Noncompliance as Deviant Behavior: An Automated Black-box Noncompliance Checker for 4G LTE Cellular Devices

Syed Rafiul Hussain*
Pennsylvania State University
hussain1@psu.edu

Imtiaz Karim*
Purdue University
karim7@purdue.edu

Abdullah Al Ishtiaq
Pennsylvania State University
abdullah.ishtiaq@psu.edu

Omar Chowdhury
University of Iowa
omar-chowdhury@uiowa.edu

Elisa Bertino
Purdue University
bertino@purdue.edu

ABSTRACT

The paper focuses on developing an automated black-box testing approach called DIKEUE that checks 4G Long Term Evolution (LTE) control-plane protocol implementations in commercial-of-the-shelf (COTS) cellular devices (also, User Equipments or UEs) for noncompliance with the standard. Unlike prior noncompliance checking approaches which rely on property-guided testing, DIKEUE adopts a property-agnostic, differential testing approach, which leverages the existence of many different control-plane protocol implementations in COTS UEs. DIKEUE uses *deviant behavior* observed during differential analysis of pairwise COTS UEs 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 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.

CCS CONCEPTS

• **Networks → Network protocols**; **Protocol testing and verification**; • **Security and privacy** → *Mobile and wireless security*.

KEYWORDS

Cellular Network, 4G, LTE, Model Learning, Vulnerabilities, Attacks

ACM Reference Format:
Syed Rafiul Hussain*, Imtiaz Karim*, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. 2021. *Noncompliance as Deviant Behavior*: An Automated Black-box Noncompliance Checker for 4G LTE Cellular Devices . In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3460120.3485388

*Both authors contributed equally to this research.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3485388

1 INTRODUCTION

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, under-specification, and intricate protocol details present in the natural languages specification [2, 8, 9]. 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 [39], location exposure of a target user [51]), 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 paper, we aim to develop an automated approach for identifying noncompliance behavior in 4G LTE UEs.

Prior research. Although prior works [16, 23, 38, 40, 41, 47, 48, 51] 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 [16, 38–40, 47, 48, 51] are completely manual and cannot uncover a myriad of *implementation-specific* behavior; (B) The analyses [39] perform semi-automated stateless testing; (C) The approaches based on formal verification [12, 30, 32] 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 [23, 41] 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 [30, 49], eavesdropping [48] and DNS redirection [49]. In summary, in this paper 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 paper, 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 [45, 54, 55]. 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 [20, 21, 25–27, 46, 53], 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 paper 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 [1] 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.

2 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 $\Psi : \mathcal{S} \times \Sigma \rightarrow \mathcal{S}$ maps the pair of a current state and an input symbol to the corresponding next state, and the output relationship $\Omega : \mathcal{S} \times \Sigma \rightarrow \Lambda$ maps the pair of a current state and an input symbol to the corresponding output symbol.

2.1 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 $x \overset{?}{\in} \mathcal{L}$ (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 [10, 34]. ***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 [50] 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 [17]. 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).

2.2 4G LTE Preliminaries

In the following, we introduce the most important network components relevant to our analysis in this paper.

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.

eNodeB. 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.

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 7).

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. We discuss in detail the relevant NAS and RRC layer procedures in Appendix A.1.

3 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.

3.1 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 [30, 39, 47, 51] 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 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.

3.2 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 $\pi_i = \sigma_1\sigma_2\sigma_3 \ldots \sigma_m$ where $\sigma_j \in \Sigma$ such that the output sequence generated by I after feeding π_i as input, $\gamma_i = \lambda_1\lambda_2\lambda_3 \ldots \lambda_m$ in which $\lambda_j \in \Lambda$, 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 $\langle I_1, I_2, \ldots, I_n \rangle$. It also requires that the

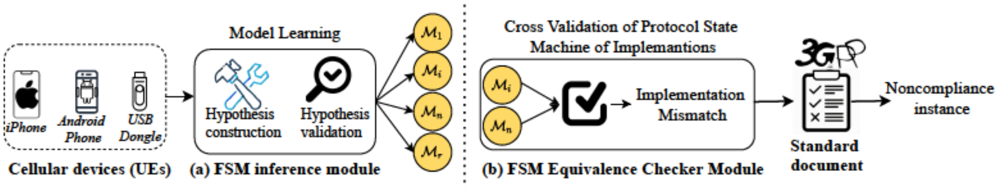

Figure 1: Workflow of DIKEUE

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 $\mathcal{I}_j$, using active automata learning, extract an approximation $\mathcal{M}_j^*$ of the underlying FSM $\mathcal{M}_j$; ❷ For each pair of extracted FSM $\mathcal{M}_j^*$ and $\mathcal{M}_k^*$, find input sequences of the form π_i such that when it is fed as input to both $\mathcal{M}_j^*$ 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.

