automata learning techniques can be classified into two major classes–passive automata learning and active automata learning. In passive learning, the system is inferred from a set of given execution traces. On the other hand, active automata learning is an interactive technique where the learner generates queries and infers the behavior of the system from the outputs of those queries.

Active automata learning techniques are mostly built upon the L* algorithm [24]. These techniques [24], [25] learn the Deterministic Finite Automaton (DFA) for a given black-box system. Given the input alphabet, Σ (e.g., a, b) where a,b are input symbols), the algorithms generate sequences (e.g., a, aa, aba, abaa, ...), and probe the black-box system by resetting it between sequences. With a series of input sequences, a hypothesis FSM consistent with input-output pairs seen so far is built. This stage is called the *hypothesis construction* phase, and the queries generated in this phase are called membership queries. The learner iteratively refines the hypothesis FSM until it is complete (i.e., the set of probing sequences cover the state space of the hypothesis). After the hypothesis FSM is consistent and complete, the learner moves on to the *model validation* phase, where it queries an *equivalence oracle*, which checks whether the inferred FSM is identical to the black-box system and provides a counterexample if they are not. If the oracle reports that the hypothesis is identical to the black-box system, the algorithm terminates. Otherwise, the learner uses the counterexample to further refine the hypothesis. This process repeats until the oracle reports no counterexamples. In real scenarios, the existence of an oracle is often not feasible. However, the lack of a deterministic oracle can be approximated with a series of membership queries cleverly produced for this purpose [26].

III. Overview

In this section, we discuss the threat model, challenges, and a high-level description of BLEDiff.

A. Scope of Analysis

Our analysis covers the security-critical layers of the host and controller of the BLE protocol. Particularly, we study interactions in the Link Layer (LL), Security Manager Protocol (SMP), Logical Link Control, and Adaptation Protocol (L2CAP). These layers manage the most critical security procedures, such as pairing, bonding, encryption, encryption pause and authentication. Our approach BLEDiff also enables the analysis of all four association methods (just works, numeric comparison, passkey entry, out-of-band) and different pairing procedures (legacy and secure connections) associated with different I/O capabilities. Last but not the least, since security-related protocol behavior is identical in both BLE centrals and peripherals and BLE peripherals are more pervasive than BLE centrals [1], this work focuses on noncompliance checking for BLE peripheral implementations only.

B. Threat Model

We consider the communication channels between the central and the peripheral subjected to adversarial influence. Our attacker model follows the one defined by previous works [2], [3], [7], [8] and comprises either a passive or an active attacker that differs in capabilities and restrictions. The passive attacker can observe arbitrary communication between the central and the peripheral. The active attacker acts as a central and can additionally intercept, replay, modify, drop, or delay messages, without knowing the key material of devices not owned by the attacker. Also, the adversary cannot replace the firmware of the peripheral. Since BLE is a short-range communication protocol, we assume that the distance between the adversary and the peripheral is within the BLE range.

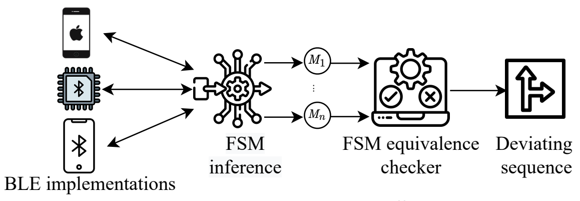

Fig. 2: Modules of BLEDiff

C. Problem and Solution Outline

For a black-box BLE protocol implementation $\mathcal{B}$ of a BLE-enabled smartphone, development board, IoT device, BLEDiff aims to find input sequences $\Delta_i = \sigma_1\sigma_2\sigma_3 \ldots \sigma_j \ldots \sigma_m$ where $\sigma_j \in \Sigma$ for which the corresponding output sequences does not follow the one provided by the standards. The first challenge for solving this problem for BLE devices is to automatically extract the behavioral abstraction (e.g., FSM) of a protocol implementation. Since the reference FSM of BLE protocol is not present yet and is hard to manually construct, the second challenge is to devise an approach for identifying diverse noncompliant behavior in the extracted implementation $\mathcal{B}$ without having access to the reference FSM.

To address the first challenge, BLEDiff extracts an approximate FSM ($\mathcal{M}_j$) for each BLE implementation $\mathcal{B}_j$ using active automata learning approach. To resolve the second challenge, BLEDiff leverages the access to multiple BLE implementations, and for each pair of extracted FSM $\mathcal{M}_j$ and $\mathcal{M}_k$, find input sequences of the form Δ_i such that for Δ_i, both $\mathcal{M}_j$ and $\mathcal{M}_k$ generate different output sequences. The output of BLEDiff is Δ_i which induces the deviant behavior.

D. Challenges of Designing BLEDiff

BLEDiff, as shown in Figure 2, works with two main modules: the BLE Learning module and the BLE checking module. The challenges of BLEDiff, therefore, can be grouped into two broad categories: (i) learning the FSM of a diverse set of BLE protocol implementations; (ii) identifying noncompliance from the learned FSMs.

1) Learning the BLE FSM of an implementation

For learning the FSM of a BLE implementation, we use an active automata learning approach. However, effectively applying active automata learning for BLE protocol implementations requires solving some non-trivial challenges. In the following, we discuss these challenges and the insights on addressing the challenges.

(C1) ***Scalability.*** Automata learning typically runs into severe scalability issues, particularly when the input/output alphabet size is large. Although this is not new, automata learning with Over-The-Air (OTA) queries and responses makes the scalability issue worse due to the highly unreliable nature of the wireless communication medium. This actually warrants running the same query multiple times to meet sufficient confidence in learning, and thus takes several days or months to extract an FSM. In case of BLE, this problem is critical due to the following reasons. First, the BLE protocol has different security procedures (secure connections, legacy pairing) based on the device's capabilities. It is essential to explore all the security procedures as it has already been shown that secure or legacy pairings can affect each other and cause severe security issues [2], [12]. Second, the BLE security procedures are distributed over multiple layers. For instance, pairing and bonding are part of the Security Manager Protocol (SMP), whereas encryption and encryption pause procedures are part of the Link Layer (LL) protocol. Exploring all critical security procedures necessitates the scope of BLEDiff to be tremendously large compared to previously tested protocols. Although previous works [14], [27] have adopted techniques such as caching and adding constraints, however, these are not enough to handle the scalability of BLE automata learning.

Insights on addressing C1: For addressing this important problem related to scalability, instead of extracting one FSM, BLEDiff utilizes the idea of a divide-and-conquer approach. In the case of BLE, applying a divide-and-conquer approach for FSM learning ensues the challenge of dividing the protocol in such a way that, later on, they can be merged together systematically. We divide the implementation space into three distinct sub-protocols: LL, SMP, and reconnection procedures. BLEDiff learns the FSM for each of them separately. The critical insight behind this divide is that one sub-protocol does not induce any change to the state machine of the other sub-protocols i.e., the FSMs do not react instantaneously. Now the next critical task is to merge the inferred FSMs together. To solve this challenge, an idea can be to perform a cross-product-based cascade composition [28]. However, that will result in a large FSM, which is not necessarily optimized. As the completion of a sub-protocol enables the execution of the next sub-protocol, we can detect the states where the different layer procedures are completed (e.g., LL procedures complete and SMP procedures start when the device responds with a *PairResp*). Coupled with this insight, we can do a sequential merging for the FSMs, which entails a minimal but complete FSM of the large protocol implementation.

(C2) ***Intertwined BLE protocol is not suitable for designing a mapper.*** Another major challenge for applying active automata learning in the context of BLE protocol state machines involves developing BLE specific mapper. The mapper facilitates communication between the learner and the BLE device. It needs to convert the abstract input symbols in the membership queries to concrete OTA packets and send them to the BLE device. In the same vein, it also needs to decode the response from the BLE device and convert it back to an abstract output symbol comprehensible to the learner. Developing such a BLE-specific mapper is challenging because protocol layers are intertwined and have strong temporal correlations among their operations. In our case, following a divide-and-conquer approach, we need to develop three separate mappers that can operate independently from the logic of the other protocol layers.

Insights on Addressing C2: We have developed three BLE-specific mappers that can set the protocol to the required state, and transparently send and receive messages based on the direction of the learner. The mapper can handle complex multi-level, stateful interactions of the BLE protocol.

(C3) ***Standard-compliant non-determinism in link layer procedures.*** Unlike BLE's other layers' procedures that are only triggered by a central, the procedures for central-peripheral connection setup at the link layer, such as feature, and version requests, can be triggered by both central and peripheral without following any strict ordering as specified by the standard. As a result, the order of the origination of such procedures at the link layer is implementation dependent. If usual model learning is applied here, due to this protocol design, this will create spurious deviations while comparing two implementations even though none of them actually deviate from the specification.

Insights on addressing C3: We design our LL mapper differently from all the previous works [14], [15], [21]. The high-level idea is to abstract the peripheral-triggered request messages from the learner (shown in Figure 4). Concretely, whenever for an input request, the mapper receives a peripheral-generated link layer request as output, the mapper takes the following steps: (i) it sends a response to the peripheral internally without notifying the learner about the output; (ii) waits for the response of previously send request; (iii) whenever it receives the response for the previous input request, the response is passed to the learner. Thus the mapper completely abstracts the peripheral-triggered LL requests and let the learner learn a consistent FSM of the peripheral.

(C4) ***BLE random addressing and human interaction affect automation.*** For automata learning, the learner needs to run a significant number of OTA messages to the SUL. Each time a query is run, the device needs to be reset. But resetting a device also changes the MAC address of a device, if a random address is used by a device to protect its privacy. On the other hand, depending on the association model used (e.g., passkey entry or numeric value comparison), human interaction may be required in the pairing procedure. These become a challenge for building a fully automated FSM learning system.

Insights on addressing C4: We design a fully automated procedure to identify the changing random MAC address of the device. To learn the address, we design a *probing and set-subtraction* method where the learner first probes for a time period T_1 to get a set of available BLE devices A. The learner then turns on the device under test and probes for another time period T_2 (T_1 and T_2 are non-overlapping) to get a set of available BLE devices B. The learner obtains the address of the target device by computing a set subtraction

$B - A$. For addressing human interactions required during pairing procedures, the learner simulates taps or keyboard inputs when prompted.

2) Identifying noncompliance from FSMs

Once we have extracted the protocol state machines of the BLE implementations under test, we need to find non-compliance instances. This would have been simpler if a reference FSM of the protocol was available. However, in the case of BLE, this poses a challenge as there is no reference FSM available from the specifications [4]. To resolve this, we use the idea of pairwise differential testing of protocol state machines extracted from different implementations to identify deviant behavior inducing input sequences. We use these input sequences as a proxy for identifying noncompliant behavior. Another major challenge for FSM comparison is how to automatically identify not only one but many diverse deviant behavior inducing input sequences. Existing equivalence checking approaches [29] are insufficient for our purpose as they neither have the notion of diversity nor the capability to provide multiple deviant behavior inducing input sequences.

