Logic Gone Astray: A Security Analysis Framework for the Control Plane Protocols of 5G Basebands

Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Yilu Dong, Weixuan Wang, Tianwei Wu, and Syed Rafiul Hussain, *Pennsylvania State University*

https://www.usenix.org/conference/usenixsecurity24/presentation/tu

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Logic Gone Astray: A Security Analysis Framework for the Control Plane Protocols of 5G Basebands

Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid
Yilu Dong, Weixuan Wang, Tianwei Wu, Syed Rafiul Hussain
Pennsylvania State University
{kjt5562, abdullah.ishtiaq, szr5848, yiludong, wjw5351, tvw5452, hussain1}@psu.edu

Abstract

We develop 5GBaseChecker— an efficient, scalable, and dynamic security analysis framework based on differential testing for analyzing 5G basebands' control plane protocol interactions. 5GBaseChecker first captures basebands' protocol behaviors as a finite state machine (FSM) through black-box automata learning. To facilitate efficient learning and improve scalability, 5GBaseChecker introduces novel hybrid and collaborative learning techniques. 5GBaseChecker then identifies input sequences for which the extracted FSMs provide deviating outputs. Finally, 5GBaseChecker leverages these deviations to efficiently identify the security properties from specifications and use those to triage if the deviations found in 5G basebands violate any properties. We evaluated 5GBaseChecker with 17 commercial 5G basebands and 2 open-source UE implementations and uncovered 22 implementation-level issues, including 13 exploitable vulnerabilities and 2 interoperability issues.

1 Introduction

5G devices are pivotal in unlocking the full potential of 5G cellular system's unique features— secure, faster, massive, and ultra-reliable low-latency data communications. These features, powered by significant paradigm shifts across all layers of cellular systems from previous generations of cellular networks, have enabled a plethora of new services, including mobile broadband, augmented reality, and mission-critical services, including public safety and emergency responses. The security of these services critically hinges on the security of 5G devices, also known as modems or basebands. Therefore, 5G baseband implementations must satisfy the security and privacy requirements mandated by the specifications. Failure to satisfy these requirements may result in compromised devices, which often serve as entry points for more catastrophic attacks on broader systems relying on 5G technology, such as healthcare and transportation. Also, the global nature of 5G supply chains further emphasizes the need for robust security, as vulnerabilities introduced at any point in the manufacturing process of 5G devices can have far-reaching consequences. However, due to the extensive volume, complexity, and ambiguities of 5G specifications, achieving an error-free and secure implementation remains a formidable challenge. Since manually analyzing large and complex implementations of 5G basebands is error-prone and time-consuming, *we aim to develop an automated technique to reason about the security and privacy of 5G baseband implementations.*

Only a handful of approaches [15, 58] have attempted so far to analyze 5G basebands. A majority of them, however, merely extend the black-box-based testing schemes developed for analyzing previous generations of cellular devices [15, 42, 44, 50, 51, 53, 54, 58]. Among those, some approaches [15, 51, 53, 58] follow semi-automated techniques, i.e., they manually design test cases by reading the specifications, whereas some others [50] require a reference state machine as a testing oracle to detect logical bugs leading to security flaws. Manually designing test cases is, however, laborious, error-prone, and ineffective in testing a broad range of security-critical behaviors of complex 5G basebands. Nonetheless, the lack of a reference state machine by the cellular standard body, i.e., the Third Generation Partnership Project (3GPP) [1], necessitates the manual construction of an oracle in most prior works, a process that is also both time-consuming and prone to errors. Contrary to black-box schemes, reverse engineering-based analysis [31, 43, 45] requires an enormous manual effort and expertise that cannot be easily transferred to the devices of other models and vendors. This leaves the massive and ubiquitous deployments of 5G devices largely untested and potentially vulnerable.

In this paper, we, therefore, aim to improve the suboptimal state of the existing works and develop a scalable and automated black-box security analysis technique dubbed 5GBaseChecker. We design 5GBaseChecker based on *automata learning* and *differential testing* principles and thus obviates constructing hand-crafted test cases or capturing reference state machines. Performing security analysis in a black-box fashion makes 5GBaseChecker agnostic to the basebands' internal details and underlying microprocessor architecture, and thus makes this framework flexible to be used across different models and vendors of 5G basebands.

At a high level, 5GBaseChecker employs differential testing, in which if two basebands produce two different output sequences for the same input sequence, one of them may violate the security specifications, with the specific one yet to be determined. This approach sidesteps the challenge of manually crafting a reference state machine to serve as an oracle. Besides, rather than manually crafting or arbitrarily searching for deviation-inducing input sequences, 5GBaseChecker first systematically learns the finite state machines (FSMs) of 5G baseband devices in a black-box manner and then checks for differences in the extracted FSMs through differential testing. Finally, 5GBaseChecker uses the deviations as guidance to extract properties from the specification and automates the process of analyzing similar deviations. It also eliminates the need for manually identifying properties prior to testing.

To construct an FSM from 5G baseband implementation, we develop a novel *hybrid and collaborative FSM learning* technique called StateSynth. It combines *passive automata* and *active automata* learning by capitalizing on their strengths, i.e., the ability to learn from traces without executing time-intensive over-the-air queries (passive) and to explore deeper state space (active), and by mitigating their limitations, i.e., poor coverage of learning and a prohibitively high amount of queries and time to construct an FSM. To achieve this, StateSynth first synthesizes an initial FSM of a 5G baseband using the passive automata learning and uses the synthesized FSM to bootstrap the active automata learning phase. This approach helps StateSynth reduce a high number of queries that would otherwise be required by a standalone active learning technique.

Although this hybrid FSM learning minimizes queries and time, still a major challenge for active automata learning is the time-consuming *hypothesis validation* stage. During this stage, the learner requires a large number of *equivalence queries* to identify a *counterexample*, which signifies that the hypothesis model inferred through *learning queries* in the *hypothesis construction* stage is not equivalent to the System Under Learning (SUL) and requires further refinement until no counterexample is observed. StateSynth handles this challenge by introducing *collaborative learning* that utilizes the following insight. The basebands from different vendors follow the same protocol specification; hence, the counterexamples observed during other basebands' FSM learning may serve as potential counterexamples for the current baseband. Based on this, StateSynth first checks if the counterexamples identified for other basebands can be reused for the current baseband under learning. Thus, by learning from each other's counterexamples, StateSynth significantly reduces the total number of queries to learn complex and large 5G FSMs. Although a few prior works [35, 39] attempted to use automata learning for analyzing implementations, they suffer from scalability challenges due to their straightforward applications of active automata learning on the target device. In contrast, 5GBaseChecker's hybrid and collaborative learning mechanism significantly enhances 5GBaseChecker 's efficiency and scalability in constructing FSMs for 5G devices.

We also design DevScan, which takes the extracted FSMs as inputs, pairwise compares them, and identifies unique paths from the initial state to the state where the deviation happens. Unlike prior works [35,39], which can only find a small subset of deviating paths, we propose a *graph traversal scheme* that enables DevScan to identify many unique deviating paths and thus uncover a broader spectrum of potential attack sequences. Since pairwise comparisons of FSMs generate many deviations, manually triaging them to identify the traces violating security properties is tedious and error-prone [35, 39, 50]. To address this challenge, we design DevLyzer, which extracts security properties by following deviating traces and uses them to analyze other deviating traces automatically.

Findings. We evaluated 5GBaseChecker with 17 commercial 5G basebands and 2 open-source UE implementations. We identified 2044 unique deviations, which helped us identify 45 properties. The identified deviations and properties led to 22 unique implementation issues, 13 of which can lead to attacks, and 2 may cause interoperability issues. One of the most critical issues identified by 5GBaseChecker is mishandling security headers of particular messages in 5G devices equipped with Exynos basebands, affecting a majority of mainstream Samsung and Google Pixel devices. Exploiting this vulnerability, the attacker can force a victim device to connect to a fake base station without any authentication. The fake base station can then inject phishing SMSs and eavesdrop on the victim's Internet data.

Contributions. This paper makes the following contributions:

- We design 5GBaseChecker— an automatic, scalable, and black-box security analysis framework based on differential testing that automatically infers the FSM of 5G basebands and uncovers deviations by comparing them.
- We design *hybrid and collaborative FSM learning*, which significantly reduces the overall time for inferring FSMs.
- We develop a differential testing mechanism that takes FSMs as inputs and automatically finds all unique deviating traces in those FSMs leading to a specific deviation.
- We design a deviation analyzer to find security properties corresponding to a deviation and use them to automatically triage similar deviating traces to aid root cause analysis.
- We evaluated 5GBaseChecker with 19 5G basebands and found 22 distinct deviations, including 13 exploitable issues and 2 potential interoperability issues.
- We open-source 5GBaseChecker at: https://github.com/SyNSec-den/5GBaseChecker.

Responsible disclosure. We have reported our findings to the affected vendors. Until now, 5 CVEs have been assigned, with 1 high and 4 medium severity ratings. We are currently cooperating with vendors on patching the affected devices.

2 Preliminaries

5G standalone (SA) preliminaries. The 5G cellular network primarily comprises three major components: 5G Core Network (5GC), 5G-NR base station (gNodeB), and User Equipment (UE). The 5G Core (5GC) consists of several Network Functions (NF), including Access and Mobility Management Function (AMF), User Plane Function (UPF), and Session Management Function (SMF) to offer a range of functionalities, including registration, authentication, mobility management, and voice and data services. The *gNodeBs* are base stations supporting 5G radio technologies using which a gNodeB communicates with UEs through the Radio Resource Control (RRC) protocols in the control plane, relays user plane traffic to UPF, and Non-Access Stratum (NAS) signaling messages to AMF in 5G Core. The *UE* is the device, also known as baseband, that enables network connectivity for end users, e.g., smartphones or other 5G-capable devices. These devices include a Subscriber Identity Module (a physical SIM card or e-SIM) containing the user's credentials, e.g., user identifier and cryptographic keys.

Finite state machine. `5GBaseChecker` uses Mealy machines/Finite State Machines (FSM) to abstract 5G UE control plane protocol implementations. A Mealy machine, M, can be defined as a 6-tuple, $(Q, q_0, \Sigma, \Lambda, \delta, \lambda)$. Here, Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is the input alphabet, and Λ is the output alphabet. The transition function $\delta : Q \times \Sigma \rightarrow Q$ maps a state and an input symbol to the corresponding next state, and the output function $\lambda : Q \times \Sigma \rightarrow \Lambda$ maps a current state and an input symbol to the corresponding output symbol.

Black-box automata learning. Automata learning infers abstract models of systems based on observable behaviors. The observation data can either be sourced from logged system traces or obtained by actively sending (receiving) queries (responses), leading to two categories — *passive* [48] and *active* automata learning [12]. Passive automata learning builds the model based on pre-existing data, such as logs or traces, which makes it suitable for learning models of systems that are difficult or impossible to interact with directly. However, if certain behaviors are not recorded or if the logs are not representative of the whole system behavior, the resulting model can be incomplete or inaccurate. On the other hand, active automata learning algorithms, such as L* [12], interact with the SUL to learn a model. The algorithm for active automata learning usually works in two iterative stages: (i) *hypothesis construction stage* where given the input alphabet I (e.g., a, b, c), the algorithm generates a series of *membership queries* (e.g., abb, abc, $bcca$, $aacb$, $\ldots$), sends them to the SUL and builds a hypothesis FSM consistent with input-output pairs seen so far until the hypothesis FSM is complete and consistent. (ii) *model validation stage* where the learner will query an *equivalence oracle* to check if the current hypothesis FSM model is identical to the black-box system or not. If identical, the algorithm terminates. Otherwise, the learner refines the current FSM hypothesis based on a returned counterexample (CE). Note that ideal *equivalence oracle*s are not available in most cases. However, it is possible to approximate an equivalence query by using a sequence of carefully constructed membership queries [21, 41]. In this work, we refer to these queries as *equivalence queries*.