3.3 Workflow of DIKEUE

DIKEUE (shown in Figure 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.

3.4 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.

3.4.1 *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* input 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 [44].

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 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 [11, 52] 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 re-transmissions 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.

3.4.2 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 [24, 42]. 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 5). 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* identify deviant-behavior-inducing input sequences from different diversity classes but also identify different instances from the same class.

4 FSM INFERENCE MODULE

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

4.1 Learner

Following the model learning algorithm [34], the learner systematically generates queries as sequences of input alphabets, and based on the outputs, infers the underlying FSM.

4.1.1 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

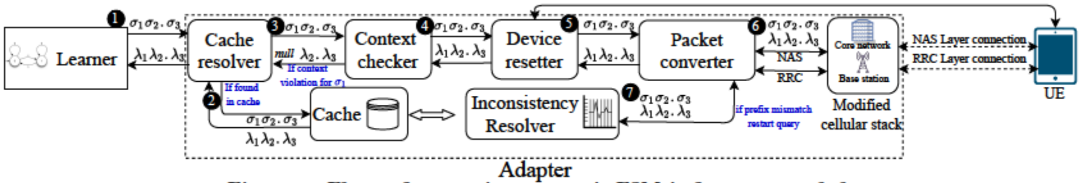
Learner
Cache resolver
Context checker
Device resetter
Packet converter
Cache
Inconsistency Resolver
Core network
Base station
Modified cellular stack
UE
NAS Layer connection
RRC Layer connection
NAS
RRC
1 $\sigma_1\sigma_2.\sigma_3$
$\lambda_1\lambda_2.\lambda_3$
2 $\sigma_1\sigma_2.\sigma_3$ $\lambda_1\lambda_2.\lambda_3$
If found in cache
3 $\sigma_1\sigma_2.\sigma_3$
$null$ $\lambda_2.\lambda_3$
If context violation for σ_1
4 $\sigma_1\sigma_2.\sigma_3$ $\lambda_1\lambda_2.\lambda_3$
5 $\sigma_1\sigma_2.\sigma_3$ $\lambda_1\lambda_2.\lambda_3$
6 $\sigma_1\sigma_2.\sigma_3$ $\lambda_1\lambda_2.\lambda_3$
7 $\sigma_1\sigma_2.\sigma_3$ $\lambda_1\lambda_2.\lambda_3$
if prefix mismatch restart query
Adapter

Figure 2: Flow of query in DIKEUE's FSM inference module

the potential input alphabet set of 261 input symbols (Section 3.4). 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*∧¬is_replay(m) ∧is_plain_text(m)∧¬is_plain_header (m) ∧¬is_null_security(m); (2) $message\_kind(m) =$ *identity_request* ∧is_replay(m) ∧ is_plain _text(m) ∧ ¬is_plain_header(m) ∧¬is _null_security(m). 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 9). 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 [7], 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 9.

4.1.2 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.

4.2 Adapter

The adapter acts as a glue between all the components of FSM inference module (shown in Figure 2), and builds a reliable interface from the learner to each control-plane layers we want to analyze.

4.2.1 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.

4.2.2 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 [2, 8, 9], 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 $Q1$ in Table 1, 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 $Q2$, 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 1, for $Q3$, the security protected *GUTI_reallocation* message requires key for integrity and encryption. However, before the authentication and security mode command 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 $Q4$, 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 $Q5$, the first *auth_request_replay* does not correspond to anything and will be discarded until an *auth_request* has been received.

4.2.3 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.