Insight on addressing the challenge. To resolve this challenge, we reduce the problem of equivalence checking to a model-checking problem of a safety property. We pose a series of model-checking queries, one for each pair of distinct output symbols. However, checking the safety property in a model usually returns the same deviant trace, which in most cases is the shortest one. To find diverse deviations, we need to define a way to modify the FSM so that we can get different deviations. For this, a recent work DIKEUE [21] proposes the idea of *elimination-based* model modification, where the transition that causes the deviation is eliminated from the model. This has a critical limitation as eliminating the transitions makes the FSMs disconnected and hinders the exploration of deviant behavior inducing input sequences deep into the FSMs, which is highly desirable in finding noncompliance in protocol implementations. To alleviate this, we adopt a *refinement* based model modification, where instead of removing the transition, we refine the transition in one of the FSMs under consideration by changing that transition's output to that of the other FSM so that two FSMs become equivalent up to that transition. Thus the same deviation will not be generated by the model checker if run again. This refinement is carried out until there are no more deviations left and both the FSMs are equivalent based on the posed safety property.

IV. Detailed Design of BLEDiff

A. Divide and Conquer Based FSM Learning

Due to the BLE protocol consisting of multiple layers and numerous procedures, it is extremely challenging to infer the whole FSM of the BLE implementation. Essentially, if all input/output symbols of both layers are used at once, it runs into state space explosion and takes an unreasonable time to infer the FSM. To resolve this, BLEDiff takes a divide-and-conquer approach to infer FSMs separately. In the divide phase, the protocol is split into three parts, and FSM for each part is inferred. In the conquering phase, the FSMs are merged together to create the large FSM of the protocol implementation.

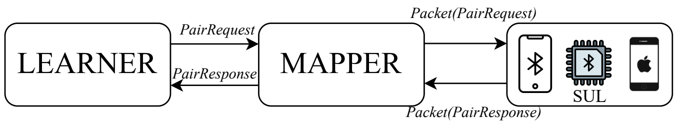

Fig. 3: Mapper for BLE Learning Module

1) Divide Phase

In this phase, following our insight of creating non-instantaneously reacting FSMs, we divide the protocol into three separate parts (i) Link Layer Control Protocol; (ii) Security Manager Protocol (SMP); (iii) BLE reconnection, and learn FSMs for them separately.

Alphabet set selection. The first decision for model learning is to select the initial alphabet set, i.e., the set of input and output symbols. The number of input symbols relies on the kinds of considered protocol messages. Once the input symbols are selected, then the output symbols are obtained from the protocol specification. In order to reason about security-critical behavior, we include several predicates of an input symbol. More elaborately, we employ (i) *field-level* predicates of an input/output message by applying different operations, including changing the value of a field either to zero or to the max, and (ii) *packet-level* predicates, e.g., changing an encrypted packet to plaintext. We apply packet-level predicates to all possible encrypted packets and field-level predicates to only security-sensitive fields, e.g., public keys, confirmation, interval, and timeout values. Note that each predicate applied to a symbol introduces a new symbol. Such a packet- and field-level predicate mechanism allows us to minimize the total number of input/output symbols. The list of all input/output symbols for all three parts of the protocol is shown in Table VII in the Appendix.

Termination. Termination is a critical issue for model learning. The termination strategy should provide a balance between termination and coverage. As we are employing a divide-and-conquer approach, we have to make sure each FSM reaches the connected state before moving on to the next FSM. For example, the FSM of the SMP procedure starts after the LL's FSM completes the link layer connection. We can detect the states where the link layer control procedure is completed based on the output symbols. The link layer procedure is completed when a *PairReq* message is responded with a *PairResp*. Similarly, the SMP procedures are completed with a *DHKeyCheckSend* responded with *DHKeyCheckRecv*, and finally, the reconnection is completed when the encryption starts, i.e., the *StartEncResp* from central is responded with a *StartEncResp* from the peripheral.We employ this domain knowledge, and as soon as the respective FSM gets these output symbols and completes the layers connection, we terminate the learning for that FSM. We utilize this termination strategy for merging the FSMs in the conquering phase of FSM learning (discussed in IV-A2).

Separate mappers for each part. One of the crucial components of BLE Learning module is the design of the mapper.

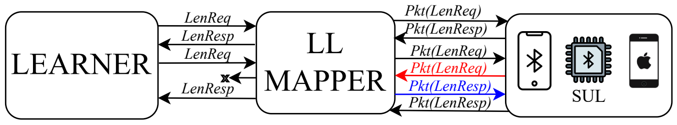

Fig. 4: Link Layer protocol mapper

The mapper acts as a glue between the SUL and the learner (shown in Figure 3) and builds a reliable interface from the learner to each protocol layer. For each input symbol from the learner, the mapper waits a pre-defined time for an output symbol to be received from the SUL. In case of a timeout, a pre-defined *Null* symbol is returned to the learner by the mapper. In our case, we design three mappers for each part of the protocol. This is necessary as each mapper has separate initial states and some unique challenges, which we discuss in detail below.

① *Link Layer (LL) Mapper.* For LL inference, the initial state is set to the beginning of the link layer procedures. The LL protocol messages are selected as input symbols, and the corresponding responses as output symbols. However, as discussed earlier, there can be inconsistencies in the inferred LL FSM due to the protocol design. This is due to the fact that the same procedure can be triggered by both the peripheral and the central. This can create spurious deviations, i.e., false positives. To resolve this, the mapper abstracts out SUL-originated LL messages and only pass on the response messages to the learner. For instance, as shown in Figure 4, in case the mapper receives an SUL-generated message *LenReq* (red colored), it automatically responds with a *LenResp*. However, this *LenReq* is not passed on to the learner. The mapper needs to respond to this *LenReq* internally because, in some implementations, if the response is not received, the device does not respond to future messages. Through this design, the mapper facilitates the learner to learn a consistent FSM for all the devices and removes the possibility of false positives, i.e., a deviation that is neither an implementation issue, nor an issue with the protocol standards.

② *SMP Layer Mapper.* As the name suggests, the security manager protocol is the most important layer for BLE implementation with respect to security. For the SMP layer inference, the initial state of the learner is set to the beginning of the SMP. Three critical security operations, pairing, bonding, and authentication, are covered here. To cover all possible association, pairing and authentication modes, we include all possible I/O capabilities (no input no output, display yes/no, keyboard). To automate the learning process, we fix the input to all zeroes in cases where an input is required from the central. Similarly, on the peripheral side, we automate the process to input the required values to complete all the SMP procedures. More on this is discussed in the following subsection on handling reset, human interaction, and BLE random addressing.

③ *Reconnection Mapper.* For this scenario, we move the initial state to the *reconnection* state, i.e., both devices have already paired and bonded and they try to reconnect with each other. To simulate the reconnection scenario, the mapper first pairs and bonds with the peripheral and then intentionally drops the connection to create the scenario of reconnection. Reconnection is critical to test device authentication, encryption and encryption pause procedures. To achieve this, we design our mapper to complete both link layer and SMP procedures and go through pairing and bonding. After bonding, the connection is forcefully disconnected and connected again to test the reconnection procedures.

Applying existing optimizations. Apart from this, we also include the previous approach to improve scalability. One of the known and most popular approaches to improving the scalability of model learning is query caching. In the model validation stage, the learner can generate the same query, which has already been resolved in the hypothesis construction phase. To avoid expensive OTA testing of these duplicate queries in the SUL, the queries from the hypothesis construction phase are cached in the database [23], [27]. In the model validation stage, if the same query is found in the cache, the query is not run OTA again, cutting down the overhead and time for repeated queries. Another approach is to minimize the time-consuming OTA transmissions by adding multiple constraints as invariants [14], [21]. For this, the mapper is provisioned with a set of invariants. In case the invariants are violated, the query is not sent OTA, and a pre-designated symbol is returned. For BLEDiff we use invariants such as: ❶ A connection has to be established before sending any other symbol; ❷ After disconnection and before establishing a connection, all the symbols will be ignored; ❸ No security protected messages will be sent without establishing the necessary keys. A prerequisite of model learning is for the SUL to be deterministic, which is not always possible due to OTA communication. To maintain such consistency, we leverage existing insight from prior works [23], [30] and run the same query twice. In case the output for both the queries are different, the query is run once more, and a majority voting scheme is applied to the results to store the correct response.

Modified BLE stack. We modify the open-sourced BLE stack provided by SwyenTooth [2] to develop the components of a central BLE device. We remove the original FSM implementation used for the SwyenTooth fuzzer and create direct interfaces to convert packets to and from the learner. We introduce the LL Encryption Pause procedure, which was missing from the open-source implementation. Furthermore, SwyenTooth's implementation is not able to communicate with devices that have Asynchronous Connection-Less (ACL) fragmentation. This is a critical limitation for SwyenTooth to work with smartphones having mandatory ACL fragmentation. To resolve this, we implement ACL fragmentation to be able to analyze all the possible devices.

Handling reset, human interaction, and BLE random addressing. For model learning, the device should be transparently reset to the known initial state. In our case, it means setting to the corresponding initial states for the corresponding mapper. Furthermore, as we are handling all the possible I/O capabilities, we are required to automate some of the user inputs. For instance, when both the devices' I/O capability is keyboard display, then in the case of LE legacy pairing, the devices use the passkey entry association method. Here, the central sends a passkey, and the peripheral needs to input this passkey. In our case, we automate this process by setting the passkey to all zero (0x000000) throughout the learning process. These automation schemes require significant engineering efforts. To achieve this, for development boards, we reset the board using software reset and set the associate passkey through UI automation, for Android smartphones, we use ADB and key press simulation, and for iPhones, we use IOS13-SimulateTouch [31] to simulate touch events. After the reset is complete, we bring the corresponding mapper to the respective initial state for learning. For instance, for the SMP learning, we complete all the link layer connections. For re-connection, we complete the pairing and bonding procedures. One of the critical challenges for most BLE devices is that after a certain threshold time, and in case of smartphones, after each time BLE is turned on/off, the BLE address is changed. This is challenging as we need to create a fully automatic system. To resolve this, before running each query, we identify the new BLE address by following our *probing and set subtraction* scheme discussed in Section III. Furthermore, in case of smartphones, different prompts pop up during the pairing procedure for different I/O capabilities, and we automatically handle them using key press simulation.

2) Conquer Phase

The task of conquer phase is to merge the three separate FSMs of the implementation to create the large protocol FSM which allows the equivalence checker to find an end-to-end trace of deviant behavior, i.e., from entry-point of BLE protocol to where deviation occurs. Such a trace can be readily converted to a concrete test case for further testing. A straightforward way to merge FSMs would be performing a cross-product-based cascade composition. But this would create an unnecessarily large FSM. As the task here is to create a merged FSM that maintains the scalability of the divided FSMs, we design a sequential merging for the FSMs, which entails a minimal FSM of the large BLE implementation.

Sequential FSM merging. As the inferred FSMs do not react instantaneously and coupled with our choice of termination (discussed in IV-A1), we can detect the corresponding terminating states in the FSMs and therefore, merge the FSMs sequentially. The terminating states can be detected based on the output symbols. For example, the link layer procedure is completed when *PairReq* responds with a *PairResp*. The SMP procedures are completed when *DHKeyCheckSend* responds with *DHKeyCheckRecv*. Upon detecting the terminating states, we connect the terminating state of the first FSM to the initial state of the second FSM (which we automatically get in the FSM). Here we formally define the conquered FSM with the separate component FSMs.