3 Overview of `5GBaseChecker`

3.1 Problem Statement

The overarching goal of `5GBaseChecker` is to uncover security and privacy flaws in 5G standalone (SA) UE implementations. For this, given a set of basebands $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$, a set of input symbols Σ, and a set of output symbols Λ, `5GBaseChecker` leverages differential testing as a gateway to detect flaws and thus decomposes the overarching goal into three sub-goals: (G1) Uncover the classes of deviant behaviors, also called deviations; (G2) Find the set of properties, i.e., security and privacy requirements whose violations lead to security or privacy flaws; and (G3) Identify instances of property violations for each class of deviant behaviors. To achieve these goals, we ask the following questions: (Q1) Is there a possible input sequence $S_{in} = \sigma_1\sigma_2\ldots\sigma_m$, with $\sigma_j \in \Sigma$ for which a baseband B_i responds with output sequence $S_{out} = \varphi_1\varphi_2\ldots\varphi_m$, where $\varphi_j \in \Lambda$ and S_{out} deviates from the outputs of other basebands B_j $(i \neq j)$? (Q2) If it deviates, which policy ρ is violated by $\langle S_{in}, S_{out}\rangle$ that others do not violate? (Q3) Are there other input/output sequences such as $\langle S'_{in}, S'_{out}\rangle$ which also violate ρ?

3.2 Solution Space

One can approach the problem defined in §3.1 from several directions. First, one can perform fuzz testing with over-the-air (OTA) messages [30] and check if the responses of different devices are different for the same randomly generated input messages. However, randomly generating OTA inputs cannot explore the state space (ineffective) and requires a prohibitively high number of test inputs (inefficient). This is because 5G UE's input space is huge, and there is no useful feedback (e.g., code coverage) to guide the fuzzer due to the black-box nature of 5G UEs.

Another approach is to extract the FSM of 5G basebands using automata learning and then analyze them against the policies (i.e., properties) through formal model checking [34, 40]. However, the 3GPP standard does not provide any curated list of security or privacy requirements, and in many cases, they are implicit. As such, manually enumerating all such requirements from the specification is extremely arduous, if not impossible. In addition, checking a large and complex model against a property often requires frequent expert intervention to avoid trivial counterexamples [32, 34]. Consequently, this approach does not scale well with an increased number of basebands under test.

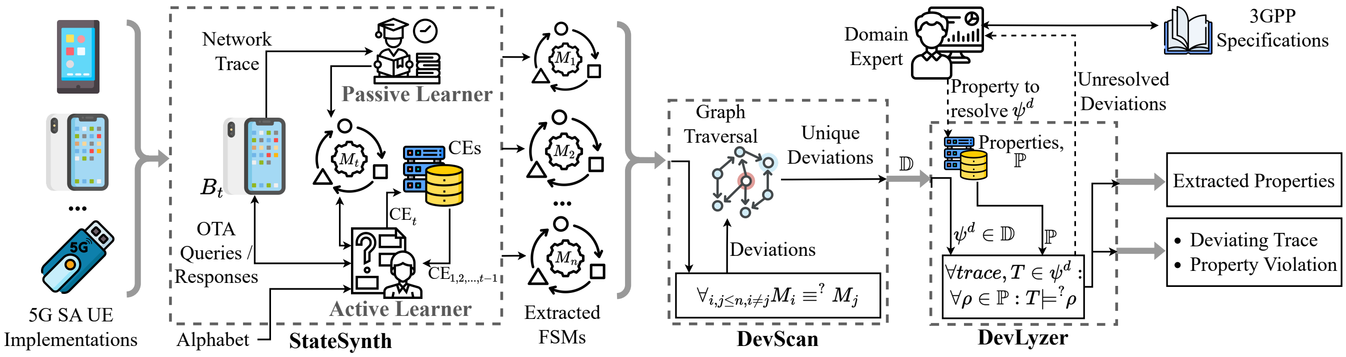


Figure 1: Overview of 5GBaseChecker.

3.3 Solution Sketch of 5GBaseChecker

5GBaseChecker works with three main components: StateSynth, DevScan, and DevLyzer as shown in Figure 1. StateSynth first extracts the FSMs $\mathcal{M} = \{M_1, M_2, \dots M_n\}$ of basebands $B_1, B_2, \dots B_n \in \mathcal{B}$. 5GBaseChecker then utilizes the extracted FSMs and finds out the security policy violations in each FSM $M_i \in \mathcal{M}$. To do this, DevScan of 5GBaseChecker takes the extracted FSMs for all implementations as input and yields a set of deviations as outputs. By deviations, we mean that for a particular input sequence, not all the FSMs respond with the same output sequences. As such, for a particular input sequence, there can be multiple distinct output sequences that do not match with each other.

Definition 3.1 (**Deviation**). Suppose for an input sequence S_{in} as defined in §3.1, the output sequences in $M_1, M_2, \dots M_n \in \mathcal{M}$ are $S^1_{out}, S^2_{out}, \dots, S^n_{out}$, respectively. Among those, let there be k distinct deviating outputs $\gamma^{d,1}_{out}, \gamma^{d,2}_{out}, \dots, \gamma^{d,k}_{out}$ for S_{in}, where $1 < k \leq n$, $(p \neq q) \implies \gamma^{d,p}_{out} \neq \gamma^{d,q}_{out}$ and $\forall M_l \in \mathcal{M} : (\exists j : (1 \leq j \leq k) \wedge S^l_{out} = \gamma^{d,j}_{out})$. A deviation ψ^d (also termed as a deviation class) is defined by the input sequence S_{in} and all k distinct outputs $\gamma^{d,1}_{out}, \gamma^{d,2}_{out}, \dots, \gamma^{d,k}_{out}$. We denote the set of all extracted deviations as $\mathbb{D}$. Moreover, we define a deviating trace T as a pair of the deviation-inducing input and a deviating output, i.e., $\langle S_{in}, \gamma^{d,j}_{out} \rangle$, where $1 \leq j < k$. For instance, a deviation $\psi^d \in \mathbb{D}$ can have the input sequence $S^d_{in} = x_1 x_2 x_3 x_4$, and three distinct outputs $\gamma^{d,1}_{out} = y_1 y_2 y_3 y_4$, $\gamma^{d,2}_{out} = y_1 y_2 y_3 y_5$, and $\gamma^{d,3}_{out} = y_1 y_6 y_3 y_7$, where $\langle S^d_{in}, \gamma^{d,1}_{out} \rangle$, $\langle S^d_{in}, \gamma^{d,2}_{out} \rangle$, and $\langle S^d_{in}, \gamma^{d,3}_{out} \rangle$ are deviating traces.

For each deviating trace $T \in \psi^d$, the DevLyzer component of 5GBaseChecker first finds a set of properties $\mathcal{P}_d$ relevant to ψ^d from the specifications. DevLyzer uses ψ^d as guidance to find such properties from specifications. The sets of all identified properties for all $\psi^d \in \mathbb{D}$ are referred to as $\mathbb{P}$. DevLyzer then automatically tests if each $T \in \psi^d$ satisfies all $\rho \in \mathbb{P}$, i.e., $T \models^? \rho$. This reduces the significant manual effort by a security expert to analyze deviations.

3.4 Threat Model

We consider an active attacker with Dolev-Yao capabilities on the wireless channel between a victim UE and a gNodeB [24]. In this model, the attacker can set up a Fake Base Station (FBS) or a Machine-in-the-middle (MitM) relay to modify, inject, replay, drop, or observe packets on the channel. However, we assume the cryptographic constructs to be perfectly secure, i.e., the attacker cannot craft a message with a valid message authentication code (MAC) or cannot encrypt/decrypt a message without cryptographic keys. In the cellular domain, these assumptions are reasonable, as seen in prior works [34, 35, 42, 50].

4 Challenges and Solutions

4.1 Challenges of 5GBaseChecker Design

To solve the problems in §3.1, we find the following challenges and limitations of the prior approaches, which motivate us to design our proposed approach 5GBaseChecker.

C1: Slow inference of 5G FSM with active learning. The standard active automata learning algorithm starts from an empty knowledge base to build the FSM. As a result, it learns the basic interactions of the protocol very slowly during its first few iterations of hypothesis construction. Further, as discussed in §2, active automata learning algorithms require an equivalence oracle to validate the hypothesis models, which is approximated using carefully crafted membership queries in prior works [21, 41]. However, this approximation results in a very large number of queries being generated in the model validation stage. For example, Wp-method [41] requires an order of $O(n^2 |\Sigma|^{k-n+1})$, where Σ is the input alphabet of the FSM, n is the number of states in the FSM, and k is the upper bound on the number of states of the target system. As a result, the model validation stage often requires 5-10 times more queries than the hypothesis construction stage [23, 35, 39, 57]. As executing an OTA query in 5G requires a substantial amount of time (∼40 seconds/query), this stage becomes a critical bottleneck for active automata learning-based FSM extraction.

C2: Difficulty in finding unique deviations. Prior works [28, 35, 39] have developed automated ways to find deviations from the extracted FSMs. Although these works are effective in identifying deviations, they do not explore unique variations of a single deviation. In other words, in case of a deviation, prior works often fail to identify multiple traces between the initial state and a deviation-inducing state— the state at which the outputs of two or more FSMs differ for the same input. Multiple such deviations imply that attackers can execute the same attack using various attack sequences.

C3: No comprehensive list of security policies. Once unique deviations are detected, one must check if deviating traces violate any security properties. However, it is difficult and time-consuming to find a comprehensive list of critical security properties without any hints as there are thousands of properties in 5G specifications; for instance, the 5G NAS specification of Release 17 itself has ∼8000 such requirements [10].

C4: Lack of automation in triaging deviations. If we test a large number of 5G UEs, there would be a large number of deviations among the extracted FSMs. However, following previous approaches [35, 39], manually triaging each such deviation consumes a huge amount of time and effort. Even when security properties are in place, one has to check the properties against complex and large FSMs, which also requires significant time and frequent expert interventions.

4.2 Our Approach

A1: Combining passive automata learning and counterexample reuse. To address **C1**, we propose the idea of *hybrid and collaborative learning*. In *hybrid learning*, we combine active and passive automata learning in order to reduce the number of OTA learning and equivalence queries. First, we leverage passive automata learning [38] to synthesize an initial FSM called M^p from the available 5G network traces. We then use M^p as a guide for active learning. In other words, M^p helps active learning optimize the number of queries and time to refine the hypothesis models. On the other hand, since all basebands implement the same protocol specified by 3GPP, the FSMs would largely be similar, and any counterexamples (CEs) found during the FSM construction of one device are highly likely to be the CEs for other basebands as well. Thus, using *collaborative learning* during the model validation stage, we first try to reuse the CEs found during the FSM constructions of other devices instead of directly executing the expensive off-the-shelf model validation algorithms (e.g., Wp-method). If any of the CEs are valid, we directly use them to refine the hypothesis model of the current SUL. Identifying CEs in such a collaborative way significantly reduces equivalence-checking queries and time.

A2: Graph traversal-based model refinement. To address **C2**, similar to previous work [35, 39], we first convert the differential testing problem to a model-checking problem with a safety property. This eliminates the requirement for expert intervention and a comprehensive list of security properties. However, to address the limitations of previous works, which cannot find all unique deviations, we propose a graph traversal mechanism. Specifically, when a deviation-inducing input sequence is found, we perform a graph traversal from the initial state to the state where the deviation is found and log all unique paths that can lead to the deviation.

A3: Guided search of security policies using deviation. To address challenge **C3**, we leverage the obtained deviations to look up the specifications and extract corresponding security policies. More specifically, we manually examine the specification to detect the correct output behavior corresponding to the input sequence of a deviation. As an example, for an input sequence that provides a plaintext `ue_info_req` without performing access stratum (AS), i.e., RRC layer's security activation, consider an output sequence where the device responds with `ue_info_resp`. This behavior contrasts with other devices that discard `ue_info_req`. When analyzing such a deviation, we can easily obtain the required security property by examining the portions of the specification that deal with unprotected `ue_info_req` messages. Thus, the deviation guides us to obtain the security property of not accepting plaintext `ue_info_req` before AS security activation.

A4: Efficient triaging of deviations. To address challenge **C4**, we adopt an iterative approach. We formally verify the extracted properties against automatically generated formal models of each of the deviating traces instead of verifying the entire FSMs of devices. This approach eliminates the time and scalability challenges of formally verifying the extracted FSMs with desired properties. Additionally, we observe that a single property often helps us automatically resolve multiple deviations. As a result, we need to manually analyze only a small subset of all identified deviations (as discussed in §9.5), significantly reducing the manual effort required.