4.2.4 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

ID	Query	ID	Output
$Q1$	*attach_accept enable_RRC_con.enable_attach*	$\mathcal{R}1$	*null_action RRC_con_request.attach_request*
$Q2$	*enable_attach enable_RRC_con enable_attach.auth_request*	$\mathcal{R}2$	*null_action RRC_con_request attach_request.auth_response*
$Q3$	*enable_RRC_con enable_attach, GUTI_reallocation.auth_request*	$\mathcal{R}3$	*RRC_con_request attach_request null_action.auth_response*
$Q4$	*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*
$Q5$	*enable_RRC_con enable_attach auth_request_replay.auth_request*	$\mathcal{R}5$	*RRC_con_request attach_request null_action.auth_response*
$Q6$	*enable_RRC_con enable_attach GUTI_reallocation.auth_request*	$\mathcal{R}6$	*RRC_con_request attach_request null_action.auth_response*
$Q7$	*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*

Table 1: Example queries and responses. "." divides the prefix and suffix of the queries and responses.

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.

4.2.5 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 [11, 52]. 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 $Q6$ is a query generated during the model validation phase, and the previous queries are generated during the hypothesis construction phase. $Q6$ is checked against queries $Q1$ - $Q5$, and as the same query is cached in $Q3$, $Q6$ will not be sent, and the saved response $\mathcal{R}3$ will be returned.

4.2.6 Resolving observational non-determinism with inconsistency resolver: As discussed in Section 3.4, 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 [44] 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 [44] 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 1) 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 $Q7$ in Table 1, the response prefix of the query is not consistent with the previously saved response of $\mathcal{R}1$. In such occurrences, the query $Q7$ 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.

4.2.7 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 [4] is used, and for USB devices, the device is toggled through the USB connection. Finally, for development boards and LTE dongles, AT commands [36] are injected through serial connections.

4.2.8 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.

5 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.

5.1 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 3). For this reduction, a symbolic model checker (e.g., nuXmV [14]) 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 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*

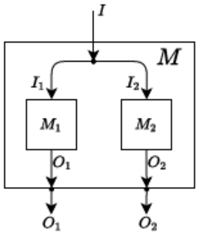

Figure 3: Equivalence Checking to Model Checking

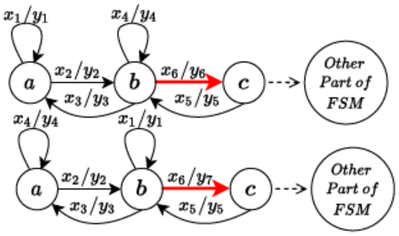

Figure 4: FSMs for understanding the challenge for identify diverse deviation-inducing input sequences.

of the execution (precisely, in linear temporal logic $\Box(O_1 = O_2)$). 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.

5.2 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 $\Box(O_1 = O_2)$ 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. 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 $s_i \overset{x_k/y_o}{\longrightarrow} s_j$, 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 $b \longrightarrow c$ (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 $b \longrightarrow c$ 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 $b \longrightarrow c$.

One may consider removing the transition $b \longrightarrow c$ 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.

5.3 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 5.1 (Diversity Class of Deviation-inducing Input Sequences). Given a fixed set of output symbols Λ where $|\Lambda| = n$, 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 $\pi_i = \sigma_1\sigma_2\sigma_3 \dots \sigma_m$ is an element of the $\langle \lambda_r, \lambda_s \rangle$-diversity class iff when π_i is executed on M_1 and M_2 to obtain output sequences $\gamma_i^1 = \lambda_1^1\lambda_2^1\lambda_3^1 \dots \lambda_m^1$ and $\gamma_i^2 = \lambda_1^2\lambda_2^2\lambda_3^2 \dots \lambda_m^2$, respectively, then there exists a $1 \leq k \leq m$ such that $\lambda_k^1 = \lambda_r$ and $\lambda_k^2 = \lambda_s$.

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 $\pi = abcc$ for M_1 and M_2 for which we obtain the output sequences $\gamma^1 = 1234$ and $\gamma^2 = 1243$ after executing π on M_1 and M_2, respectively. π is an element of the $\langle 3, 4 \rangle$-diversity class as there exists a $k = 3$ for which $\gamma_3^1 = 3$ and $\gamma_3^2 = 4$. Note that, π is also an element of $\langle 4, 3 \rangle$-diversity class as there exists $k = 4$ for which $\gamma_4^1 = 4$ and $\gamma_4^2 = 3$.