Definition 1 (Merging of BLE FSMs):

Let us assume the states where LL procedures are completed as $\mathcal{S}_{LL_{Comp}}$, and SMP procedures are completed as $\mathcal{S}_{LL_{SMP}}$. Let us also assume $\mathcal{M} = (\mathcal{S}, \mathcal{S}_0, \Psi, \Sigma, \Lambda, \Omega)$ as the merged FSM and $\mathcal{M}_{LL} = (\mathcal{S}_{LL}, \mathcal{S}_{0_{LL}}, \Psi_{LL}, \Sigma_{LL}, \Lambda_{LL}, \Omega_{LL})$, $\mathcal{M}_{SMP} = (\mathcal{S}_{SMP}, \mathcal{S}_{0_{SMP}}, \Psi_{SMP}, \Sigma_{SMP}, \Lambda_{SMP}, \Omega_{SMP})$, $\mathcal{M}_{Re} = (\mathcal{S}_{Re}, \mathcal{S}_{0_{Re}}, \Psi_{Re}, \Sigma_{Re}, \Lambda_{Re}, \Omega_{Re})$, are the LL layer, SMP and

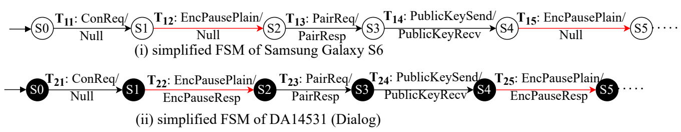

Fig. 5: BLE checking module

reconnection FSMs, respectively. Following our discussion of state merging, we define $\mathcal{M}$ as: $\mathcal{S} = \mathcal{S}_{LL} \cup \mathcal{S}_{SMP} \cup \mathcal{S}_{Re}$, $\mathcal{S}_0 = \mathcal{S}_{0_{LL}}$, $\Sigma = \Sigma_{LL} \cup \Sigma_{SMP} \cup \Sigma_{Re}$, $\Lambda = \Lambda_{LL} \cup \Lambda_{SMP} \cup \Lambda_{Re}$, $\Psi = \Psi_{LL} \cup \Psi_{SMP} \cup \Psi_{Re} \cup (\mathcal{S}_{LL_{Comp}} \times \epsilon \rightarrow \mathcal{S}_{0_{SMP}}) \cup (\mathcal{S}_{SMP_{Comp}} \times \epsilon \rightarrow \mathcal{S}_{0_{Re}})$, $\Omega = \Omega_{LL} \cup \Omega_{SMP} \cup \Psi_{Re} \cup (\mathcal{S}_{LL_{Comp}} \times \epsilon \rightarrow \mathcal{S}_{0_{SMP}}) \cup (\mathcal{S}_{SMP_{Comp}} \times \epsilon \rightarrow \mathcal{S}_{0_{Re}})$

B. BLE Checking Module

For BLE checking module, we reduce the problem to a model checking problem with a safety property.

1) Reduction to Model Checking

Suppose the two FSMs under differential test are denoted by M_1 and M_2. The input messages to these two FSMs are denoted by I_1 and I_2 and output messages as O_1 and O_2, respectively. Using M_1 and M_2 we then construct a model M, where M_1 and M_2 are sub-components. M will take a single symbolic input I, which will be fed to both I_1 and I_2, in other words, the same input is fed to both M_1 and M_2. M will have two outputs O_1 and O_2, essentially the outputs of M_1 and M_2, respectively. The model M can be viewed as a parallel composition of both M_1 and M_2. Then for each pair of different output symbols, we pose a query that *Are there any same input sequence which generates this different output?* The model checker returns the sequence if there are any such input sequences. This will be the deviating input for which the same input sequence generates different output sequences. As we are aiming to find as many deviating traces as possible, we run the model checker again. However, in most of the cases, the model checker will return the same input trace. To resolve this, we need to modify our model.

Problem with elimination-based model modification. In previous work [21], the authors use the idea of a *elimination-based* model modification by removing the deviation transition from the models. Though promising at first glance, this idea raises some issues. To illustrate, let us look into the two FSMs of Figure 5. With the model checking of a safety property, the two different outputs of the same input symbol *PauseEncReqPlain-Text* will be identified as (*PauseEncResp*, *Null*). To answer what is the input deviating sequence is, there is a high probability the model checker will return $S0 \rightarrow S1 \rightarrow S2$ and the deviating transition is T_{12} and T_{22}. Now, if we follow elimination-based model modification, then both the transitions will be removed, and model checking query will be run again. Due to the transition removal, part of the FSM becomes unreachable, and the model checker returns no more deviating traces, which is not true, as evident in Figure 5. The other and more interesting deviation is $S0 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S4 \rightarrow S5$, which would be left undetected by the previous elimination-based approach.

Refinement based model modification. BLEDiff takes a different approach, by instead of eliminating the transition, it refines the transition in one of the FSMs under consideration by changing that transition's output to that of the other FSM so that two FSMs become equivalent up to that transition. Thus the same deviation will not be generated by the model checker if run again. Continuing with our example of Figure 5, we modify the output of T_{22} as *Null* and run the model checker safety property again. As the FSMs are identical up to this point, it generates a more in-depth deviation between T_{15} and T_{25}. One thing to be noted here, our transition refinement does not affect the soundness of BLEDiff. Our goal is to find the same input traces that produces different outputs, and a deviating trace can deviate in multiple positions.

V. Implementation

The BLE Learning module is implemented on top of LearnLib [32]. For the learning algorithm, we use TTT [25] as it requires fewer queries compared to other algorithms [33], and for conformance testing, we use Wp-method [26]. We specify TTT as the learning algorithm and Wp-method as the validation approach in Learnlib. Learnlib is an abstract state learning implementation that requires a custom interface to the SUL. LearnLib sends abstract message sequences as queries. These are translated to BLE messages by the mapper. Similarly, the responses from the SUL are translated back to an abstract form by the mapper and forwarded to LearnLib. We implement our mappers in Java. We modify the implementation developed by SwyenTooth [2] as part of their fuzzer to implement our modified central implementation. We replace the LL, SMP, and ATT implementations of the SwyenTooth stack with our modified stack and create interfaces between the stack and the mapper to forward LL, SMP, and ATT packets in both directions. We also introduce additional code to handle reconnections and ACL fragmentation of BLE devices. We use nRF52840 Dongle [34] to send/receive raw link layer packets to and from the peripheral OTA. The FSM merger is implemented in Python, which identifies final states from dot representations of the inferred FSMs and merges them accordingly to create the large FSM of the implementation. The BLE checking module is developed using the NuXmv model checker [35] and a python 2.7 script as the wrapper. LearnLib outputs the FSMs as dot files. We transpile dot FSMs to the SMV specification language. Table III summarizes our efforts in modifying the tools and creating new components for BLEDiff.

VI. Evaluation

To evaluate the performance of BLEDiff, we aim to answer the following research questions: **RQ1.** How effective is BLEDiff in finding deviant behaviors in different BLE implementations? **RQ2.** How does BLEDiff perform compared to existing baseline testing approaches, i.e., BLE conformance testing suites [36] and previous works on BLE testing? **RQ3.** What is the effectiveness and performance of BLEDiff components: BLE Learning module and BLE checking module?

The experimental setup and devices their vendors and BLE versions are described in section A and Table XI in the Appendix respectively.

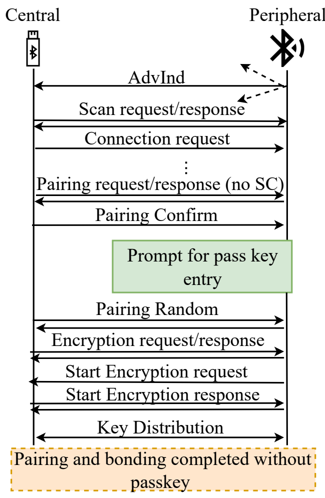
Central
Peripheral
AdvInd
Scan request/response
Connection request
⋮
Pairing request/response (no SC)
Pairing Confirm
Prompt for pass key entry
Pairing Random
Encryption request/response
Start Encryption request
Start Encryption response
Key Distribution
Pairing and bonding completed without passkey

Fig. 6: Passkey entry bypass

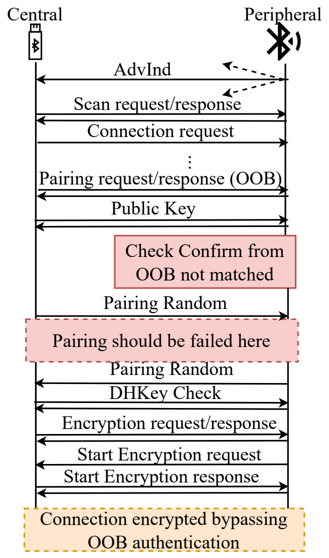
Central
Peripheral
AdvInd
Scan request/response
Connection request
⋮
Pairing request/response (OOB)
Public Key
Check Confirm from OOB not matched
Pairing Random
Pairing should be failed here
Pairing Random
DHKey Check
Encryption request/response
Start Encryption request
Start Encryption response
Connection encrypted bypassing OOB authentication

Fig. 7: Out-of-Band pairing bypass

A. RQ1. Deviations, Attacks, Impacts

BLEDiff identifies deviations between different BLE implementations. However, we observe that multiple deviations have the same root cause. We define *unique deviant behaviors* as the ones having unique root causes. For example, for two deviations $D1$ and $D2$ with two root causes $R1$ and $R2$, if $R1 \neq R2$, we consider $D1$ and $D2$ as unique deviations. Otherwise, the deviations are not considered unique. We manually identify root cause of the deviations by consulting with the 3GPP specifications. Based on the root causes, we identify unique deviant behaviors from all the deviant behaviors. The root cause can be boiled down to one of the two reasons: either the implementation deviating from the standards or the standard has ambiguities due to underspecification. It took around 2 days of human effort to identify all unique deviations from all the deviant behaviors found in 25 devices. In total, BLEDiff has identified 13 unique deviations in the 25 BLE implementations tested. Among them, 10 are exploitable attacks, 2 are potential interoperability issues, and for 1 the impact is still not evident. We define interoperability issues as deviations that can hinder the communication between two devices and cause re-pairing. Upon root cause analysis, 11 deviations were found due to the implementations deviating from the standards, and 2 were due to underspecification in the standards. The identified issues, their impacts, and the root causes are shown in Table I. We characterize the impacts into three types: security bypass, crash, and Denial-of-Service (DoS). We categorize crash as a separate class because the issues that cause the device to crash and become unresponsive require manual intervention to recover and can be seen as an enhanced form of DoS. The attacks to device mapping are shown in Table IX in the Appendix.

1) Attacks

(E1) Passkey Entry Bypass. Among the four association methods, passkey entry is considered secure against Man-in-the-Middle (MitM) attacks. In this method, the initiating device displays a randomly generated value, which the responding device has to enter. Particularly, after the central sends a *PairConfirmSend* message, a prompt is shown on the peripheral device for passkey entry. In LE legacy pairing, the peripheral device shall send a *PairRandomSend* only if the *confirm* value ($\mathsf{C_{cmp}}$) computed on the device matches the *confirm* value ($\mathsf{C_{rcv}}$) received from the central device , i.e., when $\mathsf{C_{cmp}} = \mathsf{C_{rcv}}$. If $\mathsf{C_{cmp}} \neq \mathsf{C_{rcv}}$, then the responding device would terminate the pairing. BLEDiff, however, has uncovered 13 implementations where the device completes pairing and bonding without requiring to enter the passkey in the device and thereby effectively nullifying all the security protections against MitM attacks. In this deviation as illustrated in Figure 6, if the central sends a *PairRandomSend*, setting the value of the user input passkey to zero, the deviating BLE peripheral implementation responds with a *PairRandomSend*, without sending a *PairConfirmSend* message and even before taking the input from the user (deviating from the standards). The connection persists even after the user inputs the passkey after an attack with the deviation is performed. Furthermore, the peripheral implementation completes the pairing and bonding process and enables encryption, all assuming the user input to be zero. Surprisingly, one of the devices (Pixel 4a) does not even show the prompt for passkey entry, thus effectively bypassing the MitM protection put into place through the passkey entry association method.