5 StateSynth: FSM Construction

Given a set of input and output alphabets, Σ and Λ, respectively, StateSynth aims to efficiently extract the FSM for each 5G baseband under test. Figure 1 also shows the overview and components of StateSynth, whereas Figure 2 shows its workflow with an example. In what follows, we discuss in detail the design and workflow of StateSynth.

5.1 Hybrid Learning for Bootstrapping

Passive FSM construction. As discussed in §2 and §4.2, passive automata learning synthesizes an FSM from a given execution traces instead of actively interacting with the system under learning. This approach requires a diverse set of traces to build a comprehensive FSM suitable for analyzing the security of the system. However, only a limited 5G standalone

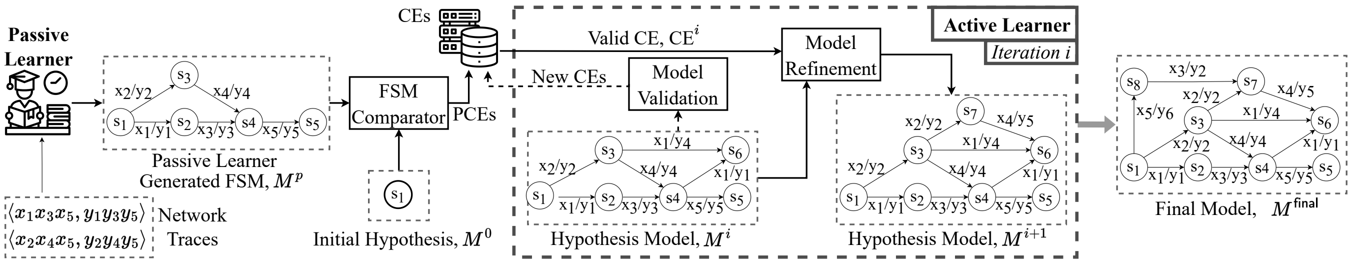


Figure 2: Workflow of StateSynth with an illustrative example.

(SA) control plane traces can be observed by monitoring a UE's regular interactions with 5G networks. Due to the absence of diverse traces for 5G UEs, it is thereby challenging to apply passive learning to comprehensively infer 5G basebands' FSMs. To tackle this, we first utilize the limited availability of 5G SA traces to construct only an initial FSM using passive learning. We then use that FSM to bootstrap/guide *Active Learner*, specifically during the first few iterations of *Active Learner*'s model validation phase. This bootstrapping helps StateSynth significantly reduce OTA queries and time to learn diverse system behaviors.

Note that the limited 5G SA traces contain only regular (i.e., positive) behavior, whereas the classic passive automata learning algorithm, such as *Regular Positive and Negative Inference (RPNI)* [49], a widely used technique in networks and cyber-physical systems, requires both positive and negative execution traces to infer a system's FSM. To sidestep this, we leverage Trace2Model [38], a passive automata learning technique that learns an FSM only from positive traces. To this end, we first collect network traces of the registration procedures between the baseband and a 5G SA network and represent each input/output in that trace as an observation for Trace2Model. Trace2Model uses a program synthesis technique called *synthesis from examples* to construct *transition predicates* and then uses the predicates to find counterexamples used for refining the hypothesis model. We use the traces of registration procedure because the scope of our analysis is mainly the NAS and RRC procedures related to registration.

As an example, consider the two traces– $\langle x_1x_3x_5, y_1y_3y_5\rangle$ and $\langle x_2x_4x_5, y_2y_4y_5\rangle$. Here, a trace is a tuple of an input sequence and corresponding output sequence. The *Passive Learner* takes these traces as input and generates the initial state machine M^p, as shown in Figure 2. In this figure, self-loop transitions are not shown for clarity and conciseness.

Trace collection. We collect 5G SA traces by connecting the devices to both (i) open-source 5G testbeds, and (ii) nearby commercial 5G base stations with valid user credentials. This ensures that the traces are diverse, covering behavior in both testbed environments and commercial network settings.

Bootstrapping active learning with passive learner-generated FSM. To complement the behavior learned and captured in M^p by *Passive Learner*, StateSynth's *Active Learner* first assimilates the information present in M^p and thus bootstraps the overall learning. We use TTT [37] as the active learning algorithm for 5G basebands. As discussed in §2 and §4.2, *Active Learner* systematically generates queries as sequences of input alphabets, observes the responses, and infers the underlying FSM by iteratively constructing and validating hypothesis models. Since 3GPP specifications do not provide a reference FSM for 5G basebands, during the model validation phase, *Active Learner* uses the Wp-method [41] as an approximate equivalence checking mechanism, which, however, requires a prohibitively high number of queries and time to find counterexamples in hypothesis models. This becomes particularly worse for 5G UEs as their protocols are complex and have a higher number of input and output alphabets than previous generations. This results in a significant slowdown during the initial iterations of FSM learning.

StateSynth bootstraps *Active Learner* by first utilizing M^p during *Active Learner*'s model validation phase. More specifically, StateSynth compares the TTT-generated initial (empty) hypothesis M^0 with M^p, and treats the differences, i.e., deviating traces between them as *potential counterexamples (PCEs)*. During model validation, instead of using the Wp-method in the first place, *Active Learner* considers these PCEs as candidate CEs and uses them first to refine the hypothesis if they are proved valid. This helps *Active Learner* quickly bootstrap the learning of regular 5G protocol interactions (e.g., a correct flow of the registration procedure) by consuming the knowledge that *Passive Learner* learns from network traces. For example, in Figure 2, initial hypothesis M^0 contains only one state with self-loop transitions for each symbol in the alphabet. By comparing M^0 with M^p, as shown in Figure 2, we find several PCEs, e.g., $\langle x_1x_3, y_1y_3\rangle$, $\langle x_2x_4x_5, y_2y_4y_5\rangle$, etc.

5.2 Collaborative Learning for Validation

Since 5G SA UEs implement the same protocol following 3GPP specifications, their FSMs are also largely similar. We use this insight to propagate the learned behavior from one UE to the next if such behavior is valid. More specifically, we store the CEs found during model validation of one baseband as potential counterexamples for other basebands. During model validation of other basebands, we first apply those PCEs if valid. To check the validity, the StateSynth executes the PCEs as OTA queries to the device under learning. If the observed response does not match with the hypothesis model, StateSynth marks those as valid CEs. Such *collaborative learning* significantly reduces the number of OTA queries and time. Finally, after applying all PCEs, to find remaining counterexamples and refine the learned model, we use Wp-method [41] in a limited way for model validation until the termination condition is met as defined in §5.4. We detail the algorithm of StateSynth in Appendix B.

We demonstrate the workflow of *Active Learner*, showing one iteration of model validation and refinement for baseband B_t in Figure 2. In the i^{th} iteration, *Active Learner* initially has the hypothesis model M^i. If there are any unused CEs remaining, *Active Learner* first checks if such CEs are valid and then uses a valid one for M^i refinement. Otherwise, as shown in Figure 2, *Active Learner* performs model validation with the Wp-method to find a new CE, uses it (if any) for model refinement, and saves it as a PCE to apply to other basebands. For example, assume that for input sequence $x_2x_2x_4$, B_t responds with $y_2y_2y_5$. However, according to M^i, it responds to the last message with y_4. Thus, in this iteration, *Active Learner* uses $\langle x_2x_2x_4, y_2y_2y_5\rangle$ as CE^i, refines the model, and generates a new hypothesis, M^{i+1}, as shown in Figure 2. Finally, this iterative model refinement and validation is performed until termination, when *Active Learner* provides the final learned model, M^{final} for B_t.

5.3 Alphabet Selection

In StateSynth, the number of queries generated by TTT and Wp-method [41] increases significantly with alphabet size (state explosion). Therefore, one needs to carefully select the alphabet in a way that can strike a balance between the scope and the termination of FSM learning.

Selection strategy. Since the goal of this work is to identify security policy violations in 5G SA control plane protocol implementations, we create alphabets based on critical NAS and RRC layers' messages and 3GPP-provided high-level security policies: confidentiality, integrity, and non-replayability. More precisely, we use the following four strategies. (A) We consider messages required to learn valid/regular protocol flows, such as the completion of the registration procedure. This includes both plaintext messages allowed before NAS and AS security activation and security-protected messages (i.e., ciphered and/or integrity protected)[1] used after security activation. (B) For messages that must not be accepted without security protection (i.e., encryption and/or integrity), we consider their plaintext and/or non-integrity-protected counterparts. (C) For non-replayability, we construct the replay version of integrity-protected messages. (D) We also take security headers of messages into account as they influence the acceptance of plaintext and security-protected messages.

Among the 16 security-header types, security headers 0 to 4 denote plain (0), integrity protected (1), integrity protected and ciphered (2), integrity protected with a new security context (3), and integrity protected and ciphered with a new security context (4), while the remaining 11 are reserved. We consider security header types 0, 2, 3, and 4 in our alphabet set. We include header 0 to test plaintext messages. Moreover, headers 2 and 3 are required to learn the regular flow of the protocols for protected messages. They are vital for security analysis because some exploitable vulnerabilities can still occur after receiving protected messages, e.g., accepting plaintext *Authentication Request* messages after security activation, and replayed *Security Mode Command* messages after registration. We do not include header 1 as it is not required to learn the regular protocol flow. Finally, header 4 is a special type used only for an uplink message *Security Mode Complete*, and thus, receiving this header may lead to unexpected behavior in the UE under test. With these strategies, we construct the alphabet based on TS 24.501 version 17.8.0 [10] and TS 38.331 version 17.6.0 [11], as shown in Table 9 in the Appendix.

Scope. The alphabet used by StateSynth spans a subset of the 5G SA control plane procedures. More specifically, in the NAS layer, the alphabet covers registration, identification, authentication, security mode control, UE configuration update, and de-registration procedures. On the other hand, in the RRC layer, the alphabet covers AS security mode control, UE capability transfer, counter check, UE information request, and RRC reconfiguration procedures. We excluded NAS layer reject messages and RRC layer reject and release messages as they would drop the connection and, consequently, hinder FSM construction. Also, currently, open-source 5G stack implementations that we employed to send messages to the UE do not support RRC Resume, paging, and handover procedures, and implementing them would require significant engineering efforts. We exclude these procedures from the scope of 5GBaseChecker as well.

5.4 Termination Strategy

The termination condition is critical in active learning. Terminating early results in incomplete FSM construction, whereas keeping StateSynth running even after capturing a representative FSM leads to superfluous OTA queries and time. As seen in prior works [23, 35, 39], most OTA queries are generated in model validation using the Wp-method [41] during FSM construction. To reduce such queries and balance between early termination and FSM completeness, in this work, after using the PCEs to refine the hypothesis models, we use Wp-method in a controlled way through configurable parameters N_0, N_1, and N_2 to further validate and refine the hypotheses. In summary, we use the following termination strategy.

- If any of the output symbols (Table 9 in Appendix) have not

[1] Security-protected messages are considered only for FSM learning. They do not reflect the attacker's capabilities.

been observed yet, StateSynth continues FSM extraction.

- For the first UE, when *Active Learner* does not have PCEs from any previously learned basebands, we limit the number of equivalence queries to a certain value N_0 for each iteration of the Wp-method. If a CE is not found within N_0, model construction terminates. Otherwise, we refine the model with the CE and re-run model validation. In contrast to DIKEUE's termination strategy [35], which terminates whenever the registration procedure is completed, this ensures learning implementation behavior even after the procedure.
- For the subsequent phones, if another baseband from the same vendor (e.g., Qualcomm, MediaTek, etc.) has already been learned, we set the limit of equivalence queries per round of model validation at N_1, such that $N_1 < N_0$. This utilizes the insight that FSMs from the same baseband vendor are similar and may require fewer equivalence queries.
- If the device under learning is the first phone from its baseband vendor, we set the above-mentioned limit to N_2, such that $N_1 < N_2 < N_0$. It ensures that the first baseband under learning from a vendor is explored sufficiently.

5.5 Efficient and Consistent Learning

OTA query caching. As discussed in prior works [13, 55], *Active Learner* can generate some equivalence queries in model validation, which are already executed during hypothesis construction. Following this insight, we store the results of all OTA queries in a cache to use in subsequent queries for the same baseband.