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 $\Box(O_1 = O_2)$ in the composed model M (as shown in Figure 3), we would pose a series of model checking queries; one for each of the following safety properties: (1) $\Box\neg(O_1 = 1 \land O_2 = 2)$ (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) $\Box\neg(O_1 = 1 \land O_2 = 3)$; (3) $\Box\neg(O_1 = 2 \land O_2 = 1)$; (4) $\Box\neg(O_1 = 2 \land O_2 = 3)$; (5) $\Box\neg(O_1 = 3 \land O_2 = 1)$; (6) $\Box\neg(O_1 = 3 \land O_2 = 2)$. 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 5.2), 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 5.2 because this phenomenon is localized to only a single equivalence class.

6 IMPLEMENTATION

The FSM inference module is implemented on top of LearnLib [35] and srsLTE [6]–an open-source 4G LTE stack. For the learning algorithm, we use TTT [34] as it requires fewer queries compared to other algorithms [33], and for conformance testing, we use Wp-method [17]. We implement our adapter in Java. We use srsLTE v19.10 as the cellular stack to implement our modified core network

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

Table 2: M = Membership and E = Equivalence queries.

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 [14] and a python 2.7 script as the wrapper. Table 7 summarizes our efforts of modifying the tools and creating new components for DIKEUE.

7 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 8) 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.

8 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 3.

8.1 Exploitable deviations

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

8.1.1 Replayed GUTI_reallocation: We identified the exploitable deviations **E1** and **E2** (from Table 3) 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.

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 [39, 51] or Man-in-the-Middle (MitM) relay [30, 49] that can replay previously saved messages.

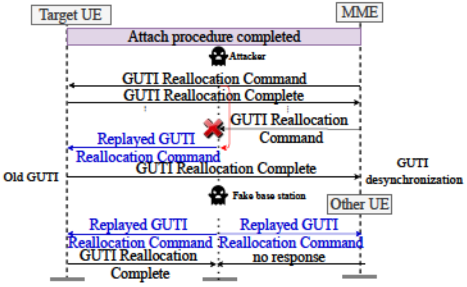
Target UE
MME
Attach procedure completed
Attacker
GUTI Reallocation Command
GUTI Reallocation Complete
GUTI Reallocation Command
Replayed GUTI Reallocation Command
GUTI Reallocation Complete
Old GUTI
GUTI desynchronization
Fake base station
Other UE
Replayed GUTI Reallocation Command
Replayed GUTI Reallocation Command
GUTI Reallocation Complete
no response

Figure 5: Steps of the replayed GUTI reallocation attack

Attack Description. This vulnerability can be exploited in two ways: (1) The adversary, using a sniffer [30, 49] or MitM relay [49], captures the *GUTI_reallocation* message for a given security context. Later on when the MME sends *GUTI_reallocation* again for refreshing the 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 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 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 [51], 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.

8.1.2 Plaintext message acceptance after security context: The deviations **EI3** and **EI4** in Table 3 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 [8], 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 [49] 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 [22].

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*.

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 [30, 31] or MitM relays [49]. Furthermore, it has been shown that *auth_request* can be used to leak private information, including subscriber activity monitoring [13], launching DoS, and tracking a user [13, 37]. Implementations accepting plaintext *auth_request* are, therefore, vulnerable to these attacks.

8.1.3 Inappropriate state reset. In exploitable issues **E11-E14** (of Table 3), 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 [2] 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.

8.2 Interoperability issues

DIKEUE uncovered 3 potential interoperability issues **EI3**, **EI4**, **I15** (shown in Table 3). 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 [2] 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.

8.3 Other deviant behaviors

DIKEUE also uncovered deviant behaviors **O6 - O10** in Table 3, 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.

8.4 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. [47] for a Huawei USB dongle.

Issue	Description	Root cause		Device													
		D	U	Nexus6	HTC1	GalaxyS6	HTC 10	Nexus6P	GalaxyS8+	Pixel 3XL	Huwayei Y5	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* any-time	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		✓			✔						✔	✔		✔		✔
(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		✓			✔											
(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* [32]	Accepts replayed *sm_command* after security context has been established		✓	✔			✔	✔	✔	✔				✔		✔	
(E17) Downgraded *RRC_sm_command* acceptance [47]	UE accepts downgraded *RRC_sm_command* and bypasses the whole RRC layer security	✓											✔				

Table 3: 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

9 COMPARISON WITH BASELINE (RQ2)

We compare the effectiveness of DIKEUE with the conformance testing framework defined in the 3GPP specification [5] and property-guided testing by previous approaches [12, 19, 30, 32, 37].