Root cause. The root cause of this issue can be attributed to implementation deviating from the specification. The BLE specification clearly states that if the confirm values do not match, the peripheral should not proceed with pairing [4, p. 1628]. *Impact.* Due to this passkey entry bypass, it is possible for the attacker to perform a MitM attack on the vulnerable BLE devices. As the key value TK is always set to zero for the vulnerable peripheral, the attacker can impersonate both legitimate central and peripheral devices. In a hindsight, this is actually worse than just works association method as the user thinks they are using a high level of protection, but actually, they are not.

(E2) Out-Of-Band Authentication Bypass. During pairing, an out-of-band (OOB) channel, e.g., NFC may be used to communicate information between central and peripheral, which is further used later in the pairing process. The OOB data flag shall be set if a device has the peer device's out-of-band authentication data. A device uses the peer device's out-of-band authentication data to authenticate the peer device. More specifically, after public key exchange when a device receives the OOB confirm value, if the confirm value does not match, or the peripheral does not have the central's OOB data, then the device should immediately abort the pairing process by sending *PairFailed* message. However, BLEDiff found 6 implementations where without receiving any OOB confirm data, the peripheral devices proceed to the next step, i.e., random value exchange, completely bypassing the authentication as shown in Figure 7. To make matters worse, the implementations even pass *DHKeyCheckSend* with rb set to zero and complete pairing and bonding altogether.

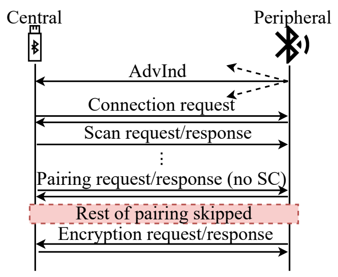

Fig. 8: Legacy pairing bypass

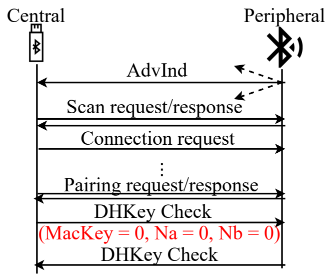

Fig. 9: Invalid DHKey Check

Root cause. The root cause of this issue is that the implementation is deviating from standards. In the specification, it is mandated that if the confirm value received from OOB does not match the calculated value, the peripheral will abort the pairing process [4, p. 1632].

Impact. An attacker in the radio range can abuse this vulnerability to completely bypass OOB authentication in the affected BLE devices, which rely on secure connections with out-of-band data to protect user privacy. As the OOB authentication is bypassed, an attacker can send the usual BLE packets, impersonate both the legitimate central and peripheral, and perform MitM attacks on BLE connections.

(E3) Legacy Pairing Bypass. In this deviation, it is possible to bypass the legacy pairing procedure and start encryption on a device. During legacy pairing, an implementation exchanges random values and confirms the values to generate Short Term Keys (STK). Without these procedures, an implementation cannot move to the encryption procedure. However, for the affected devices, the implementations skip part of the pairing procedures and directly proceed to encryption (shown in Figure 8). In the specification, the flow of pairing is clearly attributed, and hence starting the encryption procedure without even completing the pairing is a deviation from the standards. For exploiting this deviation in an attack, an attacker in the radio range can skip the pairing procedure and directly start encryption and try to bypass BLE security.

Impact. There are two impacts of this deviation. The first impact is that it can cause security bypass due to low-entropy key size. The STK is generated using $s1 = (k, r1, r2)$. Each of these parameters are 128-bit long. In the key generation phase, 64 bits of $r1$ and 64 bits of $r2$ are discarded to create a 128-bit input, which together with k generates the STK. In case the pairing procedures are bypassed, and with no input and no output capability ($k = 0$), the implementation generates a key with only the 64-bits of $r2$, thus generating a key with much smaller entropy. This can potentially lead to a security bypass. For other I/O capabilities, the entropy will be higher with different k value, but lower than the envisioned entropy when random values are exchanged. The second impact is DoS. As part of pairing can be bypassed (including authentication), an attacker can start encryption without completing the authentication. In case the attacker is unable to figure out the low-entropy key, there is a key-mismatch and the connection is dropped. Since an attacker can drop a connection without authentication, this can cause a DoS.

Issue	Impact	I/S
(E1) Bypassing passkey entry in legacy pairing	Security bypass	I
(E2) Bypassing Out-Of-Band Authentication	Security bypass	I
(E3) Bypassing legacy pairing	DoS	I
(E4) Accepts *DHKeyCheckSend* with all fields zero	DoS	I
(E5) Unresponsiveness with *PauseEncRespPlainText*	Crash	I
(E6) Unresponsiveness with *ConReqTimeoutZero* and *ConReqIntervalZero*	Crash	I
(E7) Accepts *PauseEncReqPlainText* before pairing is complete	DoS	I
(E8) Issue with incomplete *PairReq*	DoS	I
(E9) Accepts *PairRandomSend* before exchanging public keys	DoS	S
(E10) Accepts *PairConfirmSend* with wrong values	DoS	I
(I1) Issue with reject messages	Interoperability	S
(I2) Issue with OOB pairing failed	Interoperability	I
(O1) Accepting key size greater than max	-	I

TABLE I: Deviations identified by BLEDiff. E- exploitable, I- interoperability issue, O- other deviating behavior, I- Implementation issue, S- Specification issue.

(E4) Invalid DHKey Check. In this deviation, during BLE secure pairing, the BLE implementations respond to *DHKeyCheckSend* message with $MacKey, Na$, and Nb set to zero (shown in Figure 9). This behavior deviates from the standards as the implementations fail to properly check the confirmation value. As specified in the standard, if the confirmation value check fails, it indicates that the initiating device has not confirmed the pairing, and the protocol must be aborted.

Impact. Due to this noncompliance, it is possible for an attacker to inject the *DHKeyCheckSend* packet with $MacKey, Na$, and Nb set to zero during the pairing procedure, forcing the vulnerable device to stop communicating with a specific central device and causing DoS. Furthermore, this deviation can be a stepping stone for a much more severe issue as illustrated below. After *DHKeyCheckSend*, the encryption procedure is started which uses the generated LTK to encrypt subsequent packets. The task of the *DHKeyCheckSend* is to ensure the right key is generated. In case *DHKeyCheckSend* fails, the subsequent LTK is discarded due to security reasons. Since it is possible to bypass *DHKeyCheckSend* by setting $MacKey, Na$, and Nb to zero, the attacker may exploit it to bypass the security partially as well.

(E5) Device Unresponsiveness with *PauseEncRespPlainText*. The deviation happens when a BLE device receives a plaintext *PauseEncResp*. Even before pairing, if the BLE implementation receives *PauseEncRespPlainText*, then it crashes and becomes unresponsive. This is a clear deviation from the standards. In case a device handles an invalid packet, there are three ways to handle it (i) ignoring the packet, (ii) sending rejection, (iii) terminating the connection. However, in this case, the packet causes a fault in the implementation.

Impact. It is possible to cause DoS by sending this packet to the implementation. The packet is plaintext and does not have integrity protection; therefore, it can be sent by an attacker anytime to an existing BLE connection. Moreover, the packet does not show any prompt on the smartphone and turns off the Bluetooth for some time. It seems the packet causes restart of the Bluetooth daemon of the device. Therefore, sending such packets in a loop can create permanent DoS without any notification to the user.

(E6) Device unresponsiveness with *ConReqTimeoutZero* When a device receives a *ConReq* with the timeout field set to zero, the device becomes completely unresponsive. A user has to manually turn on the Bluetooth service to make the device responsive. A similar attack with invalid connection requests was shown in [2] on two development boards. We have found this issue in 5 different smartphones and 3 different development boards. Furthermore, in their attack for the invalid connection request, both the interval and timeout fields have to be set to zero. In our case, the interval field does not matter; as long as the timeout field is set to zero, the device becomes unresponsive and automatically turns off Bluetooth.

Impact. An attacker in the radio range can exploit the issue to cause a surreptitious denial of service of the Bluetooth. Though this attack is on BLE, the smartphone turns off both BLE and BR/EDR without notifying the user. To resolve this, the user has to manually restart BLE and, in some cases, the smartphone altogether.

(E7) DoS with *PauseEncReqPlainText*. In this deviation, the device responds with a *PauseEncResp* in case a plaintext *PauseEncReq* is sent. As a result, the affected device moves to an incorrect state of the implementation where it is not able to complete pairing and not able to communicate with a specific central device. As stated in the previous section, responding to an invalid message is a noncompliance. We found this issue in 5 different BLE implementations.

Impact. The implementation goes to an incorrect state and discards subsequent messages from the central. The deviation thus enables an attacker to induce DoS attacks on the affected devices. An correctly implemented device ignores plaintext *PauseEncReq* messages and does not change state.

(E8) DoS with *PairReq*. In this deviation, the implementations do not respond to subsequent *PairReq*'s if the first *PairReq* is not properly completed. In such a case, the peripherals stop advertising altogether and are not able to communicate with *any* central device within their radio range. This is a noncompliance with the standards as one connection should not affect the other subsequent connections. An attacker in radio range acting as a central can initiate a pairing but abruptly close the connection. This will create a service disruption in the affected devices as those devices will not respond to any other legitimate device in the radio range.

(E9) Accept *PairRandomSend* before *PublicKeySend*. The affected devices deviate from the standard by responding to a *PairRandomSend* message before authentication and *PublicKeySend*. Because of accepting *PairRandomSend*, the implementations move to an incorrect state from which it cannot complete the pairing procedure. Exploiting this an attacker can force the vulnerable device to stop communicating with a specific central device. Although the standard specifies the regular protocol flow, it does not explicitly state how to handle out-of-order protocol messages. Hence, this behavior can be attributed due to the underspecification of the standards.

(E10) *PairConfirmSend* Value Mismatch. The affected devices respond to *PairConfirmSend* request with wrong confirm values.

The deviation occurs when a *PairReq* is sent with the secure connection flags turned on or the OOB flag turned on. Due to this, the implementations go to an unintended state, and do not complete pairing and bonding anymore. In the correct implementations, the devices ignore *PairConfirmSend* and proceed with pairing as mandated by the standard.

2) Interoperability

(I1) Interoperability with reject messages. In case a device receives an invalid message, it can respond with a reject message. However, the specification does not specify the order of the reject messages in a order sequence. In our experiments, the implementations respond at different places in case of invalid messages and this can create a potential interoperability issue among different devices. For instance, in case a device receives a *PublicKeySend* with an invalid key, (i) some implementations send a reject message as soon as the invalid message is received; (ii) some implementations still continue with the subsequent procedures and respond to *DHKeyCheckSend* with a reject message; (iii) some implementations do not send any reject messages. We found 16 devices following (iii), 6 devices following (i), and 3 devices following (ii).

(I2) Interoperability with OOB Pairing Failed. As discussed in VI-A1, in case of pairing with OOB data, if the confirm value fails, then the pairing should be aborted right away. However, BLEDiff found implementations where even after the confirm value fails, the implementations still proceed with random value exchange. This deviates from the standards and can cause potential interoperability issues. One thing to be noted, in these implementations, the pairing eventually fails during *DHKeyCheckSend*, and it is not possible to pair and bond with the device.

