A SYSTEMATIC FRAMEWORK FOR ANALYZING THE SECURITY AND PRIVACY OF WIRELESS COMMUNICATION PROTOCOL IMPLEMENTATIONS

by

Imtiaz Karim

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

May 2023

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Elisa Bertino, Chair

Department of Computer Science

Dr. Ninghui Li

Department of Computer Science

Dr. Dave Tian

Department of Computer Science

Dr. Sonia Fahmy

Department of Computer Science

Approved by:

Dr. Kihong Park

To my family, who have always supported and encouraged me throughout my journey.

ACKNOWLEDGMENTS

I would like to express my deep gratitude to the following individuals who have supported me throughout my journey toward completing this thesis:

First and foremost, I would like to thank my advisor, Dr. Elisa Bertino, for her invaluable guidance, expertise, and encouragement. I would also like to thank my mentor, Dr. Syed Rafiul Hussain for his constant support and guidance. Their unwavering commitment to excellence and dedication to students have inspired me to strive for nothing less than the best. Furthermore, I would like to thank all my co-authors: Dr. Ninghui Li, Dr. Omar Chowdhury, Dr. Hyunwoo Lee, Fabrizio Cicala, Abdullah Al Ishtiaq, Mirza Masfiqur Mim, and Kazi Samin Mubasshir who have helped me with their expertise.

I am also grateful to the members of my thesis committee for their constructive feedback and insightful comments. Their expertise and knowledge have helped me to refine my ideas and to ensure that my work meets the highest standards.

I am indebted to my Intel and Amazon collaborators: Jason Fung, Dr. Arun Kanuparthi, Dr. Sayak Ray, Dr. Stephan Heuser, Dr. Vaibhav Sharma, and Dr. Saswat Padhi for providing me with great learning opportunities during the summer internships.

I would also like to thank the faculty and staff of the Department of Computer Science, at Purdue University for providing me with an outstanding academic environment, resources, and opportunities to pursue my research interests.

I am deeply grateful to my family for their love, support, and encouragement, especially during the challenging times of my graduate studies. Their unwavering support and belief in me have been my constant source of strength and motivation.

Last but not least, I would like to express my sincere gratitude to all my friends and colleagues who have supported me, shared their knowledge, and made my journey toward the Ph.D. degree a memorable and enriching experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Wireless communication technologies, such as cellular ones, Bluetooth, and WiFi, are fundamental for today's and tomorrow's communication infrastructure. Networks based on those technologies are or will be increasingly deployed in many critical domains, such as critical infrastructures, smart cities, healthcare, and industrial environments. Protecting wireless networks against attacks and privacy breaches is thus critical. A fundamental step for the security and privacy of these networks is ensuring that their protocols are implemented as mandated by the standards. These protocols are however quite complex and unfortunately, the lack of secure-by-design approaches for these complex protocols often induces vulnerabilities in implementations with severe security and privacy repercussions. For these protocols, the standards are thousands of pages long, written in natural language, describe the high-level interaction of the protocol entities, and most often depend on human interpretation—which is open to misunderstanding and ambiguity. This inherently entails the question of whether these wireless protocols and their communication equipment implement the corresponding standards correctly or whether the implementations introduce vulnerabilities that can have severe consequences.

In this dissertation, we systematically analyze the security and privacy of these wireless communication protocol implementations (e.g., cellular networks and Bluetooth) and develop techniques useful for both white-box and black-box analysis. For the white-box analysis, we propose a model-based testing approach–ProChecker which (1) extracts a precise semantic model as a finite state machine of the implementation by combining dynamic testing with static instrumentation, and (2) verifies the properties against the extracted model by combining a symbolic model checker and a cryptographic protocol verifier. We demonstrate the effectiveness of ProChecker by evaluating it on a closed source and two of the most popular open-source 4G LTE control plane protocol implementations with 62 properties. ProChecker unveiled 3 new protocol-specific logical attacks, 6 implementation issues, and detected 14 prior attacks. The impact of the attacks ranges from denial-of-service, broken integrity, encryption, and replay protection to privacy leakage.

For the first black-box testing approach, we develop an automated, stateful noncompliance checker–DIKEUE that can check non-compliance between different cellular device implementations. DIKEUE adopts a property-agnostic, differential testing approach, which leverages the existence of many different control-plane protocol implementations in COTS devices. For deviant behavior identification, DIKEUE first uses black-box automata learning, specialized for 4G LTE control-plane protocols, to extract input-output finite state machine (FSM) for a given UE. It then reduces the identification of deviant behavior in two extracted FSMs as a model-checking problem. We applied DIKEUE in checking noncompliance in 14 COTS UEs from 5 vendors and identified 15 new deviant behavior as well as 2 previous implementation issues. Among them, 11 are exploitable whereas 3 can cause potential interoperability issues.

Lastly, 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.

1. INTRODUCTION

Wireless communication protocols and networks such as cellular networks, Bluetooth, and WiFi are critical infrastructures. These are high-speed, high-capacity voice and data communication networks with enhanced multimedia and seamless roaming capabilities for supporting numerous devices. They are not any longer used for just phone calls and entertainment. They have become the primary communication means for finance-sensitive business transactions, lifesaving emergencies, and life-/ mission-critical services.

Recent versions of these technologies, such as the 5G New Radio (NR) for cellular networks, are further enhancing the transmission speed and capacity, as well as, reducing latency through the use of different radio technologies and are expected to provide Internet connections that are an order of magnitude faster than previous generations, such as 4G Long Term Evolution (LTE)—the previous generation of cellular networks. As a result, future generations of these protocols and networks will be able to provide ubiquitous connectivity, interoperability, and massive-scale support to numerous network services and billions of heterogeneous devices.

However, because of their ubiquitous presence, use for critical applications (e.g., emergency alert system [1]), and in low-energy communication, wireless networks are an attractive attack target for malicious parties. Furthermore, we can expect that attacks on these networks will no longer be limited to simple (albeit significantly harmful) discrete events [2], such as a distributed denial-of-service attack against a portion of a network. Rather we can expect stealthy, persistent, and sophisticated activities aiming at establishing a foothold in core networks and maintaining such a foothold to carry out massive disruption operations or sophisticated data-gathering operations. For instance, resourceful adversaries such as nation-states, foreign intelligence agencies, and terrorists can rely on an ingenious range of attack strategies and wreak havoc by exploiting vulnerabilities of the cellular ecosystem (e.g., cyberwarfare [3] and surveillance [4]). With the increasing adoption of wireless-enabled smart devices [5] and systems such as autonomous vehicles, and autonomous healthcare which reside in individuals' personal space, the potential of such attacks is increasing. Therefore, ensuring the security of the critical wireless networks and the privacy of the users is critical.

Lack of adequate protection, may result in huge monetary, and strategic advantages, and even human life losses.

1.1 Implementations of Wireless Communication Protocols

An important building block for wireless network security and privacy is ensuring that the communication protocols deployed in these networks be implemented as mandated by the standards. Unfortunately, the lack of secure-by-design approaches for these complex protocols often induces vulnerabilities in implementations with severe security and privacy repercussions. While memory corruption vulnerabilities (e.g., buffer overflows, use-after-free) can be detected without prior knowledge about the protocol, utilizing memory sanitization techniques [6], detecting logical vulnerabilities (e.g., resetting the counter to break the replay protection of protocol messages), or bypassing key establishment procedure (*aka.,* pairing procedure) and accepting messages encrypted with the default key [7] in large and complex protocol implementations is challenging since logical vulnerabilities do not have externally discernible effects such as crashes or memory leaks. Instead, they require an in-depth semantic understanding of the protocol interactions and are thus primarily detected through manual analysis. For these complex protocols, the standards are thousands of pages long, written in natural language, describe the high-level interaction of the protocol entities, and most often depend on human interpretation—which is open to misunderstanding and ambiguity. This inherently entails the question of whether the protocol devices implement the corresponding standards correctly or whether implementations introduce vulnerabilities that can be exploited by attackers. For instance, under specification, ambiguity, and implementation mismatch introduced a server vulnerability making it possible to completely bypass the authentication of very popular smartphones [8]. It is hence pivotal to not only design and improve secure protocols but also ensure that the implementations of such complex protocols comply with the specification, and the security and privacy requirements.

1.2 Challenges in Analyzing Wireless Communication Protocol Implementations

Developing methodologies to evaluate the security and privacy of wireless communication protocol implementations is challenging because it requires addressing several challenges: (i) *Lack of formal specification:* The protocols do not have formal specifications. The specifications are written in natural language, and thus have ambiguities and underspecifications [9–14]. Furthermore, intricate protocol details written in natural languages cause misinterpretations while developing the implementations; (ii) *Lack of formal implementation:* There is no formal implementations to follow. Developers are free to design and implement the protocol in their own way with the only requirement of matching input/output behavior. Thus implementations of internal protocol structures most often deviate from the standards; (iii) *Protocol complexity:* The protocols are complex and stateful. The protocol implementations are large and therefore existing static or dynamic testing approaches face scalability issues when applied to these implementations.

1.3 Existing Efforts

Although prior works [7, 15–30] focusing on the analysis of security, privacy, and noncompliance of wireless communication protocols, such as cellular protocols and Bluetooth, have identified several implementation flaws, they suffer from at least one of the following limitations: (A) The approaches [8, 15–17, 23–27, 30] are completely manual and cannot uncover a myriad of *implementation-specific* behaviors; (B) The analyses [8] perform semi-automated stateless testing; (C) The approaches use fuzzing [7, 31] through a hand-crafted bug oracle or reference state machine; (D) The approaches based on formal verification [20, 30, 32–34] only test the protocol specification for noncompliance and also heavily rely on the coverage and quality of the properties being tested—for which there is no official exhaustive list; and (E) The analyses based on re-hosting and reverse-engineering the baseband software [21, 22, 28, 29] not only require a huge manual effort and expertise but also are not general enough to be applicable to implementations from different vendors.

1.4 Dissertation Focus

It is evident that the state-of-the-art systematic analysis frameworks and vulnerability detection techniques are inadequate for the analysis of complex communication protocol implementations. Therefore, the focus of the dissertation is to provide a framework to analyze these large, stateful, and complex protocol implementations. In our framework we tackle the analysis in two broad directions: (i) *white-box setup:* analyzing the network implementations when the source-code of the implementation is available; (ii) *black-box setup:* analyzing implementations when the source code of the implementation is not available. Following these broad directions, we first address the research question: "*Is it possible to evaluate the security and privacy of a stateful communication protocol implementation to uncover logical vulnerabilities, when the source-code of the implementation is available?*"

While a white-box approach to evaluate the security and privacy properties provides assurance for the implementation, in most cases, protocol implementations are black-box, proprietary, and closed systems which necessitates the development of black-box and system-agnostic testing approaches. In this research, therefore, we develop analysis tools in the black-box setup where the implementation is not available. One of the common causes of vulnerabilities in implementations is when implementations deviate from the standards and becomes noncompliant. The ramifications of noncompliance with respect to the standard may result in: (1) critical security and privacy flaws (e.g., authentication bypass [7, 8], location exposure of a target user [25]), and (2) interoperability issues in the devices. Since manual identification of noncompliant protocol behavior in large and complex implementations is error-prone and time-consuming we tackle the second research question: *Is it possible to design an automated, black-box, and scalable protocol analysis framework that can uncover noncompliant behavior in the protocol implementations in stateful wireless communication protocols?*

We address these research questions and develop tools from the perspective of cellular networks and Bluetooth but our proposed techniques are general enough to adapt for any communication protocols, e.g., WiFi, Voice Over WiFi (VoWiFI), cellular IoT, etc.

1.5 Thesis Statement

In this thesis, we demonstrate that: (i) it is possible to extract scalable formal models from implementations leveraging static instrumentation and dynamic testing; furthermore, these models can be used to reason about security and privacy properties of large scale implementations; (ii) it is possible to detect noncompliance of implementations without the need of a formal model of the specification, by leveraging the idea that in case two implementations deviate from each other then at least one of them is deviating from the standards; and (iii) it is possible to improve the scalability of active automata learning for the large and complex protocols by using the idea of using a divide-and-conquer approach.

1.6 Contributions

In this thesis, we present our research addressing the research questions. We make the following contributions.

1.6.1 An Automated Security and Privacy Analysis Framework for Cellular Network Implementations

We investigate the security and privacy of a commercial and two open-source cellular network protocol implementations and uncover 6 implementation issues and 3 new protocol specific logical attacks that are true for all the implementations. For the analysis we propose a model-based testing approach–ProChecker which (1) extracts a precise semantic model as a finite state machine of the implementation by combining dynamic testing with static instrumentation, and (2) verifies the properties against the extracted model by combining a symbolic model checker and a cryptographic protocol verifier. For model extraction of commercial 4G LTE implementations, instead of creating a separate framework for security and privacy analysis, we capitalize on the *functional conformance* testing frameworks developed by protocol standardization bodies and/-or commercial test-case developers. We deploy a code instrumentation mechanism that automatically instruments the code and then utilizes the conformance testing framework to generate a detailed log with rich metadata. Based on such metadata, we design a model extraction algorithm that constructs the FSM of the

protocol implementation. Such an approach can be easily integrated to the mainstream functional testing framework to uncover logical vulnerabilities.

1.6.2 An Automated Black-box Noncompliance Checker for Cellular Network Implementations

We develop an automated, stateful noncompliance checker–DIKEUE that can check noncompliance between different cellular device implementations. DIKEUE adopts a property-agnostic, differential testing approach, which leverages the existence of many different control-plane protocol implementations in COTS devices. DIKEUE uses deviant behavior observed during differential analysis of pairwise COTS devices as a proxy for identifying noncompliance instances. For deviant behavior identification, DIKEUE first uses black-box automata learning, specialized for 4G LTE control-plane protocols, to extract input-output finite state machine (FSM) for a given device. It then reduces the identification of deviant behavior in two extracted FSMs as a model checking problem. We applies DIKEUE in checking noncompliance in 14 COTS UEs from 5 vendors and identified 15 new deviant behavior as well as 2 previous implementation issues. Among them 11 are exploitable whereas 3 can cause potential interoperability issues

1.6.3 Scalable and Property-Agnostic Noncompliance Checking for BLE Implementations

We develop an automated, scalable, property-agnostic, and black-box protocol noncompliance checking framework–BLEDiff that can analyze and uncover noncompliant behavior in the Bluetooth Low Energy (BLE) protocol implementations. To overcome the huge manual effort for 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 that it can uncover 13 different deviant behaviors with 10 exploitable attacks.

1.6.4 Dissertation Outline

The remainder of the thesis is organized as follows: Chapter 2 provides a background on some of the most popular wireless communication networks, e.g., cellular networks and Bluetooth. Chapter 3 presents ProChecker, a model-based testing approach for analyzing the security and privacy properties of cellular network implementations. Chapter 4 presents DIKEUE, an automated, stateful, and property-agnostic non-compliance checker for cellular-enabled 4G LTE devices. Chapter 5 presents BLEDiff, an automated, scalable, and stateful non-compliance checker for Bluetooth. Chapter 6 discusses the state-of-the-art work. Finally, Chapter 7 outlines concluding remarks and future works.

2. BACKGROUND

In this chapter, we provide a brief introduction to the cellular network protocol, specifically the 4G LTE protocol. We first discuss the simplified 4G LTE architecture (shown in Figure 2.1) and then the relevant procedures of the protocol. Following this we provide a introduction to the Bluetooth Low Energy (BLE) protocol. We refer the interested readers to the specifications [9–14, 35] to explore further details.

2.1 4G LTE Network Architecture

The 4G LTE network is broadly comprised of three components: (i) the cellular device (also known as User Equipment or UE); (ii) the radio access network (E-UTRAN); (iii) the core network or Evolved Packet Core (EPC)(shown in Figure 2.1).

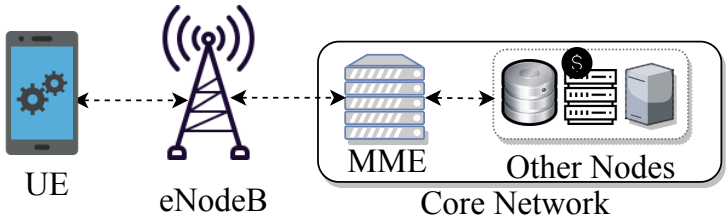


Figure 2.1. The simplified 4G LTE Network Architecture

User Equipment (UE). The UE, also called cellular device, is the user's access terminal, in most cases, a smartphone. The User Services Identity Module (USIM) stores the user identifier, the master secret key, and shared session keys. With these credentials, the user and the network performs mutual authentication. Note that we use the terms UE, device, and cellular device interchangeably in the thesis. Usually, a baseband modem inside the smartphone's system-on-chip processes all LTE-specific functionality.

E-UTRAN. A geographical area, in the context of LTE is partitioned into hexagonal cells where each cell is serviced by a single base station. The base stations, i.e., eNodeBs span the wireless cells that users connect to. An eNodeB performs all connection management through the Radio Resource Control (RRC) protocol with a UE. The UE first connects to a base station with radio connections, which forwards all user data to the core network.

Core network and MME. The operator-run core network is a server landscape that performs all management aspects of mobile networks. The Mobility Management Entity (MME) is the central component managing users access, mutual authentication, and keeping track of a user's location. Most of these functions involve many other network nodes; however, the MME orchestrates them. UE and MME communicate through Non-Access Stratum (NAS) protocol with the eNodeB as a relay. The MME is connected to eNodeBs through the S1AP protocol (shown in Figure 2.2).

2.2 Protocol Overview

When a UE is turned on, it first connects with a base station with three-way RRC layer handshaking messages. This connection allows a UE to initiate the attach procedure with the core network in which the UE and the MME mutually authenticate each other, negotiate security algorithms for both NAS and RRC layers, and complete the attach process with IP address and a temporary identifier assigned to the UE. The UE in idle-mode is notified of incoming services with paging messages, whereas a UE requesting a cellular service or moving to a different tracking area due to handover sends corresponding initiating messages to the MME. We discuss in detail the relevant NAS and RRC layer procedures in the following sections.

2.2.1 NAS Layer Procedures

We now briefly discuss the NAS layer procedures that are most relevant in the context of our paper (shown in Figure 2.2, the NAS layer procedures are shown in black).

Initial attach. After rebooting, the UE performs a radio setup procedure. After the radio setup the UE establishes communication through the RRC layer following the RRC Connection Setup. The UE starts the NAS attach procedure by sending the *attach_request* message. After successful authentication through *auth_request* and *auth_response* messages, the MME moves towards the negotiation of ciphering and integrity algorithms through the security mode command procedure. At this point, the NAS level security context is established between the UE and MME, and the selected encryption and integrity protection algorithms

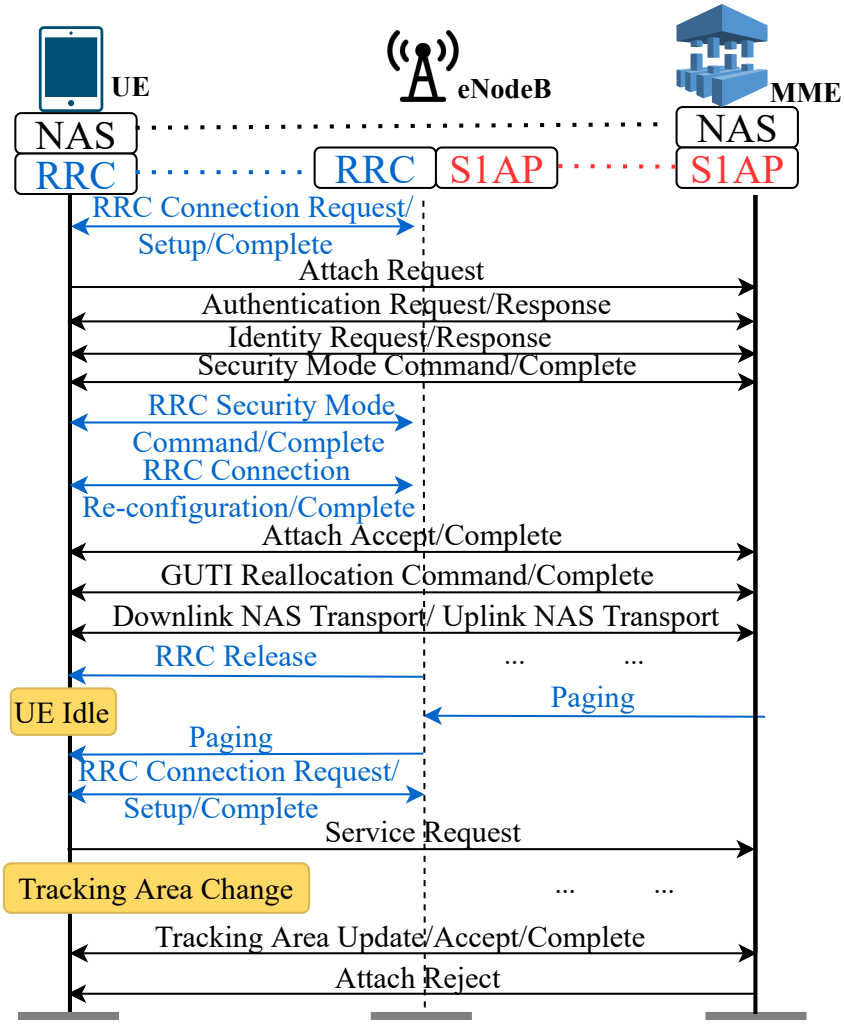


Figure 2.2. LTE control plane procedures. NAS and RRC layer procedures are shown in black and blue, respectively.

will be applied to subsequent NAS massages. The MME concludes the attach procedure by sending *attach_accept* message with a Globally Unique Temporary Identity (GUTI) and the UE responds *attach_complete* message.In case the attach cannot be accepted by the network, the MME shall send an *attach_reject* message to the UE including an appropriate cause value.

Identification procedure. The procedure is used to know the identity, in most cases, International Mobile Subscriber Identity (IMSI) of the device.

Detach procedure. To disconnect from the network, the UE can initiate a detach procedure by sending a *detach_request* to which the network is expected to respond with a *detach_accept*.

Service procedure. The UE invokes this procedure when it recieves a *paging* request from the network or the UE has pending uplink data. The UE initiates this procedure by sending the *service_request* message to the MME.

GUTI reallocation procedure. The GUTI reallocation procedure is used by the MME to reallocate a new GUTI to the UE. The procedure is started by the MME through sending a *GUTI_reallocation* and the UE acknowledges with a *GUTI_reallocation_complete*.

Tracking area update. The tracking area update procedure is a standalone procedure that occurs either when the UE detects a new tracking area (TA) or a periodic TA update timer has expired. The tracking area update procedure can also be triggered if the RRC connection is released with cause "load re-balacing TAU required".

Downlink NAS transport. Through this procedure the network can send an actual SMS message in the NAS message. The procedure start with the network sending a *DL_NAS_transport* message, the UE acknowledges with *UL_NAS_transport*.

2.2.2 RRC layer procedures

We now briefly discuss the RRC layer procedures that are most relevant in the context of this thesis (shown in Figure 2.2, the RRC layer procedures are shown in blue).

RRC setup. RRC setup procedure is the backdrop of the NAS attach procedure. The purpose of this procedure is to establish an RRC connection and to transfer the initial NAS dedicated information message from the UE to the network.

RRC security activation. RRC layer security is established through the RRC security activation procedure. The procedure is started through the *RRC_sm_command* message from the eNodeB and completed by the *RRC_sm_complete* message by the UE.

RRC release. This procedure is used by the network to release the established radio bearers as well as all radio resources to suspend the RRC connection.

RRC connection reconfiguration. The purpose of this procedure is to modify an RRC connection, e.g., to establish/modify/release radio bearers. As part of the procedure, dedi-

cated NAS information may be transferred from the network to the UE. Usually, after this RRC procedure the UE completes the initial attach. To begin this procedure, the network sends an *RRC_reconf* message which the UE replies with *RRC_reconf_complete* to complete the procedure.

RRC Connection Re-establishment. A UE in RRC Connected state, for which security has been activated, may initiate the procedure in order to continue the RRC connection. The procedure initiates from the UE with *RRC_con_reest_req* and completes with *RRC_con_reest*, and *RRC_con_reest_complete* messages.

2.3 Bluetooth Low Energy (BLE)

In this section, we discuss the other importat communication protocol we analyze in this thesis: Bluetooth. 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).

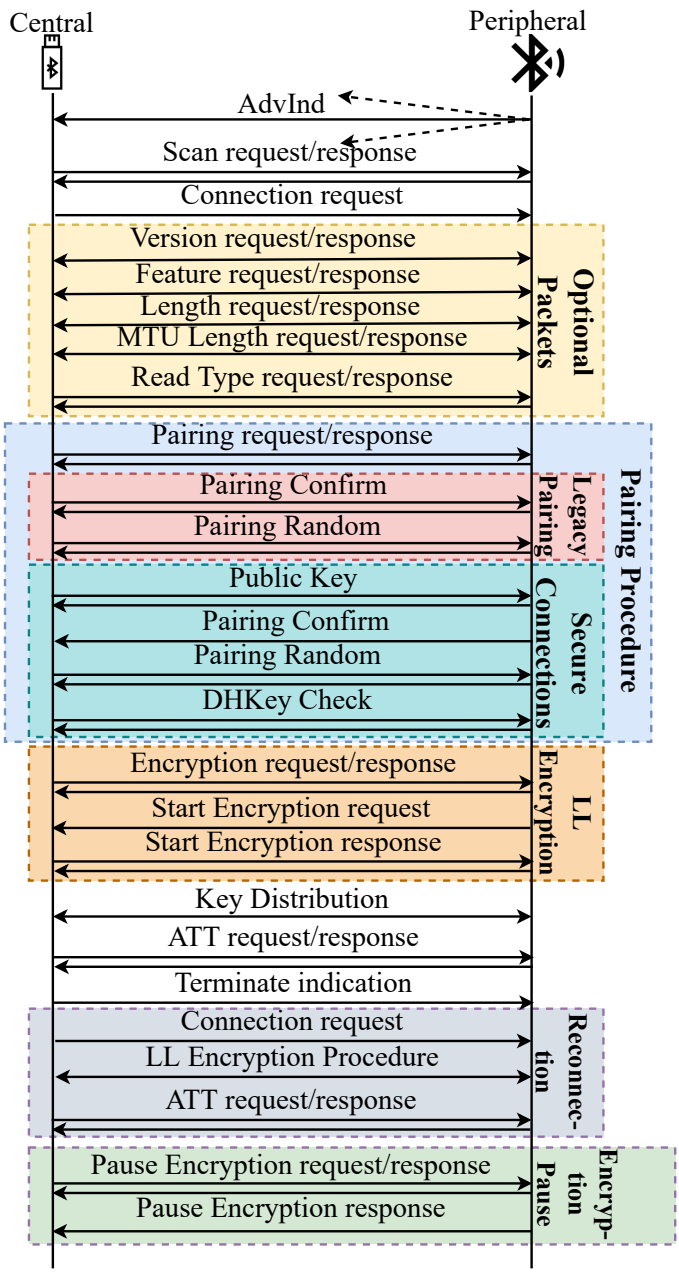


Figure 2.3. BLE procedures

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 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 2.3, where packet sequences and directions are available as well.

3. PROCHEKCER: AN AUTOMATED SECURITY AND PRIVACY ANALYSIS FRAMEWORK FOR CELLULAR PROTOCOL IMPLEMENTATIONS

Implementations of cellular network protocols, such as 4G LTE and 5G NR, must adhere to the specified security and privacy requirements. Unfortunately, lack of secure-by-design approaches for these complex protocols often induces vulnerabilities in implementations with security and privacy repercussions. While memory corruption vulnerabilities (e.g., buffer overflows, use-after-free) can be detected without prior knowledge about the protocol utilizing memory sanitization techniques [6], detecting logical vulnerabilities (e.g., resetting the counter to break the replay protection of protocol messages) in large and complex protocol implementations is challenging since logical vulnerabilities do not have externally-discernible effects such as crashes or memory leaks. Instead, they require an in-depth semantic understanding of the protocol interactions and are thus primarily detected through manual analysis.