9.1 Comparison with conformance test cases

We first compare the performance of DIKEUE with the 3GPP conformance test cases [5] 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 [6] v20.10.1– the open-source implementation by srsLTE [6]. We use the percentage of lines and functions executed, which are obtained by Gcov [3], 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 $L_e(f)$ 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: $\sum_{i=1}^{m} L_e(f_i)/\sum_{i=1}^{m} L(f_i)$ 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 [5] 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.

9.2 Comparison with existing LTE works

Table 4 compares our approach with existing LTE testing approaches based on several criteria such as automation, specification, implementation analysis, and stateful testing.

Paper	Auto-matic	Specific-ation analysis	Impleme-ntation analysis	Under-specificat-ion detection	Stateful
LTEFuzz [39]	✗	✗	✓	✗	✗
LTEInspector [30]	✗	✓	✗	✗	✓
5GReasoner [32]	✗	✓	✗	✗	✓
5G-Authentication [12]	✗	✓	✗	✓	✓
5G-AKA [19]	✗	✓	✗	✓	✓
ProChecker [37]	✗	✓	✓	✓	✓
DIKUE	✓	✓	✓	✓	✓

Table 4: Comparison with existing approaches.

9.2.1 Comparison with LTEFuzz. LTEFuzz [39] 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 8.1.1) 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.

9.2.2 Comparison with property-guided testing. Previous work [12, 19, 30, 37] has applied property-guided testing on FSMs derived from standards [12, 19, 30, 37] or extracted from white-box analysis [37]. 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.

10 COMPONENTS PERFORMANCE (RQ3)

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

10.1 FSM inference module performance

Table 2 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 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 5.

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*

Table 5: DIKEUE performance of different components. M = Membership queries and E = Equivalence queries.

	Nexus6	HTC1	GalaxyS6	HTC 10	Nexus6P	GalaxyS8+	Pixel 3XL	HuwaeiY5	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														

Table 6: Number of unique deviants.

10.1.1 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 9 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.

10.1.2 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%.

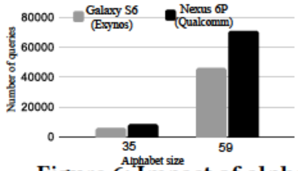

Figure 6: Impact of alphabet size

10.1.3 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%.

10.1.4 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 process, the learner grinds into complete halt due to inconsistencies in the responses (shown as N/A in Table 4). 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.

10.2 FSM equivalence checker performance

Table 6 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 3.

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 10. On an average, FSM equivalence checker takes 42 minutes to find all the deviations. The timing cost of querying to the model checker is shown in Figure 8 in Appendix A.2.

11 RELATED WORK

We divide the related work in two broad categories: (i) Model learning and protocol state fuzzing; (ii) Cellular network security.

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 [18], 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 [29], 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 [21], DTLS [26], TCP [25], IoT [53], OpenVPN [20], QUIC [46], and SSH [27]. In the area of cellular networks, recently Chlosta et al. [15] 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. [43] 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 [52] and SFADiff [11] uses cache to avoid the communication cost of repeated queries and improve performance. Furthermore, majority voting has been used to deal with non-determinism [26, 43, 44].

Cellular network security. Previous work on 4G LTE implementation security has either been found by complete manual analysis [16, 23, 28, 38, 40, 41, 48, 51] or semi automated testing [39, 47]. Other than protocol implementations, there is another body of work related to protocol specifications. Rupprecht et al. [49] 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 [30] and 5G [32] protocols.

12 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 paper, we consider the typical notion of property [12, 19, 30, 32] 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.

We also discuss parallelizing model learning and automatic exploit generation from deviant behavior in Appendix A.4.

13 CONCLUSION AND FUTURE WORK

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.

Future Work. In future, we will accommodate session management and other data layer protocols and port DIKEUE to 5G. We will also develop an automated attack strategy generator to provide end-to-end attack scenarios from the deviating behavior traces.

ACKNOWLEDGEMENTS

This work is supported by NSF grants IIS-2112471, CNS-2006556, DARPA YFA D19AP00039 and Intel. We thank GSMA and the baseband vendors and manufacturers for coordinating with us for the vulnerability disclosure process.