3) No impact

We found one deviation where the impact of the deviation is not clear. In this deviation, an implementation accepts *PairReq* with a key size greater than the max value of 16 bytes. The specification mandates using a key size of 7 to 16 bytes; however, in this case, the implementation becomes noncompliant by accepting a key size greater than the max value. A similar issue was found by Pferscher et al. in a different device [13]. Although this is a deviation from the standard, it is not evident how this can be exploited.

B. Comparison with existing testing approach

We compare the effectiveness of BLEDiff with the BLE conformance or qualification testing framework defined in the BLE standards [36] and the previous approaches on BLE testing [2], [7]–[10], [13], [37], and summarize the results in Table II. Although *line coverage* and *function coverage* of a device under test are commonly used metrics to fairly evaluate and compare these frameworks and tools, BLEDiff being a black-box noncompliance checking method poses a challenge of extracting coverage data from commercial BLE devices. To address this challenge, we run an open-source BLE implementation BTstack [38] on an Ubuntu 18.04 machine with BLE version 5.0. We run all the testing frameworks for 24 hours and compute line coverage and function coverage

Paper	Automatic	Specification analysis	Implementation analysis	Under-specification detection	Non-compliance-checking
SwyenTooth [2]	✓	✗	✓	✗	✗
BIAS [7]	✗	✓	✗	✗	✗
KNOB [8]	✗	✓	✗	✗	✗
Model-Driven [9]	✗	✓	✗	✗	✗
Fingerprinting [37]	✓	✗	✓	✗	✗
Black-box Fuzzing [13]	✓	✗	✓	✗	✗
Frankenstein [10]	✓	✗	✓	✗	✗
BLEDiff	✓	✓	✓	✓	✓

TABLE II: Comparison with existing approaches.

using LCOV [39], which is an extension of GCOV [40]. The results of this endeavor is shown in Figure 10 in the Appendix and discussed below.

1) Conformance or qualification testing framework

BLE standards [36] define conformance or qualification testing where different scenarios and expected behavior are described. For a fair comparison, we consider only the test cases which are relevant to the procedures in our scope. Results show that these standard tests cover 59.47% of lines and 68.92% of functions, whereas BLEDiff achieves 63.29% line coverage and 71.69% function coverage.

2) Previous approaches on BLE testing

Among the previous works on BLE testing, SwyenTooth [2], Fingerprinting [37], Black-box Fuzzing [13], and Frankenstein [10] are automatic approaches analyzing BLE implementations and do not require manual intervention, in general. On the other hand, BIAS [7], KNOB [8], Model-Driven [9] are manual and perform analysis on specifications. However, none of these works can identify underspecifications or noncompliance. Compared to these previous works, BLEDiff is automatic, can analyze both specifications and implementations, and can also discover underspecifications in the standards and noncompliance of implementations. These features are summarized in Table II.

Among the previous works that do automatic testing, we do not calculate coverage for Frankenstein [10] as it is a reverse-engineering based approach which requires a significant manual effort to run and is not a plug-and-play. We compute coverage for the other tools and summarize results in Figure 10. The comparison shows that BLEDiff is the most effective approach. For SwyenTooth [2], Fingerprinting [37], and Black-box Fuzzing [13], the line coverage are 59.68%, 43.79%, and 41.37%, respectively, and the function coverage are 68.92%, 53.85%, and 51.08%, respectively. Compared to these, BLEDiff has 63.29% line coverage and 71.69% function coverage. More discussion on some of the manual intervention needed in some tools is analyzed in Appendix B.

C. BLEDiff *performance*

1) BLE Learning module performance

Table V shows the summary of membership queries, equivalence queries, time, states, and transitions required for BLE Learning module to infer the FSM of a BLE implementation. In the worst case, the BLE Learning module requires 3 days to learn the FSM, with the average being 1.7 days. The device-specific details for all the specific devices are shown in Table VIII in the Appendix.

2) *Performance of the divide and conquer approach*

To improve the scalability of active automata learning, we propose the idea of using a divide and conquer approach by extracting 3 different FSMs and merging them. To evaluate the performance improvement of learning, we take a device (Nexus 6) run divide-and-conquer approach without any caching or constraints additions. For the baseline, we run a general model learning approach on the same device with all the input symbols (32) and the usual techniques to handle scalability (e.g., caching, constraints addition) in hopes of inferring a large FSM of the entire BLE implementation. We pick *StartEncResp* as the terminating symbol as it marks the completion of the scope of encryption, pairing, and bonding. However, with all the input symbols, it took the learner more than two days just to complete the link layer procedure connection. For most of the input symbols, the response is *Null*. This is because symbols of SMP or reconnection do not induce any changes to LL FSM, but the learner still has to run all the symbols over-the-air wasting precious time and queries. We estimate that with all 32 symbols, it will take the general learner more than 5 days to learn the full FSM, which is more than twice the time taken by BLEDiff with its divide-and-conquer learning approach. The comparison of both approaches is shown in Table IV.

3) *BLE checking module performance*

To evaluate the performance of BLE checking module, we pair-wise compare the number of deviations and the time required to find the deviations among all the devices with the closest implementation to our BLE checking module–the equivalence checker designed in the context of 4G LTE called DIKEUE [21]. On average BLE checking module finds 62% more deviant traces compared with the DIKEUE equivalence checker. This is due to the fact that BLE checking module finds deviant traces with higher depths, whereas DIKEUE finds the shortest trace only. On timing BLE checking module takes on an average of 17.4 sec to find all the deviating traces compared to 11.49 sec for DIKEUE. This increase can be attributed to the calls to the model checker for finding counterexamples of increasing length. Compared to finding deviation-inducing traces deep inside the FSM, this time increase is reasonable. The statistics of the number of deviations identified and the time of both the approaches are shown in Table VI. For an interested reader, the detailed pair-wise comparison of the number of deviations and time is shown in Table X and Table XII, respectively in the Appendix.

To illustrate with a concrete example, the deviation and the corresponding attack **E7** are not detectable through the elimination-based approach of DIKEUE. This deviation is detected only through BLEDiff's BLE checking module because of its ability to detect more in-depth deviation. The FSMs and the deviations are discussed in a simplified form in the running example of section IV-B1 and Figure 5.

VII. Related work

We discuss related work in two directions relevant to our work. At first, we discuss work in Bluetooth security and then in active automata learning.

BLE implementation security. BlueBorne [5] and Bleedingbit [6] manually identifies critical attack vectors that can be used to take control of affected devices even without pairing. BIAS [7], and KNOB [8] also manually analyze the BR/EDR specifications and present practical impersonation attacks. BLESA [3], on the other hand, builds a ProVerif model according to the BLE specifications and analyzes the model to find security implications. The authors present impersonation attacks using BLE spoofing in this work. Furthermore, Wu et. al. [9] introduces an extensive ProVerif model that encompasses both key sharing and data transmission phases in Bluetooth Classic, BLE, and Bluetooth Mesh. However, these works only consider the specifications, whereas we consider both implementations and specifications. SweynTooth [2] provides a testing framework to identify implementation vulnerabilities, whereas Frankenstein [10] uses firmware emulation to run fuzzing on firmware dumps. Moreover, InternalBlue [11] releases a reverse-engineered Bluetooth implementation for the research community. BLURtooth [12] analyzes the Cross-Transport Key Derivation (CTKD) feature of Bluetooth. They also uncover four different vulnerabilities in this feature and report corresponding attacks. However, none of these works aim to systematically explore protocol noncompliance.

Active autoamta learning. Active automata learning approaches are often used to analyze protocol implementations for a wide range of protocols, including OpenVPN [18], QUIC [19], TCP [16], [27], TLS [14], [23], DTLS [15], SSH [20], LTE [21], IoT [41]. Pferscher et al. [37] employ model learning for fingerprinting BLE devices. However, the scope of this work is limited to the link layer only, and it does not cover pairing, bonding, encryption, secure procedures, or different device capabilities. Therefore, the authors do not face the challenge of scalability. In the subsequent work, they use the learned model to fuzz BLE implementations in a stateful manner and consequently suffer from the same issues due to limited scope [13]. On the other hand, to mitigate some well-known drawbacks of active automata learning, several works attain different techniques. To deal with the unreliability of over-the-air communication and ensuing non-deterministic results, the majority voting scheme has proven useful in previous works [15], [21], [30], [42]. Furthermore, HVLearn [23], SFADiff [27], and DIKEUE [21] utilize caching mechanism to circumvent sending duplicate queries over-the-air which, in turn, reduces the time requirement of automata learning.

VIII. Discussion

Manual process of deviation to attack analysis. As multiple deviations may have the same root cause, we manually analyze diverse deviations uncovered through our automated technique BLEDiff and identify unique deviations. In Table X, all pairwise deviant behaviors are reported and through consultation with the specification, the unique deviant behaviors are elaborated. The high number of deviations in Table X as compared to 13 unique ones is because:

1) If an input i_j (e.g., *ConReqTimeoutZero*) in a query $q = \langle i_1 i_2 \dots i_j \dots i_m \rangle$ induces a crash to a device D_1, D_1's outputs for the remaining inputs $\langle i_{j+1} \dots i_m \rangle$ in q become *Null* as D_1 becomes unresponsive after i_j. While comparing D_1 with another device D_2 which did not crash at i_j, BLEDiff yields multiple deviant behavior inducing input sequences, for instance, $\langle i_1 \dots i_j \rangle, \langle i_1 \dots i_{j+1} \rangle, \dots, \langle i_1 \dots i_{j+1} \dots i_m \rangle$ for which the root cause is same;
2) If for an input sequence $\langle i_1 i_2 i_3 i_2 \rangle$, devices D_1 and D_2 yield $\langle o_1 o_2 o_3 o_2 \rangle$ and $\langle o_1 o_2^{'} o_3 o_2^{'} \rangle$ as outputs, respectively, BLEDiff identifies both deviations $\langle i_1 i_2 \rangle$ and $\langle i_1 i_2 i_3 i_2 \rangle$ as it aims to identify deviations of different depths. Although these are valid deviations but are not considered unique as they occur from the same root cause.

Soundness. BLEDiff does not have any false positives. If BLEDiff finds a deviation in the FSMs of two devices under consideration, for the same input sequence, two corresponding implementations indeed behave differently. False positives could have occurred if: (1) the extracted input/output FSMs had states/transitions that do not exist in devices/implementations; or (2) input/output symbols were different for different devices. BLEDiff addresses the former with formal soundness guarantees of active automata learning underpinning BLEDiff's FSM extraction process [24]. To ensure the same input/output symbols for all devices, BLEDiff defines input/output symbols (shown in Table VII) based on high-level protocol messages and their security features that are consistent across BLE versions (from 4.2 to 5.2). BLEDiff abstracts away non-security-related protocol features for instance versions and modulation schemes in input/output symbols through mappers. For instance, when Link Layer (LL) mapper receives a *FeatureResp* message from a device, it abstracts the contents of the packet and responds with a *FeatureResp* to the learner. Similarly, when the mapper receives a *VersionResp*, whatever the version number is (e.g., 4.2, 5.0, 5.1), the mapper responds to the learner with a *VersionResp* message type as output. Furthermore, assumptions made in BLEDiff do not affect soundness/correctness. Since peripheral-originated LL messages, e.g., *LenReq* are stateless and originated by a device anytime, LL mapper abstracts those messages by not modeling them as input/output symbols. To ensure sound/correct protocol flow, in response to peripheral-originated messages, the LL mapper sends valid and protocol-compliant messages to the device under test but does not send corresponding output symbols to the learner. As this learning process is consistent across all devices, the learner learns consistent and sound FSMs, and this assumption does not affect the soundness of the extracted FSMs.