Avoiding inconsistency. Due to unreliable wireless communication channels used for transmitting OTA messages and responses, *Active Learner* may experience observational inconsistencies in the responses. To ensure consistent learning, we apply majority voting following prior works [35, 47]. In this scheme, we run an OTA query twice and use the output if the outputs are the same. Otherwise, we run the query another time to resolve the correct output.

Minimizing duplicate states. We also observe that some UE implementations incorrectly reset their state machines after receiving particular messages (e.g., `auth_req_plain`) at certain states and then restart learning of the entire FSM from scratch from those states. This leads *Active Learner* to spawn many duplicate states in the hypothesis models, causing a large amount of similar query generation and delay in termination. To address these inconsistent behaviors, *Active Learner* automatically detects such behaviors during each query execution and interprets subsequent outputs in that query as `null_action`.

6 DevScan: Finding Deviations

DevScan takes the extracted FSMs of basebands as inputs and automatically identifies the set of deviations.

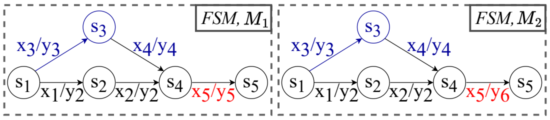


Figure 3: Example deviations in two FSMs.

6.1 Unique Paths For Deviations

Prior research has utilized deviations to detect vulnerabilities in protocol implementations [25, 35, 39]. However, efficiently identifying all deviations from a set of implementations or even a set of representative FSMs is non-trivial because the search space is exponentially large depending on the size of the alphabet and the length of the traces. Although previous works have identified deviations in a reasonable amount of time, they lack the support for identifying all unique paths leading to these deviations [35, 39]. As all these unique deviations are essential for a comprehensive analysis of the FSMs, we opt for a graph traversal algorithm to find all unique paths from the initial state to a deviation-inducing state in an FSM leading to a deviation.

For example, Figure 3 shows deviations in FSMs M_1 and M_2. Prior tools [35, 39] can only identify the deviation with deviating traces $\langle x_1x_2x_5, y_1y_2y_5\rangle$ and $\langle x_1x_2x_5, y_1y_2y_6\rangle$, whereas there is another path in these FSMs (highlighted in Figure 3) leading to another unique deviation with deviating traces $\langle x_3x_4x_5, y_3y_4y_5\rangle$ and $\langle x_3x_4x_5, y_3y_4y_6\rangle$. DevScan, with the help of graph traversal-based model refinement, successfully identifies such unique deviations in extracted FSMs.

6.2 Graph Traversal-Based Model Refinement

To identify all unique paths, DevScan at first uses a prior tool [39] to find the set of deviations by performing pairwise comparison. For each of the deviations, in both FSMs, DevScan inspects the state in which the deviation has occurred and performs a depth-first-search-based algorithm to find all paths from the initial state to the deviation-inducing state. For all these paths, DevScan identifies the sequences of input symbols along the paths and checks both FSMs to discover if these new sequences also lead to deviations.

7 DevLyzer: Triaging Deviations

Given the set of deviations $\mathbb{D}$, DevLyzer identifies a set of properties $\rho \in \mathbb{P}$ where ρ is violated by at least one deviating trace $\langle S_{in}, \gamma^{i,j}_{out}\rangle \in \psi^i$ of a baseband. It also finds deviating traces in other deviations ψ^k that also violate ρ. DevLyzer aims to minimize the manual effort required to extract relevant properties and corresponding violations of the properties.

7.1 Extracting Properties

For each deviation (also called *deviation class*) in $\mathbb{D}$, DevLyzer iteratively finds relevant properties and checks for violations of those properties by basebands. It compiles the extracted properties represented with Linear Temporal Logic (LTL) formula into $\mathbb{P}$. In each iteration, DevLyzer attempts to *resolve* any unresolved deviation $\psi^d \in \mathbb{D}$ by leveraging the properties in $\mathbb{P}$. We consider a deviation resolved when we have at least one property $\rho \in \mathbb{P}$ for each pair of deviating outputs in the deviation class such that one of the deviating outputs satisfies ρ but others violate. Formally, a deviation ψ^i is considered *resolved* only if:

$$\forall_{p \neq q} \gamma_{out}^{d,p}, \gamma_{out}^{d,q} \in \psi^d : \exists \rho \in \mathbb{P} : \langle S_{in}, \gamma_{out}^{d,p} \rangle \models \rho \wedge \langle S_{in}, \gamma_{out}^{d,q} \rangle \not\models \rho \tag{1}$$

Initially, all deviations ψ^d are *unresolved*, and $\mathbb{P} = \emptyset$. During each iteration, DevLyzer first picks an unresolved deviation ψ_u^d and provides it to the security expert for manual analysis. This is a guided property extraction procedure, as the deviating trace will provide guidance to the security expert regarding what and where (e.g., message sequences and procedures) to look for in the specification to extract properties. The expert then consults with the 5G specification to identify the correct output behavior for the input sequence S_{in} of ψ_u^d and represent that as an LTL property ρ. Also, if there are k_s number of correct output behaviors prescribed by the specification for input S_{in}, we represent k_s properties $\rho_1, \rho_2, \ldots, \rho_{k_s}$ such that each ρ_m $(1 \leq m \leq k_s)$ characterizes one of the correct behaviors. Also, if there are k_w wrong outputs for S_{in}, we write properties $\rho_1, \rho_2, \ldots, \rho_{k_w}$ such that each ρ_n $(1 \leq n \leq k_w)$ is violated only by one of the outputs $\gamma_{out}^{j,j} \in \psi_u^i$. DevLyzer adds all such properties to $\mathbb{P}$.

7.2 Checking Properties For Violation

For all trace $\langle S_{in}, \gamma_{out}^{j,j} \rangle$ in a deviation $\psi^i \in \mathbb{D}$, DevLyzer checks whether the trace satisfies all properties $\rho \in \mathbb{P}$. To perform this, DevLyzer first automatically models each trace as an FSM $M_{dev}^{i,j}$, and performs model checking (using nuXmv [16]) to determine if $M_{dev}^{i,j} \models \rho$. If a $\rho \in \mathbb{P}$ is violated by any $\langle S_{in}, \gamma_{out}^{j,j} \rangle$, DevLyzer can immediately infer that all $b_l \in \mathcal{B}$ that outputs $\varphi_{out}^l = \gamma_{out}^{j,j}$ when provided S_{in} also violates the property ρ.

Automatic construction of a model from a trace. To construct $M_{dev}^{i,j}$ for $\langle S_{in}, \gamma_{out}^{j,j} \rangle$, DevLyzer creates $|S_{in}|$ number of states for $M_{dev}^{i,j}$, where at each state s_k $(1 \leq k < |S_{in}|)$ there is only one transition that takes the k-th message of S_{in} as input, outputs the k-th message in $\gamma_{out}^{j,j}$ and goes to state s_{k+1}. Since $M_{dev}^{i,j}$ represents the model of a single trace, its size is much smaller than extracted FSMs for 5G devices. This helps DevLyzer automatically analyze them against extracted properties. DevLyzer thus iteratively triages all unresolved deviation ψ_u^i and marks them as resolved when the condition in Equation 1 is satisfied.

7.3 Illustration of DevLyzer

Consider two deviations ψ^1 and ψ^2, as shown in Figure 4. For ψ^1, the input sequence is $S_{in}^1 = x_1 x_2 x_3 x_4 x_5$, and it has two deviating output sequences $\gamma_{out}^{1,1} = y_1 y_2 y_3 y_4 y_5$ and $\gamma_{out}^{1,2} = y_1 y_2 y_3 y_4 y_7$. For ψ^2, we have $S_{in}^2 = x_1 x_2 x_8 x_3 x_5$ and two deviating outputs $\gamma_{out}^{2,1} = y_1 y_2 y_8 y_3 y_5$ and $\gamma_{out}^{2,2} = y_1 y_2 y_8 y_3 y_7$.

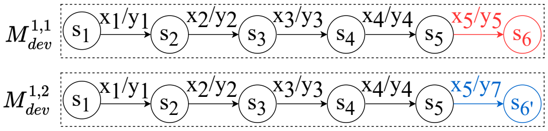


Figure 4: Example models for $M_{dev}^{1,1}$ and $M_{dev}^{1,2}$ for the example in 7.3. Colored areas show the point of deviation.

Initially, $\mathbb{P}$ is $\emptyset$, and all deviations are unresolved. Suppose DevLyzer first triages ψ^1. It creates models $M_{dev}^{1,1}$ and $M_{dev}^{1,2}$ using $(S_{in}^1, \gamma_{out}^{1,1})$ and $(S_{in}^1, \gamma_{out}^{1,2})$, respectively, as shown in Figure 4. To resolve ψ_1, we assert property ρ_1: *After response y_3, if x_5 is provided as input, it will always be responded with y_5*. We represent ρ_1 as an LTL formula as $\mathsf{G}(\mathsf{output}{=}\mathsf{y}_3) \implies (\mathsf{G}(\mathsf{input}{=}\mathsf{x}_5) \implies \mathsf{X}(\mathsf{output}{=}\mathsf{y}_5))$, which the output sequence $\gamma_{out}^{1,1}$ does not violate but $\gamma_{out}^{1,2}$ does. DevLyzer utilizes model checking and finds out $M_{dev}^{1,1} \models \rho_1$ but $M_{dev}^{1,2} \not\models \rho_1$ since output $\gamma_{out}^{1,2}$ violates property ρ_1. Now, ψ^1 will be marked as resolved, and $\mathbb{P}$ will be $\{\rho_1\}$. DevLyzer also detects the devices producing $\gamma_{out}^{1,2}$, which violates ρ_1. Next, DevLyzer examines ψ^2, and builds models $M_{dev}^{2,1}$ using $(S_{in}^2, \gamma_{out}^{2,1})$ and $M_{dev}^{2,2}$ using $(S_{in}^2, \gamma_{out}^{2,2})$. It will test all properties in $\mathbb{P}$ and observe $M_{dev}^{2,1} \models \rho_1$ but $M_{dev}^{2,2} \not\models \rho_1$. Thus, it will mark ψ^2 as resolved. violating ρ_1. Here, DevLyzer saves the manual effort to analyze ψ_2. We summarize DevLyzer in Algorithm 1 in Appendix A.

8 Implementation

StateSynth. For the *Passive Learner* of StateSynth, we use Network Signal Guru [5] to collect the network traces and Trace2Model [38] to generate an FSM. *Active Learner* is built on TTT's [37] and Wp-method's [41] implementations in LearnLib [36]. For the termination strategy, we empirically set $N_0 = 10000$, $N_1 = 1000$, $N_2 = 2000$, which preserves the balance between termination and coverage for the 5G UE implementations (listed in Table 8).

As the *Active Learner* needs active interactions with the basebands, we build an adapter consisting of gNodeB and 5G core to translate abstract symbols provided by the learning algorithm to over-the-air messages and vice-versa. For this, we modify existing open-source 5G SA gNodeB and 5G core (5GC) implementations. These modifications are required to enable the open-source gNodeB and 5GC to send arbitrary sequences of OTA messages as generated by *Active Learner*.

Moreover, available open-source gNodeB implementations do not support a broad range of frequencies. To facilitate learning of all implementations under test, we employ two different gNodeB implementations (OpenAirInterface [7] and srsRAN [8]). We modify Open5GS [6] for the 5G core of the adapter.

DevScan and DevLyzer. As discussed in §6, we extend the FSM equivalence checker of BLEDiff [39] to implement DevScan. DevLyzer uses nuXmv [16] for model checking.

9 Evaluation

We evaluate 5GBaseChecker with respect to the following research questions:

- **RQ1.** How effective is 5GBaseChecker in finding deviations and security policy violations (§9.2)?
- **RQ2.** How efficient is StateSynth in extracting FSMs from baseband implementations (§9.3)?
- **RQ3.** What is the efficacy of DevScan in detecting deviations (§9.4)?
- **RQ4.** How efficient is DevLyzer in finding security policies and analyzing deviations (§9.5)?'
- **RQ5.** How does 5GBaseChecker perform compared to existing UE testing approaches (§9.6)?