REFERENCES

[1] [n. d.]. *DIKEUE*. https://github.com/SyNSec-den/DIKEUE.

[2] [n. d.]. *Evolved Universal T Radio Re Prot (3GPP TS 36.3 TECHNICAL SPECIFICATION 136 331 V13.0.0 (2016 LTE; l Terrestrial Radio Access (E- Resource Control (RRC)*.

[3] [n. d.]. *GNU Compilers. Gcov - Using the GNU Compiler Collection (GCC))*.

[4] [n. d.]. *libimobiledevice A cross-platform protocol library to access iOS devices*. https://github.com/libimobiledevice.

[5] [n. d.]. *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)*.

[6] [n. d.]. *srsLTE*. https://github.com/srsLTE.

[7] [n. d.]. *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)*.

[8] [n. d.]. *TS 33.401 3GPP System Architecture Evolution (SAE)*.

[9] [n. d.]. *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)*.

[10] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87 – 106. https://doi.org/10.1016/0890-5401(87)90052-6

[11] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. 2016. 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* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1690–1701. https://doi.org/10.1145/2976749.2978383

[12] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. 2018. A Formal Analysis of 5G Authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1383–1396. https://doi.org/10.1145/3243734.3243846

[13] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A Messy State of the Union: Taming the Composite State Machines of TLS. In *2015 IEEE Symposium on Security and Privacy*. 535–552. https://doi.org/10.1109/SP.2015.39

[14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 334–342.

[15] Merlin Chlosta, David Rupprecht, and Thorsten Holz. 2021. 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* (Abu Dhabi, United Arab Emirates) *(WiSec '21)*. Association for Computing Machinery, New York, NY, USA, 164–174. https://doi.org/10.1145/3448300.3469133

[16] Merlin Chlosta, David Rupprecht, Thorsten Holz, and Christina Pöpper. 2019. LTE Security Disabled: Misconfiguration in Commercial Networks. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks* (Miami, Florida) *(WiSec '19)*. Association for Computing Machinery, New York, NY, USA, 261–266. https://doi.org/10.1145/3317549.3324927

[17] T. S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering* SE-4, 3 (1978), 178–187. https://doi.org/10.1109/TSE.1978.231496

[18] P. M. Comparetti, Gilbert Wondracek, C. Krügel, and E. Kirda. 2009. Prospex: Protocol Specification Extraction. *2009 30th IEEE Symposium on Security and Privacy* (2009), 110–125.

[19] C. Cremers and Martin Dehnel-Wild. 2019. Component-Based Formal Analysis of 5G-AKA: Channel Assumptions and Session Confusion. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. https://doi.org/10.14722/ndss.2019.23394

[20] L. Daniel, E. Poll, and J. de Ruiter. 2018. Inferring OpenVPN State Machines Using Protocol State Fuzzing. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. 11–19. https://doi.org/10.1109/EuroSPW.2018.00009

[21] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 193–206.

[22] Simon Erni, Patrick Leu, Martin Kotuliak, Marc Röschlin, and Srdjan Capkun. 2021. AdaptOver : Adaptive Overshadowing of LTE signals. In *https://arxiv.org/abs/2106.05039*. arxiv.

[23] CheolJun Park Insu Yun Yongdae Kim Eunsoo Kim, Dongkwan Kim. 2021. BASESPEC: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols. *NDSS 2021* (2021). https://doi.org/10.14722/ndss.2021.24365

[24] Robert B. Evans and Alberto Savoia. 2007. Differential Testing: A New Approach to Change Detection. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers* (Dubrovnik, Croatia) *(ESEC-FSE companion '07)*. Association for Computing Machinery, New York, NY, USA, 549–552. https://doi.org/10.1145/1295014.1295038

[25] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 454–471.