Problem. Recent work has demonstrated the effectiveness of formal verification in identifying logical vulnerabilities in 4G LTE [32] and 5G NR [34] protocols. Most of these proposals, however, primarily focus on developing a *standalone* security and privacy analysis framework for verifying *specifications* of protocols on a manually constructed simplified model, which is hardly an option for *commercial-scale* complex implementations. On detecting logical flaws of 4G LTE protocol implementations, previous approaches [8, 23–27] have one or more limitations: (A) The analysis [23–27] is completely manual; (B) The analysis [8] performs stateless semi-automatic dynamic testing of the implementation but requires significant manual analysis and can only test few pre-defined properties. Even though such manual or semi-automated security analyses are effective to some extent, from a commercial vendor's point-of-view the use of different test infrastructures for separate functional and security testing is often expensive and leaves security testing at a low priority. To address these challenges, this work aims at answering the following research question: *Is it possible to evaluate the security and privacy properties of a commercial-scale 4G LTE protocol im-*

plementation and integrate the evaluation with the mainstream functional testing framework to uncover logical vulnerabilities?

Challenge. Prior work [32, 34, 36, 37] evaluating the design of cellular network protocols represents the high-level protocol interactions with finite state machines (FSMs) and evaluates the FSMs against desired security and privacy properties. Such approaches can also be naturally applied to the FSM's of 4G LTE protocol implementations. One major challenge in applying such model checking based formal verification to protocol implementations is, however, the automatic extraction of the FSM from the implementation. It is critical that the extracted model (represented by a FSM) is in bounds for the state-of-the-art model checking tools, contains semantic meaning, and is explicit enough to allow one to identify logical vulnerabilities. However, due to under-specifications in the standards, developers are free to design and implement some part of the protocol in their own way— with the only requirement of matching input/output behavior. Thus implementations of internal protocols structure most often deviate from the standards. This necessitates a sophisticated and automated model extraction technique to reverse-engineer a model from the implementation to properly verify properties on protocol implementations.

Plausible approaches. Conceptually, one can extract the model using one of the following two broad approaches: (1) static analysis, and (2) dynamic analysis. For a typical industrial implementation with pointers and function redirections, static analysis techniques are unable to meet the precision required to reason about both implementation soundness [38] and completeness. On the contrary, though dynamic analysis would appear to be effective because of its high precision, it fails to scale for production-level and large-size implementations, when executing all feasible paths and suffers from state space explosion. Nonetheless, existing popular dynamic extraction techniques such as active-automata learning [39, 40] are used to extract FSM's of the implementations of other protocols e.g., TLS, SSH in a black-box setting. However, such approaches are prohibitively expensive as they require a significantly high time and number of queries to infer the target implementation's FSM. Moreover, the inferred FSM is not sufficiently large and semantically rich compared to that of the white-box settings. For the FSM extraction, our goal is, therefore, to achieve the accuracy of dynamic

analysis without falling into state explosion [6] and utilize the white-box information to create a semantically rich model.

Our approach. We propose an automated white-box framework, ProChecker, that allows developers to check whether a 4G LTE protocol implementation violates the desired security and privacy guarantees. The violations can either mean the implementation deviates from the standards, the protocol is underspecified or the vulnerability is in the protocol design. ProChecker works with two major components: (1) *model extraction*, and (2) *model checking*.

For model extraction of commercial 4G LTE implementations, instead of creating a separate framework for security and privacy analysis, we capitalize on the *functional conformance testing* frameworks developed by protocol standardization bodies and/or commercial test-case developers. We deploy a code instrumentation mechanism that automatically instruments the code and then utilizes the conformance testing framework to generate a detailed log with rich metadata. Based on such metadata, we designed a model extraction algorithm that constructs the FSM of the protocol implementation.

For model checking, like LTEInspector [32], we combine the reasoning powers of the symbolic model checker and a cryptographic protocol verifier to detect logical vulnerabilities that adhere to the cryptographic constructs of the protocol. The reason behind combining the model checker and cryptographic protocol verifier is to: (i) efficiently capture all the desired properties that we have observed; (ii) reason about rich temporal properties (e.g., safety, liveliness, correspondence) that could not be captured if one of them is solely used.

Implementation. We evaluate the effectiveness of ProChecker on a closed-source and two open-source (srsLTE [41] and OpenAirInterface [42]) 4G LTE implementations. We instantiate the model checking component of ProChecker with the nuXmv infinite-state model checker [43] and the ProVerif cryptographic protocol verifier [44]. For properties, we use the conformance test suite [9] suggested by the standard along with the properties which are implicit in the standard. The key properties and insights leveraged by ProChecker and the major procedures discussed here remain unchanged in the upcoming 5G deployment, making our framework directly applicable to 5G and securing upcoming generations.

Contributions. This work makes the following contributions:

- We propose ProChecker, a framework for property-guided formal verification of commercial 4G LTE implementations.
- We design a novel model extraction tool as part of the framework. It is scalable and leverages the functional testing infrastructure (inherent to commercial products) to extract a detailed formal model, e.g., a FSM, from the commercial and complex codebase. This FSM can also be used to enhance testing by detecting missing test cases.
- We evaluate ProChecker by implementing and integrating it into the existing functional testing framework of a closed-source and two open-source LTE implementations and analyze their implementations. We evaluate our extracted models against 62 properties. Along with uncovering 3 new protocol-specific logical attacks, 6 implementation issues, ProChecker identified 14 prior attacks in the FSM's derived from implementations. The issues range from denial-of-service attacks, broken integrity, encryption, and, replay protection to severe privacy leakage.

Responsible Disclosure. We have reported the protocol vulnerability findings of ProChecker to GSMA through the coordinated vulnerability disclosure (CVD) program and are actively coordinating with GSMA regarding the issues. The CVD submission (CVD-20201-0043) has been awarded Mobile Security Hall of Fame status by GSMA [45]. We have also reported implementation issues to open-source 4G LTE protocol stack developers [41, 42].

3.1 Background

We introduce logical vulnerabilities, and elaborate the key properties of cellular network protocols leveraged by ProChecker.

3.1.1 Logical Vulnerabilities

Logical vulnerabilities are issues that force the protocol to deviate from (i.e., yield a trace that violates) basic security (confidentiality, integrity, availability) and privacy guarantees without having an externally-discernible effect such as crash or memory corruption. The de-

viations can be attributed to protocol level design-flaws, underspecifications in the standards, and implementation mismatch. For instance, underspecifications and inadequate checks in replay protection induce logical vulnerabilities in 4G/5G protocols enabling an adversary to force a user to use the same session keys [23] (also known as *key-reinstallation* attack) and reset the replay protection counters [34]. Note that all these previously uncovered issues have been identified manually or from manually derived models.

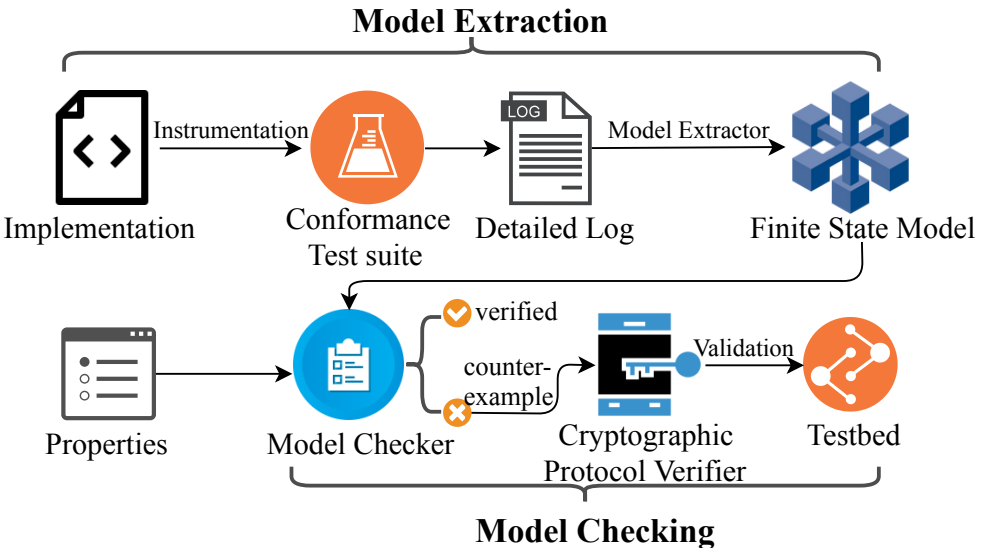


Figure 3.1. Architecture of ProChecker

3.1.2 Properties of LTE Protocol Implementation

We now briefly discuss common properties of 4G LTE implementations that ProChecker leverages to extract a FSM of a given implementation. The properties are identified by analyzing sample protocol implementations followed by commercial and most open-source protocol implementations.

Event-driven communication architecture. 4G LTE follows an event-driven communication paradigm. For instance, whenever a protocol entity receives a message, it reciprocates with a reply message. At a high-level, it means that every action by an entity depends on the action taken by the other participating entity. We can thus translate the action of one entity to the event (or condition) of the other communicating entity.

Statefulness of the protocol. As the 4G LTE protocol is stateful, every action of a participant is decided based on the current state and the external/internal event (e.g., packet reception or timer expiration) that occurred at the protocol level. Since events may be triggered at different components of the protocol implemented/managed by different source files, from an implementation's design perspective, state variables or pointers to them are represented with global variables so that they can be accessible from all the source files. This observation holds irrespective of language or design patterns used for any implementation. Besides, for tractability and efficient interoperation, implementations try to use the standard names of the protocol states and messages that are explicitly defined in the protocol specifications.

Validation of well-formedness. Implementation guidelines for 4G LTE recommend checking the well-formedness of cryptographic primitives (e.g., authenticity/integrity) of incoming messages. For instance, when a message is received, the participant first unpacks the message, checks the well-formedness and the sanity of specific fields of the payload, and then validates the message authentication code (MAC). Therefore, whenever a packet is received, it is passed to the respective *message handler* for performing these tasks.

3.2 Overview of ProChecker

In this section, we discuss the threat model followed by our definition of a FSM, challenges in designing such a system and overview of our framework.

3.2.1 Threat Model

We consider a Dolev-Yao adversary model [46] in which the communication channel between the client and the server is subject to the following adversary actions: arbitrary packet dropping, injection, or modification while impersonating a legitimate participant. In this model, the adversary adheres to cryptographic assumptions, i.e., it can decrypt a packet only if it has the keys.

3.2.2 Protocol Finite State Machine

We model the 4G LTE protocol abstractly as a set of deterministic FSM's. A state machine α^{μ} communicates with another state machine β^{μ} with two unidirectional channels, one carrying message from α^{μ} to β^{μ} and vice-versa. Each state machine is a 5-tuple $(\Sigma, \Gamma, \mathcal{S}, s_0, \mathcal{T})$, where Σ and Γ are the non-empty sets of conditions and actions for the protocol respectively, $\mathcal{S}$ is a finite set of states in which the protocol can reside, s_0 is the initial state of the protocol, and $\mathcal{T}$ is a finite set of transitions in $\mathcal{S}$. We consider a transition as a 4-tuple $(s_{\text{in}} \in \mathcal{S}, s_{out} \in \mathcal{S}, \sigma \subset \Sigma, \gamma \subset \Gamma)$. Here, s_{in} is the source state, s_{out} the destination state, σ and γ are the condition and action defined on this transition respectively.

3.2.3 Challenges

The most difficult problem in the design of ProChecker is to extract a high-level protocol model of the implementation with minimal knowledge of the protocol code. Solving such a problem requires addressing several challenges:

C1: *(Colossal codebase).* The main challenge is the sheer scale of the complex 4G LTE protocol implementation; in the case of industrial implementations with legacy codes this problem becomes intractable. For such implementations, both static and dynamic analysis results in imprecision [38] or state space explosion. The models generated from exclusively applying different static or dynamic analysis techniques either contain low-level intricacies of intra- and inter-procedural interactions and thus result in scalability issues.

C2: *(Pointer aliases and cryptographic constructs).* 4G LTE implementations contain large amounts of pointer aliases and cryptographic constructs. This makes the extraction of FSM challenging and results in intractability and false positives.

C3 *(Semantic model).* To detect logical vulnerabilities the extracted model must have semantic meaning and should not include low-level details, such as parsers, cryptographic protocol implementations, etc. Providing such an abstract model requires that someone with the implementation knowledge extracts the high-level protocol semantics for the resulting model to be amenable for automated analysis.

C4 *(Layered protocol).* 4G LTE has a layered architecture. A generated model that contains the interaction and information of all the layers would break the scalability of general-purpose model checkers. It is therefore important to extract the different layers separately to be in bounds with model checkers limits. However, this imposes an additional requirement when extracting a model of the underlying implementation, where all the layers are intertwined.

3.2.4 Insights on Addressing Challenges

For addressing C1 and C2, ProChecker does not rely directly on the codebase to extract the FSM of the implementation; rather it leverages the execution logs of the protocol interaction. The logs are captured from the execution of conformance test cases provided by the protocol standard body and/or the code manufacturers. 3GPP, the 4G LTE standard body, provides conformance test suites for protocol implementation verification [47]. Also, commercial vendors have their own functional testing infrastructure and code coverage information. To address C3 we automatically instrument the codebase to inscribe necessary information in the execution logs to create a FSM with semantic information. For C4 we only extract interactions of a particular layer from the execution logs and utilize the state and protocol message names from the standards [9].

3.2.5 High Level Description of ProChecker

ProChecker comprises of two components: (i) Model extraction; and (ii) Model checking (see Figure 3.1). For inferring a 4G LTE implementation's FSM, the model extraction leverages the *testing logs* generated from the functional conformance test suite. It is, therefore, important to provide a detailed execution log enriched with a sufficient-level of semantic information to the model extractor. For this, our simple source-code level instrumentor automatically instruments the code to dump the values of global and local variables. Note that, our instrumentor does not require any knowledge about the implementations, such as control-flow, program-dependency or call graphs. The information-rich log is then passed to the model extractor, which from the log extracts the specific state, condition, and actions of the FSM following a generic algorithm (see Algorithm 1). The algorithm utilizes

the traits of a generic 4G LTE implementation (that holds for both commercial and open-source implementations) and state and protocol message names from the specification. Our extracted model abstracts out all cryptographic assumptions and for all encrypted/integrity-protected messages, the plain-text counterpart is extracted. For instance, a specific protocol message may always be encrypted and transmitted with integrity protection; our extracted model does not include that information and only includes the interaction as plain-text. for analysis.

For model checking, our approach is based on the counter-example-guided-abstraction-refinement principle (CEGAR) [48]. In the CEGAR framework, an initial abstract model, and property are passed to the verifier. If the abstract model results in erroneous (or "spurious") counterexamples, the model is revised to rule out the spurious counterexamples. This continues until the verification goes through or a realizable counterexample is found. Based on CEGAR, in ProChecker, (1) our extracted 4G LTE model abstracts out the cryptographic assumptions. (2) We then instrument that model with Dolev-Yao [46] adversarial assumptions and call it a threat-enhanced model. (3) The threat-enhanced model and properties to check are passed to a general-purpose symbolic model checker. Note that the model may generate a spurious counterexample due to the absence of cryptographic abstractions. (4) To resolve this, we use a *cryptographic protocol verifier*. If the protocol verifier confirms that all the steps in the counterexample adhere to the cryptographic assumptions, then the counterexample (alternatively, the attack) is reported by ProChecker. Otherwise, we refine the property to ensure this spurious counterexample is never generated again. Like the CEGAR framework, this loop continues until the verification completes or a realizable counterexample is found.

3.3 Detailed Design of ProChecker

In this section, we dive deep into both the components of ProChecker: model extraction and model checking.

3.3.1 Model extraction

ProChecker leverages the properties discussed in Section 3.1.2 and extracts a scalable and verifiable FSM from the logs with the following high-level operations.

(1) *Creating an information-rich log.* To build a FSM, we extract information on the current/next protocol state, condition variables defining the next state, and corresponding actions from the log. The default execution log, however, only provides whether a particular function is executed, which is used for obtaining the coverage information. This information, however, is not enough for obtaining protocol states, conditions, and actions. To address this problem, we develop a source code-level instrumentation mechanism to automatically incorporate certain information into the log.

(2) *Code instrumentation.* The challenge for code instrumentation is to add information to the log with minimal implementation knowledge. To address this challenge, our code instrumentation prints only the values of global variables, local variables and function entrance/entry points in the log for each function. The value of global variables on the entry and exit for each function is used to detect state transitions, whereas the output of local variables right before the exit of a function is used to detect those variables' last value in the current function scope. The instrumented source code, when executed through the conformance test cases, thus creates a log containing the state information obtained from global variables, condition/action as protocol interaction inferred from function entrance, and even more detailed information, such as packet parsing/processing results, as local variables. This information-rich log is then used for extracting the FSM of the implementation. The only required manual intervention is the identification of the specific source files of a specific layer of the protocol that requires instrumentation. From our experience of industrial and open-source code, protocol source files of a specific layer are always located in separate directories and to make the instrumentation scalable and automatic, it is recommended to apply the instrumentation to the particular layer of the 4G LTE implementation under analysis. To achieve this instrumentation with minimal knowledge of the source code, we leverage insights from standard C/C++ coding practices such as (1) global variables defined

Algorithm 1 ProChecker Model Extractor

Require: Log, state_signatures, incoming_signatures, outgoing_signatures
Ensure: FSM$(\Sigma, \Gamma, \mathcal{S}, s_0, \mathcal{T})$

```
while end of Log not reached do
    B ← DivideBlock(Log, incoming_signatures)
    for each line L ∈ B do
        if L contains any s ∈ state_signatures then
            append s to FSM.S
            if s is the first state_signature ∈ B then
                s_in ← s
            else
                s_out ← s
            end if
        else if L contains any σ ∈ incoming_signatures then
            append σ to conditions set FSM.Σ
        else if L contains any γ ∈ outgoing_signatures then
            append γ to conditions set FSM.Γ
        end if
        if γ is empty then
            γ ← null_action
        end if
        append transition tuple (s_in, s_out, σ, γ) to FSM.T
        remove B from Log
    end for
end while
```

in separate header (.h) files, (2) local variables defined in the first basic block in each function.

(3) *Dissecting the log to detect relevant states, conditions and actions.* The log created through code-instrumentation and conformance test suite contains all the global, local variable values, and function entrance indications that are executed/accessed during the test case execution. The next challenge is to use this information with minimal implementation knowledge to extract the FSM. We leverage key insights from the 4G LTE protocol and their implementations for this step. As the 4G LTE protocol follows an event-driven paradigm, we can dissect the log into blocks based on each incoming message to the protocol. After the packet is received by the implementation, it is passed to the corresponding *incoming mes-*

sage handler designated for unpacking, decrypting, sanity checking (e.g., packet type and well-formedness), and validation of cryptographic primitives (e.g., message authentication code or MAC). Depending on which checks are passed, the internal state of the protocol is changed accordingly and the control moves to the corresponding *outgoing message handler* designated for taking the responsive action. Depending on the results of checks performed by the incoming message handler and protocol context, the receiver may take an action, i.e., send a response packet (*accept* or *reject* based on the validation results) to the other communicating entity (UE/MME) or take no action at all (referred to in our FSM as null_action). For example, whenever an authentication challenge is received, it is passed to the incoming message handler for processing authentication challenges. Upon completion of sanity checking and internal state transition, the authentication completion message is sent as a response from the outgoing message handler. Since the condition variables used in the sanity checking are local variables, we obtain their values from the information-rich log containing values of all the local variables declared and defined in the corresponding message handler. In a similar vein, we extract the current and next state information from the inscribed global state variables in the log. Based on the incoming message handler (from the function entrance indication in the log) and the outgoing message handler execution, we extract the type of message received i.e., the condition and sent i.e., the action of the FSM.

(4) *Mapping protocol specific variables to implementation.* To map the 4G LTE protocol/-standard specific state variables, incoming and outgoing messages, and condition variables to the myriad of implementation-specific variables in the log, we leverage the following intuitions: (1) 4G LTE state names defined in the standards [9] are directly used in the implementations to ensure interoperability. Therefore, by simply knowing the name of each state defined in the standards, we can detect the corresponding state represented with global variables. (2) Similarly, incoming/outgoing message names defined in the protocol specification are indirectly used in the implementation as function signatures. For industrial implementations, the same signature is followed consistently throughout the implementation and even for the open-sourced implementations, consistent signatures have been used. The consistency aides tractability, efficient portability, and interoperability. For instance, a sample signature is to prepend `send_/recv_` (based on whether the protocol message is incoming or

outgoing) as a prefix before the actual protocol message name. Instances of this signature can be `send_authentication_request`, `recv_ authentication_response`. Leveraging this insight, we use the function entrance information to extract both the type of message received and sent during protocol interaction and represent them as conditions and actions in a transition of the FSM. The algorithm for model extraction from the log is shown in Algorithm 1. The algorithm takes the generated Log, state, and incoming/outgoing message signatures as inputs and outputs the FSM. First, the log is divided into a block based on the incoming message signature that caused the protocol interaction. The block is then scanned line by line to extract states ($FSM.\mathcal{S}$), conditions ($FSM.\Sigma$), and actions ($FSM.\Gamma$) [line (4-18)]. Intuitively, the first extracted state of the block is denoted as the incoming state and the second one as the outgoing state [line (6-11)]. As already discussed, there might be the case when the incoming message does not trigger any action for the protocol (due to failed validation); in that case, the action is denoted as null_action [line (20-21)]. At the end of the extraction, the tuple $(s_{in}, s_{out}, \sigma, \gamma)$ is added to $FSM.\mathcal{T}$ to keep track of the transition relation system.

3.3.2 Model checking

Our approach combines a symbolic model checker (MC) and a cryptographic protocol verifier (CPV). As the 4G LTE protocol can be considered as a set of communicating FSM's, we model each communication between two FSM's, for instance, the communication between the UE and MME as, UE^{μ} and MME^{μ}, with two uni-directional channels; one from UE^{μ} to MME^{μ} and another from MME^{μ} to UE^{μ}. The choice of using two unidirectional channels instead of a single bidirectional channel provides more flexibility (e.g., one direction of the public channels to be adversary controlled whereas the other to be reliable) in reasoning about specific scenarios and filtering spurious counterexamples. From the extracted models UE^{μ} and MME^{μ} and including the two uni-directional channels, we enhance the model to include a Dolev-Yao-Style adversary and create a threat instrumented model IMP^{μ}. We then use a general-purpose model checker [43] and a property to check whether the model satisfies the property. If the model satisfies the property, we adjudicate the property to be verified

on the model. If, however, a counterexample is generated, there can be two possibilities: (a) the implementation model violates the property; (b) due to the abstraction of cryptographic-constructs, a spurious counterexample was generated. To prevent spurious counterexamples we run steps of the counterexample to a symbolic CPV. If the CPV confirms that all steps conform to the cryptographic assumptions, the counterexample can be considered valid. If the CPV adjudicates one of the steps taken by the adversary to be infeasible, we refine the property to ensure that the adversary does not exercise offending action in the future iterations of the verification. The verification loop continues until either the property is satisfied or a realizable counterexample is found.

3.4 Running Example

To illustrate our model extraction approach, we walk through a simplified example code (see Figure 3.2) of the attach procedure for a device in 4G LTE UE. The code is abstracted to include only the protocol interactions of Non-Access Stratum (NAS) layer of the cellular stack. The same algorithm can be utilized for other protocol layers. For our running example, we focus on the code of a UE for the final phase of the attach procedure, i.e., the protocol interaction through `attach_accept`/`attach_complete`. For ease of exposition, we assume that the simplified implementation contains three functions (see Figure 3.2). *air_msg_handler* takes a message from the MME, parses the message, identifies its type, and passes it to the corresponding handler associated with it. For our example, the incoming message is an `attach_accept` and it is thus routed to *recv_attach_accept*. The first task in any implementation of *recv_attach_accept* is to check whether the message contains a valid MAC. If the MAC check is passed, the control is transferred to the respective outgoing message handler that sends an appropriate response– which in our case is *send_attach_complete*. In this example code snippet, our instrumentation tool automatically includes few print statements that inscribe all global and local variables values, and function entrance information (see Figure 3.2, the instrumented lines are shown in blue) when relevant test cases are executed. For instance, consider a simple test case: "When a properly formatted `attach_accept` message

```
air_msg_handler(air_msg){
    print "air_msg_handler"
    print current_state
    ... ... ... ... ... ...
    ... ... ... ... ... ...
    air_msg_id = parse(air_msg)
    case(air_msg_id){
        attach_accept:
            recv_attach_accept( )
        authentication_request:
            recv_auth_request( )
             ... ... ... ... ... ...
             ... ... ... ... ... ...
             }
     print air_msg_id
     print current_state

}
```

(a) air_msg_handler

```
recv_attach_accept(air_msg){
    print "recv_attach_accept"
    print current_state
    ... ... ... ... ... ...
    ... ... ... ... ... ...
    mac_valid = extract(air_msg)
    if(mac_valid){
        send_attach_complete( )
    }else{
         send_emm_status( )
    }
    ... ... ... ... ... ...
    ... ... ... ... ... ...
    print mac_valid
    print current_state
}
```

(b) recv_attach_accept

```
send_attach_complete( ){
    print "send_attach_complete"
    print current_state
    ... ... ... ... ... ...
    ... ... ... ... ... ...
    #create attach_complete packet
    send_tx_conf( ) #send to MME
    ... ... ... ... ... ...
    ... ... ... ... ... ...
    print current_state
}
```

(c) send_attach_complete

```
air_msg_handler
current_state: UE_REGISTERED_INIT
recv_attach_accept
current_state: UE_REGISTERED_INIT
send_attach_complete
current_state: UE_REGISTERED_INIT
current_state: UE_REGISTERED
mac_valid: True
current_state: UE_REGISTERED
air_msg_id: attach_accept
current_state: UE_REGISTERED
```

(d) Generated detailed log

Figure 3.2. Instrumented generic example implementation (instrumented lines in the code are colored as blue)

with appropriate MAC is sent to the UE, the UE responds with an `attach_complete`". As the test case gets executed with the instrumented code, we get a detailed log (see Figure 3.2(d)).

Now the task of the model extractor is to build the FSM from the log. For building the FSM, we need to extract four specific pieces of information from the log: (1) incoming state, (2) outgoing state, (3) conditions, and (4) actions. in our example, line 3 of the log (Figure 3.2(d)) indicates that the control has moved to *recv_attach_accept* handler, which essentially means that the condition for this transition is the incoming `attach_accept`

message. Down the trace, line 8 indicates that the MAC for the message is computed as valid. Note that the initial state for this transition is extracted from line 6 and identified as `UE_REGISTERED_INIT`. The final state is extracted from line 9 as before completing the specific test case the state transitions to `UE_REGISTERED` state. Line 5 manifests that an `attach_complete` message was sent by the device in response to this particular test case. This example shows the effectiveness of our approach in building a FSM of an implementation without requiring detailed knowledge about the source code. In a practical case, the generated log will contain information about multiple rounds of interaction between the UE and the MME. In that case, the log can be divided into blocks based on the incoming message signature names. From the blocks, a similar strategy can be applied to extract the entire state machine.

3.5 Implementation

We now discuss the implementation of ProChecker. Though completed for LTE, we are adapting the framework for 5G.

Formal property gathering. The set of properties that ProChecker aims to check includes authenticity (e.g., disallowing impersonation attacks), availability(e.g., preventing denial of service attacks), integrity(e.g., restricting unauthorized messages), privacy of user's sensitive information (e.g., preventing location data, activity profiling, and preserving users soft identity), and replay protection. (e.g., restricting reception of the same message more than once). We identify and extract the precise and formal security goals from the informal and high-level descriptions given in the conformance test suites [47] and technical specification documents [9] provided by 3GPP and translate them into properties. We extracted, formalized, and verified a total of 62 properties among them 25 are related to privacy and 37 related to security.