9.1 Evaluation Setup

To evaluate 5GBaseChecker, we analyze 17 Commercial off-the-shelf (COTS) 5G SA capable devices, and 2 open-source 5G UE implementations— srsRAN UE (srsUE) [8] and OpenAirInterface UE (OAI UE) [7], as listed in Table 8. We use USRP B210 as our radio front end to test COTS 5G SA devices and leverage software-simulated radio connection to test srsUE and OAI UE. We perform the evaluations on an Ubuntu 20.04 machine with an AMD Ryzen 9 7950X3D CPU and 64 GB DDR5 RAM.

9.2 Deviations and Flaws

To address **RQ1**, we use StateSynth to extract FSMs from the implementations and analyze the deviations found by DevScan through DevLyzer. 5GBaseChecker uncovers 22 unique issues, 20 of which are newly discovered in 5G UEs, including 13 attacks and 2 potential interoperability issues. We present these issues in Table 2 and categorize them into 12 types according to the flaw types in Table 1.

9.2.1 Mishandling Messages ($I_1 - I_7$)

● **Mishandle unprotected message before AS security activation (I_1-I_2).** As per TS 38.331 [11], *UE Information Request* and *Counter Check* messages should not be accepted before AS security activation. Correctly implemented devices shall drop these messages before AS security activation. By observing the responses to these messages, the attacker can first fingerprint devices up to the baseband manufacturer level [50]. The attacker can then launch other attacks such as 5G AKA bypass discussed in §10.1.

Flaw Type		**Description**
M	M1	Mishandle unprotected message before authentication
	M2	Mishandle unprotected message after authentication but before NAS security activation
	M3	Mishandle unprotected message after NAS security activation, before AS security activation and before registration completed
	M4	Mishandle unprotected message after NAS security activation and registration completed, but RRC security not activated
	M5	Mishandle protected message after NAS and RRC security activation, but before registration completed
	M6	Mishandle sequence number for certain message type
S	S	Accept invalid security header type for certain message type
I	I	Accept prohibited IE values
U	U1	Incomplete information for mandatory features/messages
	U2	Incomplete information for optional feature
	U3	Conflicting information
D	D	Design specification issue

Table 1: Description of flaw types. **M**: mishandling messages, **S**: mishandling security headers, **I**: mishandling information elements (IEs), **U**: underspecification, **D**: Design flaw.

● **Mishandle unprotected message after NAS security activation (I_3, I_4).** Based on TS 24.501 [10], all NAS communications should be ciphered and integrity protected after NAS security activation. However, some UEs accept *Identity Request* and *Authentication Request* in plaintext even after NAS security context establishment. As a result, the attacker can capture the plaintext *Authentication Request* message for the victim UE from a benign 5G network and send this message later to the victim UE. Since the victim UE accepts and responds to this message, an attacker can exploit this vulnerability to launch traceability [14] and SUCI-Catching [20, 33] attacks or regenerate session keys, causing a key desynchronization between the victim UE and the AMF.

● **Mishandle protected message after NAS and AS security activation (I_5).** Several UEs accept *Deregistration Request* and deregister from the core network before the registration procedure is completed. However, the 5G NAS specification [10] mandates that the UE has to proceed with the registration process by discarding the *Deregistration Request*. This may result in interoperability issues during handover.

● **Mishandle sequence number for certain message types (I_6, I_7).** According to TS 24.501 [10], a replayed NAS message cannot be accepted after security establishment. However, some UEs accept replayed *NAS Security Mode Command* messages during the registration procedure (**I_6**), and some UEs accept this message even after the registration is complete (**I_7**). This is fatal since the attacker can exploit the *NAS Security Mode Command Replay* message to perform traceability attacks [32].

9.2.2 Mishandling Security Headers ($I_8 - I_9$)

The *Registration Accept* and *Configuration Update Command* messages should only be accepted using security header type 2 (integrity protected and ciphered) [10]. However, in **I_8**, we

Issue	Description	Flaw Type	Impact	New
I_1	UE accepts `ue_info_req_plain` before AS security activation	M1, M2, M3, M4	F	●
I_2	UE accepts the `counter_check_plain` before AS security activation	M1, M2, M3, M4	F	○
I_3	UE accepts `identity_req_plain` after NAS security activation	M3, M4	IL	●
I_4	UE accepts `auth_req_plain` after NAS security activation	M3, M4	LL, DoS	●
I_5	UE accepts `dereg_req_protected` before completing the registration procedure	M5	II	●
I_6	UE accepts `nas_sm_cmd_replay` when sent immediately after `nas_sm_cmd_int`	M6	F	●
I_7	UE accepts `nas_sm_cmd_replay` even after registration complete	M6	LL	●
I_8	UE accepts the `reg_accept_header4_plain`, bypassing the authentication and security activation	S	PS, IL	●
I_9	Plain header NAS messages (i.e., `reg_accept_plain`, `conf_update_cmd_plain`) bypass the integrity check	S	IS, DoS	●
I_{10}	UE accepts `nas_sm_cmd_NIA0_plain`, bypassing NAS layer security activation	I	IL	●
I_{11}	UE accepts the `rrc_sm_cmd_NIA0_plain`, bypassing RRC layer security activation	I	IL, DG	○
I_{12}	UE responds to `nas_sm_cmd_replay` differently, enabling device fingerprinting	U1	F	●
I_{13}	UE does not release the radio connection immediately after accepting `dereg_req_protected`	U1	-	●
I_{14}	Upon receiving the second `rrc_sm_cmd_int`, UE replies with `rrc_sm_failure`	U1	-	●
I_{15}	UE does not accept `dereg_req_protected` (with “switch off” bit set to 1) after registration	U1	-	●
I_{16}	Before AS security activation, UE not accepting RRC messages with non-zero MAC	U1	-	●
I_{17}	UE does not accept `reg_accept_protected` inside *DL NAS Transport* message	U1	-	●
I_{18}	Out-of-sequence `ue_info_req_protected` interrupts ongoing registration procedure	U1	-	●
I_{19}	UE accept `nas_sm_cmd_header2_protected`	U1	-	●
I_{20}	UE replies with `conf_update_comp` when acknowledgment indicator not present	U2	II	●
I_{21}	UE responds to `nas_sm_cmd_NIA0_plain` differently, enabling device fingerprinting	U3	F	●
I_{22}	`rrc_sm_cmd_NIA0_plain` after `rrc_sm_cmd_int` makes all subsequent symbols unresponsive	D	DoS	●

Table 2: Property violations identified by `5GBaseChecker`. IL: information leak, LL: location leak, IS: identity spoofing, PS: phishing SMS, DoS: denial-of-service, DG: downgrade, F: fingerprinting device, II: interoperability issue. ●: not found in previous works on 5G. ○: found in previous works on 5G.

found some UEs accept plaintext *Registration Accept* with security header 4 before NAS security activation, which enables an attacker to bypass the whole NAS layer Authentication and Key Agreement (AKA) procedure. The attacker can exploit this vulnerability to provide a victim UE to access the Internet through a fake base station without data ciphering and integrity. She can also send fake SMS to the victim from the fake base stations. We elaborate on these in §10.1. On the other hand, in **I_9**, we found that one UE accepts plaintext *Registration Accept* and *Configuration Update Command* with security header type 0. This can lead to AKA bypass in the NAS layer and DoS for the victim.

9.2.3 Mishandling Information Elements (IEs) ($I_{10} - I_{11}$)

According to TS 24.501 [10], *NAS Security Mode Command* messages in 5G are not ciphered but are integrity-protected, ensured by a message authentication code (MAC) [10]. In **I_{10}**, an attacker can craft a *NAS Security Mode Command* with NIA0 and NEA0. Acceptance of this message enables the attacker to establish NAS layer security context with null cipher and null integrity and to bypass NAS layer security. In **I_{11}**, we found that some UEs, despite being prohibited by the specification [11], accept *AS Security Mode Command* messages with *NIA0* IE, leading to AS security bypass. This allows the attacker to inject any RRC layer messages without integrity protection. Exploiting this, the attacker may send an *RRC Reconfiguration* or *RRC Release* message to get sensitive UE information or downgrade the victim UE to 4G/2G.

9.2.4 Underspecified Specifications ($I_{12} - I_{21}$)

- **Incomplete information for mandatory features (I_{12}-I_{19}).** For **I_{12}**, clause 4.4.3.2 in TS 24.501 [10] states that no specific mechanism is required for implementations to ensure replay protection. However, TS 24.501 [10] also mentions that when a *Security Mode Command* message cannot be accepted, the UE has to respond with a *Security Mode Reject* message. Based on the specification, it is not clear whether the UE should drop the replayed *NAS Security Mode Command* message or reply with *NAS Security Mode Reject*. Due to such ambiguous and incomplete instructions, some devices become vulnerable to fingerprinting attacks. In addition, devices responding with *NAS Security Mode Reject* are vulnerable to uplink NAS counter desynchronization attacks leading to denial of services [34]. For some other issues (**I_{13}-I_{19}**), the implementations deviate from each other, but the security impacts are not evident. For example, in **I_{13}**, after receiving the *Deregistration Request* message, the device's RRC layer connection is not dropped instantly, and it can still respond to some RRC layer messages, e.g., *AS Security Mode Command*. In **I_{14}**, some UEs respond with *Security Mode Failure* upon receiving a *second AS Security Mode Command*, while others do not. As for **I_{15}**, we also discovered that some UEs do not accept *Deregistration Request* message with “switch off” bit set to 1, while others do not check the value of this IE.
- **Incomplete information for optional features (I_{20}).** We found a UE implementation unexpectedly sends a response to a *Configuration Update Command* message even though the

# Iteration	**Cumulative # queries w/o hybrid learning**	**Cumulative # queries w/ hybrid learning**
1	29	29
2	208	116
3	786	612
4	1111	750
5	3016	959
6	3244	1547
7	3837	6630
8	8920	14864
9	17155	15529
10	17820	16280
11	18593	17084
12	19397	27084 (terminate)
13	29397 (terminate)	-

Table 3: Queries required w/ and w/o hybrid learning.

"acknowledgment indicator" field is not present ($\boldsymbol{I_{20}}$). Since the core network does not expect any response to this message, it may behave unexpectedly upon receiving such a response from the UE and cause interoperability issues.

- **Conflicting information in the specifications ($\boldsymbol{I_{21}}$).** In $\boldsymbol{I_{21}}$, we observed that before authentication, some devices respond to *NAS Security Mode Command* with *NIA0*. Devices equipped with Exynos baseband follow TS 24.501 [10], clause 5.4.2.5, and respond with a *Security Mode Reject* message. The rest of the devices follow TS 24.501 [10], clause 7.5.1, and respond with *5GMM Status*. This deviating behavior enables an attacker to fingerprint the Exynos devices.

9.2.5 Flawed Specifications (I_{22})

According to TS 38.331 [11], clauses 5.3.7.1 and 5.3.7.2, after AS security activation, integrity check failure on *RRC messages* leads to an RRC connection release. However, this enables an adversary to force the victim UE to drop the connection upon receiving messages with wrong MAC values ($\boldsymbol{I_{22}}$). The attacker can repeat this attack for prolonged DoS.

9.3 Efficiency of StateSynth

To answer **RQ2**, we measure the number of queries, i.e., input message sequences, generated by StateSynth, and the total time required to extract each FSM. We summarize the results for each device in Table 4, along with the number of states and transitions. We also evaluate the effectiveness of different design decisions in StateSynth in this section.

Evaluating hybrid learning. To evaluate *hybrid learning*, we compute the cumulative number of queries in each iteration (hypothesis construction and validation) of *Active Learner* for one of the implementations– *Motorola Edge+*. We show the results in Table 3, where it is evident that hybrid learning reduces the number of queries by ~2,300. As the passive FSM construction takes only ~2 minutes, this improvement essentially translates to a ~1,450 minutes decrease in total time for FSM extraction. Thus, this approach significantly improves StateSynth's efficiency.