[26] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2523–2540. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[27] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. 2017. Model Learning and Model Checking of SSH Implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software* (Santa Barbara, CA, USA) *(SPIN 2017)*. Association for Computing Machinery, New York, NY, USA, 142–151. https://doi.org/10.1145/3092282.3092289

[28] Grant Hernandez and Kevin R. B. Butler. 2019. Basebads: Automated Security Analysis of Baseband Firmware: Poster. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks* (Miami, Florida) *(WiSec '19)*. Association for Computing Machinery, New York, NY, USA, 318–319. https://doi.org/10.1145/3317549.3326310

[29] Yating Hsu, Guoqiang Shu, and David Lee. 2008. A model-based approach to security flaw detection of network protocol implementations. In *2008 IEEE International Conference on Network Protocols*. 114–123. https://doi.org/10.1109/ICNP.2008.4697030

[30] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. 2018. 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. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_02A-3_Hussain_paper.pdf

[31] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. 2019. 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. https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_05B-5_Hussain_paper.pdf

[32] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 2019. 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* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 669–684. https://doi.org/10.1145/3319535.3354263

[33] Malte Isberner. 2015. *Foundations of active automata learning: an algorithmic perspective*. Ph. D. Dissertation.

[34] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In *Runtime Verification*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.). Springer International Publishing, Cham, 307–322.

[35] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-Source LearnLib. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 487–495.

[36] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. 2019. Opening Pandora's Box through ATFuzzer: Dynamic Analysis of AT Interface for Android Smartphones. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 529–543. https://doi.org/10.1145/3359789.3359833

[37] Imtiaz Karim, Syed Hussain, and Elisa Bertino. 2021. 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*.

[38] Hongil Kim, Dongkwan Kim, Minhee Kwon, Hyungseok Han, Yeongjin Jang, Dongsu Han, Taesoo Kim, and Yongdae Kim. 2015. Breaking and Fixing VoLTE: Exploiting Hidden Data Channels and Mis-Implementations. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 328–339. https://doi.org/10.1145/2810103.2813718

[39] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. 2019. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1153–1168. https://doi.org/10.1109/SP.2019.00038

[40] Chi-Yu Li, Guan-Hua Tu, Chunyi Peng, Zengwen Yuan, Yuanjie Li, Songwu Lu, and Xinbing Wang. 2015. Insecurity of Voice Solution VoLTE in LTE Mobile Networks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and*

Component	Tools	Lines of Code
Learner	LearnLib [35]	248 (Java)
Adapter	–	1807 (Java)
Membership Cache	–	507 (Mysql and Java)
Modified cellular stack	srsLTE [6]	~4000 (C++)
Device resetter	–	640 (Python 2.7)
FSM Equivalence Checker	–	2240 (Python 2.7)

Table 7: Additions/modifications to the tools used in DIKEUE.

Communications Security (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 316–327. https://doi.org/10.1145/2810103.2813618

[41] Dominik Maier, Lukas Seidel, and Shinjo Park. 2020. BaseSAFE: Baseband Sanitized Fuzzing through Emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (Linz, Austria) *(WiSec '20)*. Association for Computing Machinery, New York, NY, USA, 122–132. https://doi.org/10.1145/3395351.3399360

[42] William M. McKeeman. 1998. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL* 10, 1 (1998), 100–107.

[43] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. 2018. Extending Automated Protocol State Learning for the 802.11 4-Way Handshake. In *Computer Security*, Javier Lopez, Jianying Zhou, and Miguel Soriano (Eds.). Springer International Publishing, Cham, 325–345.

[44] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. 2019. Alembic: Automated Model Inference for Stateful Network Functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 699–718.

[45] Harald Raffelt, Bernhard Steffen, and Margaria Tiziana. 2007. Dynamic Testing Via Automata Learning. 136–152. https://doi.org/10.1007/978-3-540-77966-7_13

[46] Abdullah Rasool, Greg Alpár, and Joeri de Ruiter. 2019. State machine inference of QUIC. *CoRR* abs/1903.04384 (2019). arXiv:1903.04384 http://arxiv.org/abs/1903.04384

[47] David Rupprecht, Kai Jansen, and Christina Pöpper. 2016. Putting LTE Security Functions to the Test: A Framework to Evaluate Implementation Correctness. In *Proceedings of the 10th USENIX Conference on Offensive Technologies* (Austin, TX) *(WOOT'16)*. USENIX Association, USA, 40–51.

[48] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. 2020. Call Me Maybe: Eavesdropping Encrypted LTE Calls With ReVoLTE. In *USENIX Security Symposium (SSYM)*. USENIX Association.

[49] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. 2019. Breaking LTE on Layer Two. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1121–1136. https://doi.org/10.1109/SP.2019.00006

[50] Muzammil Shahbaz and Roland Groz. 2009. Inferring Mealy Machines. In *FM 2009: Formal Methods*, Ana Cavalcanti and Dennis R. Dams (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–222.