Codebases. For the closed-source implementation, the complete size of the codebase is around 80 GB (including all testing infrastructure and legacy support). We integrate ProChecker with the mainstream functional testing framework of the implementation. For the open-source implementations we use the two most popular ones, srsLTE [41] and OpenAirInterface(OAI) [42]. All the codebases are written in C++.

Conformance test suite. For the closed-source codebase, the conformance test suite we leveraged is part of the codebase and contains 7087 test cases. These test cases can be considered as protocol level functional test cases, testing a separate protocol interaction. The test suite is completely automatic and all test cases can be run together to get a detailed log. Both srsLTE and OAI also have completely automatic testing environments as part of their codebase but do not have the implementations of all the conformance test cases. To test all the procedures of NAS layer and generate enough coverage we add 9 test cases to srsLTE (getting to 84% coverage for the NAS layer), and 7 test cases to OAI. Note that these additional test cases are not required for ProChecker, as any commercial LTE implementation must include the conformance testing framework following the 3GPP standards. This part is included only for demonstrating the viability of ProChecker on open-source LTE implementations.

Code instrumentation. We developed our instrumentation tool which takes the code directory of the specific protocol layer as input, and instruments the code with print statements for function entrance, global and local variables. For all three of our implementations, source files of a specific layer are located together in separate directories. We only instrument the NAS layer of the protocol. After the source code of the NAS layer is instrumented, the whole code is put through the conformance test suite to generate a detailed Log.

Model extractor. We implement the model extractor in Python 2.7 with around 1000 lines of code. We leveraged the protocol state names directly from the standards [9] as the implementations use identical names. We mapped the incoming/outgoing message signatures, sanity checking variable names following the incoming/outgoing message names from the standards, and a manual inspection of the source files of the NAS layer. For future generations, this mapping can be documented with minimal effort while designing the implementation, thus eliminating this one-time manual intervention altogether. For the largest log from the closed-source implementation, it takes our model extractor around 5 minutes to analyze the log and generate the semantic model.

Adversarial model instrumentor. The adversarial model instrumentor takes two FSM's— UE^{μ} and MME^{μ} for UE and MME as input and returns another model IMP^{μ} which is an extension of UE^{μ} and MME^{μ} containing explicit adversarial influence. Given two public com-

munication channels– c_1 from UE^{μ} to MME^{μ}, and c_2 from from MME^{μ} to UE^{μ}, our ProChecker incorporates adversarial capabilities into UE^{μ} and MME^{μ} and thus combine them all to build a new threat-instrumented abstract model IMP^{μ}. For instrumenting threat to a given transition, the adversary non-deterministically decides either to drop/pass/change the message. We have developed a model generator that takes as input the state machine of the protocol written in Graphviz-like language and outputs a SMV [43] description of the model. For our implementation, we extracted the models of the UE by using our proposed model extraction module. We, however, did not have access to the commercial/closed-sourced implementation of a core network and thus used the open-source core network's FSM manually constructed by Hussain et al. [32], which precisely served our purpose as we were interested in identifying vulnerabilities on the UE side. But it is evident that, given the implementation and the test cases, this approach can also be applied to the core network's implementation.

Model checker (MC). To model check IMP^{μ}, we use NuXmv[43]. A major challenge in formal verification is scalability; the model checker may not be able to terminate when the model is large. We want to report that it was indeed possible to run a model checker on the model extracted from an industry-level large codebase. This is because of our semantic model-extraction based on high-level protocol interactions from the log and abstracting out low-level details of implementation.

Cryptographic protocol verifier (CPV). The counterexample generated from MC is fed to the ProVerif [44] CPV to determine its validity. For each adversary action in the model checker provided as a counterexample, we query the CPV to check its feasibility. If all adversarial actions can be proven feasible, then the counterexample is presented as a feasible attack and tested on the testbed. Otherwise an invariant is added to the property ruling out the infeasible adversarial action to refine the property. The verification loop between MC and CPV is continued until either the property is satisfied by the model or a realizable counterexample is found.

Testbed. We build a testbed using low-cost software-defined radios and open-source LTE software stack, srsLTE [41], totaling to a cost of around USD $4000. After verifying the counterexample, we manually analyze the counterexample to determine whether it is due

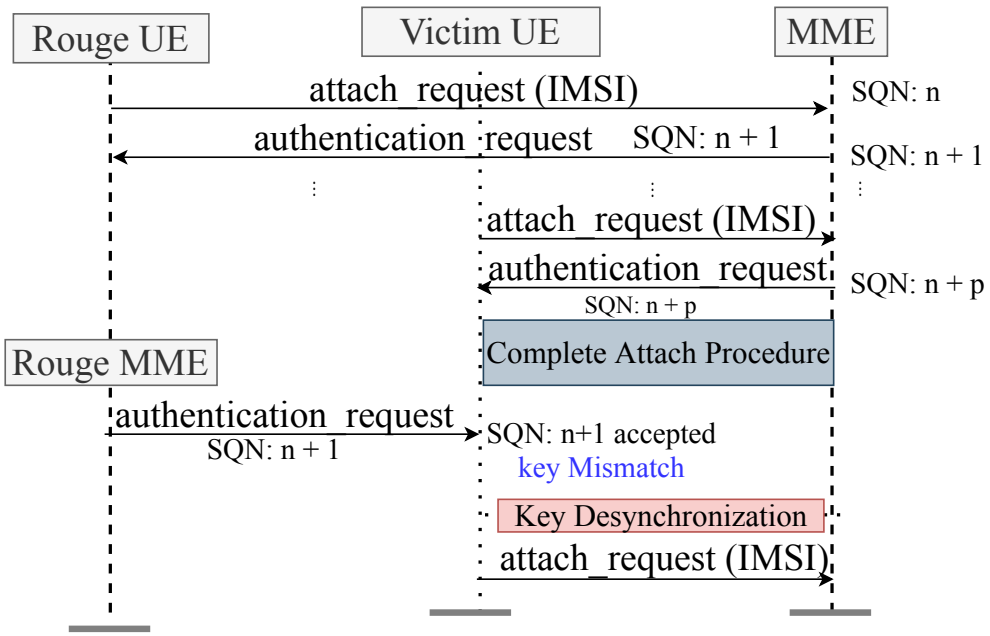


Figure 3.3. Service disruption using `authentication_request`

to the implementation deviating from the protocol specification or the deviation is due to underspecification, design-flaw in the standards.

3.6 Evaluation and Findings

The main goals of our evaluation of ProChecker are to answer the following research questions:

- **RQ1. Logical Vulnerability Detection:** How effective are the extracted models of ProChecker? Is it possible to reason about security and privacy properties to detect logical vulnerabilities with the models?
- **RQ2. Model Comparison:** How expressive is the automatically extracted FSM of ProChecker compared to the state-of-the-art model [32] for formal verification?
- **RQ3. Scalability:** Is the generated model scalable with the COTS model checking tools?

3.6.1 RQ1. Logical Vulnerability Detection

ProChecker uncovered 3 new protocol specific logical attacks (applicable to any implementation), 6 implementation issues, and identified 14 prior attacks (Table 3.1).

New counterexamples: We first discuss the 3 protocol specific (P1-P3) logical attacks, which are true for all three implementations, and then 4 most severe implementation specific (I1-I4) vulnerabilities.

(P1) Service disruption using `authentication_request`: With this attack, the adversary exploits a potential vulnerability of `authentication_request` sequence number (`SQN`) handling. The attack utilizes previously captured `authentication_request` to desynchronize keys, disrupt service, and force the UE to go through power-consuming authentication procedure over and over again, causing a denial of service and battery depletion.

Detection and attack description. We uncover this attack by model checking IMP^{μ} with respect to the property: "*If the UE is in the registered initiated state, it will get authenticated with an authentication sequence number (`SQN`) which is greater than the previously accepted `SQN`*". We observe a counterexample in IMP^{μ} where the UE accepts a `SQN` value smaller than the current value. We validate the capability of the adversary to fabricate `attach_request` message to generate legitimate `authentication_request` by the MME using ProVerif. The steps of the attack are shown in Figure 4.5. The adversary using a malicious UE sends `attach_request` message to the MME to capture `authentication_request` message to be used later in the attack. At the time of the attack, the attacker replays such captured `authentication_request` to the victim UE. Due to the specific design of the generation and verification of `authentication_request` message's `SQN` (which we describe in detail in the next section), the victim UE accepts and processes this stale `authentication_request` message, and regenerates all session keys causing a key desynchronization between the UE and the legitimate MME. As a result, the UE will also keep discarding actual packets from the legitimate MME until the connection is dropped and the authentication procedure is invoked from scratch by the legitimate MME. By analyzing the traces of real operational networks, we uncover that it is possible to use previously captured `authentication_request` messages– which are days old to carry out such an attack. To extend the attack impact, the adversary

can keep using previously captured `authentication_request` messages and replay them to the victim UE recurrently.

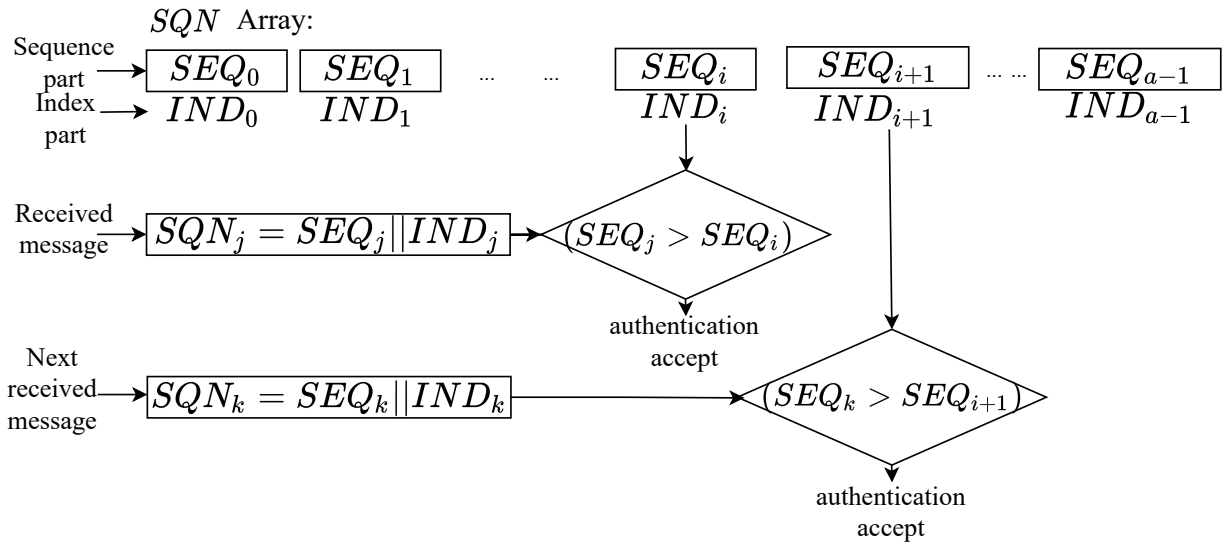


Figure 3.4. Sequence number handling in USIM

Vulnerability. The `SQN` generation and verification for `authentication_request` message is performed through a complex scheme defined in TS 33.102 [49] Annex C. The `SQN` is divided into two concatenated parts SQN = SEQ || IND, namely the *sequence* (`SEQ`) and the *index* (`IND`). To generate a fresh `SQN`, the core network increments both `IND` and `SEQ`, concatenates them together and sends to the UE. For verification on the UE side, the USIM keeps track of a `SQN_array` of $a = 2^{\texttt{IND}}$ items, where each item is a `SQN` as shown in Figure 3.4. Whenever a new $\text{SQN}_\text{j} = \text{SEQ}_\text{j} \,||\, \text{IND}_\text{j}$ $(0 \leq \text{IND}_\text{j} \leq (\text{a} - 1))$ in an `authentication_request` is received by the UE, the USIM in the UE looks up its `SQN_array`, checks if $\text{IND}_\text{i} == \text{IND}_\text{j}$, and based on that it retrieves the corresponding SEQ_i value from the `SQN_array`. After comparing this saved SEQ_i with the received SEQ_j, the UE either accepts or rejects the `SQN` number and the `authentication_request` message. In case $\text{SEQ}_\text{j} \leq \text{SEQ}_\text{i}$, the USIM generates an authentication synchronization failure message using the highest previously accepted `SQN` anywhere in the `SQN_array`. Due to this design, the UE allows out-of-order `SQN` values.

Following our example, in case $\text{SQN}_\text{j} = \text{SEQ}_\text{j} \,||\, \text{IND}_\text{j}$ is captured and dropped by an attacker, the MME would generate another $\text{SQN}_\text{k} = \text{SEQ}_\text{k} \,||\, \text{IND}_\text{k}$ (where $\text{IND}_\text{k} = (\text{IND}_\text{j} + 1)\%\text{a}$

and $0 \leq \mathrm{IND_k} \leq (\mathrm{a}-1)$) and send it to the UE. Upon receiving the message, the USIM would look up the $\mathrm{IND_k} = \mathrm{IND_{i+1}}$ index of the `SQN_array`, retrieve $\mathrm{SEQ_{i+1}}$, compare with the received $\mathrm{SEQ_k}$, and accept the `SQN`. Now, when the previously captured and dropped $\mathrm{SQN_j} = \mathrm{SQE_j} \mid\mid \mathrm{IND_j}$ is replayed back to the UE by the attacker, the USIM would look up `SQN_array` at index $\mathrm{IND_i} = \mathrm{IND_j}$ and as the sequence part $\mathrm{SEQ_i}$ at this index is still unchanged and smaller than the received $\mathrm{SEQ_j}$ the UE would accept this stale $\mathrm{SQN_j}$.

According to the specification, this design to accept `authentication_request` messages with out-of-order `SQN` was designed to allow more efficient authentication of UEs that move between different regions of a serving network or between different serving networks in roaming scenarios and thus frequently run into `SQN` desynchronization issues. From our experiments, we, however, uncovered that COTS UEs choose 5 bits for `IND` and the rest for `SEQ`, which results in a `SQN_array` of $a = 2^5 = 32$ values. With this values, the USIM accepts 31 previously captured stale `authentication_request` message. From the captured traffic of commercial network operators, we observed that it takes at least a few (in some cases far more) days to receive this much `authentication_request` from the MME. Therefore, the majority of the COTS UE implementations accept a couple of days old `authentication_request` message sent by the network, making it possible for the attacker to carry out attacks.

Interestingly, in TS 33.102 [49] Annex C 2.2, there is a freshness limit (L) on the range of old accepted `SQN`. If the difference between the saved sequence part SEQ_{i} at index IND_{i} and the received sequence part, SEQ_{j} is $SEQ_{\mathrm{j}} - SEQ_{\mathrm{i}} > L$, it will be rejected. However, the use of such a range is completely optional and the value is undefined for both 4G and 5G. The specification states – "*The use of such a limit is optional. The choice of a value for the parameter L affects only the USIM. It has no impact on the choice of other parameters and it is entirely up to the operator, depending on his security policy.* ***Therefore no particular value is suggested here***". Apparently, being optional and unspecified none of the major vendors are implementing such a check, paving the way to the acceptance of old sequence numbers.

(P2) Linkability attack using `authentication_response`: This attack can enable an adversary to track the presence of a user's device in a particular cell area violating the user's privacy by exploiting the different responses of `authentication_request` message. From the counterex-

ample of the previous attack (**P1**), we identified that the UE accepts previously captured stale `authentication_request` with a `SQN` value smaller than the current accepted value. We utilize ProVerif's capability to reason about observational equivalence and pose the query: "*is it possible to distinguish two UE's based on their responses to an* `authentication_request`*?*", to identify this attack. The first phase of the attack to capture `authentication_request` is similar to **P1** (see Figure 4.5).

For the next step, the attacker with a malicious base station connects to all the UEs in a particular cell area and replays the captured `authentication_request` to all of them. The victim UE will accept this message and respond with `authentication_response` whereas all the other UEs in the cell will respond with MAC failure due to integrity check failure (see Figure 3.5) identifying and tracking the presence of the victim user. This attack is inspired by the linkability attack using `auth_sync_failure` shown in 3G [50] with the caveat that the distinction between different messages and a different vulnerability is utilized by the attacker for the two attacks. In this attack, the attacker differentiates between an out-of-order accepted `authentication_request` and a synchronization failure `auth_sync_failure` message to detect the presence of a user, whereas in the 3G attack two failure messages (`auth_sync_failure` and `auth_MAC_failure`) are utilized for the attack. The root cause of this attack is same as **P1**.

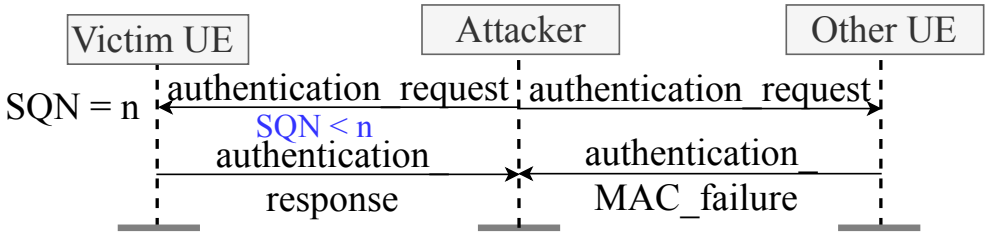


Figure 3.5. Linkability using `authentication_response`

Impact on 5G. The generation and verification scheme of sequence number (`SQN`) in `authentication_request` message described in **P1** (Section 3.6.1) is exactly the same in the 5G specifications, thus making the 5G rollout directly vulnerable to **P1** and **P2** attacks.

(P3) Selective security procedure denial: In this attack, the adversary exploits the underspecification of the sequence number checking to prevent important security procedures

entirely causing severe security and privacy issues.

Detection and attack description. We model check IMP^{μ} against the property "*If the MME initiates a common procedure (e.g.,* Security Mode Command *or* GUTI Reallocation*), the UE will complete that procedure*". This is violated by a trivial counterexample where the adversary drops packets in transit, but neither UE nor MME could detect such occurrences. We, in fact, observed that the adversary can even drop an arbitrarily large number of packets at once since the UE always accepts packets with higher sequence numbers, but does not check the difference of sequence numbers of two consecutive received packets. As a result, for the security procedures where a fixed number of trials is attempted, it is possible to drop packets and surreptitiously prevent the security procedure altogether. To carry out this attack, the attacker sets up a man-in-the-middle (MITM) relay between the UE and MME and drops packets related to important security procedures, such as GUTI reallocation or security mode command. The attacker, by inferring the message type (from the packet metadata, e.g., packet-length and temporal order of the encrypted/plaintext packets in transit), can selectively drop relevant packets until the security procedure is abandoned by the MME (in most of the times it is tried 4-5 times). This forces the victim UE and the core network to keep using previous security contexts or the temporary identifier GUTI.

Vulnerability. Such kind of selective service denial attack is possible because of the underspecification of the standards. In TS 24.301 [9] it is specified that "*Replay protection must assure that one and the same NAS message is not accepted twice by the receiver. Specifically, for a given NAS security context, a given NAS COUNT value shall be accepted at most one time and only if message integrity verifies correctly.*" However, the case where an adversary is dropping packets surreptitiously is not handled in the specification. Due to the higher sequence number being the only satisfying condition and the inadequate check on the sequence numbers of two consecutive packets, it is possible to carry out this attack without detection.

Impact. The impact of this attack can be catastrophic as it affects multiple crucial security and privacy-preserving procedures. For instance, when the MME assigns a new GUTI to the UE with `GUTI_reallocation_command` message, the adversary can drop the message without being detected by the UE/MME and thus can induce both parties to use the same GUTI for a longer time than expected. The consequence of such packet dropping on the

Table 3.1. : Summary of ProChecker's findings. ● yes, ○ no, – not implemented

Attack	Property Type	Implication	Vulnerability Type	srsLTE [41]	OAI [42]
New Attacks					
(P1) Service disruption using authentication_request	Security	Service disruption	Standards	●	●
(P2) Linkability using authentication_response (Inspired by [50])	Privacy	Location privacy leakage	Standards	●	●
(P3) Selective service dropping	Security	Surreptitious service disruption	Standards	●	●
(I1) Broken replay protection with all protected messages	Security	Broken replay protection	Implementation	●	●
(I2) Broken integrity, confidentiality with all protected messages	Security-Privacy	Integrity, encryption broken	Implementation	○	●
(I3) Counter-reset with replayed authentication_request	Security	Breaks replay protection	Implementation	●	○
(I4) Security bypass with reject messages	Security	Security bypass	Implementation	●	○
(I5) Privacy leakage with identity_request	Privacy	IMSI leaking	Implementation	○	●
(I6) Linkability with security_mode_command	Privacy	Location tracking	Implementation	●	●
Previous Attacks					
Authentication sync. failure [32]	Security	Denial of Service	Standards	●	●
Stealthy kicking-off [32]	Security	Detaching victim surreptitiously	Standards	●	●
Panic attack [32]	Security	Creating artificial chaos	Standards	●	●
Linkability using TMSI_reallocation [51]	Privacy	Location privacy leak	Standards	–	–
Linkability using IMSI to GUTI using paging_request [50]	Privacy	Location privacy leak	Standards	●	●
Linkability using auth_sync_failure [50]	Privacy	Location privacy leak	Standards	●	●
Authentication relay [32]	Security-Privacy	DoS, location history poisoning	Standards	●	●
Numb Attack [32]	Security	Prolonged DoS, batter depletion	Standards	●	●
Downgrade using tracking_area_reject [25]	Security	DoS	Standards	–	–
Denial of all services [25]	Security	DoS	Standards	●	●
Paging hijacking [32]	Security	Stealthy DoS, panic	Standards	●	●
Detach/Downgrade [32]	Security	DoS, battery depletion	Standards	●	●
Service Denial [32]	Security	DoS	Standards	●	●
Linkability (GUTI/TMSI) [32]	Privacy	Location Tracking	Standards	●	●

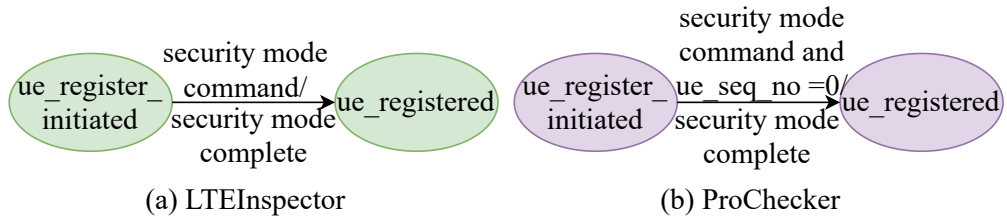
GUTI reallocation procedure is critical because of the following specification - TS 24.301 [9]: "*The GUTI reallocation procedure is supervised by the timer T3450. The network shall, on the first expiry of timer T3450, reset and restart timer T3450 and shall retransmit the* `GUTI_reallocation_command`*. This retransmission is repeated four times, i.e. on the fifth expiry of timer T3450, the network shall abort the reallocation procedure*". This implies that an adversary can surreptitiously drop five consecutive `GUTI_reallocation_command` messages and prevent the procedure entirely. After the five tries, the MME thus aborts the procedure and both MME and UE will keep using the previous GUTI. Since frequent updates of GUTI are

mandated by the standard to prevent user tracking, this attack forces the GUTI reallocation to fail and thus enables the adversary to track the victim for a long period of time. Similar implications also apply to the *security mode command* procedure, where it is also possible to surreptitiously prevent the UE and MME from re-negotiating the keys. Such kind of selective procedure denial enables the adversary to force a device to reuse the same GUTI or session keys for an elongated time period and thus to track the victim device easily.

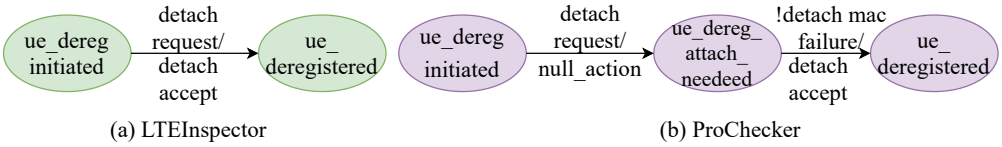
Impact on 5G. For the vulnerability and attack described here, the same procedures exist with the same design in 5G [52] as well, making it vulnerable to selective security procedure denial attack. Moreover, 5G introduces some new procedures which are also vulnerable to this attack. For instance, in TS 24.501 [52] the *5G Configuration Update Procedure* it is stated–"*The network shall, on the first expiry of the timer T3555, retransmit the* `configuration_update_command` *message and shall reset and start timer T3555. This retransmission is repeated four times, i.e. on the fifth expiry of timer T3555, the procedure shall be aborted*" making it possible to drop five messages, deny the procedure entirely and force the use of the same 5G-GUTI. Similar to this attack on 4G LTE, the adversary exploiting the vulnerability in the 5G network can track the user for long periods of time.

(I1) Broken replay protection with all protected messages: As discussed in the previous attack (P3), the UE should never accept any replayed packet after the security context has been established. We, however, found that both srsUE [41] and OAI [42] implementations allow the adversary to replay packets. We tested the FSMs of these implementations with the replay protection property and observed that OAI accepts only the last message when replayed, whereas srsUE accepts any replayed messages and resets the downlink counter with the counter value given in the replayed packet.

(I2) Broken integrity, confidentiality with all protected messages: The standard [9] also specifies a primitive property that a UE must not accept any plain-text messages after the security context is established. However, while testing the FSM of OAI with this property, we found a counterexample where the UE accepts plain-text messages with *plain-NAS (0x0)* as the packet header after the security context is established. We validate the counterexample in the testbed and indeed the OAI implementation accepts all security-protected messages in plain-text and un-cyphered after establishing the security context, effectively



(a) Mapping transitions with new conditions



(b) Mapping transitions with new states

Figure 3.6. Transition refinement between ProChecker and LTEInspector

breaking integrity and confidentially protection of the protocol implementation.

(I3) Counter-reset with replayed `authentication_request`: We uncover this attack by model checking IMP$^{\mu}$ with respect to the property: "*If the UE is in the registered initiated state, it will get authenticated with an authentication sequence number (`SQN`) which is greater than the previously accepted `SQN`*". We observe counterexamples where the implementations of srsUE accept the same sequence number and reset the counter. Due to this attack, it is possible to break the replay protection by sending replayed packets over and over again.

(I4) Security bypass with reject messages: As per the specification, the UE after receiving a release/reject message (e.g., `attach_reject`) should delete all the security contexts, move to the de-registered state and perform authentication and security mode command procedures again to reconnect to the network. While checking this property with the srsUE model, we, however, found counterexamples in which the UE directly moves from de-registered to registered state without completing authentication and security mode command procedures. Thus the adversary can bypass the whole security and authentication procedure.

Proving previous attacks: Along with detecting new logical issues, ProChecker is able to automatically identify 14 design-level logical vulnerabilities uncovered by previous works [25, 32, 50, 51] (see Table 3.1). These vulnerabilities were previously identified through manual inspection or from models that were manually derived.

3.6.2 RQ2. Model Comparison

To evaluate the expressiveness of ProChecker's automatically extracted models we compare the extracted model from the closed-source implementation to the closest available 4G LTE model from LTEInspector [32]. We compare the extracted model from the closed-source implementation because it implements all the procedures and has a complete conformance test suite. For the comparison we first introduce a notion of refinement for FSMs and use it to compare the models.