Device	**# MQ**	**# EQ**	**Time (min)**	**# (States)**	**# (Trans)**
Motorola Edge+	1860	25239	17615	14	81
Redmagic 8 Pro	1856	1030	1968	14	85
Oneplus10 pro	1856	1030	1924	14	85
iPhone 14	1860	1030	2023	14	81
BlackShark	1856	1030	1924	14	86
Quectel	1856	1030	1535	14	86
RedMi K40S	1856	1030	1924	14	86
Samsuang S20+	1856	1030	1924	14	86
Nothing Phone (1)	1856	1030	1828	14	86
OnePlus Nord 5G	1856	1030	1972	14	86
Huawei P40 Pro	1351	2030	2202	11	54
Hisense F50+	1662	3269	3205	13	75
Oppo Reno 7 Pro 5G	2117	2497	3230	15	88
Rog Phone 6D	2117	1030	2203	15	88
OAIUE	687	3602	2145	6	16
Pixel 7	1207	2024	2262	9	65
Pixel 6	1207	1024	1562	9	65
Samsuang S21	1193	1404	1818	9	48
srsUE	710	3732	2221	6	20

Table 4: Queries, time, states, and transitions. MQ: membership queries, EQ: equivalence queries.

Evaluating collaborative learning. To evaluate *collaborative learning*, we extract the FSMs sequentially, as shown in Table 4, and present the required number of queries and time. For the first device, when potential counterexamples from other devices are not available, StateSynth requires ~27,100 queries to extract the FSM. For subsequent devices, this number is reduced by 1.6 – 12.3 times compared to approaches without collaborative learning [35], cutting the time requirements accordingly. This technique thus significantly boosts StateSynth's efficiency.

Comparison with existing FSM constructors. We compare StateSynth with an existing framework for FSM extraction from UE implementations, namely, DIKEUE [35]. Although this framework is for 4G implementations, we use its FSM inference module with our 5G OTA message infrastructure and the input alphabet of StateSynth. We refer to this modified version as DIKEUE*. Moreover, DIKEUE has a different termination condition for its FSM extraction. Thus, for a proper comparison, we use the same termination condition from DIKEUE in both StateSynth and DIKEUE* to extract the same level of information from the implementations.

Figure 5 shows the cumulative number of queries required for FSM extraction. For the first device, StateSynth clearly performs better than DIKEUE* (14,864 vs. 17,155 queries), which shows the impact of *hybrid learning*. For the subsequent devices, the improvement in efficiency is even more significant, as seen in the growth in the number of queries in the figure. Overall, DIKEUE* requires 73,657 queries to extract all the FSMs, whereas StateSynth requires only 26,715. This demonstrates that StateSynth brings significant improvement in FSM extraction of UE implementations.

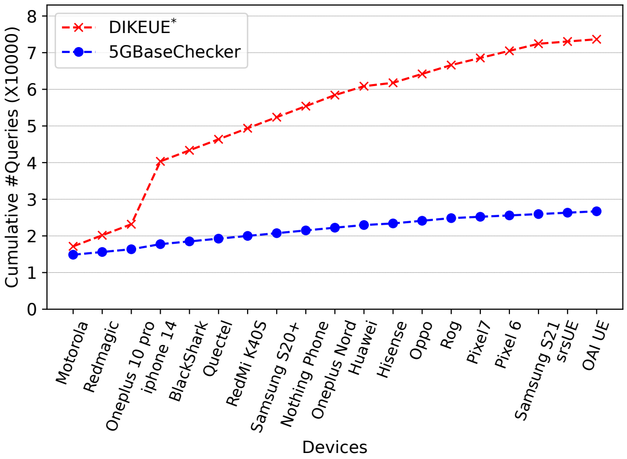


Figure 5: Comparison of FSM extraction.

9.4 Efficacy of DevScan

To address **RQ3**, we compare DevScan with the FSM equivalence checker components of BLEDiff [39], and DIKEUE [35]. We compute the number of deviations identified and the required time by StateSynth, BLEDiff, and DIKEUE and present them in Table 5. The results show that DevScan identifies ∼700 more deviations than the other approaches. For each pairwise FSM comparison, compared to DIKEUE, DevScan, however, requires ∼ 15 seconds more to find deviations. This is reasonable as we find more unique deviations by comprehensively traversing the FSMs.

Approach	# Deviations	Avg. time (sec.)
DIKEUE [35]	1325	83
BLEDiff [39]	724	118
5GBaseChecker	2044	98

Table 5: Number of deviations and time requirement.

9.5 Effectiveness and Efficiency of DevLyzer

To answer **RQ4**, we summarize how DevLyzer reduces the manual effort by the domain expert to analyze deviations and helps extract properties.

Reduction in manual effort. A new property in DevLyzer can resolve multiple deviations. As such, DevLyzer resolves multiple deviations within one iteration and thus requires smaller than $|\mathbb{D}|$ iterations. In our experiments, we ran DevLyzer on 2044 deviations found by DevScan and resolved them in 36 iterations by manually analyzing 36 deviations (one deviation $\psi^i \in \mathbb{D}$ per iteration) and identifying 45 properties in total, as the analysis of 9 deviations rendered more than one property. It took, on average, ∼2 minutes to analyze each deviation and extract the required properties. If we needed to analyze all of them manually, it would take ∼67 man-hours. Thus, we saved ∼65 man-hours of work using DevLyzer.

Approach	Dynamic	Guided property extraction	Partial positive tests	Interoperability detection	Stateful testing
BASECOMP [42]	✗	✗	✗	✗	✓
DoLTEst [50]	✗	✓	✗	✗	✓
DIKEUE [35]	✓	✗	✓	✓	✓
UE security reloaded [15]	✗	✗	✗	✗	✗
SecChecker [58]	✗	✗	✓	✗	✗
5GBaseChecker	✓	✓	✓	✓	✓

Table 6: Comparison with existing testing approaches.

Extracted properties. With DevLyzer, we extracted a total of 45 properties, as listed in [2]. However, as we release these properties as one of our contributions, subsequent analysis of 5G devices using 5GBaseChecker can start with our extracted property set $\mathbb{P}$. In that case, most of the deviations would be automatically resolved even before any manual analysis, further lessening the manual effort required.

9.6 Comparison with Current Testing Methods

We address **RQ5** by comparing 5GBaseChecker with existing works in several directions. Table 6 shows the summary of these comparisons. Among the approaches compared with, several only statically generate, i.e., pre-compute test cases or perform static analysis on the baseband software [15, 42, 50, 58], whereas 5GBaseChecker analyzes implementations dynamically during testing. Some existing approaches, such as DoLTEst [50], use keywords, e.g., 'shall not', 'security activation', and 'integrity protected' to search security policies from specifications. This may, however, yield a large amount of policies to test that are either not security critical or duplicate. In contrast, 5GBaseChecker dynamically extracts policies based on deviating traces. Moreover, 5GBaseChecker performs some regular/positive tests (indicated by *partial positive testing* in Table 6) during FSM construction along with the negative ones. In contrast, a few approaches test the implementations with only negative test cases [15, 42, 50], avoiding positive testing altogether. Notably, some of these works also do not perform stateful analysis [15, 58], although 5G implementations are highly stateful. Further, 5GBaseChecker can discover interoperability issues as DIKEUE [35], whereas others cannot.

Effectiveness. Other than the above-mentioned comparisons, we also compare if these works are capable of identifying the exploitable and interoperability issues identified by 5GBaseChecker. We show the results in Table 7. Among these works, BASECOMP [42] performs tests only on integrity protection. Several works have a fixed set of properties or test cases [15, 50, 58], which do not include many of the issues found by 5GBaseChecker. Finally, DIKEUE [35] also misses several of these issues because of a limited alphabet.

Coverage on open-source UE implementation. Although 5GBaseChecker primarily focuses on finding security pol-

Approach	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{20}	I_{21}	I_{22}
BASECOMP [42]			✓	✓				✓	✓						
DoLTEst [50]	✓	✓	✓	✓				✓	✓	✓	✓			✓	✓
DIKEUE [35]			✓	✓		✓	✓		✓	✓	✓	✓		✓	✓
UE security reloaded [15]		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓			
SecChecker [58]		✓							✓	✓	✓				✓
5GBaseChecker	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 7: Comparison of the effectiveness of 5GBaseChecker on finding security, privacy, and interoperability issues with existing UE implementation testing approaches.

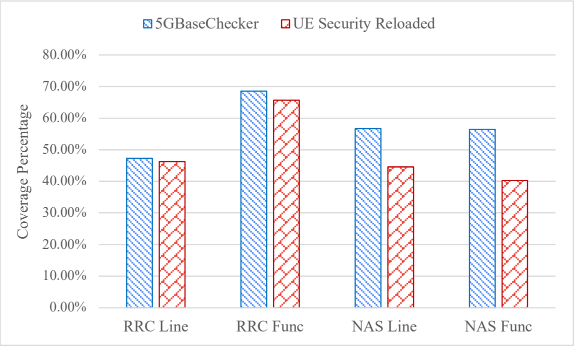


Figure 6: NAS and RRC implementation coverage.

icy violations in black-box UE implementations, we also compute the coverage information [4] for an open-source UE implementation, srsUE [8], using LCOV. We compare 5GBaseChecker with an existing open-source 5G UE testing framework, namely, UE security reloaded [15], for this evaluation. Figure 6 shows that 5GBaseChecker improves coverage in the NAS layer by ∼12% and in the RRC layer by ∼1%. This demonstrates that, in both layers, 5GBaseChecker improves the coverage, which essentially translates to a more comprehensive testing approach.

10 Attack Case Study

10.1 5G AKA Bypass (I_8)

Exynos basebands accept plaintext *Registration Accept* with header 4 (integrity protected and ciphered with new 5G NAS security context) from a fake base station after sending a *Registration Request* without establishing NAS or RRC layer security. The adversary can exploit this behavior to grant unprotected data sessions and send phishing SMS's to victims.
Attack. Upon receiving the plaintext *Registration Accept* with security header 4, the UE replies with a plaintext *Registration Complete*, and a *PDU Session Establishment Request* message. The attacker then crafts a plaintext Downlink NAS Transport message (with security header 4) containing a plaintext *PDU Session Establishment Accept* and encapsulates that in an *RRC Reconfiguration* message to send it to the victim UE. This allows the attacker to bypass the 5G NAS AKA procedure completely. After that, the victim UE gains unprotected access to the Internet through the fake base station. The attack steps are shown in Figure 7 (a), which we also confirm end-to-end. Further, Figure 8 shows the trace, where we find that the victim UE sends DNS queries and starts accessing the Internet over the unprotected 5G connection.

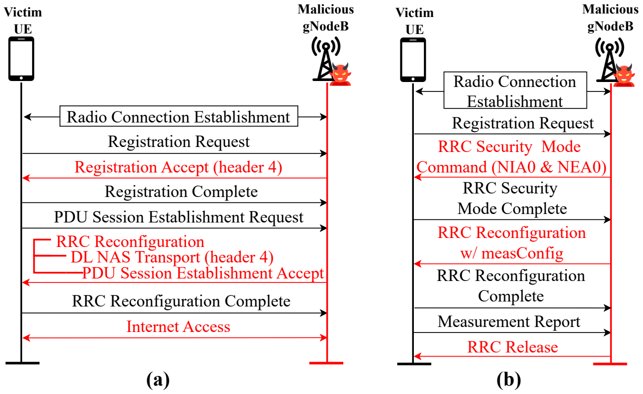

Victim UE
Malicious gNodeB
Radio Connection Establishment
Registration Request
Registration Accept (header 4)
Registration Complete
PDU Session Establishment Request
RRC Reconfiguration
DL NAS Transport (header 4)
PDU Session Establishment Accept
RRC Reconfiguration Complete
Internet Access
(a)
Victim UE
Malicious gNodeB
Radio Connection Establishment
Registration Request
RRC Security Mode Command (NIA0 & NEA0)
RRC Security Mode Complete
RRC Reconfiguration w/ measConfig
RRC Reconfiguration Complete
Measurement Report
RRC Release
(b)

Figure 7: (a) 5G AKA bypass, (b) AS security bypass.

Exploiting the same vulnerability, we have also experimentally confirmed that the attacker can send a plaintext *DL NAS Transport* message with a phishing SMS to the victim [42].
Impact. As the adversary provides unprotected 5G data sessions, the attack leads to severe information leaks. For example, by intercepting the DNS queries, she can lead the victim to phishing websites to acquire sensitive credentials. Further, with fake SMSs, the adversary can cause panic and lure the victim into sharing sensitive information or sending money.