[51] Altaf Shaik, Jean-Pierre Seifert, Ravishankar Borgaonkar, N. Asokan, and Valtteri Niemi. 2016. 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.

[52] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. 2017. HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. 521–538. https://doi.org/10.1109/SP.2017.46

[53] M. Tappler, B. K. Aichernig, and R. Bloem. 2017. Model-Based Testing IoT Communication via Active Automata Learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 276–287. https://doi.org/10.1109/ICST.2017.32

[54] Mark Utting and Bruno Legeard. 2006. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[55] Frits Vaandrager. 2017. Model Learning. *Commun. ACM* 60, 2 (Jan. 2017), 86–95. https://doi.org/10.1145/2967606

A APPENDIX

A.1 NAS and RRC Layer procedures

A.1.1 NAS Layer Procedures. We now briefly discuss the NAS layer procedures that are most relevant in the context of our paper (shown in Figure 7, 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 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. Later we show how this message is utilized to create a transparent reset for DIKEUE.

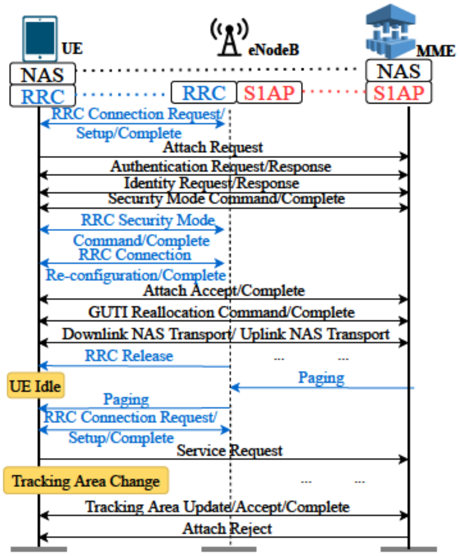

Figure 7: LTE control plane procedures. NAS and RRC layer procedures are shown in black and blue, respectively.

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

Table 8: List of tested devices

Other procedures. The *identification* procedure is used to know the identity, in most cases, the International Mobile Subscriber Identity (IMSI) of the device. 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*. 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 *downlink NAS transport* procedure can be used by the network to send an actual SMS message in the NAS message.

A.1.2 *RRC layer procedures.*

We now briefly discuss the RRC layer procedures that are most relevant in the context of our paper (shown in Figure 7, 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, dedicated 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.

A.2 Model checker performance in equivalence checking

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 8. 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.

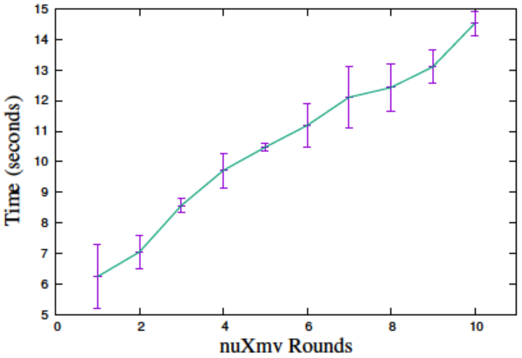

Figure 8: Time required for each round of nuXmv query

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 10. 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 8.

A.3 ACRONYMS

3GPP	Third generation partnership project
ADB	Android Debug Bridge
C-RNTI	Cell Radio Network Temporary Identity
COTS	Commercial Off-The-Shelf
EEA	EPS Encryption Algorithm
EIA	EPS Integrity Algorithm
FSM	Finite State Machine
eNodeB	Evolved NodeB
EPC	Evolved Packet Core
GUTI	Globally Unique Temporary Identity
IMSI	International Mobile Subscriber Identity
LTE	Long Term Evolution
MAC	Message Authentication Code
MitM	Man-in-the-Middle
NAS	Non Access Stratum
OTA	Over-The-Air
RNTI	Radio Network Temporary Identity
RRC	Radio Resource Control
TMSI	Temporary Mobile Subscriber Identity
SDR	Software Defined Radio
SUL	System Under Learning
UE	User Equipment
USIM	Universal Subscriber Identity Module

A.4 Additional discussion

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.

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
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 9: List of input symbols and possible output symbols for each of them. 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

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

Table 10: Performance of FSM equivalence checker.