Completeness. Testing a complex system is inherently an incomplete process and so is BLEDiff. Our approach cannot uncover all possible deviations in different implementations because: (1) the predicates included to reason about security-critical behavior may not be complete. For instance, there may be other predicates apart from the field and packet level predicates we used in BLEDiff, that can cause deviant behavior; (2) the abstractions made to handle peripheral-originated messages from learner may also miss some deviant behaviors. As discussed in the previous section, to handle peripheral originated messages such as *LenReq*, LL mapper abstracts those messages by not modeling them as input/output symbols and in turn causes incompleteness; (3) limitations of differential testing, especially, not having access to a reference FSM from the specification. As a result, if two implementations deviate in the same way, then the differential testing might miss it. But as we do pairwise differential testing among all 25 device implementations, at least one pair of comparison will yield the deviation if there is any. In a nutshell, BLEDiff is incomplete in the sense that it can not guarantee all deviations, but in practice, it can identify the majority of them.

FSM Merging. BLEDiff first merges the FSMs of individual sub-protocols and compare the entire FSM with that of other BLE implementations. An alternative design could be other way around, i.e., instead of merging the FSMs of sub-protocols, comparing them separately. From the perspective of finding deviations, this will not affect the results. However, the reason for merging to create a complete protocol is twofold: (1) it allows the equivalence checker to find an end-to-end trace of deviant behavior (i.e., from entry-point of BLE protocol to where deviation occurs) that can be readily converted to a concrete test case for further testing; (2) the complete protocol can be further leveraged by developers for other analysis, e.g., stateful fuzzing.

Cross-sub protocol analysis. BLEDiff does not model cross-sub protocol interactions. There can be deviations where one sub-protocol affects others and BLEDiff currently cannot detect those. We leave it for future work.

IX. Conclusion

We present BLEDiff a scalable, property-agnostic, and black-box protocol noncompliance checking framework for BLE implementations. We also introduce the idea of divide-and-conquer-based automata learning, where a protocol is divided into multiple sub-protocols, for each sub-protocol, a separate FSM is learned, and then merged together to form the large protocol FSM.

Future Work. In future, we will port this approach to BLE central implementations. Furthermore, we will develop new techniques to further improve the scalability of active automata learning approaches and model cross-sub-protocol interactions.

Acknowledgement

We thank the anonymous reviewers and the shepherd for their feedback and suggestions. We also thank all the vendors for their cooperation during the responsible disclosure. The work reported in this paper has been supported by NSF under grant 2112471 “AI Institute for Future Edge Networks and Distributed Intelligence (AI-EDGE)” by Defense Advanced Research Projects Agency (DARPA) under contract number D22AP00148. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily reflect the views of United States Government or any agency thereof.

References

[1] *2022 Bluetooth Market Update*, https://www.bluetooth.com/2022-market-update/.

[2] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan, "SweynTooth: Unleashing mayhem over bluetooth low energy," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 911–925. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/garbelini

[3] J. Wu, Y. Nan, V. Kumar, D. J. Tian, A. Bianchi, M. Payer, and D. Xu, "BLESA: Spoofing attacks against reconnections in bluetooth low energy," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/wu

[4] *Bluetooth Special Interest Group, Core Specification 5.3*, https://www.bluetooth.com/specifications/specs/core-specification-5-3/.

[5] B. Seri and G. Vishnepolsky, "Blueborne: The dangers of bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern bluetooth stacks," *Department of Computer Science, Michigan State University, East Lansing, Michigan, Tech. Rep*, 2017.

[6] B. Seri, G. Vishnepolsky, and D. Zusman, "Bleedingbit: The hidden attack surface within ble chips," 2019.

[7] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "Bias: Bluetooth impersonation attacks," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 549–562.

[8] D. Antonioli, N. O. Tippenhauer, and K. 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)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1047–1061. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/antonioli

[9] J. Wu, R. Wu, D. Xu, D. J. Tian, and A. Bianchi, "Formal model-driven discovery of bluetooth protocol design vulnerabilities," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 2285–2303.

[10] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 19–36. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ruge

[11] D. Mantz, J. Classen, M. Schulz, and M. Hollick, "Internalblue - bluetooth binary patching and experimentation framework," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 7990. [Online]. Available: https://doi.org/10.1145/3307334.3326089

[12] D. Antonioli, N. O. Tippenhauer, K. Rasmussen, and M. 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*, ser. ASIA CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 196207. [Online]. Available: https://doi.org/10.1145/3488932.3523258

[13] A. Pferscher and B. K. Aichernig, "Stateful black-box fuzzing of bluetooth devices using automata learning," in *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 2427, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 373392. [Online]. Available: https://doi.org/10.1007/978-3-031-06773-0_20

[14] J. De Ruiter and E. Poll, "Protocol state fuzzing of tls implementations," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. USA: USENIX Association, 2015, p. 193206.

[15] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2523–2540. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[16] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 454–471.

[17] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-based testing iot communication via active automata learning," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 276–287.

[18] L. Daniel, E. Poll, and J. de Ruiter, "Inferring openvpn state machines using protocol state fuzzing," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2018, pp. 11–19.

[19] A. Rasool, G. Alpár, and J. de Ruiter, "State machine inference of QUIC," *CoRR*, vol. abs/1903.04384, 2019. [Online]. Available: http://arxiv.org/abs/1903.04384

[20] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of ssh implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 142151. [Online]. Available: https://doi.org/10.1145/3092282.3092289

[21] S. R. Hussain, I. Karim, A. A. Ishtiaq, O. Chowdhury, and E. Bertino, "Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4g lte cellular devices," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 10821099. [Online]. Available: https://doi.org/10.1145/3460120.3485388

[22] M. Chlosta, D. Rupprecht, and T. Holz, "On the challenges of automata reconstruction in lte networks," in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 164174. [Online]. Available: https://doi.org/10.1145/3448300.3469133

[23] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 521–538.

[24] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87 – 106, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0890540187900526

[25] M. Isberner, F. Howar, and B. Steffen, "The ttt algorithm: A redundancy-free approach to active automata learning," in *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds. Cham: Springer International Publishing, 2014, pp. 307–322.

[26] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, 1978.

[27] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias, "Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 16901701. [Online]. Available: https://doi.org/10.1145/2976749.2978383

[28] S. L. Harris and D. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2015.

[29] *Formal equivalence checking*, https://en.wikipedia.org/wiki/Formal_equivalence_checking.

[30] S.-J. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, and Y. Zhang, "Alembic: Automated model inference for stateful network functions," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. USA: USENIX Association, 2019, p. 699718.

[31] *IOS13-SimulateTouch-iOS Automation Framework iOS Touch Simulation Library*, https://github.com/xuan32546/IOS13-SimulateTouch.

[32] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 487–495.

[33] M. Isberner, "Foundations of active automata learning: An algorithmic perspective," Ph.D. dissertation, 10 2015.

[34] *nRF52840 Dongle*, https://www.nordicsemi.com.

[35] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 334–342.

[36] *Bluetooth Qualification Test Requirements*, https://www.bluetooth.com/specifications/qualification-test-requirements/.

[37] A. Pferscher and B. K. Aichernig, "Fingerprinting bluetooth low energy devices via active automata learning," in

Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 2026, 2021, Proceedings. Berlin, Heidelberg: Springer-Verlag, 2021, p. 524542. [Online]. Available: https://doi.org/10.1007/978-3-030-90870-628

[38] BlueKitchen, *BTstack: Dual-mode Bluetooth stack, with small memory footprint.*, https://github.com/bluekitchen/btstack.

[39] *LTP GCOV extension (LCOV)*, https://github.com/linux-test-project/lcov.

[40] *Gcov (Using the GNU Compiler Collection (GCC))*, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[41] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-based testing iot communication via active automata learning," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 276–287.

[42] C. McMahon Stone, T. Chothia, and J. de Ruiter, "Extending automated protocol state learning for the 802.11 4-way handshake," in *Computer Security*, J. Lopez, J. Zhou, and M. Soriano, Eds. Cham: Springer International Publishing, 2018, pp. 325–345.

[43] *Fluoride Bluetooth stack*, https://android.googlesource.com/platform/system/bt/+/181144a50114c824cfe3cdfd695c11a074673a5e/README.md.

[44] *iOS BLE Stack*, https://developer.apple.com/documentation/corebluetooth.

Appendix A
Evaluation Setup.

For all the evaluations **RQ1** - **RQ3**, we use 3 laptops with Intel i7-3750QCM CPU and 32 GB DDR3 RAM. For the BLE Learning module we use three nRF52840 Dongles to send/receive raw link layer packets. All the experiments are done in a laboratory environment with our own BLE devices without affecting any other BLE devices nearby.

Appendix B
Manual intervention for comparison with existing testing approach

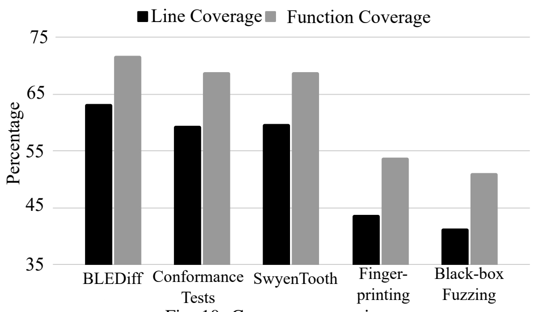

Fig. 10: Coverage comparison

During the comparison with existing approaches on BLE testing, though some of the tools and approaches are automatic required some manual interventions in our experiments. For instance, in the case of Fingerprinting [37], and Black-box Fuzzing [13] the testing apparatus' frequent crashed or froze. Moreover, both these approaches: Fingerprinting and Black-box Fuzzing have limited scope compared to our work and do not incorporate secure pairing or encryption. Accordingly, these are not effective in discovering vulnerabilities at a complex level. Also, Black-box Fuzzing requires that the finite state machine of the device under test is first learned and then fuzzed. However, in our experiments, the tool could not complete learning the finite state machine within 24 hours. Instead, we let the fuzzer use a pregiven complete finite state machine so that it can explore further and get a fair opportunity. Despite the efforts, its effectiveness was the worst among the tools we tested in terms of coverage.

Component	Tools	Lines of Code
Learner	LearnLib [32]	2157 (Java)
Mapper	–	2288 (Java)
Modified BLE stack	SwyenTooth [2]	4488 (Python 2.7) & 15215 (C)
Device resetter	–	965 (Python 2.7)
FSM Merger	–	302 (Python 3.10)
FSM Equivalence Checker	–	2240 (Python 2.7)

TABLE III: Additions/modifications to the tools used in BLEDiff.

Approach	Membership Queries	Equivalence Queries	Time (min)	States	Transitions
Divide and conquer learning	323	1946	2448	7	121
Automata learning with caching and constraints	3077	1495	3077*	6	192

TABLE IV: Comparison between divide and conquer learning and general model learning for Nexus 6
* The learner just completed link layer connection

Statistic	Membership Queries	Equivalence Queries	Time (min)	States	Transitions
Max	1045	4022	4805	9	152
Min	113	277	274	5	77
Average	519.32	2552.6	2546.92	7.44	128.68
Median	339	2042	2400	7	121
Standard Deviation	327.85	979.99	994.95	0.82	15.73

TABLE V: Summary of time, membership, and equivalence queries.

Statistic	BLEDiff Deviations	DIKEUE Deviations	BLEDiff Time	DIKEUE Time
Max	43	42	63.64	46.97
Min	11	8	17.4	11.49
Average	26.96	16.72	41.72	25.71
Median	27	16	42.25	24.65
Standard Deviation	5.96	4.78	9.70	6.85

TABLE VI: Summary of deviations and time.