Refinement. Let, $\mathsf{M_1}^{\mu} = (\Sigma_1, \Gamma_1, \mathcal{S}_1, s_{0_1}, \mathcal{T}_1)$ and $\mathsf{M_2}^{\mu} = (\Sigma_2, \Gamma_2, \mathcal{S}_2, s_{0_2}, \mathcal{T}_2)$ be two protocol FSM's. We say that $\mathsf{M_2}^{\mu}$ is a refinement of $\mathsf{M_1}^{\mu}$ if the following properties hold: (1) The set of states $\mathcal{S}_1$ is a subset of the set of states $\mathcal{S}_2$. For this property to be true, for each state, $s \in \mathcal{S}_1$, there is an one-to-one mapping to a state $s' \in \mathcal{S}_2$. (2) The sets of conditions Σ_2 and actions Γ_2 are a strict supersets of Σ_1 and Γ_1 respectively, containing new constraints on the transitions. (3) The transitions in set $\mathcal{T}_1$ can be mapped to transitions in $\mathcal{T}_2$. For each transition $t_{\mathrm{i}} = (s_{\mathrm{i}_{in}}, s_{\mathrm{i}_{out}}, \sigma_{\mathrm{i}}, \gamma_{\mathrm{i}}) \in \mathcal{T}_1$ there can be several cases: (i) t_{i} can be directly mapped onto a transition in $\mathcal{T}_2$; (ii) t_{i} can be mapped to a transition $t_{\mathrm{j}} = (s_{\mathrm{i}_{in}}, s_{\mathrm{i}_{out}}, \sigma_{\mathrm{j}}, \gamma_{\mathrm{j}}) \in \mathcal{T}_2$, where t_{j} has the same incoming and outgoing state as t_{i}. However the condition, σ_{j} has the form $\sigma_{\mathrm{i}} \wedge \phi$, where ϕ is a new condition defined in Σ_2, thus making the condition of t_{j} stricter than t_{i}; (iii) t_{i} can be mapped onto multiple transitions (based on the new states) in $\mathcal{T}_2$. The transition t_{i} from $s_{\mathrm{i}_{in}}$ to $s_{\mathrm{i}_{out}}$ can go through multiple new intermediate states such as $s_{\mathrm{i}_{\mathrm{i}}}$, generating transitions of the form $t_{\mathrm{i}1} = (s_{\mathrm{i}_{in}}, s_{\mathrm{i}_{\mathrm{i}}}, \sigma_{\mathrm{i}1}, \gamma_{\mathrm{i}1})$ and $t_{\mathrm{i}2} = (s_{\mathrm{i}_{\mathrm{i}}}, s_{\mathrm{i}_{out}}, \sigma_{\mathrm{i}2}, \gamma_{\mathrm{i}2})$, which can be mapped onto transitions in $\mathcal{T}_2$. The mapped transitions in $\mathcal{T}_2$ contain all the previous conditions and actions on t_{i} together with new conditions and actions defined on $t_{\mathrm{i}1}$ and $t_{\mathrm{i}2}$.

Comparison of the models. We now show that the model of the closed-source implementation extracted by ProChecker (Pro^{μ}) is a refinement of the model of LTEInspector (LTE^{μ}). First, the majority of the states in the set $\mathcal{S}_{LTE}$ of LTE^{μ} can be directly mapped onto the states in the set $\mathcal{S}_{Pro}$ of Pro^{μ}. States in $\mathcal{S}_{LTE}$ that do not have a direct mapping in $\mathcal{S}_{Pro}$ (such as *ue_registered* and *ue_deregistered*) can be mapped onto the set of sub-states of the respective states. This mapping from states to sub-states is done following the standards [9].

Table 3.2. : Common properties of ProChecker and LTEInspector

High Level Properties common to ProChecker **and** LTEInspector
① If the UE is in the deregistered state, it is always the case that the UE initiates authentication and moves to the registered initiated state from there on, eventually the UE gets authenticated and moves to the registered state
② When the MME is in the tracking area update initiated state and the UE sends tracking_area_update_request message, the MME will eventually move to registered state.
③ The UE sends a service_request only if the MME sent the paging message that is pending
④ If the MME sends a security_mode_command message, the security context will be eventually updated.
⑤When the MME is in the service initiated state and the UE sends service_request_message, the MME will eventually move to the registered state
⑥ The UE will respond with the GUTI_reallocation_complete message only if the MME sends GUTI_reallocation_command

Specifically, due to the automated extraction of FSM by ProChecker, *it was possible to extract sub-states of several procedures*; manually generating such sub-states would be severely cumbersome. Second, the condition Σ_{Pro} and action Γ_{Pro} sets of Pro^{μ} are strict supersets of Σ_{LTE} and Γ_{LTE} of LTE^{μ}, respectively. Furthermore, as *data and packet payload information* were also extracted in Pro^{μ}, *new constraints* (such as sequence numbers and back-off counters) and new actions are included in Σ_{Pro} and Γ_{Pro}. Finally, some transitions follow one-to-one mapping between Pro^{μ} and LTE^{μ}. Others can be mapped based on new states or new conditions following our definition of refinement. The transitions defining new conditions impose stricter constraints (using predicates) that are based on the data and packet payload. For instance, the transition $t_{LTE} \in \mathcal{T}_{LTE}$ presents a change of state from *ue_register_initiated* to *ue_registered* for the condition `security_mode_command` and action `security_mode_complete`. In $t_{Pro} \in \mathcal{T}_{Pro}$, t_{LTE} is mapped to t_{Pro}, which is the refined version of t_{LTE} where both states and actions remain the same but the condition has the form `security_mode_command` and `ue_sequence_number=0`, and it is thus stricter. These example transitions for both LTEInspector and ProChecker are shown in Figure 3.6a. The rest of the transitions in $\mathcal{T}_{LTE}$ can also be mapped onto transitions on $\mathcal{T}_{Pro}$ based on *new states*. To illustrate this, let us consider the transition from *ue_dereg_initiated* to *ue_deregistered* having the condition `detach_request` and action `detach_accept` in $\mathcal{T}_{LTE}$. In $\mathcal{T}_{Pro}$ a new intermediate state is introduced *ue_dereg_attach_needed* and the transition is broken into two, introducing new conditions (shown in Figure 3.6b). Pro^{μ} is, therefore, a refinement of LTE^{μ}

and considers more procedures and critical aspects, including transitions based on data and packet payload.

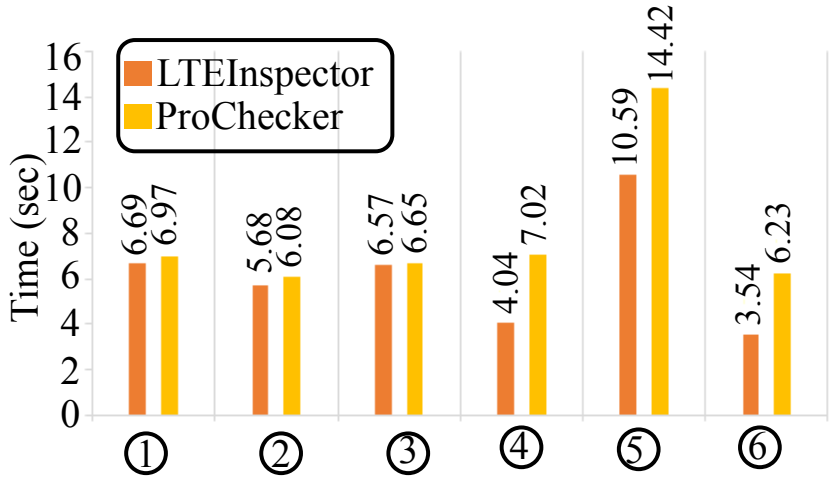


Figure 3.7. Execution time of the common properties used in ProChecker and LTEInspector. Properties are numbered according to Table 3.2.

3.6.3 RQ3. Scalability

We take our largest and most detailed extracted model– from the closed-source implementation, and record the execution times for verifying the properties common to the LTEInspector model (see Table 3.2). We used a laptop with an Intel i7-3750QCM CPU and 32 GB DDR3 RAM. The results in Figure 3.7 show that the time required by ProChecker for each property is only a fraction higher than LTEInspector. This result also signifies the scalability of our framework since it can run a COTS model checker on the automatically extracted model from an implementation with negligible overhead.

3.7 Discussion and Limitations

Completeness of our model. Model completeness depends on the coverage of test suites. In our experiments, we managed to extract state machines detailed enough to reason about protocol aspects critical for security. For commercial 4G LTE implementations, having a

complete conformance testing suite is a must; therefore this automatically corresponds to high coverage. We also showed that in the case of open-source implementations, it is possible to add some procedure-specific test cases and extract a formal model. Even though ProChecker's ability to extract details is limited by the coverage of the test suite, its true strength lies in its efficiency. For a given test environment, it adds negligible resource overhead to extract a state machine. As the test suite grows in coverage, ProChecker can generate increasingly detailed FSMs.

Consistent message name signatures. ProChecker leverages consistent protocol message and packet extraction signatures for extracting the FSM from the generated log. Our case study of industrial and open-source 4G LTE implementations, in fact, substantiates this assumption since those implementations follow a consistent signature because of tractability, efficient portability, and interoperability. For instance, srsLTE and OAI use the consistent signature of `send_/parse_` and `emm_send_/emm_recv_` followed by actual protocol message name from the standards respectively.

FSM for both communicating parties. Since we did not have access to the source code for the core network, we had to use an open-source standard model derived by the research community. For protocols, such as Wi-Fi and Bluetooth, where both communicating parties of the protocol are implemented by the same vendor, the same methodology can be applied.

ProChecker for 5G implementations. The design requirements of ProChecker for analyzing 5G implementations are similar to that of 4G (i.e., properly defined protocols states, protocol message names [52], conformance test case suite [53]). Therefore, this framework can easily be adapted to evaluate any 5G implementations. More precisely, since ProChecker works with a very minimal overhead with the existing testing infrastructure, it can be easily adapted to verify the security and privacy properties of the 5G protocol implementations from the get-go.

Access to the code and testing infrastructure of closed-source implementation. We got access to the closed-source source-code and the conformation/functional testing infrastructure, through a collaboration with industry.

Threat to validity. We used automatically extracted models for UE FSMs. Due to the low coverage of test suites and manually constructed MME FSMs extracted from the 3GPP

standard, the counterexamples derived from the model may not completely reflect the behavior of real operational networks. Thus inaccuracies in the model may induce false positives in commercial networks, although, we have not observed any such behavior. Due to ethical considerations, we validate the attacks in a custom-built network, which may not faithfully capture the operation network behavior.

3.8 Summary

We presented ProChecker—a framework for automatically verifying cellular network protocol implementations to uncover logical vulnerabilities. On the horizon of 5G deployment ProChecker can have important impact in securing 5G implementations from the very start. The properties discussed here for cellular networks apply to any communication protocol in general; therefore, in the future, we plan to use ProChecker on other protocols such as Bluetooth, and WiFi.

We present DIKEUE which can automatically infer the FSMs of 4G LTE UE implementations, and identify deviant behaviors among the implementations in a property-agnostic way. To show the viability, we applied DIKEUE to 14 COTS devices from 5 vendors. DIKEUE uncovered 15 deviant behaviors; among them 11 are exploitable. We have responsibly disclosed the vulnerabilities to the affected stakeholders and they have acknowledged our findings.

4. NONCOMPLIANCE AS DEVIANT BEHAVIOR: AN AUTOMATED BLACK-BOX NONCOMPLIANCE CHECKER FOR CELLULAR DEVICES

4G Long-Term Evolution (LTE), developed by the 3rd Generation Partnership Project (3GPP), is a global standard for cellular networks. 4G LTE protocols provide ubiquitous connectivity, interoperability, and massive scale support to numerous network services and billions of heterogeneous devices. As the security of cellular devices (also known as, User Equipment or ***UE***) is of utmost importance in this ecosystem, it is imperative that devices correctly implement the cellular protocols as mandated by the standard. Faithful implementation of the cellular protocol is, however, challenging due to the ambiguities, underspecification, and intricate protocol details present in the natural languages specification [9–11]. As a consequence, misinterpretations of the standard are commonplace, which result in implementations demonstrating *noncompliant behavior* with the cellular standard. As an example, if a device responds to a particular message in a state whereas the standard prescribes ignoring the message, it gives rise to a noncompliant behavior. The ramifications of noncompliance with the standard may result in (1) critical security and privacy flaws (e.g., authentication bypass [8], location exposure of a target user [25]), and (2) interoperability issues in the UEs. Since manual identification of noncompliant protocol behavior in large and complex implementations is error-prone and time-consuming, in this work, we aim to develop an automated approach for identifying noncompliance behavior in 4G LTE UEs.

Prior research. Although prior works [15, 23–28, 54] analyzing security and noncompliance of cellular proptocols have identified several implementation flaws, they suffer from at least one of the following limitations: (A) The approaches [8, 15, 23–27] are completely manual and cannot uncover a myriad of *implementation-specific* behavior; (B) The analyses [8] perform semi-automated stateless testing; (C) The approaches based on formal verification [32–34] only test the protocol specification for noncompliance and also heavily rely on the coverage and quality of the properties being tested—for which there is no official exhaustive list; and (D) The analyses based on re-hosting and reverse-engineering the baseband software [28, 54]

not only require a huge manual effort and expertise but also are not general enough to be applicable to implementations from different vendors.

Problem and scope. Since implementations of commercial base stations and core networks are not publicly accessible, we focus only on analyzing the commercial 4G LTE device implementations. Among many different procedures, we further focus on the *connection management* and the *mobility management* components of a UE. These components manage the most critical control-plane procedures, including connection setup, termination, mobility, hand-off, service notification, and setup procedures. Without the correct and reliable operations of these *stateful* procedures, most of the other control-plane (e.g., call setup) and data plane (e.g., browsing Internet) operations are susceptible to critical security attacks, such as MitM relay [32, 55], eavesdropping [23] and DNS redirection [55]. In summary, in this research we address the following research question: *Is it possible to design an automated , black-box, and stateful protocol analysis framework that can uncover noncompliant behavior in the control-plane protocol implementations in 4G LTE UEs?*

Challenges. The first critical challange for developing a black-box noncompliance checker for UEs is to automatically extract a behavioral abstraction of the protocol implementation. Once we have extracted the behavioral abstraction from an implementation, the second challenge is to devise an approach for identifying diverse noncompliant behavior in a property-agnostic way.

Our approach. In this work, for our automated and black-box efficient compliance checker DIKEUE (in Greek mythology, Dike refers to the goddess of justice), we use the input-output protocol finite state machine (*FSM*) as the behavioral abstraction. One can consider automatically extracting the protocol FSM from the implementation in one of the following two ways: (1) passive trace-based learning approach; (2) active-learning based approach. The effectiveness of learning the protocol FSM with the trace-based approach, however, critically hinges on the diversity and coverage of the input traces. Although it is possible to obtain a large number of crowd-sourced traces to be used as input to the passive learning algorithm, these traces often only exercise expected behavior and miss out on capturing corner-cases where noncompliance occurs.

DIKEUE thus relies on an active FSM learning approach for which we use an existing automated black-box FSM learning technique [56–58]. Our FSM Learner starts from the UE's initial state, and using a controlled LTE network, sends *queries* (i.e., sequences of over-the-air protocol messages) to the device-under-test; dubbed *System Under Learning* (SUL). Based on the observed *responses* to the queries (i.e., sequence of protocol messages from the SUL), it infers the FSM of the underlying implementation. Although automata learning has been used in the context of testing various protocols [39, 40, 59–63], applying it in 4G LTE domain requires taking into account some protocol-specific challenges. First, 4G LTE is a complex *multi-layer* protocol. Second, protocols in each layer entail multiple timers and re-transmission counters, whose values are unobservable from the output interface, making the device's protocol FSM seem to behave in a nondeterministic way, violating one of the pre-requisites of applying active, black-box automata learning approaches (i.e., deterministic behavior). Third, after each sequence of messages, the SUL needs to reset *transparently*–deleting all internal states and context information without any modification on the device. Fourth, in addition to the general behavior, i.e., regular protocol flow of the SUL, the learner needs to infer the implementation-specific atypical behavior, e.g., response to a replay packet, to further aid the noncompliance checking. Finally, a substantial amount of engineering effort is needed to develop an *adapter*, which facilitates the communication between the learning algorithm and the SUL by converting abstract symbols to over-the-air messages. We rely on some existing efforts and also develop some new insights to address the above aspects.

Once we have extracted the FSMs of the devices' LTE control-plane protocol implementations, DIKEUE takes advantage of having access to multiple COTS UEs. Particularly, it relies on the concept of *deviant behavior* as a proxy for identifying noncompliant behavior in a property-agnostic way during the differential analysis of two FSMs belonging to two different UEs. In our context, a deviant behavior is a sequence of inputs for which the two FSMs that are being compared, when executed from the initial state, generate distinct output sequences. When comparing two FSMs, if a deviant behavior is observed, then it is clear that at least one of the implementations is noncompliant even though it is not clear which one. These deviant traces are then triaged through consultation with cellular protocol standards to classify them into one of the following two root causes: (1) the implementation

deviates from a clear specification; (2) the specification suffers from under-specification or ambiguity. Automatic identification of diverse deviant traces between any two FSMs, however, is challenging, especially in the presence of loops in the FSMs. DIKEUE addresses this challenge by reducing the problem of identifying deviant behavior in two different FSMs to a model checking problem. The model checking problem checks the safety properties of a model which parallelly composes the two FSMs under analysis.

Findings. To test the effectiveness of our system, we evaluate DIKEUE with 14 popular UEs from 5 vendors, including Qualcomm, MediaTek, Exynos, HiSilicon, and Intel. DIKEUE has uncovered 15 new distinct deviations and two previously reported issues. Some of these issues are only evident when the implementation reaches a specific state and can only be uncovered through stateful testing. We classify these deviant behavior based on root causes and impacts. Among the reported issues 11 are exploitable, and 3 are susceptible to interoperability issues between UEs and network operators. The implications of these deviations include implementations accepting replayed messages and plaintext messages, exposing private information, and causing denial-of-service attacks.

Responsible disclosure. We have responsibly disclosed our findings to all the affected stakeholders (i.e., GSMA, Qualcomm, MediaTek, Exynos, HiSilicon, Intel, Apple, Samsung, Huawei, HTC, Android). GSMA has acknowledged with CVD-2021-0050 for all the 15 newly discovered deviating behavior. The affected vendors are in the process of patching the issues in future versions.

Contributions. To summarize, this work makes the following technical contributions:

- We propose DIKEUE— which, to the best of our knowledge, is the first tool that designs a black-box FSM inference module to automatically infer the FSM from a UE's implementation without any manual interventions or modifications to the devices. DIKEUE will be publicly available at [64] after all the affected UEs are patched and the responsible disclosure is completed.

- We design an FSM equivalence checking algorithm that automatically detects and reports diverse deviant behavior of two FSMs by reducing it to a symbolic model checking problem.

- We evaluate DIKEUE with 14 different devices from 5 vendors, and demonstrate that it can uncover 17 deviant behaviors, including 11 exploitable weaknesses and 3 interoperability issues.

4.1 Background

DIKEUE infers the model of a protocol implementation in the form of a Mealy machine, also known as a finite state machine (FSM). In the following, we define a Mealy machine, provide an overview of model learning, and discuss relevant technologies in 4G LTE.

Finite State Machine (FSM). We define an FSM ($\mathcal{M}$) 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. Σ and Λ are the sets of input and output alphabets representing the set of possible input and output messages, respectively. The transition relation $\Psi : \mathcal{S} \times \Sigma \to \mathcal{S}$ maps the pair of a current state and an input symbol to the corresponding next state, and the output relationship $\Omega : \mathcal{S} \times \Sigma \to \Lambda$ maps the pair of a current state and an input symbol to the corresponding output symbol.

4.1.1 Active Automata Learning

Active automata learning approaches such as L^* aim to learn the deterministic finite automata (DFA) representation of an unknown regular language $\mathcal{L}$ for a given input alphabet from a minimal adequate teacher (MAT). The learner asks the MAT the following two types of queries, namely, *membership queries* and *equivalence queries*. A membership query is of the form $x \in^? \mathcal{L}$ (i.e., the learner wants to check whether a concrete string x is a member of the unknown language $\mathcal{L}$). The MAT responds with a *yes* iff $x \in \mathcal{L}$; otherwise, it responds with a *no*. An equivalence query, on the other hand, checks whether a hypothesis DFA $\mathcal{H}$ is equivalent to the DFA of the language $\mathcal{L}$ denoted by $D_{\mathcal{L}}$, i.e., both $\mathcal{H}$ and $D_{\mathcal{L}}$ accept the same set of strings. If $\mathcal{H}$ is not equivalent to $D_{\mathcal{L}}$, then the MAT should provide a concrete string y that is accepted by one but rejected by another as a counterexample.

A majority of the automata learning approaches work iteratively in the following two stages [65, 66]. ***Hypothesis construction stage:*** In this stage, the learner asks a series of membership queries to build a closed and consistent hypothesis DFA $\mathcal{H}$ for $\mathcal{L}$. ***Model***

validation stage: In this stage, the learner poses an equivalence query to the MAT to check whether $\mathcal{H}$ is equivalent to $D_{\mathcal{L}}$. If $\mathcal{H}$ is equivalent to $D_{\mathcal{L}}$, the learning concludes, and $\mathcal{H}$ is provided as the learned DFA. Otherwise, the approach goes back to the first stage to create a new hypothesis based on the provided counterexample and additional membership queries. This learning approach can be extended in the standard way [67] to learn Mealy machines instead of a DFA.

In practice, directly applying active automata learning as discussed above is not feasible. This is because obtaining a MAT with the capability of answering an equivalence query (needed for the model validation stage) is absent in the majority of the cases. One can, however, *approximate* an equivalence query with a series of carefully constructed membership queries [68]. We refer to this relaxed MAT (without equivalence query stage) as the *System-Under-Learning* (SUL). Due to the approximate equivalence checking, the learned model in such a case is not guaranteed to be *correct* but instead assured to be *observationally equivalent* (i.e., the learned and original model behave equivalently for strings whose membership results the learner has observed during learning).

4.2 Design of DIKEUE

We now present the threat model, formally define our problem, discuss the workflow of DIKEUE, and outline the challenges of designing DIKEUE as well as insights on addressing them.

4.2.1 Threat Model

We consider the communication channels between the UE and base station, and between the UE and core network subjected to adversarial influence. Our attacker model follows the one defined by previous works [8, 15, 25, 32] and comprises of either a passive or an active attacker that differs in capabilities and restrictions. The passive attacker can observe arbitrary communication between the UE and the LTE network over the radio layer. The active attacker can additionally intercept, replay, modify, drop or delay message, without knowing the key material of devices not owned by the attacker. Moreover, the attacker

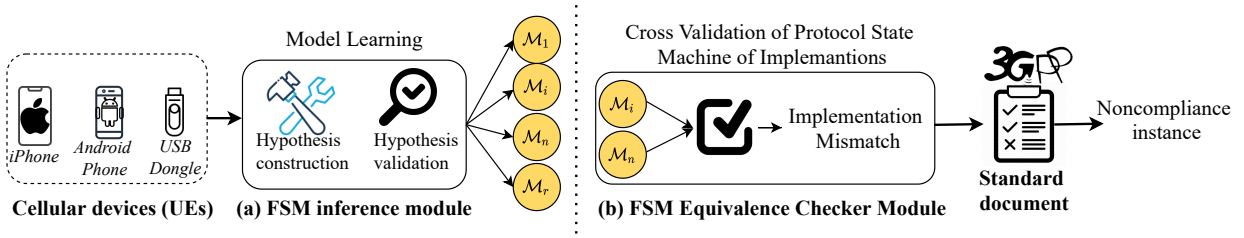


Figure 4.1. Workflow of DIKEUE

can deploy a fake LTE base station impersonating a real LTE network. Note that, the cryptographic constructs are considered to be perfectly secure. We also consider the core network components, target user's UE, and the USIM to be part of the trusted computing base and free of adversarial influence.

4.2.2 Problem Statement and Approach Skeleton

Problem. DIKEUE aims to solve the following noncompliance problem. Given black-box access to a LTE control-plane protocol implementation I of a UE, the noncompliance asks is there an input sequence $\boldsymbol{\pi}_{\mathrm{i}} = \sigma_1\sigma_2\sigma_3\ldots\sigma_m$ where $\sigma_{\mathrm{j}} \in \Sigma$ such that the output sequence generated by I after feeding $\boldsymbol{\pi}_{\mathrm{i}}$ as input, $\gamma_{\mathrm{i}} = \lambda_1\lambda_2\lambda_3\ldots\lambda_m$ in which $\lambda_{\mathrm{j}} \in \Lambda$, is not the one prescribed by the standard.

Approach skeleton. For addressing the above noncompliance problem, DIKEUE takes advantage of its black-box access to multiple UE implementations $\langle \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_n \rangle$. It also requires that the input and output interfaces of these implementations are the same; that is, the set of input and output symbols are Σ and Λ across all implementations. Suppose the implementations simulate the following protocol state machines $\langle \mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n \rangle$, respectively. DIKEUE's approach has the following two steps: ❶ For each implementation $\mathcal{I}_{\mathrm{j}}$, using active automata learning, extract an approximation $\mathcal{M}^*_{\mathrm{j}}$ of the underlying FSM $\mathcal{M}_{\mathrm{j}}$; ❷ For each pair of extracted FSM $\mathcal{M}^*_{\mathrm{j}}$ and $\mathcal{M}^*_k$, find input sequences of the form $\boldsymbol{\pi}_{\mathrm{i}}$ such that when it is fed as input to both $\mathcal{M}^*_{\mathrm{j}}$ and $\mathcal{M}^*_k$, the output sequences they generate are γ_{j} and γ_k,

respectively, and $\gamma_j \neq \gamma_k$. In such a case, π_i is called a ***deviant-behavior-inducing input sequence***, and it also serves as an example of a noncompliant behavior.

4.2.3 Workflow of DIKEUE

DIKEUE (shown in Figure 5.1) works mainly with two components, namely, the FSM inference module and the FSM equivalence checker module. The FSM inference module requires black-box access to one or more UE implementations to be checked for noncompliance. For each of these implementations, it uses active automata learning to extract a protocol state machine of the input UE implementation. Once the protocol state machines of all the implementations have been extracted, each pair of the state machines are fed into the FSM equivalence checker module. The FSM equivalence checker module then tries to identify a diverse set of deviant-behavior-inducing input sequences. Each of these sequences denotes a sequence of input protocol messages for which the two input state machines disagree. For each such input sequence, the outputs of the two state machines are manually compared to the standard to identify which of these implementations deviate from the standard; identifying the noncompliant behavior which is displayed as output.

4.2.4 Challenges and Insights

For realizing the skeleton approach for noncompliance detection presented just above, DIKEUE has to address the following two sets of challenges. In addition, we also discuss how we address these challenges using existing approaches as well as novel insights.

4.2.5 Learning the 4G LTE Protocol State Machine of a UE

As we have hinted before, we use an existing active automata learning algorithm for extracting the 4G LTE protocol state machine of a UE. Effectively applying active automata learning for 4G LTE protocol machine has the following three classes of challenges.

Challenge C_1: Satisfying Pre-requisites of Automata Learning Algorithms. The first challenge involves ensuring that the (implicit) prerequisites for active automata learning are satisfied so that one can apply L* like algorithms for learning the protocol state

machine. There are three prerequisites for applying L* like algorithms, namely, (P_1) identifying the input and output alphabet, (P_2) ensuring that the SUL is deterministic, and (P_3) the membership queries are run from the known initial state of the protocol.

First, the number of input symbols relies on the kinds of considered protocol messages, procedures, and also predicates over messages. Once the input symbols are selected, then the output symbols can be obtained from the protocol specification. Note that, the considered input symbols are exponential to the number of considered predicates over messages and linear to the kinds of messages. Let us consider an example protocol that has three kinds of messages k_1, k_2, and k_3, but the protocol transition conditions also rely on two predicates over messages $p_1(\cdot)$ and $p_2(\cdot)$. In this case, we can have a total of 12 ($= 3 \times 2 \times 2$) input symbols based on which message kind (synonymously, message type) it is and whether $p_1(\cdot)$ and $p_2(\cdot)$ are true. As an example, two different input symbols are needed to capture the following two conditions, namely, $message\_kind(m) = k_1 \wedge p_1(m) \wedge p_2(m)$ and $message\_kind(m) = k_1 \wedge \neg p_1(m) \wedge p_2(m)$ (m is a variable of type message). There are 12 such possible conditions requiring 12 input symbols. Note that, the size of the input alphabet impacts both termination of the learning and coverage of learned protocol behavior. The larger the alphabet size the more of the protocol behavior will be covered, but it will negatively impact the termination.