10.2 AS Security Bypass (I_{11})

Several basebands (shown in Table 8) accept *RRC Security Mode Command* with the null integrity protection algorithm (NIA0) and the null encryption algorithm (NEA0). The adversary can exploit this vulnerability to bypass the RRC layer security and inject any plaintext RRC messages without a valid MAC.
Attack. When the victim UE initiates the connection to the malicious gNodeB, the adversary crafts a *RRC Security Mode Command* message with null integrity protection and the null encryption algorithms (NIA0 & NEA0) and sends it to the victim UE (shown in Figure 7(b)). After that, the victim UE uses null integrity protection and null encryption in subsequent communication. The attacker can send plaintext *RRC Reconfiguration* with *measConfig* IE to retrieve measurement information from the victim UE. In addition, the attacker can also send plaintext *RRC Release* message with *redirectedCarrierInfo* IE to downgrade the victim UE to LTE or even 2G network, as shown in a prior work [58]. Furthermore, AS security bypass can lead to eavesdropping [50], as shown by previous work on LTE.
Impact. Bypassing the AS security and retrieving the measurement report from the victim UE, the adversary can infer

the location of the victim based on the signal strength information in the measurement report [54]. Moreover, the victim can be downgraded to prior-generation networks [58], and the communication can be eavesdropped by the adversary [50].

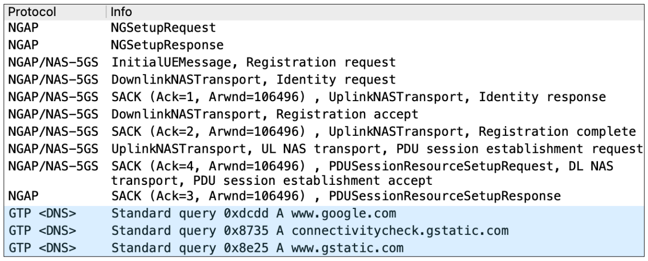
11 Related Work

Finite state machine (FSM) extraction. Several works perform automata learning to extract FSMs from protocol implementations of various protocols such as 4G LTE [35], TLS [23], DTLS [27], IoT [56], SSH [29], BLE [39], etc. In the area of cellular networks, Chlosta et al. [19] perform automata learning to model open-source core network projects but fail to extend to real-world implementations. ProCheker [40] extracts formal models through program analysis on open-source implementations and performs security analysis. DIKEUE [35], on the other hand, performs active automata learning on black box devices and leverages deviations among testing devices to find attacks through manual analysis.

Security analysis of cellular protocols. Many works [14, 22, 32, 34, 46, 52] manually create formal models from cellular specifications and perform model checking against manually crafted security properties. On the other hand, 5GBaseChecker automatically extracts the models from black-box implementations. Kim et al. [44] identify three basic properties from 4G specifications and generate test cases based on them. DoLTEst [50] provides a stateful and comprehensive set of test cases to detect vulnerabilities for black-box 4G UE devices focusing on negative test cases. However, it does not provide positive test cases. UE Security Reloaded [15] and SecChecker [58] create 5G SA security testing frameworks with static lists of test cases. Contester [17] and Atomic [18], on the other hand, create test cases from the specifications using natural language processing techniques. However, these works do not create test cases dynamically depending on the implementations. BASECOMP [42] analyzes the integrity protection of baseband software through probabilistic inference and comparative analysis and checks for inconsistencies by consulting the specifications after locating them. Some approaches [31, 43, 45] require significant manual effort to analyze baseband firmware. Other approaches for baseband firmware include the usage of SDRs [3, 9], sending crafted messages to analyze vulnerability, and generating unusual messages [26]. However, these works do not work with black-box UE implementations.

12 Discussion and Limitations

Manual effort. 5GBaseChecker requires domain expertise to define alphabet selection criteria and manual inspection of message formats to create an alphabet for testing. Additionally, the expert also needs to implement a 5G-specific adapter for converting symbols to concrete packets and vice versa. In addition, despite adopting the majority voting scheme of prior works [35, 47], observational inconsistencies may still occur in the over-the-air queries/responses. Once an inconsistency is automatically detected, the user needs to manually analyze the inconsistent responses and rectify them before resuming the FSM construction. An expert also needs to manually identify security properties for unresolved deviations and represent corresponding properties in LTL for automated model checking with DevLyzer. However, compared to previous approaches [35, 39], DevLyzer significantly reduces such efforts by providing a semi-automated framework to resolve many deviations with one iteration of manual analysis.



Protocol	Info
NGAP	NGSetupRequest
NGAP	NGSetupResponse
NGAP/NAS-5GS	InitialUEMessage, Registration request
NGAP/NAS-5GS	DownlinkNASTransport, Identity request
NGAP/NAS-5GS	SACK (Ack=1, Arwnd=106496) , UplinkNASTransport, Identity response
NGAP/NAS-5GS	DownlinkNASTransport, Registration accept
NGAP/NAS-5GS	SACK (Ack=2, Arwnd=106496) , UplinkNASTransport, Registration complete
NGAP/NAS-5GS	UplinkNASTransport, UL NAS transport, PDU session establishment request
NGAP/NAS-5GS	SACK (Ack=4, Arwnd=106496) , PDUSessionResourceSetupRequest, DL NAS transport, PDU session establishment accept
NGAP	SACK (Ack=3, Arwnd=106496) , PDUSessionResourceSetupResponse
GTP <DNS>	Standard query 0xdcdd A www.google.com
GTP <DNS>	Standard query 0x8735 A connectivitycheck.gstatic.com
GTP <DNS>	Standard query 0x8e25 A www.gstatic.com

Figure 8: Internet access through AKA bypass

Possible missing deviations. 5GBaseChecker relies on deviations in UEs' behavior to find security and privacy properties and corresponding flaws. If all UEs have the same insecure/incorrect behavior, 5GBaseChecker will not identify the corresponding properties and flaws. However, practically, this is highly unlikely for a diverse set of implementations.

Incompleteness of properties. As 5GBaseChecker's property extraction is guided by implementation deviations, the tested properties are not exhaustive.

Termination vs. FSM completeness. 5GBaseChecker empirically chooses termination parameters N_0, N_1, and N_2 in FMS construction to balance between early termination and completeness of FSM learning. Although larger values of the parameters may increase the coverage of FSM construction and thus aid in finding more deviations, they will also increase the number of OTA queries and time required.

13 Conclusion and Future Work

We develop a scalable and efficient security analysis framework 5GBaseChecker for testing 5G basebands. Our proposed hybrid and collaborative FSM learning technique significantly outperforms existing black-box automata learning-based approaches. Unlike prior works, 5GBaseChecker's graph traversal-based model refinement for pairwise FSM comparison enables it to identify many unique deviations in 5G basebands. Finally, 5GBaseChecker utilizes deviations to find relevant security properties and design a semi-automated deviation analyzer that reduces manual effort in triaging and characterizing deviating traces. In the future, we will extend 5GBaseChecker to analyze 5G session management procedures.

Acknowledgements

We thank the anonymous reviewers and the shepherd for their feedback and suggestions. We also thank the baseband vendors for cooperating with us during the responsible disclosure. This work has been supported by the NSF under grants 2145631, 2215017, and 2226447, the Defense Advanced Research Projects Agency (DARPA) under contract number D22AP00148, and the NSF and Office of the Under Secretary of Defense– Research and Engineering, ITE 2326898, as part of the NSF Convergence Accelerator Track G: Securely Operating Through 5G Infrastructure Program.

References

[1] *3GPP Standard*. www.3gpp.org.

[2] *5GBaseChecker*. https://github.com/SyNSec-den/5GBaseChecker.

[3] *bladeRF*. https://www.nuand.com/bladerf-2-0micro/.

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

[5] *Network Signal Guru*. https://www.qtrun.com/en/?page_id=34.

[6] *Open5GS*. https://open5gs.org/.

[7] *OpenAirInterface*. https://gitlab.eurecom.fr/oai/openairinterface5g.

[8] *srsRAN*. https://github.com/srsran/srsRAN_4G.

[9] *USRP B210*. https://www.ettus.com/all-products/ub210-kit/.

[10] 3GPP. Non-access-stratum (nas) protocol for 5g system (5gs); stage 3. Technical Specification (TS) 24.501, 3rd Generation Partnership Project (3GPP), 2023. Version 17.8.0.

[11] 3GPP. Nr; radio resource control (rrc); protocol specification. Technical Specification (TS) 38.331, 3rd Generation Partnership Project (3GPP), 2023. Version 17.6.0.

[12] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[13] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. SFADiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16. Association for Computing Machinery, 2016.

[14] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 1383–1396, 2018.

[15] Evangelos Bitsikas, Syed Khandker, Ahmad Salous, Aanjhan Ranganathan, Roger Piqueras Jover, and Christina Pöpper. Ue security reloaded: Developing a 5g standalone user-side security testing framework. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 121–132, 2023.

[16] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Computer Aided Verification: 26th International Conference, CAV 2014*.

[17] Yi Chen, Di Tang, Yepeng Yao, Mingming Zha, XiaoFeng Wang, Xiaozhong Liu, Haixu Tang, and Baoxu Liu. Sherlock on specs: Building LTE conformance tests through automated reasoning. In *32nd USENIX Security Symposium (USENIX Security 23)*, August.

[18] Yi Chen, Yepeng Yao, XiaoFeng Wang, Dandan Xu, Chang Yue, Xiaozhong Liu, Kai Chen, Haixu Tang, and Baoxu Liu. Bookworm game: Automatic discovery of LTE vulnerabilities through documentation analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[19] Merlin Chlosta, David Rupprecht, and Thorsten Holz. On the challenges of automata reconstruction in LTE networks. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 164–174, 2021.

[20] Merlin Chlosta, David Rupprecht, Christina Pöpper, and Thorsten Holz. 5G SUCI-Catchers: Still catching them all? In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 359–364, 2021.

[21] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978.

[22] Cas Cremers and Martin Dehnel-Wild. Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[23] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium*, 2015.

[24] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[25] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[26] Kaiming Fang and Guanhua Yan. Emulation-instrumented fuzz testing of 4G/LTE android mobile devices guided by reinforcement learning. In *23rd European Symposium on Research in Computer Security*.

[27] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of dtls implementations using protocol state fuzzing. In *29th USENIX Security Symposium, Online, August 12–14, 2020*, pages 2523–2540, 2020.

[28] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Automata-based automated detection of state machine bugs in protocol implementations. In *NDSS*, 2023.

[29] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of ssh implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017.

[30] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 2:971–978, 1994.

[31] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin Butler. Firmwire: Transparent dynamic analysis for cellular baseband firmware. In *Network and Distributed Systems Security Symposium (NDSS) 2022*, 2022.

[32] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. LTEInspector: A systematic approach for adversarial testing of 4G LTE. In *Network and Distributed Systems Security (NDSS) Symposium 2018*.

[33] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. Privacy attacks to the 4G and 5G cellular paging protocols using side channel information. *Network and distributed systems security (NDSS) symposium2019*, 2019.

[34] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19.

[35] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4G LTE cellular devices. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1082–1099, 2021.

[36] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib: a framework for active automata learning. In *Computer Aided Verification: 27th International Conference, CAV 2015*.

[37] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: a redundancy-free approach to active automata learning. In *Runtime Verification: 5th International Conference, RV 2014*.

[38] Natasha Yogananda Jeppu, Thomas Melham, Daniel Kroening, and John O'Leary. Learning concise models from long execution traces. In *57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020.

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

[40] Imtiaz Karim, Syed Rafiul Hussain, and Elisa Bertino. ProChecker: An automated security and privacy analysis framework for 4G LTE protocol implementations. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 773–785. IEEE, 2021.

[41] Fujiwara Bochmann Khendek, S Fujiwara, GV Bochmann, F Khendek, M Amalou, and A Ghedamsi. Test selection based on finite state models. *IEEE Transactions on software engineering*, (591-603):10–1109, 1991.

[42] Eunsoo Kim, Min Woo Baek, CheolJun Park, Dongkwan Kim, Yongdae Kim, and Insu Yun. BASECOMP: A comparative analysis for integrity protection in cellular baseband software. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3547–3563, 2023.

[43] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. Basespec: Comparative analysis of baseband software and cellular specifications for l3 protocols. In *NDSS*, 2021.