Message	**Input Symbol**	**Adversarial Symbols**	**Output Symbols** (Λ)
Link Layer Control Protocol			
Feature Request	*FeatureReq*		*FeatureResp*
Exchange MTU Request	*MTUReq*		*MTUResp*
Length Request	*LenReq*		*LenResp*
Read by Group Type Request	*ReadTypeReq*		*ReadTypeResp*
Connection Request	*ConReq*	*ConReqIntervalZero*, *ConReqTimeoutZero*	
Version Request	*VersionReq*	*VersionReqMaxLen*	*VersionResp*
Security Manager Protocol (SMP)			
Pairing Request (SC) (NoInput NoOutput)	*PairReq*	*PairReqKeyZero*, *PairReqKeyMax*	*PairResp*
Pairing Request (SC) (Display Yes/No)	*PairReq*		*PairResp*
Pairing Request (SC) (Keyboard Display)	*PairReq*		*PairResp*
Pairing Request (Legacy) (NoInput NoOutput)	*PairReqLegacy*		*PairResp*
Pairing Request (Legacy) (Keyboard Display)	*PairReqLegacy*		*PairResp*
Pairing Request (Legacy) (Display Yes/No)	*PairReqLegacy*		*PairResp*
Pairing Request (OOB)	*PairReqOOB*		*PairResp*
Public Key Exchange	*PublicKeySend*	*PublicInvalidKeySend*	*PublicKeyRecv*
Pair Confirm	*PairConfirmSend*	*PairConfirmWrongValueSend*	*PairConfirmRecv*
Pair Random	*PairRandomSend*		*PairRandomRecv*
Diffie-Hellman Key Check	*DHKeyCheckSend*	*DHKeyCheckInvalidSend*	*DHKeyCheckRecv*
Reconnection			
Encryption Request	*EncReq*		*EncResp*, *StartEncReq*
Start Encryption Response	*StartEncResp*	*StartEncRespPlainText*	*StartEncResp*
Encryption Pause Request	*PauseEncReq*	*PauseEncReqPlainText*	*PauseEncResp*
Encryption Pause Response	*PauseEncResp*	*PauseEncRespPlainText*	

TABLE VII: List of input, adversarial and output symbols. In case there is a timeout the default output symbol is *Null*

Device	**Membership**			**Equivalence**			**#States**			**#Transition**			**Time**			**Total Time**	**Merged States**	**Merged Trans**
	LL	SMP	Recon	LL	SMP	Recon	LL	SMP	Recon	LL	SMP	Recon	LL	SMP	Recon			
Nexus 6	30	276	17	15	1904	27	3	4	2	30	88	13	38	2180	76	2294	7	121
DA14531	50	377	17	27	3644	27	3	5	2	30	110	13	25	1340	50	1415	9	152
NRF5340-DK	30	287	21	27	1989	17	3	4	2	30	88	13	19	1517	51	1587	8	130
CC2640R2	30	66	17	27	223	27	3	5	2	30	110	13	27	192	55	274	5	77
CYBLE-416045-EVAL	30	998	17	27	3869	27	3	5	2	30	110	13	19	2858	52	2929	8	143
Pixel 4a	30	282	17	27	1936	27	2	4	2	20	88	13	57	2957	88	3102	7	121
STEVAL-IDB008V2	30	990	17	27	3666	27	3	5	2	30	110	13	76	4656	73	4805	8	143
OnePlus 8	30	994	17	27	3442	27	3	5	2	30	110	13	38	4436	58	4532	8	143
CY8CPROTO-063-BLE	30	990	17	27	3254	27	3	5	2	30	110	13	19	2829	59	2907	8	143
NRF52-DK	30	325	21	27	2098	17	3	4	2	30	88	13	23	1615	63	1701	8	130
Galaxy S6	30	286	17	15	1902	27	3	4	2	30	88	13	42	2188	73	2243	7	121
Desire 10 Lifestyle	30	295	17	15	1876	27	3	4	2	30	88	13	30	2171	77	2278	7	121
Pixel 3XL	30	292	17	27	1996	27	2	4	2	20	88	13	38	2288	74	2400	7	121
Galaxy S8+	30	290	17	27	1886	27	2	4	2	20	88	13	57	2901	73	3031	7	121
Y5 Prime	30	290	17	15	2103	27	3	4	2	30	88	13	30	3190	88	3308	7	121
8X	50	342	17	27	3788	27	3	5	2	30	110	13	51	4130	74	4255	9	152
Mi A1	30	268	17	27	1842	27	2	4	2	20	88	13	32	1813	74	1919	7	121
iPhone XS	30	244	17	27	1920	27	2	4	2	20	92	13	57	2164	69	2263	7	123
Pixel 3XL	30	188	17	27	1845	27	2	4	2	20	88	13	57	2710	66	2833	7	121
G Power	30	298	17	27	1934	27	2	4	2	20	88	13	45	1876	66	1987	7	121
7T	30	276	17	27	1988	27	2	4	2	20	88	13	66	2641	62	2769	7	121
Ubuntu 18.04	30	954	17	27	3562	27	3	5	2	30	110	13	21	2258	43	2322	8	143
Ubuntu 20.04	30	986	17	27	3958	27	3	5	2	30	110	13	23	2342	45	2410	8	143
ESP32-C3	30	940	17	27	3968	27	3	5	2	30	110	13	19	2543	38	2600	8	143
DT100112	30	226	17	15	1952	27	3	4	2	30	88	13	38	1452	19	1509	7	121

TABLE VIII: Time, membership, and equivalence queries.

	D1 Nexus6	**D2** DA14531	**D3** CC2640R2	**D4** NRF5340-DK	**D5** NRF52-DK	**D6** CYBLE-416045	**D7** CY8CPROTO-063-BLE	**D8** STEVAL-IDB008V2	**D9** DT100112	**D10** ESP32-C3	**D11** Galaxy S6	**D12** Desire 10 Lifestyle	**D13** Galaxy S8+	**D14** Pixel 3XL	**D15** Pixel 4a	**D16** Y5 Prime	**D17** 8X	**D18** Mi A1	**D19** iPhone XS	**D20** Galaxy A21	**D21** G Power	**D22** 7T	**D23** OnePlus 8	**D24** Laptop (18.04)	**D25** Laptop (20.04)
E1	✓								✓		✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓		
E2						✓	✓	✓						✓									✓	✓	
E3		✓																							
E4			✓																						
E5																✓				✓					
E6	✓			✓					✓		✓	✓				✓									
E7		✓		✓	✓				✓								✓								
E8								✓																	
E9			✓																						
E10			✓						✓																
I1	✓		✓	✓	✓				✓		✓	✓												✓	✓
I2						✓	✓	✓						✓									✓	✓	✓
O1																			✓						

TABLE IX: Attacks to device mapping

	Nexus6	DA14531	CC2640R2	NRF5340-DK	NRF52-DK	CYBLE-416045	CY8CPROTO-063-BLE	STEVAL-IDB008V2	DT100112	ESP32-C3	Galaxy S6	Desire 10 Lifestyle	Galaxy S8+	Pixel 3XL	Pixel 4a	Y5 Prime	8X	Mi A1	iPhone XS	Galaxy A21	G Power	7T	OnePlus 8	Laptop (18.04)	Laptop (20.04)
Nexus6	–	14	19	25	24	14	14	15	11	18	8	9	15	15	14	8	14	13	14	8	16	14	18	19	16
DA14531	**31**	–	22	23	23	18	18	18	15	18	14	16	16	17	17	14	19	17	17	14	17	17	18	18	18
CC2640R2	**34**	**43**	–	25	25	18	18	18	18	18	19	19	19	21	19	19	19	19	19	19	19	19	18	18	18
NRF5340-DK	**33**	**24**	**32**	–	9	23	23	23	26	23	24	25	25	23	25	24	24	25	25	24	25	25	23	23	23
NRF52-DK	**33**	**24**	**32**	**13**	–	23	23	23	13	23	21	21	21	23	23	21	24	23	23	21	25	25	20	23	23
CYBLE-416045	**29**	**37**	**25**	**30**	**30**	–	13	12	14	10	14	15	15	19	17	14	14	17	17	14	23	23	10	12	13
CY8CPROTO-063-BLE	**24**	**37**	**29**	**30**	**30**	**21**	–	12	16	12	14	14	15	19	17	14	14	14	16	14	17	17	10	14	15
STEVAL-IDB008V2	**27**	**37**	**26**	**30**	**30**	**16**	**15**	–		13	15	16	19	15	16	15	14	15	20	15	14	14	12	17	16
DT100112	**24**	**16**	**32**	**33**	**32**	**31**	**22**	**26**	–	16	9	10	14	15	9	13	16	9	14	19	16	16	19	18	14
ESP32-C3	**25**	**37**	**26**	**30**	**30**	**31**	**29**	**32**	**24**	–	16	19	18	23	14	15	18	25	42	16	23	15	16	10	13
Galaxy S6	**14**	**30**	**33**	**33**	**24**	**25**	**22**	**25**	**16**	**33**	–	12	16	16	14	9	10	12	15	23	25	16	15	24	23
Desire 10 Lifestyle	**12**	**30**	**30**	**35**	**24**	**25**	**23**	**27**	**11**	**28**	**22**	–	15	15	16	18	19	22	16	23	16	19	14	20	21
Galaxy S8+	**19**	**37**	**23**	**35**	**32**	**33**	**24**	**26**	**22**	**31**	**27**	**29**	–	9	12	16	15	16	18	17	18	17	24	19	25
Pixel 3XL	**24**	**39**	**23**	**35**	**32**	**33**	**28**	**23**	**26**	**27**	**27**	**29**	**26**	–	8	13	9	15	13	15	10	12	16	19	22
Pixel 4a	**25**	**37**	**26**	**30**	**29**	**29**	**28**	**23**	**26**	**27**	**25**	**27**	**24**	**15**	–	14	8	14	13	8	14	13	15	14	10
Y5 Prime	**16**	**30**	**33**	**33**	**24**	**25**	**22**	**25**	**18**	**25**	**28**	**29**	**28**	**26**	**32**	–	16	14	23	25	28	14	16	19	16
8X	**31**	**21**	**43**	**24**	**24**	**37**	**37**	**37**	**33**	**25**	**30**	**30**	**39**	**39**	**37**	**30**	–	19	15	18	14	16	13	14	19
Mi A1	**21**	**37**	**26**	**30**	**27**	**29**	**15**	**16**	**23**	**29**	**25**	**27**	**37**	**15**	**29**	**25**	**37**	–	10	14	13	14	12	10	14
iPhone XS	**21**	**37**	**26**	**30**	**29**	**30**	**15**	**21**	**25**	**29**	**25**	**27**	**33**	**33**	**29**	**25**	**37**	**29**	–	8	14	13	14	22	9
Galaxy A21	**14**	**30**	**33**	**33**	**24**	**25**	**22**	**25**	**16**	**14**	**31**	**25**	**14**	**31**	**25**	**25**	**14**	**25**	**25**	–	14	13	14	10	13
G Power	**23**	**35**	**26**	**30**	**23**	**29**	**15**	**16**	**26**	**26**	**24**	**27**	**26**	**33**	**31**	**27**	**30**	**26**	**29**	**25**	–	9	12	9	14
7T	**27**	**33**	**26**	**30**	**28**	**28**	**15**	**16**	**29**	**26**	**24**	**27**	**29**	**33**	**29**	**26**	**29**	**26**	**29**	**27**	**29**	–	9	8	10
OnePlus 8	**24**	**35**	**29**	**24**	**27**	**15**	**16**	**15**	**23**	**27**	**26**	**25**	**34**	**31**	**29**	**29**	**29**	**26**	**28**	**25**	**29**	**16**	–	8	12
Laptop (18.04)	**25**	**36**	**26**	**30**	**30**	**20**	**21**	**18**	**22**	**31**	**24**	**25**	**32**	**33**	**29**	**25**	**34**	**29**	**34**	**26**	**28**	**15**	**20**	–	14
Laptop (20.04)	**22**	**34**	**23**	**33**	**31**	**22**	**24**	**18**	**26**	**31**	**25**	**25**	**31**	**32**	**29**	**25**	**37**	**28**	**28**	**25**	**29**	**15**	**16**	**16**	–