Second, despite the deterministic nature of the 4G LTE protocol state machine of a UE, due to the unreliable over-the-air (OTA) transmission, link-failures, re-transmissions, and timers, the outputs observed from the UE may not be deterministic, violating P_2. Such observational-nondeterminism causes the learned protocol state machine to never converge as it spawns new states/transitions with a new observation of nondeterministic behavior.

Finally, in case of 4G LTE, satisfying P_3 requires deleting all the keys, resynchronizing the USIM sequence number, and taking the cellular device to the initial registration phase, which require time and manual intervention to turn on/off the device and deleting information from non-volatile memory.

LTE-specific Insight for P_1. For input symbols, we consider a total of 16 protocol message kinds and the following four unary predicates over messages: is_replay($\cdot$), is_plain_text($\cdot$), is_plain_h− eader($\cdot$), and is_null_security($\cdot$). This gives us a *potential* in-

put alphabet size of 256 (= $16 \times 2 \times 2 \times 2 \times 2$). We also need to consider an additional 5 input symbols that trigger different procedures. As an example, one such input symbol is to induce the UE to send an *attach_request* message and initiate the protocol session. The different predicates we consider have the following semantics. is_replay(m) is true *iff* m is replay of a previously sent message. is_plain_text(m) is true *iff* the content of m is in clear. is_plain_header(m) is true *iff* the content of m should be encrypted and integrity protected with value of the message authentication code (MAC) to be set to 0 but the value of security header refers to a plaintext message message. Finally, is_null_security(m) is true *iff* null security is chosen as the chosen ciphersuite in the *sm_command* message. The output symbols are chosen accordingly from these possible input symbols.

Existing insight on satisfying P_2**.** For addressing the observational nondeterministic behavior of a UE, we conservatively pose each membership query twice. In case the outputs for both these membership queries agree, we update the observational table. In case of a conflict, however, we use the existing approach of using a *majority voting scheme* to resolve conflicting output sequences [69].

Novel LTE-specific insight on satisfying P_3**.** For satisfying P_3, we discovered a protocol-specific behavior to transparently reset the device and take it to an initial state. Having a software solution allows us to avoid the expensive approach of manually rebooting the device; positively impacting the termination of learning.

Challenge C_2: Balancing Termination and Coverage of Learning. Another major challenging aspect of effectively applying automata learning for extracting the 4G LTE protocol state machine of a UE is achieving the right balance between termination and coverage. On one hand, aiming to achieve a high coverage of the behavior negatively impacts the termination. Premature termination, on the other hand, negatively impacts coverage. The termination of the learning algorithm is impacted by the following factors: (1) number of posed membership queries (reliant on the input alphabet size); (2) the time to run each membership query and obtaining a response; (3) the time it takes to resolve observational nondeterminism.

Novel LTE-specific insight of input alphabet selection. Although we can potentially have a total of 261 ($= 256 + 5$) input symbols, some of the input symbols are irrelevant. As an example, consider a condition where $message\_kind(m) \neq sm\_command$ in which case the value of the predicate $\mathsf{is\_null\_security}(m)$ is not relevant as it only applies to the *sm_command* message. In addition, to reduce the model learning time, we *heuristically* prune away other input symbols that may not trigger interesting security-sensitive behavior. After pruning, we end up with a list of 35 input symbols which is much smaller than the original set of 261.

Novel LTE-specific insight of context checker. We develop a context-checker with a set of invariants to automatically deduce outputs for certain input message sequences posed as membership queries without having to run them in the UE. These invariants are conservative rules (i.e., ruling out certain infeasible orderings of protocol messages) that one can reasonably expect a UE to satisfy (e.g., not receiving certain protocol packets without an established connection). Input sequences violating these invariants can be considered to have the output sequence $null\_action^n$ where n is the length of the input message sequence. Note that, *null_action* is a special output symbol that refers to the UE not generating any outputs.

Existing insight on caching results. Running a query in the device is expensive. We thus follow an existing approach [70, 71] of maintaining a cache of membership queries, i.e., input sequences and their corresponding outputs encountered during the *hypothesis construction* stage. Equivalence queries posed during *model validation* stage are first consulted with the cache. If the cache is hit, then the response stored in the cache is used. Note that, the cache is not used during the hypothesis construction stage.

Challenge C_3: Designing a Protocol-specific Adapter. The final challenge for applying active automata learning in the context of 4G LTE protocol state machines involve developing a 4G LTE-specific adapter. The adapter facilitates communication between the learner and the UE device. It needs to convert the abstract input symbols in the membership queries to concrete OTA packets and send them to the UE. In the same vein, it also needs to decode the response from the UE and convert it back to abstract output symbols

comprehensible to the learner. Developing such a 4G LTE-specific adapter is challenging because protocol layers are intertwined and have strong temporal correlations among their operations. As an example, some NAS layer messages can only be sent after particular RRC layer messages, and vice versa. Also, messages of both layers contain timers and retransmissions but, internal protocol states, e.g., transmission failures and timeouts, are not observable from the input/output messages. In addition, for analyzing *communication* and *mobility management* protocols, the adapter needs to trigger certain behavior and corner cases in the UE that pose physical constraints on the UE. For instance, testing handover scenarios requires the UE to be physically moved between multiple base stations, which is not practical and non-trivial to test in any controlled environment.

LTE-specific adapter. We have developed a LTE-specific adapter by enhancing an open-source protocol stack that can transparently send and receive messages based on the directions of the learner. The adapter can handle the complex multi-level, stateful interactions in 4G LTE, including different error conditions.

Novel LTE-specific insight on triggering complex operations. We developed an adapter that can trigger complex 4G LTE behavior in the software that would otherwise require physically moving the UE, e.g., similar to ones for analyzing the handover procedure.

Identifying Noncompliance from Protocol State Machines: Recall that, once we have extracted the protocol state machines of the UE implementations under test, we use differential testing of pairwise protocol state machines from different implementations to identify deviant-behavior-inducing input sequences [72, 73]. We use these input sequences as a proxy for noncompliant behavior. The main challenge for achieving this goal is how to *automatically* identify a *diverse* set of deviant-behavior-inducing input sequences. Existing equivalence checking approaches are insufficient for our purpose as they neither have the notion of diversity nor the capability to provide multiple deviant-behavior-inducing input sequences.

Novel insight on differential testing. We propose a notion of diversity classes for deviant-behavior-inducing input sequences (see Section 4.4). We use this notion of diversity classes to develop a novel approach that reduces identifying deviant-behavior-inducing input sequences to a model checking problem. This approach enables us to not only *automatically*

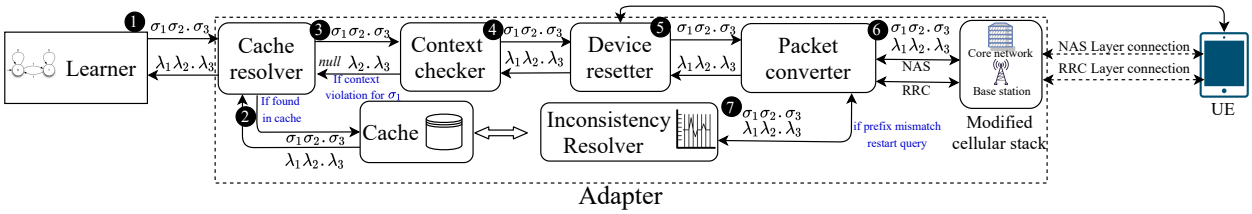


Figure 4.2. Flow of query in DIKEUE's FSM inference module

identify deviant-behavior-inducing input sequences from different diversity classes but also identify different instances from the same class.

4.3 FSM inference module

We now explain in details the components that leverage LTE-specific insights to enable a practical FSM inference module.

4.3.1 Learner

Following the model learning algorithm [66], the learner systematically generates queries as sequences of input alphabets, and based on the outputs, infers the underlying FSM.

Taming the time and state explosion with alphabet set optimization: The time and the number of queries required to learn the model are directly proportional to the number of input alphabets. We, therefore, first leverage LTE-specific insights to reduce the potential input alphabet set of 261 input symbols (Section 4.2.4). We discard the symbols that are irrelevant in the context of LTE. For example, is_null_security(m) does not apply to messages other than RRC and NAS layers' *sm_command* messages. Also, some potential symbols generated by combining multiple predicates together eventually refer to the same symbol. To illustrate, the following two conditions yield the same input symbol: (1) $message\_kind(m)$= *identity_request*$\wedge\neg$is_replay(m) $\wedge$is_plain_text(m) $\wedge$ $\neg$is_plain_header (m) $\wedge\neg$is_null_security(m); (2) $message\_kind(m)$ = *identity_request* $\wedge$is_replay(m) $\wedge$ is_plain _text(m) $\wedge$ $\neg$is_plain_header(m) $\wedge\neg$is _null_security(m).

Table 4.1. : List of input symbols and possible output symbols for each of them for NAS layer. From the input symbols from predicates column only blue color symbols are included in the optimized input alphabet set.
*Protected implies ¬is_plain_header(m) meaning the message is integrity protected and encrypted
± Replay messages are only true for protected messages, plain text messages do not have sequence numbers and replay protection

Message	Input Symbols (After irrelevant message pruning)	Input Symbols (After final optimization)	Output Symbols (Λ)
NAS			
Enable Attach Request	*enable_attach*	*enable_attach*	*attach_request*
Identity Request	*identity_request_replay*±, *identity_request_plain_text*, *identity_request_plain_header*, *identity_request_protected**	*identity_request_plain_text*	*identity_response*
Authentication Request	*auth_request_replay*, *auth_request_plain_text*, *auth_request_protected*, *auth_request_plain_header*	*auth_request_plain_text*	*auth_response*, *auth_MAC_failure*, *auth_seq_failure*
Security Mode Command	*sm_command_replay*, *sm_command_plain_text*, *sm_command_plain_header*, *sm_command_protected*, *sm_command_null_security*	*sm_command_replay*, *sm_command_plain_text*, *sm_command_plain_header*, *sm_command_protected*, *sm_command_null_security*	*sm_complete*, *sm_reject*
Attach Accept	*attach_accept_protected*, *attach_accept_replay*, *attach_accept_plain_text*, *attach_accept_plain_header*	*attach_accept_protected* *attach_accept_plain_text*	*attach_complete*
Enable Tracking Area Update	*enable_tau*	*enable_tau*	*tau_request*
Tracking Area Update Accept	*tau_accept_replay*, *tau_accept_plain_text*, *tau_accept_protected*, *tau_accept_plain_header*	*tau_accept_protected* *tau_accept_plain_header*	*tau_complete*
GUTI Reallocation Command	*GUTI_reallocation_replay*, *GUTI_reallocation_plain_header*, *GUTI_reallocation_protected*, *GUTI_reallocation_plain_text*	*GUTI_reallocation_replay*, *GUTI_reallocation_protected*	*GUTI_reallocation_complete*
Downlink NAS Transport	*DL_NAS_tansport_replay*, *DL_NAS_transport_plain_text*, *DL_NAS_transport_plain_header*, *DL_NAS_transport_protected*	*DL_NAS_transport_protected*	*UL_NAS_transport*
Paging	*paging*	*paging*	*service_request*
Authentication Reject	*auth_reject*	*auth_reject*	*null_action*
Tracking Area Update Reject	*tau_reject*	*tau_reject*	*null_action*

Since plaintext messages do not have any replay protection, replaying a previously sent plaintext *identity_request* message is equivalent to sending a new plaintext *identity_request* message. As such, we prune these irrelevant and redundant messages to reduce the alphabet set to 59 symbols (listed in column 2 in Table 4.1 and Table 4.2). Since the predicates are common to most of the messages in both NAS and RRC layers (except *sm_command* and *RRC_sm_command*), to further minimize the input alphabet set, instead of considering these variants for each input symbol, we consider testing one variant symbol per layer. For instance, one replayed symbol in one RRC layer message is enough to test RRC layer replay protection. Since NAS and RRC layers' *sm_command* are special message kinds in LTE as they are used to navigate the protocol from an unprotected state to security protected state and create dependencies among layers [74], we consider these two messages separately and also include their variants into the alphabet set. Hence, in total, our input alphabet set includes 35, as shown in the third column of Table 4.1 and Table 4.2.

Balancing termination and coverage: When the SUL completes exploring the control-plane procedures of our interest, we terminate the learning and take the last inferred model for equivalence checking. From empirical evaluation, we observed the learner needs running queries for up to length 12 to explore the procedures.

4.3.2 Adapter

The adapter acts as a glue between all the components of FSM inference module (shown in Figure 4.2), and builds a reliable interface from the learner to each control-plane layers we want to analyze.

Addressing multi-layer protocol:

The adapter flattens the multi-layer protocol interactions by combining all layers under a central component and controls the interactions of two interfaces between (i) base station and UE, and (ii) core network and UE. Based on messages in queries issued by the learner, it directs the message to appropriate interface and waits until the response or timeout occurs. It thus enables learning multi-layer protocol.

Table 4.2. : List of input symbols and possible output symbols for each of them for RRC layer. From the input symbols from predicates column only blue color symbols are included in the optimized input alphabet set.

Message	**Input Symbols (After irrelevant message pruning)**	**Input Symbols (After final optimization)**	**Output Symbols (Λ)**
RRC			
Enable RRC Connection Request	*enable_RRC_con*	*enable_RRC_con*	*RRC_con_request*
RRC Connection Setup	*RRC_connection_setup_replay*, *RRC_connection_setup_plain_text*, *RRC_connection_setup_protected*, *RRC_connection_setup_plain_header*	*RRC_connection_setup_plain_text* *RRC_connection_setup_plain_header*	*RRC_connection_setup_complete*
RRC Security Mode Command	*RRC_sm_command_replay*, *RRC_sm_command_protected*, *RRC_sm_command_plain_text*, *RRC_sm_command_plain_header*, *RRC_sm_command_null_security*	*RRC_sm_command_replay*, *RRC_sm_command_plain_text*, *RRC_sm_command_plain_header*, *RRC_sm_command_protected*, *RRC_sm_command_null_security*	*RRC_sm_failure*, *RRC_sm_complete*
RRC Connection Reconfiguration	*RRC_reconf_replay*, *RRC_reconf_plain_text*, *RRC_reconf_protected*, *RRC_reconf_plain_header*	*RRC_reconf_replay*, *RRC_reconf_plain_text*	*RRC_reconf_complete*
Enable RRC Reestablishment	*enable_RRC_reest*	*enable_RRC_reest*	*RRC_con_reest_req*
Enable RRC Measurement Report	*enable_RRC_mea_report*	*enable_RRC_mea_report*	*RRC_mea_report*
RRC Connection Reestablishment	*RRC_con_reest_replay*, *RRC_con_reest_plain_text*, *RRC_con_reest_protected*, *RRC_con_reest_plain_header*	*RRC_con_reest_plain_text*, *RRC_con_reest_protected*	*RRC_con_reest_complete*, *RRC_con_reest_reject*
RRC UE Information Request	*RRC_ue_info_req_replay*, *RRC_ue_info_req_protected*, *RRC_ue_info_req_plain_text*, *RRC_ue_info_req_plain_header*	*RRC_ue_info_req_protected*	*RRC_ue_info_req*
RRC Connection Release	*RRC_release*	*RRC_release*	*null_action*

Table 4.3. : Example queries and responses. "." divides the prefix and suffix of the queries and responses.

ID	Query	ID	Output
$\mathcal{Q}1$	*attach_accept* *enable_RRC_con.enable_attach*	$\mathcal{R}1$	*null_action* *RRC_con_request.attach_request*
$\mathcal{Q}2$	*enable_attach enable_RRC_con* *enable_attach.auth_request*	$\mathcal{R}2$	*null_action RRC_con_request* *attach_request.auth_response*
$\mathcal{Q}3$	*enable_RRC_con enable_attach,* *GUTI_reallocation.auth_request*	$\mathcal{R}3$	*RRC_con_request attach_request* *null_action.auth_response*
$\mathcal{Q}4$	*enable_RRC_con enable_attach* *RRC_release* *auth_request.enable_RRC_con*	$\mathcal{R}4$	*RRC_connection_setup attach_request* *null_action null_action.RRC_con_request*
$\mathcal{Q}5$	*enable_RRC_con enable_attach* *auth_request_replay.auth_request*	$\mathcal{R}5$	*RRC_con_request attach_request* *null_action.auth_response*
$\mathcal{Q}6$	*enable_RRC_con enable_attach* *GUTI_reallocation.auth_request*	$\mathcal{R}6$	*RRC_con_request attach_request* *null_action.auth_response*
$\mathcal{Q}7$	*attach_accept enable_RRC_con* *enable_attach.auth_request*	$\mathcal{R}7$	*null_action RRC_con_request null_action.* (query terminated)
		$\mathcal{R}8$	*null_action RRC_con_request* *attach_request.auth_response*

Improving time of learning with context-checker: To enhance the performance of the FSM inference, the adapter tries to minimize the time-consuming OTA transmissions. For this, the adapter is provisioned with a set of invariants extracted from cellular specifications [9–11], which are used to decide if an input symbol's communication context set by previous symbols in the query is valid for OTA transmission. Whenever an input symbol violates the context, it is dropped, and the default– *null_action* is returned immediately. In case an input symbol passes all these context checks, it is transmitted OTA. The invariants defined in the adapter are: ① input symbols corresponding to common control-plane procedures cannot appear before connection establishment symbols. For instance, for $\mathcal{Q}1$ in Table 4.3, the input symbol *attach_accept* is not propagated forward as the control-plane connection has not been established yet with the connection initiation symbols (e.g., *enable_RRC_con* or *enable_attach*). ② Lower layer connection (RRC) has to be established before upper layer (NAS) connection establishment. To illustrate, for $\mathcal{Q}2$, the first *enable_attach* does not have any semantic meaning and will be responded with the default *null_action* symbol; all symbols prior to the first *enable_RRC_con* in a query will thus result in *null_action* as responses. ③ Security protected messages require proper security keys to be established. Turning to Table 4.3, for $\mathcal{Q}3$, the security protected *GUTI_reallocation* message requires key for integrity and encryption. However, before the authentication and security mode com-

mand procedures, session keys have not been established. Therefore, this *GUTI_reallocation* violates the context check and the context-checker will return the default output symbol. ④ After a connection closing symbol, a new connection has to be established before transmitting the subsequent symbols. For example, in the query shown in $\mathcal{Q}4$, after the RRC connection is released, all other symbols do not have any semantic meaning and will not be propagated further until a new connection has been established with *enable_RRC_con* input symbol. ⑤ A replay symbol has to come after its original counterpart. For instance, for $\mathcal{Q}5$, the first *auth_request_replay* does not correspond to anything and will be discarded until an *auth_request* has been received.

Encoding and decoding custom NAS and RRC layer packets containing predicates:

For an input symbol forwarded by the context checker, the packet converter builds the corresponding NAS and RRC layer payload and header based on the current context. For instance, it saves the previously sent packets so that it can replay those packets later. For plain header, plaintext, and null security packets, the packet converter creates the fields as per the input symbol requirements. For example, if a plain header input symbol is received, instead of the usual integrity protected and ciphered header, the message is sent with plain header. For plaintext messages, the packet is crafted by removing the MAC and without encryption. For null security packets, the integrity and encryption algorithms are set to null-integrity (EIA0) and null-encryption (EEA0), respectively.

Triggering complex protocol interactions:

The packet converted in the adapter also has to automatically trigger certain complex interactions, which are often hard to test as they require physical movements of the SUL or manual interventions. For instance, testing handover requires the user to move from one cell/tracking area to another, whereas triggering a service request (e.g., making a phone call and text) warrants a user to tap on the call button of the phone, dial numbers or enter texts. For side-stepping such physical constraints and manual interventions, the converter crafts specialized packets without requiring any mobility or special hardware. To illustrate, if the

learner issues *enable_tracking_area_update* to begin a handoff, the packet converter sends the special RRC connection release message with cause "load re-balancing TAU required". For triggering the service procedure without any manual interaction, the controller crafts *paging* packets and send them to the SUL to trigger a service request. Also, the responses received from the SUL are converted back to the output symbols by the packet converter.

Optimizing queries during model validation with cache:

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 [70, 71]. 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 the repeated queries. For instance, let us assume $\mathcal{Q}6$ is a query generated during the model validation phase, and the previous queries are generated during the hypothesis construction phase. $\mathcal{Q}6$ is checked against queries $\mathcal{Q}1$ - $\mathcal{Q}5$, and as the same query is cached in $\mathcal{Q}3$, $\mathcal{Q}6$ will not be sent, and the saved response $\mathcal{R}3$ will be returned.

Resolving observational non-determinism with inconsistency resolver:

As discussed in Section 4.2.4, a prerequisite for deterministic model learning is to observe consistent behavior of the SUL for the same sequence of input messages. To maintain such consistency, we leverage existing insight from the prior work [69] and develop an *inconsistency resolver* that primarily performs two operations: (i) It lets the adapter run each new query (i.e., not present in the cache) twice. If both the responses are the same, it saves the query in the database. Otherwise, it triggers the adapter to run the query again. The inconsistency resolver applies a majority voting scheme [69] on the results and stores the majority output as a response to the query. (ii) It checks if the prefix of every response (a query and response is divided into prefix and suffix as shown in Table 4.3) is consistent with the previously learned results. To check this, the inconsistency resolver compares the response prefix of each query with the previously reported results saved in the cache. If there is a mismatch, the adapter

restarts this query from scratch. For instance, for $\mathcal{Q}7$ in Table 4.3, the response prefix of the query is not consistent with the previously saved response of $\mathcal{R}1$. In such occurrences, the query $\mathcal{Q}7$ is terminated and started again from scratch. When the prefix of the new response $\mathcal{R}8$ is consistent with the previous result $\mathcal{R}1$, the response is considered valid and saved in the cache.

Transparent reset without manual intervention or rebooting the device:

The *device resetter* resets the SUL to the initial state and clears the security context from the non-volatile memory of the device by only sending an OTA *attach_reject* message with EMM cause#11 “PLMN not allowed". To further ensure that both UE and adapter are synchronized with the same sequence number, the resetter sends *auth_request* to the UE. Nevertheless, as the initial connection has to be initiated by the UE under test, the resetter has to trigger the UE to generate an initial connection request (e.g., *attach_request* for NAS or *RRC_connection_setup* for RRC) without any manual intervention. To achieve this without any modification on the device, for Android devices key press events are simulated through the ADB connection. For iPhones, libimobiledevice– a library to communicate with iPhone to restart the device [**libimobiledevice**] is used, and for USB devices, the device is toggled through the USB connection. Finally, for development boards and LTE dongles, AT commands [**atcommand**] are injected through serial connections.

OTA packet encoding/decoding with modified cellular stack:

We modify an existing open-source cellular stack to set up the components of a base station and a core network that DIKEUE controls. We remove the original FSM implementations of both the NAS and RRC layers from the open-source LTE stack and create direct interfaces with the packet converter to use it only for encoding/decoding lower-layer payloads (e.g., PDCP, RLC, MAC, and PHY) of a packet. The cellular stack receives the concrete values for some specific fields of packets from the packet converter, and communicates with UE through OTA-transmission.

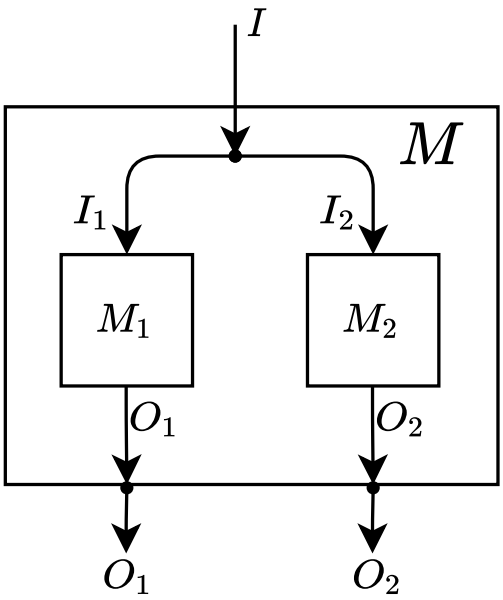


Figure 4.3. Equivalence Checking to Model Checking

4.4 FSM equivalence checker

The FSM equivalence checker module of DIKEUE takes as input two protocol FSMs, in the form of Mealy Machines and automatically identifies a diverse set of deviation-inducing input sequences, if present. In what follows, we assume that the input FSMs have the same input and output alphabet, denoted by Σ and Λ, respectively.

4.4.1 Reduction to Model Checking

We reduce this equivalence checking problem to a model checking problem of a safety property in the following way (see Figure 4.3). For this reduction, a symbolic model checker (e.g., nuXmV [43]) that is able to reason about safety-properties would suffice.

Reduction. Suppose the two FSMs under differential test are denoted by M_1 and M_2. The inputs to these two FSMS (downlink messages they can receive) are denoted by I_1 (for M_1) and I_2 (for M_2), respectively. Similarly, let us denote their outputs (messages they can send) as O_1 (for M_1) and O_2 (for M_2), respectively. We then construct a model M which contains M_1 and M_2 as sub-components. M will take a single symbolic input I which will be fed

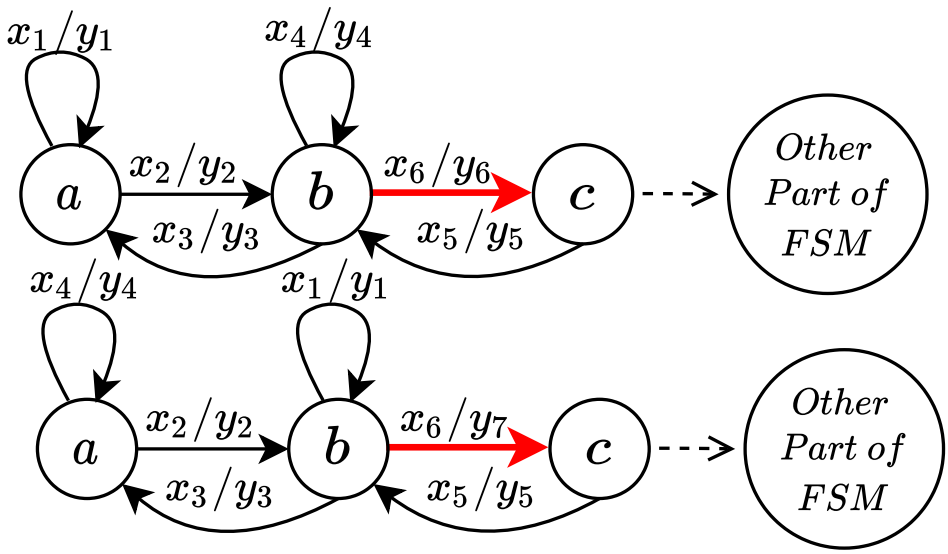


Figure 4.4. FSMs for understanding the challenge for identify diverse deviation-inducing input sequences.

to both I_1 and I_2 (i.e., the same input for both M_1 and M_2). M will have two outputs O_1 and O_2, essentially outputs of M_1 and M_2, respectively. The model M can be viewed as composing M_1 and M_2 with a parallel composition. We then assert the following property of the model M: *It is always the case that O_1 and O_2 should be equal in each step of the execution* (precisely, in linear temporal logic $\square(O_1 = O_2)$). We want to emphasize that the input I (which is essentially I_1 and I_2) is an environmental variable, i.e., we do not need to provide any concrete inputs for I. The model checker aims to find a sequence of I values for which the property is violated (i.e., $O_1 \neq O_2$ in some steps). A counterexample idenfied by the model checker suggests essentially a deviation-inducing input.

4.4.2 Challenge of Obtaining Diverse Deviations

Note that, we are interested in discovering many *diverse deviation-inducing inputs*. If we want the model checker to give us diverse counterexamples, we have to somehow inform it of the concept of diverse counterexamples. If we were to invoke the model checking multiple times, it is highly likely that it will give the same counterexample, the shortest in many cases.

We indeed need the notion of diversity, but it is unclear how to precisely define it. After getting a counterexample c_1, one may consider updating the original property $\square(O_1 = O_2)$ by blocking c_1. This will make the model checker find a different counterexample if present. However, the obtained counterexample may not match our intuitive notion of diversity. To explain this situation, let us consider the following example.

Example. Suppose we have the two partial FSMs M_1 (i.e., the top one) and M_2 (i.e., the bottom one), as shown in Figure 4.4. For this example, let us only focus on the states a, b, and c of M_1 and M_2. The transitions are denoted as $s_{\mathrm{i}} \overset{x_k/y_o}{\rightarrow} s_{\mathrm{j}}$, which refers to a transition that moves the current state from s_{i} to s_{j} after receiving input x_k, and in the process generating output y_o. In the example, M_1 and M_2 behave in the same way for all transitions except for $b \rightarrow c$ (shown in red color). M_1 and M_2 generate two different output messages (i.e., y_6 and y_7, respectively) when taking the transition $b \rightarrow c$ under input x_6. Using the above approach, if we were to ask the model checker to find a counterexample, it would likely give us the input sequence in which both FSMs traverse the following states: *abc*; as it is the shortest one. Now when we block *abc*, the model checker may give a counterexample where M_1 and M_2 traverse states *abbc*; being the next counterexample closest to the previous one. This loop can go on where it spits out a variant of the $(a^+b^+)^+c$ counterexample ('+' signifies one or more occurrences). These counterexamples show the same problem of the transition $b \rightarrow c$.

One may consider removing the transition $b \rightarrow c$ altogether from both M_1 and M_2. This may, however, result in a disconnected model in which the rest of the states become unreachable making it infeasible to find other noncompliance instances infeasible.

4.4.3 Identifying Diverse Deviations

To identify diverse deviation-inducing input sequences, we propose the notion of *diversity classes*. We use this notion to identify different noncompliance instances in a given pair of FSMs.

Definition 4.4.1 (Diversity Class of Deviation-inducing Input Sequences)**.** *Given a fixed set of output symbols* Λ *where* $\mid \Lambda \mid = n$*, there are a total of* $n \times (n-1)$ *possible diversity classes for*

deviation-inducing input sequences; one for each pair of distinct output symbols (i.e., $\langle \lambda_r, \lambda_s \rangle$ where $\lambda_r, \lambda_s \in \Lambda$ and $\lambda_r \neq \lambda_s$). For any pair of FSMs M_1 and M_2, a deviation-inducing input sequence $\boldsymbol{\pi}_{\text{i}} = \sigma_1\sigma_2\sigma_3 \ldots \sigma_m$ is an element of the $\langle \lambda_r, \lambda_s \rangle$-diversity class iff when $\boldsymbol{\pi}_{\text{i}}$ is executed on M_1 and M_2 to obtain output sequences $\gamma^1_{\text{i}} = \lambda^1_1\lambda^1_2\lambda^1_3 \ldots \lambda^1_m$ and $\gamma^2_{\text{i}} = \lambda^2_1\lambda^2_2\lambda^2_3 \ldots \lambda^2_m$, respectively, then there exists a $1 \leq k \leq m$ such that $\lambda^1_k = \lambda_r$ and $\lambda^2_k = \lambda_s$.

As an example, suppose we are given two FSMs M_1 and M_2 for which $\Sigma = \{a, b, c\}$ and $\Lambda = \{1, 2, 3, 4\}$. Let us consider a deviation-inducing input sequence $\boldsymbol{\pi} = abcc$ for M_1 and M_2 for which we obtain the output sequences $\gamma^1 = 1234$ and $\gamma^2 = 1243$ after executing $\boldsymbol{\pi}$ on M_1 and M_2, respectively. $\boldsymbol{\pi}$ is an element of the $\langle 3, 4 \rangle$-diversity class as there exists a $k = 3$ for which $\gamma^1_3 = 3$ and $\gamma^2_3 = 4$. Note that, $\boldsymbol{\pi}$ is also an element of $\langle 4, 3 \rangle$-diversity class as there exists $k = 4$ for which $\gamma^1_4 = 4$ and $\gamma^2_4 = 3$.

We use the above notion of diversity classes to identify a diverse set of deviation-inducing input sequences. Without loss of generality, we use an example to explain our approach. Suppose we are given two FSMs M_1 and M_2 with $\Lambda = \{1, 2, 3\}$. Instead of asserting the safety property $\square(O_1 = O_2)$ in the composed model M (as shown in Figure 4.3), we would pose a series of model checking queries; one for each of the following safety properties: (1) $\square\neg(O_1 = 1 \wedge O_2 = 2)$ (read, it is not the case that at any step of the execution the output of M_1 is 1 whereas the output of M_2 is 2); (2) $\square\neg(O_1 = 1 \wedge O_2 = 3)$; (3) $\square\neg(O_1 = 2 \wedge O_2 = 1)$; (4) $\square\neg(O_1 = 2 \wedge O_2 = 3)$; (5) $\square\neg(O_1 = 3 \wedge O_2 = 1)$; (6) $\square\neg(O_1 = 3 \wedge O_2 = 2)$. Each of the queries aims to find at least an element, if present, for each of the diversity classes. As an example, any violation of property (1) above will result in an input sequence that is part of the $\langle 1, 2 \rangle$-diversity class.

We go a step further by trying to identify multiple elements of each diversity class. Finding other elements of a diversity class is important as the same deviation can happen in different parts of the FSMs. Once we have obtained an element of a given diversity class, for identifying other elements of that diversity class, we use the idea of removing the transition responsible for the deviation from both FSMs (see Section 4.4.2), and posing the appropriate model checking query again. Although removing the transition may result in disconnected FSMs, it is not as disruptive as the approach discussed in Section 4.4.2 because this phenomenon is localized to only a single equivalence class.

Table 4.4. : Additions/modifications to the tools used in DIKEUE.

Component	Tools	Lines of Code
Learner	LearnLib [75]	248 (Java)
Adapter	–	1807 (Java)
Membership Cache	–	507 (Mysql and Java)
Modified cellular stack	srsLTE [41]	∼4000 (C++)
Device resetter	–	640 (Python 2.7)
FSM Equivalence Checker	–	2240 (Python 2.7)

4.5 Implementation

The FSM inference module is implemented on top of LearnLib [75] and srsLTE [41]–an open-source 4G LTE stack. For the learning algorithm, we use TTT [66] as it requires fewer queries compared to other algorithms [76], and for conformance testing, we use Wp-method [68]. We implement our adapter in Java. We use srsLTE v19.10 as the cellular stack to implement our modified core network and base station. We replace the NAS and RRC FSM implementations of the canonical srsLTE stack with our modified stack and create interfaces between the stack and adapter to forward NAS and RRC packets in both directions. The other layers of srsLTE are kept intact. We use USRP B210 as the software-defined radio peripheral for OTA transmission. The FSM equivalence checker is developed using the NuXmv model checker [43] and a python 2.7 script as the wrapper. Table 5.2 summarizes our efforts of modifying the tools and creating new components for DIKEUE.

4.6 Evaluation

To evaluate the performance of DIKEUE, we aim to answer the following research questions in the subsequent sections:

- **RQ1.** How effective is DIKEUE in finding deviant behaviors?
- **RQ2.** How does DIKEUE perform compared to the existing baseline testing approaches?
- **RQ3.** What are the effectiveness and performance of DIKEUE components, i.e., FSM inference module and equivalence checker?

Evaluation setup. We use a laptop with Intel i7-3750QCM CPU and 32 GB DDR3 RAM to run the FSM inference module with USRP. We use the same configuration laptop for FSM equivalence checker.

Devices. We use 14 different COTS devices from 5 vendors (shown in Table 5.3) for evaluation. Our test corpus includes basebands from 5 vendors: Qualcomm, Intel, MediaTek, HiSilicon, and Exynos. The devices range from Android 6.0 to Android 9.0, Apple iPhone XS, USB Wi-Fi Modem, and to a cellular development board.

4.7 Deviations (RQ1)

DIKEUE has been able to uncover 17 distinct deviations in all the 14 devices tested. Among them 15 are new and 2 are uncovered in previous works but on different devices. Based on the root cause, we categorize the issues into two groups: (i) deviations from the standards; (ii) underspecifications. Note that, we consider conflicting specifications as a part of underspecifications. Furthermore, based on the impact we categorize the issues as: *exploitable* attacks and *interoperability* issues. The attacks are constructed manually from the deviant traces. We summarize DIKEUE's findings in Table 4.5 and Table 4.6.

4.7.1 Exploitable deviations

Among the deviations identified by DIKEUE, 11 are exploitable. In the following we discuss some of the issues in detail.

Replayed *GUTI_reallocation*:

We identified the exploitable deviations **E1** and **E2** (from Table 5.8) in total 9 devices from 2 different vendors. In **E2**, the implementation accepts replayed *GUTI_reallocation* anytime after the attach procedure, whereas in **E1** the implementation accepts *GUTI_reallocation* at a specific state– after every *sm_command* message. Note that, all the devices affected by **E2** are also affected by **E16** and accept replayed *sm_command* as well, posing the implementations in vulnerable situations.

Table 4.5. : Deviations identified by DIKEUE. E- exploitable, I- interoperability issue, EI- both exploitable and an interoperability issue, O- other deviating behavior, D- deviation from standards, U- underspecification

Issue	Description	Root cause		Device												
		D	U	Nexus6	HTC1	GalaxyS6	HTC 10	Nexus6P	GalaxyS8+	Pixel 3XL	HuwayeiY5	Honor8X	HuwaeiP8	MiA1	Iphone Xs	USB	Fibocom
NAS																	
(E1) Replayed *GUTI_reallocation* at specific sequence	Accepts replayed *GUTI_reallocation* when sent immediately after a *sm_command*	✓		✔			✓	✔	✔	✔		✔		✔		✔	
(E2) Replayed *GUTI_reallocation* anytime	Accepts replayed *GUTI_reallocation* when sent immediately after a *sm_command*	✓											✔				
(EI3) Plaintext *auth_request*	Accepts plaintext *auth_request* after security context has been established		✓		✔	✔					✔						
(EI4) Plaintext *identity_request*	Accepts plaintext *identity_request* (identification parameter IMSI) after security context has been established		✓		✔	✔					✔						
(E5) Selective replay of *sm_command*	UE accepts replayed *sm_command* up to the completion of of the attach procedure. After attach procedure, the replayed *sm_command* is not accepted anymore	✓				✔											
(O6) *DL_NAS_transport* without RRC security	UE performs Downlink NAS Transport procedure even before RRC layer security has been established		✓			✔									✓		✓
(O7) Attach procedure without RRC security	UE completes the attach procedure before RRC layer security		✓			✔						✓	✓		✓		✓

Table 4.6. : Deviations identified by DIKEUE. E- exploitable, I- interoperability issue, EI- both exploitable and an interoperability issue, O- other deviating behavior, D- deviation from standards, U- underspecification

Issue	Description	Root cause		Device												
		D	U	Nexus6	HTC1	GalaxyS6	HTC 10	Nexus6P	GalaxyS8+	Pixel 3XL	HuwayeiY5	Honor8X	HuwaeiP8	MiA1	Iphone Xs	USB	Fibocom
NAS																	
(O8) *GUTI_reallocation* before attach procedure completion	UE performs *GUTI_reallocation* even before the attach procedure has been completed or RRC security has been established	✓		✔			✔	✔	✔	✔		✔		✔		✔	
(O9) *auth_response* after *sm_reject*	UE replies to replayed *auth_request* even after security mode command procedure	✓		✔		✔	✔	✔	✔	✔				✔	✔	✔	
(O10) *auth_seq_failure* reply	After secure context has been established, some implementations reply with *auth_MAC_failure* while others do not reply		✓		✔						✔						
RRC																	
(E11) Out-of-sequence *RRC_reconf* causes unresponsiveness	*RRC_reconf* before *RRC_sm_command* makes all other symbols unresponsive		✓	✔	✔	✔	✔	✔	✔	✔	✔			✔	✔	✔	✔
(E12) Replayed *RRC_reconf* causes unresponsiveness	Replayed *RRC_reconf* causes the UE to be unresponsive until new attach procedure is started		✓	✔	✔	✔	✔	✔	✔	✔	✔			✔	✔	✔	✔
(E13) Out-of-sequence *RRC_sm_command* causes unresponsiveness	*RRC_sm_command* before NAS *sm_command* makes the device unresponsive		✓			✔											

Table 4.7. : Deviations identified by DIKEUE. E- exploitable, I- interoperability issue, EI- both exploitable and an interoperability issue, O- other deviating behavior, D- deviation from standards, U- underspecification

Issue	Description	Root cause		Device												
		D	U	Nexus6	HTC1	GalaxyS6	HTC 10	Nexus6P	GalaxyS8+	Pixel 3XL	HuwayeiY5	Honor8X	HuwaeiP8	MiA1	Iphone Xs	USB	Fibocom
RRC																	
(E14) Downgraded *RRC_sm_command* causes unresponsiveness	After a downgraded *RRC_sm_command*, the device has to start attach procedure again		✓	✓			✓	✓	✓	✓		✓		✓		✓	
(I15) Overly restrictive *RRC_reconf*	For some UE, *RRC_reconf* works exclusively before or only after the attach procedure is completed		✓		✓						✓	✓	✓				
Previous issues																	
(E16) Replayed *sm_command* [34]	Accepts replayed *sm_command* after security context has been established		✓	✓			✓	✓	✓	✓				✓		✓	
(E17) Downgraded *RRC_sm_command* acceptance [15]	UE accepts downgraded *RRC_sm_command* and bypasses the whole RRC layer security	✓											✓				

Root cause analysis. In TS 24.301 [9], section 4.4.3.2 it is explicitly stated- "*Replay protection must assure that one and the same NAS message is not accepted twice by the receiver. Specially, for a given security context.*" The deviant behavior, therefore, is a clear mismatch from the standards.

Adversary assumptions. To successfully carry out an attack exploiting this vulnerability, the adversary is required to set up a fake base station [8, 25] or Man-in-the-Middle (MitM) relay [32, 55] that can replay previously saved messages.

Attack Description. This vulnerability can be exploited in two ways: (1) The adversary, using a sniffer [32, 55] or MitM relay [55], captures the *GUTI_reallocation* message for a given security context. Later on when the MME sends *GUTI_reallocation* again for refreshing the

Table 4.8. : List of tested devices

Device	OS Version	Baseband
Motorola Nexus 6	Android 7.1.1	Qualcomm APQ8084 Snapdragon 805
HTC One E9+	Android 6.0	Mediatek MT6795M Helio X10
Samsung Galaxy S6	Android 8.0	Exynos 7420 Octa
HTC Desire 10 Lifestyle	Android 6.0	Qualcomm MSM8928 Snapdragon 400
Huawei Nexus 6P	Android 8.0	Qualcomm MSM8994 Snapdragon 810
Samsung Galaxy S8+	Android 9.0	Qualcomm MSM8998 Snapdragon 835
Google Pixel 3 XL	Android 11	Qualcomm SDM845 Snapdragon 845
Huawei Y5 Prime	Android 8.1	Mediatek MT6739
Honor 8X	Android 8.1	HiSilicon Kirin 710
Huawei P8lite	Android 6.0	HiSilicon Kirin 620
Xiaomi Mi A1	Android 9.0	Qualcomm MSM8953 Snapdragon 625
Apple iPhone XS	iOS 12	Intel XMM 7660 (Apple A12 Bionic)
Yoidesu 4G LTE USB WiFi Modem	–	Not known
Fibocom L860-GL	–	Intel XMM 7560

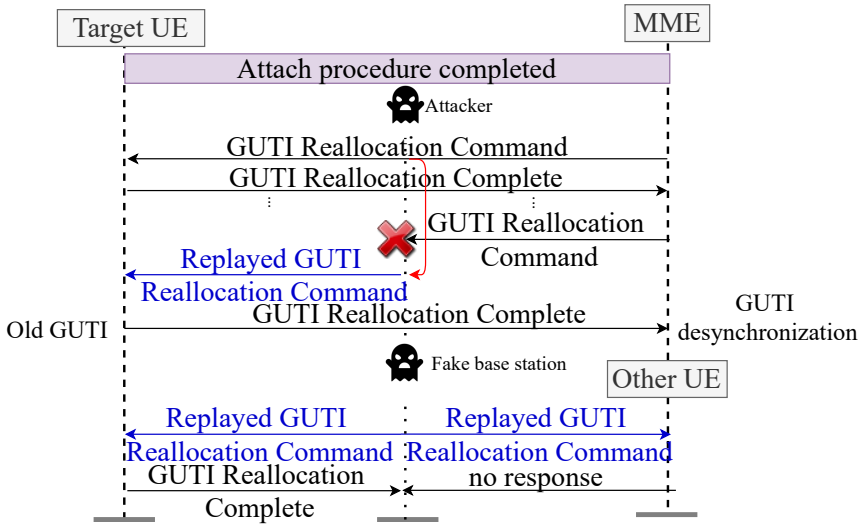


Figure 4.5. Steps of the replayed GUTI reallocation attack

GUTI, the attacker drops this packet and replays the saved *GUTI_reallocation* to the UE. The replayed packet will be successfully accepted by the victim UE. Since the *GUTI_reallocation_complete* message does not contain the agreed-upon GUTI, the MME also assumes the completion of the procedure causing a GUTI mismatch between the UE and the core network; (2) For the second attack, the adversary, using a fake base station, connects to all the UEs in a particular cell area and replays captured *GUTI_reallocation* to all of them. The victim UE accepts this message and responds with *GUTI_reallocation_complete*, whereas all

Table 4.9. : M = Membership and E = Equivalence queries.

Device	M	E	Time (min)	# of states	# of transitions
Motorola Nexus 6	3129	21300	37620	21	556
HTC One E9+	8060	42432	77757	35	1172
Samsung Galaxy S6	3097	10612	21111	20	529
HTC Desire 10 Lifestyle	3129	21300	37676	21	560
Huawei Nexus 6P	3129	21300	37450	21	568
Samsung Galaxy S8+	2908	20961	36762	21	554
Google Pixel 3 XL	3110	20501	36345	21	548
Huawei Y5 Prime	8100	44432	80899	35	114
Honor 8X	4623	16813	33011	28	725
Huawei P8lite	6228	7863	21700	34	1054
Xiaomi Mi A1	3105	21045	37191	21	570
Apple iPhone XS	2340	22450	75361	17	448
4G LTE USB Modem	2905	18332	39953	21	562
Fibocom L860-GL	2322	20470	35099	16	430

the other UEs in the cell do not respond, violating the unlinkability property and exposing the victim's presence in the cell area. The steps of both the attacks are shown in Figure 4.5.

Impact. The first attack causes a GUTI mismatch between the UE and MME and forces a victim user to use a fixed GUTI for an extended time. During this time, if the core network tries paging the UE with new GUTI, the UE will not be able to receive any such notifications or incoming services up to the point the device initiates an attach procedure (which can be done by restarting the phone) or a tracking area update procedure (due to handover), or a service procedure (initiating a service from the phone), or a UE initiated detach procedure (detaching from the core network). Since a UE often does not invoke a tracking area update even up to a week [25], and may not generate service during idle hours, during the period the GUTI remains desynchronized and the UE will keep running into this silent consistent denial-of-service attack. Using the second attack, it is also possible for an adversary to track or detect the presence of a victim UE in a cell utilizing the different responses of the same *GUTI_reallocation* packet.

Plaintext message acceptance after security context:

The deviations **EI3** and **EI4** in Table 4.5 are identified in two different vendors. The affected devices respond to plaintext *identity_request* and *auth_request* messages even if the

security context has been established. No other vendors accept plaintext messages after the establishment of the security context. Note that previous work has shown attacks exploiting the plaintext *identity_request* and *auth_request* messages. But those messages are sent by the adversary *before* the security context is established, whereas our findings show some devices accept those plaintext messages even *after* the security context is set up.

Root cause analysis. Initially, it may appear to be a straightforward deviation from the specification; however, a deeper analysis of the specification paints out a different picture. In TS 24.301 [9]– the specification for the NAS layer, it is stated that plaintext *identity_request* shall be processed by the UE until the secure exchange of NAS messages for the NAS signaling connection. Once the secure exchange of NAS messages has been established, the receiving entity shall not process any plaintext NAS message. However, in the security specification TS 33.401 [10], it is explicitly stated that all NAS signaling messages *except* the listed messages in TS 24.301 (the list includes *identity_request*, *auth_request*) as exceptions shall be integrity-protected. This implies that plaintext *identity_request* and *auth_request* can be accepted by the UE even after the security context has been established. These conflicting standards cause the developers to pick one of the options, and in this case, it seems the security standard (TS 33.401) has been followed. Therefore, conflicting specifications are the root cause of this issue.

Adversary assumptions. The attacker needs the capability to set up a fake base station and craft plaintext messages. We assume the adversary knows the victim UE's C-RNTI [55] but does not need to eavesdrop or capture any messages apriori. The adversary can also overshadow any downlink message between the network and the UE to carry out the attack [**adaptover**].

Attack description. The adversary uses a fake base station to connect to a victim UE and sends a crafted plaintext *auth_request* or *identity_request* message. Alternatively, the adversary can also overshadow any downlink message with plaintext *identity_request* or *auth_request* even after the security context is established. The UE accepts these messages and replies with plaintext *identity_response* containing the IMSI/IMEI of the victim device, or replies with plaintext *auth_response*.

Table 4.10. : Comparison with existing approaches.

Paper	Automatic	Specification analysis	Implementation analysis	Underspecification detection	Stateful
LTEFuzz [8]	✗	✗	✓	✗	✗
LTEInspector [32]	✗	✓	✗	✗	✓
5GReasoner [34]	✗	✓	✗	✗	✓
5G-Authentication [33]	✗	✓	✗	✓	✓
5G-AKA [37]	✗	✓	✗	✓	✓
ProChecker [77]	✗	✓	✓	✓	✓
DIKUE	✓	✓	✓	✓	✓

Impact. The exposure of IMSI even after security context establishment is particularly fatal. This is because the illegal exposure of IMSI provides an edge to the adversary to further track the location of the user or intercept phone calls and SMS using fake base stations [32, 78] or MitM relays [55]. Furthermore, it has been shown that *auth_request* can be used to leak private information, including subscriber activity monitoring [79], launching DoS, and tracking a user [77, 79]. Implementations accepting plaintext *auth_request* are, therefore, vulnerable to these attacks.

Inappropriate state reset.

In exploitable issues **E11-E14** (of Table 4.6), out-of-sequence, downgraded, or replayed RRC layer messages induce unwarranted reset of the affected devices' state machines, causing connection drops.

Root cause analysis and impact. The root cause for all four issues boils down to the underspecification of the standard. In the RRC [11] specification, it is stated that whenever a device receives a message not compatible with the protocol state, the actions are implementation dependent. Due to this underspecification, different implementations treat these non-compatible messages in different ways. Devices that are more restrictive than others reset the FSM state, restart the connection, go through authentication and key agreement again whenever such a non-compatible message is received. This creates the pathway to

unintentional DoS in which an attacker can send such unwarranted (plaintext/replayed/out-of-sequence) messages from a fake base station intermittently.

Adversary assumptions and attack description. Similar to previous attacks, this attack assumes the adversary knows the victim's C-RNTI and can craft plaintext messages or replay previously captured messages. The attacker connects to the victim device and based on the implementation, either sends a replayed or an out-of-sequence or a downgraded or a plaintext RRC message. Each time the attacker sends a new adversarial RRC message, the victim just becomes unresponsive for 4-5 seconds and then reconnects to the actual base station. To maintain a semi-persistent DoS, the attacker will have to keep replaying plaintext/replayed/out-of-sequence messages at every 4-5 seconds interval, causing disruption of regular operations and fast battery depletion of the victim UE.

4.7.2 Interoperability issues

DIKEUE uncovered 3 potential interoperability issues **EI3**, **EI4**, **I15** (shown in Table 4.5 and Table 4.6). Due to space constraints, we discuss only **I15** related to the handling of *RRC_reconf* message. RRC Reconfiguration is the key step in establishing/modifying radio connections between the UE and network. In most of the devices, *RRC_reconf* message is accepted both *before* and *after* the attach procedure to create/modify a radio connection. However, DIKEUE identified two UEs where either *RRC_reconf* message is exclusively accepted either before (MediaTek) or after the attach procedure (HiSilicon) is completed. This may create interoperability issues if the core network sends *RRC_reconf* in the other way around. In such a case, devices from one of the vendors (i.e., MediaTek or HiSilicon) may fall into certain connectivity issues. From our experiments, a major network operator sends the *RRC_reconf* exclusively before the attach procedure is completed. The root cause of these issues is underspecification as TS 36.311 [11] states that the only condition for RRC connection reconfiguration is the UE has to be in the connected state with the base station. But a UE can be in the connected state both *before* and *after* the attach procedure is completed.

4.7.3 Other deviant behaviors

DIKEUE also uncovered deviant behaviors **O6 - O10** in Table 4.5 and Table 4.6, whose implications are not yet certain. For instance, in **O9**, some devices respond to replayed *auth_request* messages even after an *invalid sm_command* is received, whereas other devices do not. In the former case, the device accepts such replayed *auth_request* message until a *valid sm_command* message is received. The acceptance of these replayed messages in that short time interval do not apparently induce state changes or undesired behavior. Nonetheless, these issues resulting from underspecification of the standards should be further analyzed for verifying the impact of these deviant behaviors.

4.7.4 Previous issues

We have also found 2 previously discovered issues (**E16** and **E17**), that have not been resolved yet. For instance, in **E17**, Huawei P8lite accepts downgraded *RRC_sm_command* with the choice of integrity algorithm EIA0. This makes the implementation vulnerable to Man-in-the-Middle attacks. The attack was first identified and described by Rupprecht et al. [15] for a Huawei USB dongle.

4.8 Comparison with Baseline (RQ2)

We compare the effectiveness of DIKEUE with the conformance testing framework defined in the 3GPP specification [80] and property-guided testing by previous approaches [32–34, 37, 77].

4.8.1 Comparison with conformance test cases

We first compare the performance of DIKEUE with the 3GPP conformance test cases [80] based on two criteria: (i) test coverage; (ii) identified deviant behavior issues. Since it is not possible to calculate coverage from a black-box UE implementation, such as an iPhone, we apply DIKEUE to srsUE [41] v20.10.1– the open-source implementation by srsLTE [41]. We use the percentage of lines and functions executed, which are obtained by Gcov [81], as the

indicator for code coverage. Since we are considering only the NAS and RRC layers of the UE implementation, we do not compute the percentage of lines covered with respect to the total number of lines and functions in srsUE. Instead, we calculate the percentage of lines covered within each function and only take into account the functions that are related to our analysis. Therefore, let $L_{\mathrm{e}}(f)$ be the number of lines executed of function f in the srsUE implementation and $L(f)$ be the total number of lines of f, we define the line coverage as: $\sum_{\mathrm{i}=1}^{m} L_{\mathrm{e}}(f_{\mathrm{i}}) / \sum_{\mathrm{i}=1}^{m} L(f_{\mathrm{i}})$ and function coverage as: n/m where $f_1, f_2, \ldots, f_m$ are the functions relevant to NAS and RRC layer and $f_1, f_2, \ldots, f_n$ are functions executed in srsUE. For the baseline coverage, we identify the 88 test cases related to the RRC and NAS analysis from the 3GPP conformance test cases [80] and run them on the srsUE implementation and calculate the line and function coverage of all the test cases. The rationale is to compare how DIKEUE covers compared to the standard defined test cases. The conformance testing has line coverage of 82.58% and function coverage of 83.4375%, whereas DIKEUE performs significantly better with 89.47% line coverage and 89.185% function coverage.