TABLE X: Number of deviant issues comparison. Bold values are for BLEDiff and non-bold values are for DIKEUE

Development Boards			
Board	**Vendor**	**Sample Code**	**BLE Ver.**
DA14531	Dialog	ble_app_security	5.1
NRF52-DK	Nordic	ble_app_multirole_lesc	5
NRF5340-DK	Nordic	ble_app_multirole_lesc	5.2
CYBLE-416045-EVAL	Cypress	BLE_4.2_DataLength_Security_Privacy01	4.2
CY8CPROTO-063-BLE	Cypress	BLE_Pulse_Oximeter_Sensor	5.0
CC2640R2	Texas In.	simple_peripheral_app	5.0
STEVAL-IDB008V2	STM	security_peripheral	5.0
ESP32-C3	Espressif	ble_ancs	5.0
DT100112	Microchip	PIC_LightBlue_Explorer_Demo	4.2
Devices			
Device	**Vendor**	**OS/Stack**	**BLE Ver.**
Nexus 6	Motorola	Android 7.1.1	4.2
Galaxy S6	Samsung	Android 8.0	4.2
Desire 10 Lifestyle	HTC	Android 6.0	4.2
Galaxy S8+	Samsung	Android 9.0	5.0
Pixel 3 XL	Google	Android 11	5.0
Pixel 4a	Google	Android 11	5.0
Y5 Prime	Huawei	Android 8.1	4.2
8X	Honor	Android 8.1	4.2
Mi A1	Xiaomi	Android 9.0	4.2
iPhone XS	Apple	iOS 12	5.0
Galaxy A21	Samsung	Android 10	5.0
G Power	Motorola	Android 10	5.0
7T	OnePlus	Android 10	5.0
8	OnePlus	Android 12	5.1
Laptop	Lenovo	Ubuntu 18.04	Bluez 5.48
Laptop	Lenovo	Ubuntu 20.04	Bluez 5.53

TABLE XI: List of tested devices. Fluoride [43] and iOS-BLE-Stack [44] are the BLE stacks for Android and iPhone respectively

	Nexus6	DA14531	CC2640R2	NRF5340-DK	NRF52-DK	CYBLE-416045	CY8CPROTO-063-BLE	STEVAL-IDB008V2	DT100112	ESP32-C3	Galaxy S6	Desire 10 Lifestyle	Galaxy S8+	Pixel 3XL	Pixel 4a	Y5 Prime	8X	Mi A1	iPhone XS	Galaxy A21	G Power	7T	OnePlus 8	Laptop (18.04)	Laptop (20.04)
Nexus6	–	23.55	20.93	27.84	26.85	17.64	17.54	19.34	27.3	23.6	16.65	18.23	17.29	17.29	18.48	16.5	23.55	18.48	18.48	16.5	26.3	24.6	19.54	24.5	19.63
DA14531	**47.52**	–	27.86	46.97	45.67	34.42	33.42	35.62	24.53	23.63	24.4	25.86	35.10	35.10	34.09	24.4	19.8	34.09	34.09	24.4	37.53	24.63	34.52	26.64	24.54
CC2640R2	**36.95**	**45.7**	–	26.29	28.31	18.85	17.55	16.88	14.3	18.64	20.68	21.9	21.7	21.7	20.14	20.68	20.93	20.14	20.14	20.68	23.4	24.64	16.78	24.52	23.53
NRF5340-DK	**38.36**	**56.14**	**45.63**	–	21.20	32.29	33.21	35.24	25.53	24.52	29.3	29.54	32.68	32.68	31.89	29.3	27.84	31.89	31.89	29.3	26.55	35.43	31.19	24.2	29.65
NRF52-DK	**38.4**	**55.19**	**44.52**	**25.4**	–	31.8	31.93	36.42	24.53	26.64	25.43	26.35	32.8	32.8	31.06	25.43	26.85	31.06	31.06	25.43	24.64	23.53	32.8	35.64	29.65
CYBLE-416045	**38.17**	**47.62**	**20.41**	**41.91**	**40.71**	–	14.64	17.82	24.42	28.53	16.23	17.32	18.76	18.76	19.5	16.23	17.64	19.05	17.64	16.23	24.65	25.65	12.53	23.64	25.54
CY8CPROTO-063-BLE	**39.2**	**43.71**	**21.29**	**38.6**	**39.42**	**17.4**	–	16.49	25.54	25.42	18.49	19.32	19.8	19.8	20.2	18.49	17.54	20.2	20.2	18.49	23.54	24.64	11.49	37.65	36.64
STEVAL-IDB008V2	**39.42**	**45.62**	**21.42**	**41.94**	**40.67**	**22.64**	**23.69**	–	26.75	24.54	19.42	21.3	25.52	24.54	27.54	19.42	19.34	28.4	35.3	19.42	25.2	19.87	16.82	25.53	26.64
DT100112	**40.72**	**54.56**	**56.78**	**53.74**	**52.39**	**36.93**	**37.34**	**48.43**	–	26.39	14.67	16.8	23.5	24.6	26.5	28.74	29.76	26.52	23.65	26.74	25.63	26.4	25.43	25.47	28.39
ESP32-C3	**20.25**	**39.52**	**45.29**	**52.2**	**59.2**	**40.52**	**51.29**	**52.6**	**59.6**	–	24.52	35.52	36.24	25.67	24.26	24.63	24.25	29.55	31.52	25.63	25.62	36.73	29.62	23.52	29.63
Galaxy S6	**21.98**	**49.72**	**35.25**	**37.83**	**39.9**	**39.34**	**36.21**	**39.2**	**34.63**	**45.64**	–	21.63	18.76	18.76	19.49	22.65	31.52	19.49	19.49	29.63	26.63	29.63	18.62	29.53	30.63
Desire 10 Lifestyle	**22.64**	**45.64**	**37.13**	**36.74**	**38.12**	**37.32**	**35.14**	**37.9**	**35.64**	**35.52**	**22.42**	–	20.23	20.23	23.63	24.63	16.4	19.6	22.52	22.5	35.7	46.7	22.6	42.6	36.5
Galaxy S8+	**36.62**	**56.64**	**25.65**	**41.32**	**41.54**	**29.16**	**27.61**	**52.5**	**53.64**	**53.64**	**36.52**	**34.43**	–	32.53	21.63	25.74	31.42	32.74	36.53	28.63	24.54	29.53	24.64	28.46	46.64
Pixel 3XL	**35.69**	**57.64**	**35.67**	**42.37**	**40.6**	**29.54**	**29.66**	**32.3**	**53.44**	**55.53**	**39.4**	**34.56**	**46.3**	–	42.35	26.53	27.63	27.65	29.63	29.53	23.53	15.53	37.53	35.65	29.56
Pixel 4a	**35.54**	**53.74**	**24.32**	**40.38**	**42.57**	**27.27**	**28.24**	**40.53**	**39.67**		**34.43**	**36.52**	**45.53**	**42.63**	–	24.32	25.53	36.52	37.74	25.63	24.63	24.63	22.9	37.63	35.63
Y5 Prime	**21.98**	**49.72**	**35.25**	**37.83**	**39.9**	**39.34**	**36.21**	**39.2**	**39.52**	**24.53**	**52.4**	**53.6**	**49.63**	**36.22**	**52.5**	–	42.1	24.52	23.56	32.63	24.42	36.52	24.58	36.42	36.42
8X	**45.34**	**22.9**	**45.9**	**56.14**	**55.19**	**47.62**	**43.71**	**45.62**	**42.52**	**42.63**	**49.72**	**45.64**	**56.64**	**53.74**	**49.72**	**22.9**	–	15.5	26.4	39.54	26.63	29.67	26.64	26.53	29.63
Mi A1	**35.54**	**53.74**	**24.32**	**40.38**	**42.57**	**27.27**	**28.24**	**27.65**	**29.63**	**31.64**	**34.43**	**36.52**	**33.63**	**45.53**	**52.66**	**52.63**	**63.4**	–	12.46	18.64	16.63	15.74	25.74	35.7	29.43
iPhone XS	**31.02**	**52.19**	**28.64**	**42.39**	**44.59**	**29.36**	**29.62**	**48.46**	**45.64**	**57.74**	**37.34**	**36.55**	**39.63**	**46.46**	**47.63**	**42.63**	**39.76**	**40.74**	–	19.63	20.52	31.73	29.63	36.52	29.74
Galaxy A21	**21.98**	**49.72**	**35.25**	**37.83**	**39.9**	**39.34**	**36.21**	**39.2**	**47.57**	**45.63**	**62.63**	**45.63**	**56.46**	**56.46**	**45.65**	**63.64**	**54.53**	**39.64**	**29.64**	–	22.64	27.74	29.53	30.32	43.53
G Power	**38.62**	**59.29**	**43.65**	**50.4**	**39.03**	**48.69**	**25.42**	**27.1**	**36.84**	**40.29**	**44.52**	**44.89**	**54.43**	**52.51**	**44.42**	**43.27**	**29.73**	**46.42**	**53.12**	**52.1**	–	15.03	20.148	15.12	23.85
7T	**44.28**	**54.3**	**42.77**	**48.3**	**46.06**	**47.02**	**24.75**	**26.35**	**41.52**	**44.122**	**39.52**	**44.42**	**48.691**	**56.01**	**49.01**	**42.64**	**47.77**	**42.87**	**49.126**	**45.75**	**49.123**	–	15.22	13.36	16.69
OnePlus 8	**39.51**	**57.59**	**47.966**	**39.48**	**46.12**	**25.05**	**26.8**	**24.84**	**42.1**	**44.45**	**43.09**	**36.14**	**55.93**	**50.9**	**47.70**	**47.71**	**42.79**	**41.41**	**47.72**	**45.62**	**43.42**	**26.32**	–	13.16	19.74
Laptop (18.04)	**40.25**	**58.32**	**42.64**	**49.5**	**48.69**	**32.6**	**34.34**	**29.41**	**37.18**	**51.53**	**39.68**	**41.33**	**53.01**	**54.66**	**47.72**	**33.67**	**55.96**	**47.74**	**56.53**	**42.9**	**46.39**	**24.99**	**33.12**	–	23.04
Laptop (20.04)	**35.2**	**54.74**	**37.26**	**54.12**	**51.15**	**36.74**	**40.56**	**30.42**	**42.42**	**49.6**	**40.25**	**40.75**	**50.84**	**52.8**	**47.56**	**42.25**	**59.2**	**45.08**	**45.36**	**42.25**	**46.4**	**25.35**	**25.76**	**27.04**	–

TABLE XII: Timing comparison. Bold = BLEDiff, non-bold = DIKEUE