We also apply the 88 test cases to the 14 devices. In case the same conformance test case induces different outputs in different implementations, we note it as a deviant behavior. Through the conformance test cases, only 2 deviating behavior can be captured, compared to the 17 issues automatically identified by DIKEUE.

4.8.2 Comparison with existing LTE works

Table 5.9 compares our approach with existing LTE testing approaches based on several criteria such as automation, specification, implementation analysis, and stateful testing.

Comparison with LTEFuzz

LTEFuzz [8] is a recent approach for dynamic testing of LTE protocol based on stateless dynamic testing with pre-generated test cases. In contrast to LTEFuzz, DIKEUE is different from few angles. First, DIKEUE not only performs dynamic testing but also *automatically* reconstructs the FSM of the underlying UE implementation, allowing in depth analysis. Second, DIKEUE can uncover stateful vulnerabilities, whereas the analysis done by LTEFuzz is

stateless. For instance, it is not possible for LTEFuzz to uncover the *Replayed GUTI_reallocation* (discussed in section 4.7.1) attack discovered by DIKEUE and acknowledged by both Qualcomm and Samsung as a high-severity issue. This is because the attack is triggered only at a specific state of the protocol implementation, not for a *GUTI_reallocation* packet replayed at an arbitrary protocol state. Therefore, the testcases generated by stateless property guided testing of LTEFuzz will not be able to generate such a stateful testcase that can trigger such a vulnerability.

Comparison with property-guided testing

Previous work [32, 33, 37, 77] has applied property-guided testing on FSMs derived from standards [32, 33, 37, 77] or extracted from white-box analysis [77]. To compare DIKEUE with the property-guided testing approaches, we test the properties from previous approaches and run model checking on the FSMs derived from the implementations. As the previous properties are all for the NAS layer only, for a fair comparison, we only test for NAS layer property violations. Through property-guided testing, we identify 3 deviations (**E2**, **E5**, **O9**) among the 10 issues found by DIKEUE in the NAS layer.

4.9 Components performance (RQ3)

We now evaluate the performance of DIKEUE's main components.

4.9.1 FSM inference module performance

Table 4.9 shows the number of states and transitions in the inferred models for 14 devices. Each model includes on an average 22 states and around 600 transitions. There are certain notable exceptions in the model learning phase for different devices. For instance, both the MediaTek phones (HTC One E9+ and Huwaei Y5) require substantially more queries and time to learn the models. This is because MediaTek phones require at most 6 alphabets (i.e., input symbols), including *RRC_sm_command* and *RRC_reconf* in a specific sequence, to complete the attach procedure. Consequently, it takes the learner more time to generate

Table 4.11. : DIKEUE performance of different components. M = Membership queries and E = Equivalence queries.

Approach	# Queries						Time (min)
	Total	M	E	Adapter context-violations	Read from cache	OTA	
DIKEUE	5756	1416	4340	1620	1141	9392	11490
DIKEUE w/o cache	5756	1416	4340	1968	0	11796	15552
DIKEUE w/o optimizations	5756	1416	4340	0	1141	9392	14072
DIKEUE w/o inconsistency resolver	5756	1416	4340	896	1141	5025	N/A*

this specific sequence of messages, and without it none of the future procedures, i.e., GUTI reallocation, tracking area update, service procedure, etc., can proceed.

We now evaluate the effect of different components of the adapter in FSM inference module applying different domain-specific optimizations. The results of these evaluations are shown in Table 4.11.

RQ3.1. Impact of optimal alphabet set:

In case all the feasible input symbols from the predicates are included in the alphabet set, the size of the input alphabet set would be 59 (Table 5.1 shows all the possible symbols from the predicates and the symbols picked for the optimized alphabet set). With our optimized design choice, we reduce the alphabet size to 35. To show the impact of the alphabet size, we infer the model of two different devices of two different vendors with an alphabet set of 35 and 59 respectively up to the attach procedure. Note that with the optimized alphabet set, we are able to reduce the queries required to learn the attach procedure by at least 35%. As the number of queries directly correlates to time, this substantially improves the performance of DIKEUE.

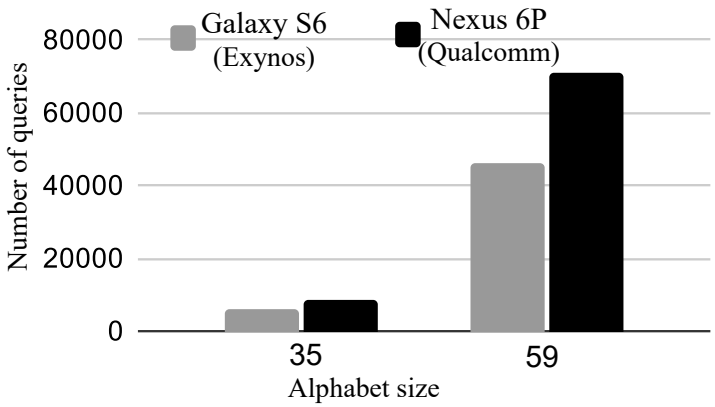


Figure 4.6. Impact of alphabet size

RQ3.2. Adapter context checking:

To evaluate the performance improvement of the context checker, we create a variation of FSM inference module with all the optimizations in the context checker turned off and compare it with the proposed FSM inference module's performance. With optimizations the system found 1620 invariant violations out of 5756 queries up to the attach procedure and thus improved the time performance by 22%.

RQ3.3. Impact of cache:

To evaluate the performance improvement of the cache, we turn off caching and compare it with the original FSM inference module performance. About 19% of the queries are cached, which reduces the over-the-air queries by 20% and improves the performance of the system by 26%.

RQ3.4. Impact of inconsistency-resolver:

To calculate the overhead of the inconsistency resolver, we disable the resolver and compare it with the general system where each query is sent only once and the result is saved in the cache. However, without the inconsistency resolver, after a certain time of the learning

Table 4.12. : Number of unique deviants.

	Nexus6	HTC1	GalaxyS6	HTC 10	Nexus6P	GalaxyS8+	Pixel 3XL	HuaweiY5	Honor8X	Huwaei P8	MiA1	Iphone Xs	USB	Fibocom
Nexus6		8	11	0	0	0	0	8	9	12	0	6	2	6
HTC1			7	8	8	8	8	0	10	10	8	8	8	8
GalaxyS6				11	11	11	11	6	12	12	11	5	12	5
HTC 10					0	0	0	8	9	12	0	6	0	6
Nexus6P						0	0	8	9	12	0	6	0	6
GalaxyS8+							0	8	9	12	0	6	2	6
Pixel 3XL								8	9	12	0	6	0	6
HuwaeiY5									10	10	8	8	8	8
Honor8X										6	10	9	10	9
Huwaei P8											12	10	13	10
MiA1												6	0	6
Iphone Xs													6	0
USB														6
Fibocom														

process, the learner grinds into complete halt due to inconsistencies in the responses (shown as N/A in Table 5.9). At that time, someone has to manually analyze the queries in the cache and remove the inconsistent responses, which requires domain knowledge and time. In our experiments, the learner without inconsistency resolver got stuck 15 times to learn up to the attach procedure.

4.9.2 FSM equivalence checker performance

Table 4.12 presents pairwise all possible deviant behaviors among 14 devices identified by our FSM equivalence checker. For instance, Nexus 6 and Samsung Galaxy S6 have 11 discrepancies, whereas Nexus 6 and Nexus 6P has no discrepancy. This is consistent because Nexus 6 and Nexus 6P have the same vendor (Qualcomm) and a similar version of baseband. Interestingly, among the devices from the same vendor, all the devices behave similarly except HiSilicon. Particularly, two devices from HiSilicon– Huwaei Honor 8X (Kirin 710) and Huwaei P8lite (Kirin 620) behave quite differently and DIKEUE identifies 6 unique differences among them. We manually analyze all the discrepancies and report 17 unique issues in Table 4.5.

To evaluate the timing performance of FSM equivalence checker, we calculate the time required for all pairwise deviation checking 5 times and report the average, max, min and standard deviation in Table 4.13. On an average, FSM equivalence checker takes 42 minutes to find all the deviations.

For further analysis, on the timing performance of the FSM equivalence checker, for each output pair, we calculate the time required for the model checker for repeated queries and take the average of each round. The results are shown in Figure 4.7. After each round of queries, a new invariant is added to the model and the search space is reduced. In case there are multiple traces for the same input, and output pair, the model checker goes deeper into FSMs and it requires much more time. This, in return, contributes to the time of our FSM equivalence checker.

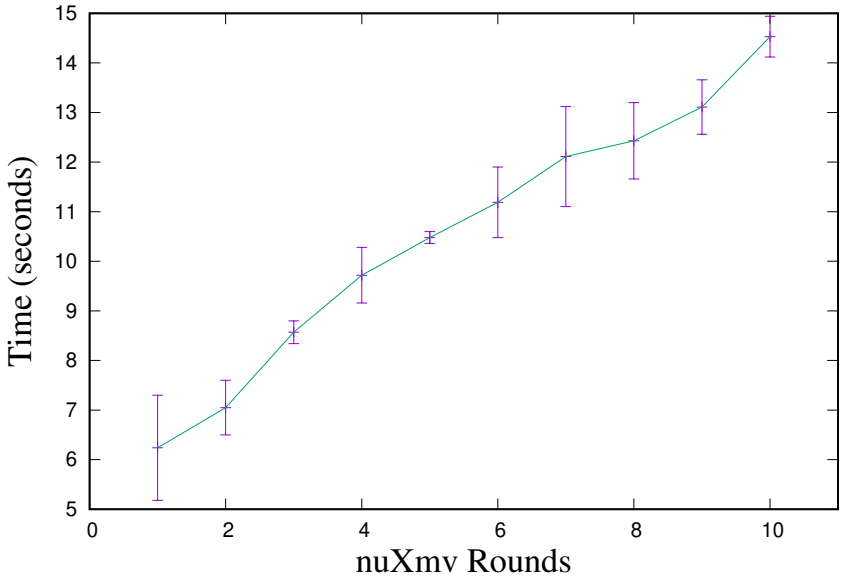


Figure 4.7. Time required for each round of nuXmv query

To evaluate the timing performance of the FSM equivalence checker, we calculate the time required for all the pairwise deviant checking 5 times and report the average, max, min, and standard deviation in Table 4.13. On an average, it takes our FSM equivalence checker 42 minutes to find all the deviations. Furthermore, the timing cost of repeated querying to the model checker is shown in Figure 4.7.

Table 4.13. : Performance of FSM equivalence checker.

Time (min)				
Max	Min	Mean	Median	Standard deviation
82.51	13.08	41.84	35.975	21.3

4.10 Discussion

Limitations of DIKEUE. Similar to any testing paradigm, our approach is incomplete and may result in false negatives due to— (1) not considering all possible message predicates in model learning; (2) precluding infeasible message sequences from testing; (3) use of custom termination condition for model learning to balance scalability and coverage; (4) disconnected FSMs resulting from removing a deviation-inducing transition used for identifying other noncompliance instances of the same diversity class; and (5) inherent limitation of not being able to detect noncompliance instances when both implementations under test are noncompliant to standard but are equivalent. DIKEUE, however, pairwise checks the equivalence of devices drawn from 14 different UE models belonging to 5 vendors (i.e., $\binom{14}{2}$ = 91 pairwise comparisons). It is, therefore, highly unlikely that all devices deviate from the standard in the same way. If one device deviates from standard in a different way than the rest, our equivalence checker can identify it.

Property agnostic. DIKEUE is not entirely property-agnostic if predicates (e.g., is_null_security(m)) of messages are considered as properties. In this work, we consider the typical notion of property [32–34, 37] which refers to stateful end-to-end guarantees of a system. Since DIKEUE does not require any such properties to identify noncompliance instances between any two implementations, we consider DIKEUE to be property agnostic.

Applicability on 5G. To the best of our knowledge, there is no open-source protocol stack for the standalone 5G core network that can be used to develop a 5G-adapter. Therefore, we leave testing of 5G cellular devices with DIKEUE as future work. Our LTE-specific insights, although are based on LTE protocol invariants, are equally applicable to 5G. As an example, similar to LTE, 5G has a multi-layer design with most of the procedures unchanged from

LTE. Thus, the multi-layer protocol handling, context-checker, and other insights will largely remain the same when adopting DIKEUE to 5G.

Parallelization. Parallelizing model learning by distributing different membership queries from a learner to different UEs is plausible. This necessitates complex coordination for maintaining soundness and efficiency of learning which is, however, challenging when inconsistencies are detected due to observational nondeterminism across different instances. In exceptional cases (e.g., a majority of the UE instances having their timers fire at the same time), In that case, it will also take a long time to complete the learning because of the majority voting mechanism culminating in a wrong result. For this to resolve, learning has to revert back. Restarting the learning process from the place of the wrong majority voting result, however, may end up nullifying the performance gain due to parallelization. These complex cases require more investigation and thus we leave it as future work.

Deviant behavior to automatic exploitation. DIKEUE automatically provides traces depicting the deviant implementation specific behavior. This is a concrete evidence of either implementation deviating from the specifications or the standards being underspecified or containing conflicting specifications. Currently, we manually construct the attack strategies from these traces, which we plan to automate in the future.

4.11 Summary

We present DIKEUE which can automatically infer the FSMs of 4G LTE UE implementations, and identify deviant behaviors among the implementations in a property-agnostic way. To show the viability, we applied DIKEUE to 14 COTS devices from 5 vendors. DIKEUE uncovered 15 deviant behaviors; among them 11 are exploitable. We have responsibly disclosed the vulnerabilities to the affected stakeholders and they have acknowledged our findings.

5. BLEDIFF: SALABLE AND PROPERTY-AGNOSTIC NONCOMPLIANCE CHECKING FOR BLE IMPLEMENTATIONS

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 [82]. 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 [7, 30], 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 [14] (3000+ pages) written in natural language with underspecifications, ambiguities, and in some cases conflicting specifications [30]. 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 [7, 16–22, 30] 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 [16, 17, 30]; (ii) only analyze the specifications either manually [18, 19, 83] or using formal verification [20, 30] with the manually extracted abstract protocol model and security properties; (iii) use fuzzing [7, 31] through a hand crafted bug oracle or reference state machine; (iv) use reverse-engineering [21, 22], 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 $\mathcal{B}_{\mathrm{i}}$. 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 $\mathcal{B}_{\mathrm{i}}$ equivalent to $\mathcal{B}_{\mathrm{j}}$ ($\mathrm{i} \neq \mathrm{j}$), failure of which approximates that at least one of $\mathcal{B}_{\mathrm{i}}$ and $\mathcal{B}_{\mathrm{j}}$ 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 [39, 40, 59–64, 84],

in most cases, automata learning has been shown to be viable for only a specific layer [84], or for specific procedures [39, 59] or within a limited scope [71]. 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 [64]. 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 [14] 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 14 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 FSM equivalence checker 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.

5.1 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.

5.1.1 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.

5.1.2 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 [65]. These techniques [65, 66] 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 [68].

5.2 Overview

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

5.2.1 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 [82], this work focuses on noncompliance checking for BLE peripheral implementations only.

5.2.2 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 [7, 18, 19, 30] 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.

5.2.3 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 $\Delta_{\mathrm{i}} = \sigma_1\sigma_2\sigma_3 \ldots \sigma_{\mathrm{j}} \ldots \sigma_m$ where $\sigma_{\mathrm{j}} \in \Sigma$ 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 automat-

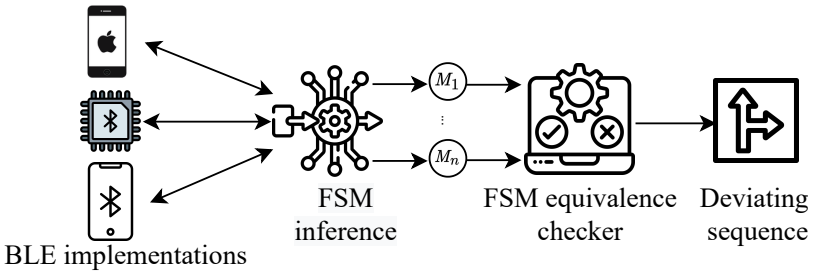


Figure 5.1. Modules of BLEDiff

ically 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 ($\mathcal{M}_\mathrm{j}$) for each BLE implementation $\mathcal{B}_\mathrm{j}$ 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 $\mathcal{M}_\mathrm{j}$ and $\mathcal{M}_k$, find input sequences of the form Δ_i such that for Δ_i, both $\mathcal{M}_\mathrm{j}$ and $\mathcal{M}_k$ generate different output sequences. The output of BLEDiff is Δ_i which induces the deviant behavior.

5.2.4 Challenges of Designing BLEDiff

BLEDiff, as shown in Figure 5.1, works with two main modules: the FSM inference module and the FSM equivalence checker. 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.

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 imple-

mentations 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 [7, 83]. 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 [39, 70] 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 [85]. 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 [39, 59, 64]. The high-level idea is to abstract the peripheral-triggered request messages from the learner (shown in Figure 5.3). 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.

Identifying noncompliance from FSMs

Once we have extracted the protocol state machines of the BLE implementations under test, we need to find noncompliance 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 [14]. 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 [86] 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 [64] 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.

5.3 Detailed Design of BLEDiff

5.3.1 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.

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 5.1 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

Table 5.1. : List of input, adversarial and output symbols. In case there is a timeout the default output symbol is *null_action*

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*	

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 5.3.1).

Separate mappers for each part. One of the crucial components of FSM inference module is the design of the mapper. The mapper acts as a glue between the SUL and the

Figure 5.2. Mapper for BLE Learning Module

learner (shown in Figure 5.2) 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_action* 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 5.3, 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.

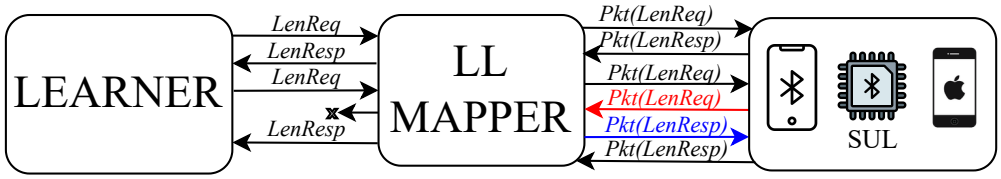


Figure 5.3. Link Layer protocol mapper

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 [70, 71]. 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 [39, 64]. 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 [69, 71] 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 [7] 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 [87] 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 reconnection, 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 5.2. 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.

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 5.3.1), 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

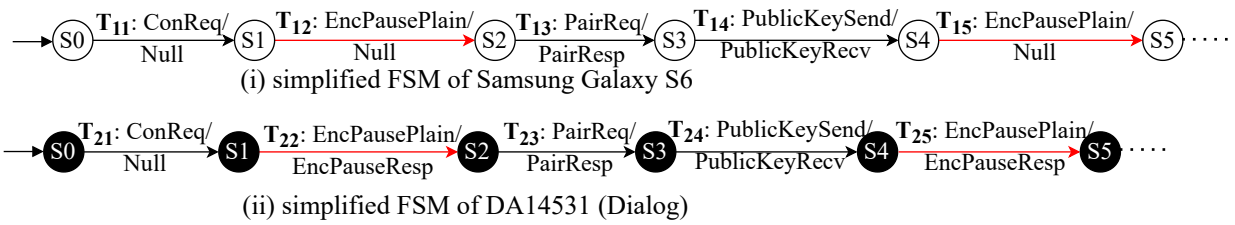


Figure 5.4. FSM equivalence checker

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

5.3.2 BLE Checking Module

For FSM equivalence checker, we reduce the problem to a model checking problem with a safety property.

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 [64], 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 5.4. With the model checking of a safety property, the two different outputs of the same input symbol *PauseEncReqPlainText* will be identified as (*PauseEncResp*, *null_action*). 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 5.4. 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 5.4, we modify the output of T_{22} as *null_action* 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.

5.4 Implementation

The BLE Learning module is implemented on top of LearnLib [75]. For the learning algorithm, we use TTT [66] as it requires fewer queries compared to other algorithms [76], and for conformance testing, we use Wp-method [68]. 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 [7] 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 [88] 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 [43] 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 5.2 summarizes our efforts in modifying the tools and creating new components for BLEDiff.

Table 5.2. : Additions/modifications to the tools used in BLEDiff.

Component	Tools	Lines of Code
Learner	LearnLib [75]	2157 (Java)
Mapper	–	2288 (Java)
Modified BLE stack	SwyenTooth [7]	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)

5.5 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 [35] and previous works on BLE testing? **RQ3.** What is the effectiveness and performance of BLEDiff components: FSM inference module and FSM equivalence checker?

The experimental setup and devices their vendors and BLE versions are described in section 5.6 and Table 5.3 in the Appendix respectively.

5.6 Evaluation Setup.

For all the evaluations **RQ1** - **RQ3**, we use 3 laptops with Intel i7-3750QCM CPU and 32 GB DDR3 RAM. For the FSM inference 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.

5.6.1 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,

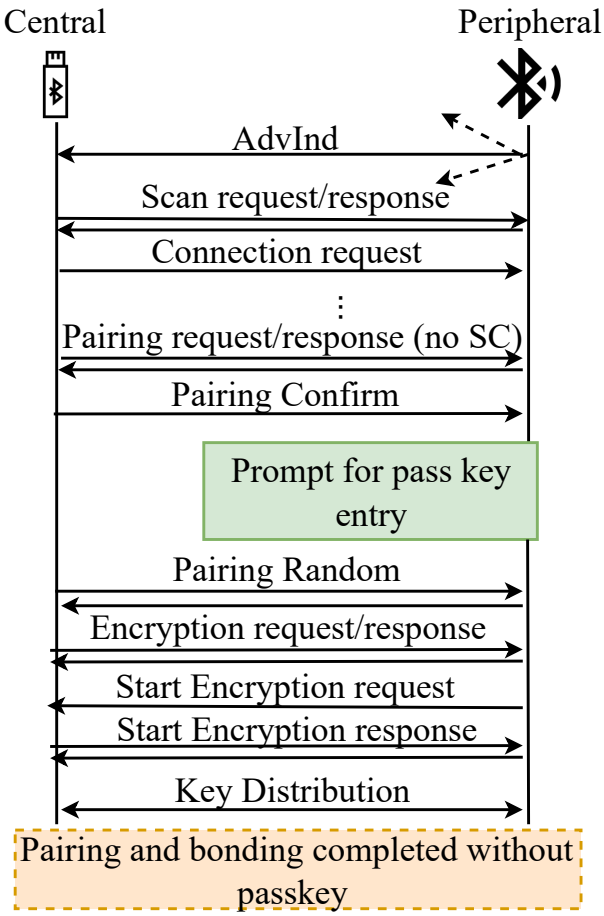


Figure 5.5. Passkey entry bypass

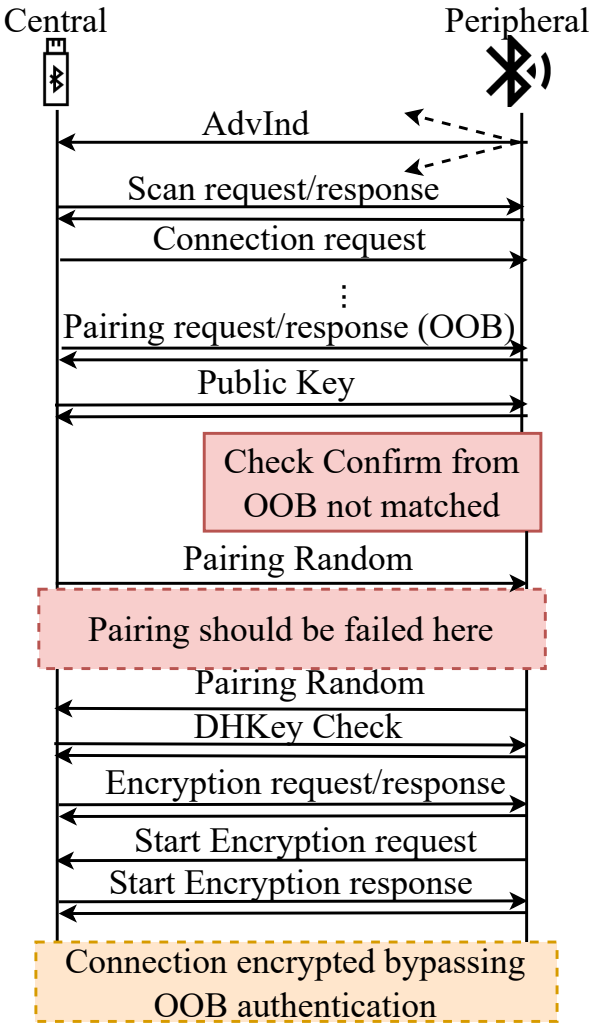


Figure 5.6. Out-of-Band pairing bypass

Table 5.3. : List of tested devices. Fluoride [89] and iOS-BLE-Stack [90] are the BLE stacks for Android and iPhone respectively

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

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

Table 5.4. : Attacks to device mapping

	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																			✓						

are shown in Table 5.8. 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 5.4 in the Appendix.

Attacks

(E1) Passkey Entry Bypass. mong 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. However, we have found 13 implementations where the device completes pairing and bonding without needing to enter the passkey in the device and therefore effectively nullifying all the security protection against MitM attacks. The steps of the deviation are shown in Figure 5.5. In regular workflows of the protocol, after the central sends a *PairConfirmSend* message, a prompt is shown on the peripheral device for passkey entry. In LE legacy pairing, the peripheral

Table 5.5. : Timing comparison. Bold = BLEDiff, non-bold = DIKEUE

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

	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**	–

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. However, in this deviation, before the user can input anything on the prompt, if the central sends a *PairRandomSend*, setting the value of $TK = 0$ 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 the attack is performed. Furthermore, the peripheral implementation completes the pairing and bonding process and enables encryption, all assuming TK to be zero. Surprisingly, one of the devices does not even show the prompt for passkey entry. Thus effectively bypassing the MitM protection put into place through the passkey entry as-

sociation method. 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 5.5, 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.

Table 5.7. : Summary of deviations and time.

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

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 [14, 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 5.6. To make matters worse, the implementations even pass *DHKeyCheckSend* with rb set to zero and complete pairing and bonding altogether.

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 [14, 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 5.7). In the specification, the flow of pairing is clearly attributed, and hence starting the encryption procedure without even

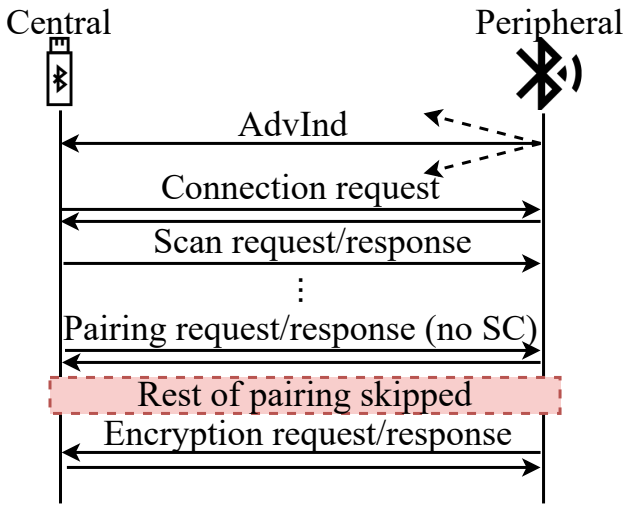


Figure 5.7. Legacy pairing bypass

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.

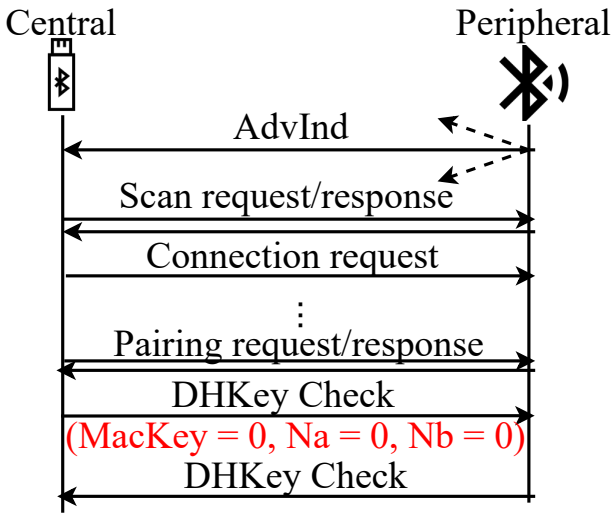


Figure 5.8. Invalid DHKey Check

Table 5.8. : Deviations identified by BLEDiff. E- exploitable, I- interoperability issue, O- other deviating behavior, I- Implementation issue, S- Specification issue.

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

(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 5.8). 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 [7] 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.

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 5.6.1, 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.

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 [31]. Although this is a deviation from the standard, it is not evident how this can be exploited.

5.6.2 Comparison with existing testing approach

We compare the effectiveness of BLEDiff with the BLE conformance or qualification testing framework defined in the BLE standards [35] and the previous approaches on BLE testing [7, 18–21, 31, 91], and summarize the results in Table 5.9. e aim to compute line coverage and function coverage of a device under testing as the metric to evaluate and compare these frameworks and tools. However, we provide a black-box noncompliance checking method, where extracting coverage data is infeasible. To address this issue, we run an open-source BLE implementation, BTstack [92], on an Ubuntu 18.04 machine with BLE version 5.0. We run the testing frameworks for 24 hours and compute line coverage and function coverage using LCOV [93], which is an extension of GCOV [81]. The results of this endeavor is shown in Figure 5.9 in the Appendix and discussed below. Although *line coverage*

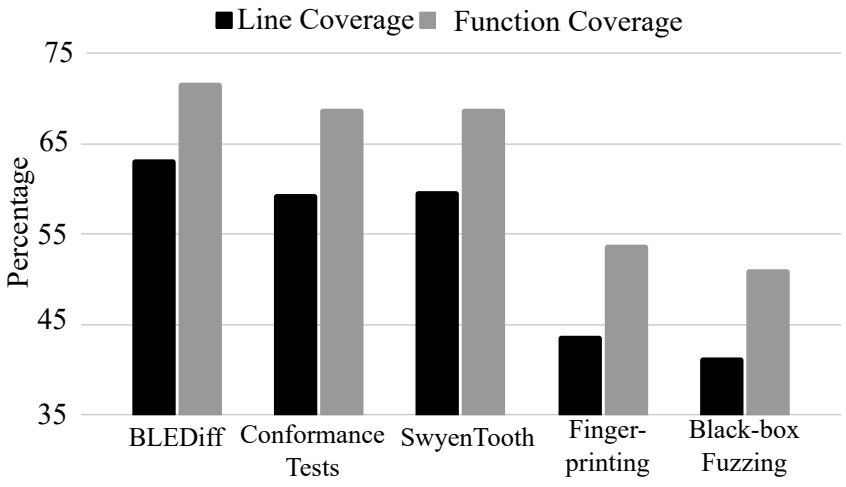


Figure 5.9. Coverage comparison

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 [92] 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 using LCOV [93], which is an extension of GCOV [81]. The results of this endeavor is shown in Figure 5.9 in the Appendix and discussed below.

Conformance or qualification testing framework

BLE standards [35] 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.

Previous approaches on BLE testing

Among the previous works on BLE testing, SwyenTooth [7], Fingerprinting [91], Black-box Fuzzing [31], and Frankenstein [21] are automatic approaches analyzing BLE implementations and do not require manual intervention, in general. On the other hand, BIAS [18], KNOB [19], Model-Driven [20] 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 5.9.

Among the previous works that do automatic testing, we do not calculate coverage for Frankenstein [21] 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 5.9. The comparison shows that BLEDiff is the most effective approach. For SwyenTooth [7], Fingerprinting [91], and Black-box Fuzzing [31], the line cov-

Table 5.9. : Comparison with existing approaches.

Paper	Auto-matic	Specific-ation analysis	Impleme-ntation analysis	Under-specificat-ion detection	Non-compliance-checking
SwyenTooth [7]	✓	✗	✓	✗	✗
BIAS [18]	✗	✓	✗	✗	✗
KNOB [19]	✗	✓	✗	✗	✗
Model-Driven [20]	✗	✓	✗	✗	✗
Fingerprinting [91]	✓	✗	✓	✗	✗
Black-box Fuzzing [31]	✓	✗	✓	✗	✗
Frankenstein [21]	✓	✗	✓	✗	✗
BLEDiff	✓	✓	✓	✓	✓

erage 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.

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 [91], and Black-box Fuzzing [31] 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.

5.6.3 BLEDiff performance

FSM inference module performance

Table 5.12 shows the summary of membership queries, equivalence queries, time, states, and transitions required for FSM inference module to infer the FSM of a BLE implementation. In the worst case, the FSM inference module requires 3 days to learn the FSM, with

Table 5.10. : Time, membership, and equivalence queries. S = States, T = Transitions

Device	Membership			Equivalence			#S			#T			Time			Time	S	T
	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
CY8CPROT063-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

the average being 1.7 days. The device-specific details for all the specific devices are shown in Table 5.10 in the Appendix.

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_action*. 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 5.11.

Table 5.11. : Comparison between divide and conquer learning and general model learning for Nexus 6
* The learner just completed link layer connection

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

FSM equivalence checker performance

To evaluate the performance of FSM equivalence checker, 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 FSM equivalence checker–the equivalence checker designed in the context of 4G LTE called DIKEUE [64]. On average FSM equivalence checker finds 62% more deviant traces compared with the DIKEUE equivalence checker. This is due to the fact that FSM equivalence checker finds deviant traces with higher depths, whereas DIKEUE finds the shortest trace only. On timing FSM equivalence checker 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 5.7. For an interested reader, the detailed pair-wise comparison of the number of deviations and time is shown in Table 5.6 and Table 5.5, respectively in the Appendix.

Table 5.12. : Summary of time, membership, and equivalence queries.

Statistic	Member-ship Queries	Equival-ence Queries	Time (min)	States	Transi-tions
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

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 FSM equivalence checker 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 5.3.2 and Figure 5.4.

5.7 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 5.6, 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 5.6 as compared to 13 unique ones is because:

1. If an input $\mathrm{i_j}$ (e.g., *ConReqTimeoutZero*) in a query $q = \langle \mathrm{i_1 i_2 \ldots i_j \ldots i_m} \rangle$ induces a crash to a device D_1, D_1's outputs for the remaining inputs $\langle \mathrm{i_{j+1} \ldots i_m} \rangle$ in q become *null_action* as D_1 becomes unresponsive after $\mathrm{i_j}$. While comparing D_1 with another device D_2 which did not crash at $\mathrm{i_j}$, BLEDiff yields multiple deviant behavior inducing input sequences, for instance, $\langle \mathrm{i_1 \ldots i_j} \rangle, \langle \mathrm{i_1 \ldots i_{j+1}} \rangle, \ldots, \langle \mathrm{i_1 \ldots i_{j+1} \ldots i_m} \rangle$ for which the root cause is same;

2. If for an input sequence $\langle \mathrm{i_1 i_2 i_3 i_2} \rangle$, 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 $\langle \mathrm{i_1 i_2} \rangle$ and $\langle \mathrm{i_1 i_2 i_3 i_2} \rangle$ 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 [65]. To ensure the same input/output symbols for all devices, BLEDiff defines input/output symbols (shown in Table 5.1) 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.

5.8 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.

6. RELATED WORK

In this chapter, we discuss existing efforts that focus on the security and privacy of cellular network and Bluetooth implementations. Furthermore, we discuss the research that is relevant to this work. The discussion is divided into few broad categories:

Security of Cellular Network Implementations. Previous work on 4G LTE implementation security has either been found by complete manual analysis [23–28, 54, 94] or semi automated testing [8, 15]. Compared to previous works ProChecker can automatically extract the FSM, reason about any security and privacy property, and easily scales to the future generation and new releases of cellular network implementations such as 5G. Other than protocol implementations, there is another body of work related to 4G protocol specifications. Rupprecht et al. [55] showed missing integrity allows the redirection of malicious websites by an active attacker. Hussain et. al. used manually constructed models for verifying certain parts of the 4G [32] and 5G [34] protocols.

Formal Verification of Cellular Networks and Other Protocols. Approaches using formal verification either rely on manually extracted models from specifications [32–34, 37], require models in formally-verifiable languages [95, 96], or require a reference implementation of the protocol in a custom language [97, 98]. These approaches are not scalable for commercial protocol implementations [32–34, 37, 95–98]. Model-checking has also been applied to verify properties of protocols [99–101]. But these approaches are unable to reason about properties that depend on protocol events. Execution-based model checking approaches [102, 103] do not require an explicit model but are prone to state-space explosion.

FSM extraction. There are approaches that infer protocol specifications as a model from traces [104], from network traces [105–107], or using program analysis [108]. However, the FSM's extracted through such approaches represent discernible external interactions of the protocol (e.g., the sequence of exchanged messages) and do not contain enough semantic meaning to reason about security and privacy properties. In a black-box setting active-learning [39] has been used to extract the FSM of a system. However, the extracted FSM does not have a proper indication of states and in our white-box setup, we have a lot more information to utilize. Symbolic execution has also been used to generate formally analyz-

able models of protocol implementations. Aizatulin et al. [109] combined symbolic execution with proof techniques for extracting a ProVerif model from implementations in C. However, their technique is limited to protocols without branching.

BLE implementation security. BlueBorne [16] and Bleedingbit [17] manually identifies critical attack vectors that can be used to take control of affected devices even without pairing. BIAS [18], and KNOB [19] also manually analyze the BR/EDR specifications and present practical impersonation attacks. BLESA [30], 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. [20] 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 [7] provides a testing framework to identify implementation vulnerabilities, whereas Frankenstein [21] uses firmware emulation to run fuzzing on firmware dumps. Moreover, InternalBlue [22] releases a reverse-engineered Bluetooth implementation for the research community. BLURtooth [83] 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.

Model learning in different domains. Model learning can be distinguished between a passive and an active approach. In passive learning, only existing data is used and based on the data, a model is constructed. For example, in [110], passive learning techniques are used on observed network traffic to infer a state machine of the protocol used by a botnet. This approach has been combined with the automated learning of message formats in [111], which then also used the model obtained as a basis for fuzz testing. When using active automated learning techniques, as done in this paper, an implementation is actively queried by the learning algorithm and based on the responses, a model is constructed. State machines learning has lately become a tool for analyzing the security protocol implementations of various protocols, such as: TLS [39], DTLS [59], TCP [60], IoT [61], OpenVPN [62], QUIC [63], and SSH [40]. In the area of cellular networks, recently Chlosta et al. [112] aimed

to apply model learning to a component of the core network (MME). However, they only apply to open-source MME networks and do not experiment with real-world implementations and therefore do not face a lot of challenges that DIKEUE encounters and solves. Stone et al. [113] extend state learning to analyze implementations of the 802.11 4-way handshake. In practice, model learning often falls to non-determinism due to unreliable commuinication medium and requires an prohibitively large number of queries to learn an FSM of a protocol implementation. Several approaches have been developed by the research community to deal with these issues. HVLearn [71] and SFADiff [70] uses cache to avoid the communication cost of repeated queries and improve performance. Furthermore, majority voting has been used to deal with non-determinism [59, 69, 113].

7. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In this dissertation, we develop systematic frameworks for analyzing the security and privacy of important wireless communication protocol implementations such as cellular networks and Bluetooth.

We propose ProChecker—a framework to formally reason about communication protocol implementations. ProChecker aims to automatically verify cellular network protocol implementations to uncover logical vulnerabilities. We evaluate ProChecker by implementing and integrating it into the existing functional testing framework of a closed-source and two open-source LTE implementations and analyze their implementations. We evaluate our extracted models against 62 properties. Along with uncovering 3 new protocol-specific logical attacks, 6 implementation issues, ProChecker identified 14 prior attacks in the FSM's derived from implementations. The issues range from denial-of-service attacks, broken integrity, encryption, and, replay protection to severe privacy leakage. On the horizon of 5G deployment ProChecker can have an important impact in securing 5G implementations from the very start.

We propose DIKEUE— which, to the best of our knowledge, is the first tool that designs a black-box FSM inference module to automatically infer the FSM from a UE's implementation without any manual interventions or modifications to the devices. DIKEUE will be publicly available at [64] after all the affected UEs are patched and the responsible disclosure is completed. We design an FSM equivalence checking algorithm that automatically detects and reports diverse deviant behavior of two FSMs by reducing it to a symbolic model checking problem. We evaluate DIKEUE with 14 different devices from 5 vendors, and demonstrate that it can uncover 17 deviant behaviors, including 11 exploitable weaknesses and 3 interoperability issues.

Lastly, we present 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 FSM

equivalence checker 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.

Future work. In the future, we would like to explore the following directions: (i) **Analyzing other communication protocol implementations:** The techniques proposed in this thesis are applicable to any stateful communication protocol implementation. Therefore we would like extend our analysis to other important protocols such as WiFi and important IoT-based protocols; (ii) **Automatic root cause analysis:** Though the developed techniques can automatically uncover issues in the protocol implementation, it still requires manual root cause analysis to uncover the root cause of the issue. In the future, we aim to develop machine learning-based techniques for automatic root cause analysis. More specifically we would like to develop Reinforcement Learning (RL) agents to automate this task; (iii) **Automatic property extraction:** One of the major issues for protocol implementation testing is to manually identify the important security and privacy properties from the implementation. In the future, we aim to automatically analyze protocol specifications to extract important properties for implementation verification; (iv) **Automatic implementation patching:** Following the techniques discussed in this thesis, when an implementation vulnerability is identified, it needs to be manually patched. One interesting and important direction can be to develop approaches for automatic protocol implementation patching. In that case, the code will be either automatically or semi-automatically patched to resolve the implementation issue.

REFERENCES

[1] *2018 hawaii false missile alert*, https://en.wikipedia.org/wiki/2018_Hawaii_false_missile_alert.

[2] J. Brtis and M. Mcevilley, *Systems engineering for resilience. the MITRE Corporation: Bedford, MA, 2013.*

[3] *Hackers take down the most wired country in europe*, https://www.wired.com/2007/08/ff-estonia/.

[4] *Hackers are tapping into mobile networks backbone, new research shows*, https://www.forbes.com/sites/parmyolson/2015/10/14/hackers-mobile-network-backbone-ss7.

[5] *Major ddos attacks involving iot devices*, https://www.enisa.europa.eu/publications/info-notes/major-ddos-attacks-involving-iot-devices.

[6] G. Candea and P. Godefroid, "Automated software test generation: Some challenges, solutions, and recent advances," in *Computing and Software Science*, Springer, 2019, pp. 505–531.

[7] 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, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/garbelini.

[8] H. Kim, J. Lee, E. Lee, and Y. Kim, "Touching the untouchables: Dynamic security analysis of the LTE control plane," in *2019 IEEE Symposium on Security and Privacy, SP 2019.*

[9] *3gpp. non-access-stratum (nas) protocol for evolved packet system (eps); stage 3 specification 3gpp ts 24.301 version 12.8.0 release 12.*

[10] *Ts 33.401 3gpp system architecture evolution (sae).*

[11] *3gpp. evolved universal terrestrial radio access (e-utra) and evolved universal terrestrial radio access network (eutran); overall description; stage 2, specification 3gpp ts 36.300 version 12.4.0 release 12.*

[12] *3gpp. technical specification group services and system aspects; study on the security aspects of the next generation system (3gpp tr 33.899 v1.3.0 release 14).*

[13] *3gpp release 15*, http://www.3gpp.org/release-15.

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

[15] D. Rupprecht, K. Jansen, and C. Pöpper, "Putting lte security functions to the test: A framework to evaluate implementation correctness," in *Proceedings of the 10th USENIX Conference on Offensive Technologies*, ser. WOOT'16, Austin, TX: USENIX Association, 2016, pp. 40–51.

[16] 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.

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

[18] 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. DOI: 10.1109/SP40000.2020.00093.

[19] 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, ISBN: 978-1-939133-06-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/antonioli.

[20] 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. DOI: 10.1109/SP46214.2022.9833777.

[21] 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, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ruge.

[22] 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, Seoul, Republic of Korea: Association for Computing Machinery, 2019, pp. 79–90, ISBN: 9781450366618. DOI: 10.1145/3307334.3326089. [Online]. Available: https://doi.org/10.1145/3307334.3326089.

[23] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper, "Call Me Maybe: Eavesdropping Encrypted LTE Calls With ReVoLTE," in *USENIX Security Symposium (SSYM)*, USENIX Association, Aug. 2020.

[24] M. Chlosta, D. Rupprecht, T. Holz, and C. Pöpper, "Lte security disabled: Misconfiguration in commercial networks," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '19, Miami, Florida: Association for Computing Machinery, 2019, pp. 261–266, ISBN: 9781450367264. DOI: 10.1145/3317549.3324927. [Online]. Available: https://doi.org/10.1145/3317549.3324927.

[25] A. Shaik, J. Seifert, R. Borgaonkar, N. Asokan, and V. Niemi, "Practical attacks against privacy and availability in 4g/lte mobile communication systems," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, The Internet Society, 2016.

[26] C.-Y. Li *et al.*, "Insecurity of voice solution volte in lte mobile networks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 316–327, ISBN: 9781450338325. DOI: 10.1145/2810103.2813618. [Online]. Available: https://doi.org/10.1145/2810103.2813618.

[27] H. Kim *et al.*, "Breaking and fixing volte: Exploiting hidden data channels and misimplementations," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 328–339, ISBN: 9781450338325. DOI: 10.1145/2810103.2813718. [Online]. Available: https://doi.org/10.1145/2810103.2813718.

[28] D. Maier, L. Seidel, and S. Park, "Basesafe: Baseband sanitized fuzzing through emulation," in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '20, Linz, Austria: Association for Computing Machinery, 2020, pp. 122–132, ISBN: 9781450380065. DOI: 10.1145/3395351.3399360. [Online]. Available: https://doi.org/10.1145/3395351.3399360.

[29] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim, "Basespec: Comparative analysis of baseband software and cellular specifications for l3 protocols," in *Symposium on Network and Distributed System Security (NDSS)(San Diego, CA, USA). ISOC*, 2021.

[30] J. Wu *et al.*, "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.

[31] 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*, Pasadena, CA, USA: Springer-Verlag, 2022, pp. 373–392, ISBN: 978-3-031-06772-3. DOI: 10.1007/978-3-031-06773-0_20. [Online]. Available: https://doi.org/10.1007/978-3-031-06773-0_20.

[32] S. R. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino, "Lteinspector: A systematic approach for adversarial testing of 4g LTE," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, The Internet Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018%5C_02A-3%5C_Hussain%5C_paper.pdf.

[33] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5g authentication," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 1383–1396, ISBN: 9781450356930. DOI: 10.1145/3243734.3243846. [Online]. Available: https://doi.org/10.1145/3243734.3243846.

[34] S. R. Hussain, M. Echeverria, I. Karim, O. Chowdhury, and E. Bertino, "5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 669–684, ISBN: 9781450367479. DOI: 10.1145/3319535.3354263. [Online]. Available: https://doi.org/10.1145/3319535.3354263.

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

[36] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5g authentication," ser. CCS '18, 2018.

[37] C. Cremers and M. Dehnel-Wild, "Component-based formal analysis of 5g-aka: Channel assumptions and session confusion," 2019.

[38] P. Godefroid, "Higher-order test generation," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 258–269.

[39] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*.

[40] P. Fiteru-Brotean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of ssh implementations," in *SPIN 2017*.

[41] *srsLTE*, https://github.com/srsLTE.

[42] *OpenAirInterface*, https://www.openairinterface.org/.

[43] R. Cavada *et al.*, "The nuxmv symbolic model checker," in *International Conference on Computer Aided Verification*, Springer, 2014, pp. 334–342.

[44] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, Jun. 2001, pp. 82–96. DOI: 10.1109/CSFW.2001.930138.

[45] *Gsma mobile security hall of fame*, https://www.gsma.com/security/gsma-mobile-security-hall-of-fame/.

[46] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[47] *3gpp conformance testing*, https://www.3gpp.org/technologies/keywords-acronyms/108-conformance-testing-ue.

[48] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*, Springer, 2000, pp. 154–169.

[49] *Digital cellular telecommunications system (phase 2+); universal mobile telecommunications system (umts); 3g security; security architecture (3gpp ts 33.102 version 11.5.1 release 11).*

[50] M. Arapinis *et al.*, "New privacy issues in mobile telephony: Fix and verification," in *CCS 12*.

[51] M. Arapinis, L. I. Mancini, E. Ritter, and M. Ryan, "Privacy through pseudonymity in mobile telephony systems.," in *NDSS*, 2014.

[52] *5g; non-access-stratum (nas) protocol for 5g system (5gs); stage 3 (3gpp ts 24.501 version 15.0.0 release 15))*, https://www.etsi.org/deliver/etsi_ts/124500_124599/124501/15.00.00_60/ts_124501v150000p.pdf.

[53] *5g; user equipment (ue) conformance specification; part 1: Protocol (3gpp ts 38.523-1 version 15.3.0 release 15)*. http://www.3gpp.org/ftp//Specs/archive/32_series/32.899/32899-f10.zip.

[54] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim, "Basespec: Comparative analysis of baseband software and cellular specifications for l3 protocols," in *Symposium on Network and Distributed System Security (NDSS)(San Diego, CA, USA). ISOC*, 2021.

[55] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper, "Breaking lte on layer two," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1121–1136.

[56] H. Raffelt, B. Steffen, and M. Tiziana, "Dynamic testing via automata learning," Oct. 2007, pp. 136–152, ISBN: 978-3-540-77964-3. DOI: 10.1007/978-3-540-77966-7_13.

[57] F. Vaandrager, "Model learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, Jan. 2017, ISSN: 0001-0782. DOI: 10.1145/2967606. [Online]. Available: https://doi.org/10.1145/2967606.

[58] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006, ISBN: 0123725011.

[59] 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, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[60] P. Fiteru-Brotean, 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, ISBN: 978-3-319-41540-6.

[61] 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. DOI: 10.1109/ICST.2017.32.

[62] 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. DOI: 10.1109/EuroSPW.2018.00009.

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

[64] *Dikeue*, https://github.com/SyNSec-den/DIKEUE.

[65] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987, ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(87)90052-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0890540187900526.

[66] 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, ISBN: 978-3-319-11164-3.

[67] M. Shahbaz and R. Groz, "Inferring mealy machines," in *FM 2009: Formal Methods*, A. Cavalcanti and D. R. Dams, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 207–222, ISBN: 978-3-642-05089-3.

[68] 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. DOI: 10.1109/TSE.1978.231496.

[69] S.-J. Moon *et al.*, "Alembic: Automated model inference for stateful network functions," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19, Boston, MA, USA: USENIX Association, 2019, pp. 699–718, ISBN: 9781931971492.

[70] 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, Vienna, Austria: Association for Computing Machinery, 2016, pp. 1690–1701, ISBN: 9781450341394. DOI: 10.1145/2976749.2978383. [Online]. Available: https://doi.org/10.1145/2976749.2978383.

[71] 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. DOI: 10.1109/SP.2017.46.

[72] W. M. McKeeman, "Differential testing for software," *DIGITAL TECHNICAL JOURNAL*, vol. 10, no. 1, pp. 100–107, 1998.

[73] B. Evans and A. Savoia, "Differential testing: A new approach to change detection," Jan. 2007, pp. 549–552. DOI: 10.1145/1295014.1295038.

[74] *Ts 24.301 universal mobile telecommunications system (umts); lte; 5g; non-access-stratum (nas) protocol for evolved packet system (eps); stage 3 (3gpp ts 24.301 version 15.4.0 release 15).*

[75] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib," in *Computer Aided Verification*, D. Kroening and C. S. Psreanu, Eds., Cham: Springer International Publishing, 2015, pp. 487–495, ISBN: 978-3-319-21690-4.

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

[77] I. Karim, S. Hussain, and E. Bertino, "Prochecker: An automated security and privacy analysis framework for 4g lte protocol implementations," in *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021*, 2021.

[78] S. R. Hussain, M. Echeverria, O. Chowdhury, N. Li, and E. Bertino, "Privacy attacks to the 4g and 5g cellular paging protocols using side channel information," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019%5C_05B-5%5C_Hussain%5C_paper.pdf.

[79] B. Beurdouche *et al.*, "A messy state of the union: Taming the composite state machines of tls," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 535–552. DOI: 10.1109/SP.2015.39.

[80] *Lte; evolved universal terrestrial radio access (e-utra) and evolved packet core (epc); user equipment (ue) conformance specification; part 1: Protocol conformance specification (3gpp ts 36.523-1).*

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

[82] *2022 Bluetooth Market Update*, https://www.bluetooth.com/2022-market-update/.

[83] 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, Nagasaki, Japan: Association for Computing Machinery, 2022, pp. 196–207, ISBN: 9781450391405. DOI: 10.1145/3488932.3523258. [Online]. Available: https://doi.org/10.1145/3488932.3523258.

[84] 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, Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2021, pp. 164–174, ISBN: 9781450383493. DOI: 10.1145/3448300.3469133. [Online]. Available: https://doi.org/10.1145/3448300.3469133.

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

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

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

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

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

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

[91] 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, pp. 524–542, ISBN: 978-3-030-90869-0. DOI: 10.1007/978-3-030-90870-6_28. [Online]. Available: https://doi.org/10.1007/978-3-030-90870-628.

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

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

[94] G. Hernandez and K. R. B. Butler, "Basebads: Automated security analysis of baseband firmware: Poster," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '19, Miami, Florida: Association for Computing Machinery, 2019, pp. 318–319, ISBN: 9781450367264. DOI: 10.1145/3317549.3326310. [Online]. Available: https://doi.org/10.1145/3317549.3326310.

[95] C. Hawblitzel *et al.*, "Ironfleet: Proving practical distributed systems," in *SOSP 15*.

[96] J. R. Wilcox *et al.*, "Verdi: A framework for implementing and formally verifying distributed systems," in *PLDI 15*.

[97] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular code-based cryptographic verification," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 341–350.

[98] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Transactions on Programming Languages and Systems (TOPLAS)*,

[99] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2014, pp. 389–391.

[100] M. Canini, D. Venzano, P. Pereni, D. Kostiundefined, and J. Rexford, "A nice way to test openflow applications," in *NSDI12*.

[101] M. Musuvathi, D. R. Engler, *et al.*, "Model checking large network protocol implementations.."

[102] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 174–186.

[103] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker b last," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[104] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 408–428, 2015.

[105] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *2009 30th IEEE Symposium on Security and Privacy*, IEEE, 2009, pp. 110–125.

[106] C. Y. Cho, D. Babi , E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[107] R. J. Walls, Y. Brun, M. Liberatore, and B. N. Levine, "Discovering specification violations in networked software systems," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 496–506.

[108] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Extraction of protocol message format using dynamic binary analysis," in *CCS 2007*.

[109] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and verifying cryptographic models from c protocol code," in *CCS 2011*.

[110] P. M. Comparetti, G. Wondracek, C. Krügel, and E. Kirda, "Prospex: Protocol specification extraction," *2009 30th IEEE Symposium on Security and Privacy*, pp. 110–125, 2009.

[111] Y. Hsu, G. Shu, and D. Lee, "A model-based approach to security flaw detection of network protocol implementations," in *2008 IEEE International Conference on Network Protocols*, 2008, pp. 114–123. DOI: 10.1109/ICNP.2008.4697030.

[112] 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, Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2021, pp. 164–174, ISBN: 9781450383493. DOI: 10.1145/3448300.3469133. [Online]. Available: https://doi.org/10.1145/3448300.3469133.

[113] 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, ISBN: 978-3-319-99073-6.