[44] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the untouchables: Dynamic security analysis of the LTE control plane. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[45] Dominik Maier, Lukas Seidel, and Shinjo Park. Basesafe: Baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM conference on security and privacy in wireless and mobile networks*.

[46] Rhys Miller, Ioana Boureanu, Stephan Wesemeyer, and Christopher JP Newton. The 5G key-establishment stack: In-depth formal verification and experimentation. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022.

[47] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated model inference for stateful network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 699–718. USENIX Association, 2019.

[48] José Oncina and Pedro Garcia. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*, pages 99–108. World Scientific, 1992.

[49] José Oncina and Pedro Garcia. Inferring regular languages in polynomial updated time. In *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*. World Scientific, 1992.

[50] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. DoLTEst: In-depth downlink negative testing framework for LTE devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1325–1342, 2022.

[51] Shinjo Park, Altaf Shaik, Ravishankar Borgaonkar, and Jean-Pierre Seifert. White rabbit in mobile: Effect of unsecured clock source in smartphones. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 13–21, 2016.

[52] Aleksi Peltonen, Ralf Sasse, and David Basin. A comprehensive formal analysis of 5G handover. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2021.

[53] David Rupprecht, Kai Jansen, and Christina Pöpper. Putting LTE security functions to the test: A framework to evaluate implementation correctness. In *10th USENIX Workshop on Offensive*, 2016.

[54] Altaf Shaik, Ravishankar Borgaonkar, N Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical attacks against privacy and availability in 4G/LTE mobile communication systems. *arXiv:1510.07563*.

[55] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 521–538, 2017.

[56] Martin Tappler, Bernhard K Aichernig, and Roderick Bloem. Model-based testing iot communication via active automata learning. In *2017 IEEE International conference on software testing, verification and validation (ICST)*, pages 276–287. IEEE, 2017.

[57] Qinying Wang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Yuhong Kan, Zhaowei Lin, Changting Lin, Shuiguang Deng, Alex X Liu, et al. MPInspector: A systematic and automatic approach for evaluating the security of IoT messaging protocols. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4205–4222, 2021.

[58] Chuan Yu, Shuhui Chen, Ziling Wei, and Fei Wang. SecChecker: Inspecting the security implementation of 5g commercial off-the-shelf (cots) mobile devices. *Computers & Security*, page 103361, 2023.

A Algorithm of DevLyzer

Algorithm 1 DevLyzer Algorithm

```
Require: Deviation Set ($\mathbb{D}$)
Output: Violation Instance Set $\mathcal{V}_i$ for each device $b_i \in \mathcal{B}$
$\mathbb{P} \leftarrow \emptyset$
$\mathbb{D}_u \leftarrow \mathbb{D}$                              ▷ set of unresolved deviating traces
while $\mathbb{D}_u$ is not empty do
    $\psi_u^i \leftarrow PickRandom(\mathbb{D}_u)$            ▷ pick a random deviation unresolved
    $\mathbb{P}_{new} \leftarrow manualAnalysis(\psi_u^i)$ ▷ manually analyze deviation and find new properties
    $\mathbb{P}' \leftarrow \mathbb{P} \cup \mathbb{P}_{new}$
    for each $\psi^i$ in $\mathbb{D}$ do
        for each $\gamma_{out}^{i,j}$ in $\psi^i$ do
            $M_{dev}^{i,j} \leftarrow createFSM(S_{in}, \gamma_{out}^{i,j})$
            for each $\rho \in \mathbb{P}$ do
                $recordViolation(M_{dev}^{i,j}, \rho)$          ▷ model checking
            end for
        end for
    end for
    for each $\psi^i$ in $\mathbb{D}_u$ do
        $resolved \leftarrow checkResolve(\psi^i, \rho)$         ▷ Equation 1
        if resolved then
            $\mathbb{D}_u \leftarrow \mathbb{D}_u - \psi^i$
        end if
    end for
end while
```

B StateSynth Algorithm

The FSM extraction algorithm for StateSynth is provided in Algorithm 2. It takes as input a set of UEs and benign network traces. For each UE (b_i), StateSynth first builds an initial FSM M_i^p with *Passive Learner* and generates an initial hypothesis model M_i^0 with *Active Learner* (lines 4-5). The deviations between these two FSMs are saved as potential CEs (TCE) (lines 6-8). *Active Learner* then checks if these TCEs are valid for M_i and the implementation and refines

#	Device Name	Baseband Vendor	SoC Model	Baseband Model	Baseband Version	Found Issues
1	Huawei P40 Pro	HiSilicon	Kirin 990 5G	Balong 5000	21C93B373S000C000	I_{22}
2	Hisense F50+	Unisoc	Tiger T7510	UDX710	5G_MODEM_20C_W21.12.3_P5	$I_2, I_5, I_{11}, I_{13}, I_{22}$
3	Samsung Galaxy S21	Samsung	Exynos 2100	Exynos 5123	G991BXXU5CVF3	$I_2, I_3, I_4, I_5, I_7, I_8, I_{11}, I_{13}, I_{16}, I_{21}, I_{22}$
4	Google Pixel 6	Samsung	Google Tensor	Exynos 5123	g5123b-116954-230524-B-10194842	$I_2, I_3, I_4, I_5, I_7, I_8, I_{14}, I_{16}, I_{21}, I_{22}$
5	Google Pixel 7	Samsung	Google Tensor G2	Exynos 5300	g5300g-230323-230525-B-10200345	$I_2, I_3, I_4, I_5, I_7, I_8, I_{14}, I_{16}, I_{21}, I_{22}$
6	OPPO Reno7 Pro 5G	MediaTek	Dimensity 1200	MediaTek M70	M_V3_P10	$I_6, I_{14}, I_{15}, I_{19}, I_{22}$
7	ROG Phone 6D	MediaTek	Dimensity 9000+	MediaTek M80	M2.6.9.22-MT6983_V30	$I_1, I_6, I_{14}, I_{15}, I_{19}, I_{22}$
8	OnePlus Nord 5G	Qualcomm	Snapdragon 765G	Snapdragon X52	.c9-00023-SAIPAN_GEN_PACK-1.422164.2.446371.2	$I_5, I_{12}, I_{16}, I_{18}, I_{22}$
9	Nothing Phone (1)	Qualcomm	Snapdragon 778G+	Snapdragon X53	MPSS.HI.4.3.3-00781-LC_ALL_PACK-1.567500.4.570503.2	$I_5, I_{12}, I_{16}, I_{18}, I_{22}$
10	Samsung Galaxy S20+	Qualcomm	Snapdragon 865	Snapdragon X55	G9860ZCU3GVL1	$I_5, I_{12}, I_{16}, I_{18}, I_{22}$
11	RedMi K40S	Qualcomm	Snapdragon 870	Snapdragon X55	MPSS.HI.2.5.1.c1-03.59-0810_2007_fde5216cf0	$I_5, I_{12}, I_{16}, I_{18}, I_{22}$
12	Quectel RM500Q-GL	Qualcomm	N/A	Snapdragon X55	RM500QGLABR11A06M4G	$I_5, I_{12}, I_{16}, I_{18}, I_{22}$
13	Black Shark 4 Pro	Qualcomm	Snapdragon 888	Snapdragon X60	MPSS.HI.2.0.c7-00222-0306_0118_9cec56919e	$I_5, I_{12}, I_{16}, I_{18}, I_{22}$
14	OnePlus 10 Pro	Qualcomm	Snapdragon 8 Gen 1	Snapdragon X65	Q_V1_P14	$I_5, I_{12}, I_{16}, I_{22}$
15	Motorola Edge+ (2022)	Qualcomm	Snapdragon 8 Gen 1	Snapdragon X65	M8450_DE10_13.2183.01.98.18R HIPHI_PVT2_NA_CUST	I_{12}, I_{16}, I_{22}
16	iPhone 14	Qualcomm	Apple A15	Snapdragon X65	1.70.02	I_{12}, I_{16}, I_{22}
17	REDMAGIC 8 Pro	Qualcomm	Snapdragon 8 Gen 2	Snapdragon X70	NX729J_Z69_UN_ZML1T_v312	$I_5, I_{12}, I_{16}, I_{22}$
18	OAI UE	N/A	N/A	N/A	2023.w33	$I_3, I_4, I_6, I_{10}, I_{11}, I_{17}, I_{19}$
19	srsUE	N/A	N/A	N/A	release_22_10	$I_3, I_4, I_7, I_9, I_{10}, I_{11}, I_{19}, I_{20}$

Table 8: List of tested devices.

M_i accordingly (lines 11-16). StateSynth also saves any CE extracted by the Wp-method as TCEs (line 19), and consequently, previously learned CEs are also utilized in this phase (lines 12-13). After all TCEs are consumed, StateSynth further attempts to refine M_i using the Wp-method (lines 17-24) until the termination condition is reached. Finally, after the first pass of learning all implementation FSMs, another round of model refinement is performed to ensure subsequent TCEs are utilized for prior implementations as well (lines 26-32).

Algorithm 2 StateSynth Algorithm

```
Require: Network Traces ($NT$), Basebands Set ($\mathcal{B}$)
Output: FSM model $M_i$ for each device $B_i \in \mathcal{B}$
 1: $TCE \leftarrow \emptyset$                    ▷ TCE is a ordered set of total CEs
 2: $UCE \leftarrow \emptyset$                    ▷ UCE is a dictionary of used CEs
 3: for $B_i \in \mathcal{B}$ do
 4:     $M_i^p \leftarrow PassiveLearning(NT[B_i])$
 5:     $M_i^0 \leftarrow InitializeHypothesisModel$        ▷ Start Active Learning
 6:     $T \leftarrow FSMComparator(M_i^p, M_i^0)$
 7:     $TCE \leftarrow TCE \cup T$
 8:     $UCE[B_i] \leftarrow \emptyset$
 9:     $M_i \leftarrow M_i^0$
10:     while learning not terminate do
11:         for $CE \in TCE \setminus UCE[B_i]$ do
12:             if $CE$ is valid then
13:                 $M_i \leftarrow RefineModel(M_i, CE)$
14:                 $UCE[B_i] \leftarrow UCE[B_i] \cup \{CE\}$
15:             end if
16:         end for
17:         $CE \leftarrow WPmethod.Search()$
18:         while $CE \neq Null$ do
19:             $M_i \leftarrow RefineModel(M_i, CE)$
20:             $TCE \leftarrow TCE \cup \{CE\}$
21:             $UCE[B_i] \leftarrow UCE[B_i] \cup \{CE\}$
22:             $CE \leftarrow WPmethod.Search()$
23:         end while
24:     end while
25: end for
26: for $B_i \in \mathcal{B}$ do
27:     for $CE \in TCE \setminus UCE[B_i]$ do
28:         if $CE$ is valid then
29:             $M_i \leftarrow RefineModel(M_i, CE)$
30:         end if
31:     end for
32: end for
```

Message	Input Symbols	Output Symbols
Special Symbol		
Enable Registration	enable_reg	reg_req reg_req_guti
RRC Symbols		
Security Mode Command	rrc_sm_cmd_int rrc_sm_cmd_replay rrc_sm_cmd_NIA0_int rrc_sm_cmd_NIA0_plain	rrc_sm_comp rrc_sm_failure null_action
RRC Reconfiguration	rrc_reconf_protected rrc_reconf_plain rrc_reconf_replay	rrc_reconf_comp null_action
UE Capability Inquiry	ue_cap_enquiry_protected ue_cap_enquiry_plain	ue_cap_info null_action
UE information Request	ue_info_req_protected ue_info_req_plain	ue_info_resp null_action
Counter Check	counter_check_protected counter_check_plain	counter_check_resp null_action
NAS Symbols		
Identity Request	identity_req_plain	identity_resp null_action
Authentication Request	auth_req_plain	auth_resp null_action
Security Mode Command	nas_sm_cmd_int nas_sm_cmd_replay nas_sm_cmd_NIA0_int nas_sm_cmd_NIA0_plain nas_sm_cmd_header2_protected	nas_sm_comp nas_sm_reject null_action
Registration Accept	reg_accept_protected reg_accept_header4_plain reg_accept_plain	reg_comp null_action
Configuration Update Command	conf_update_cmd_plain conf_update_cmd_protected conf_update_cmd_replay	conf_update_comp null_action
Deregistration Request	dereg_req_protected dereg_req_plain	dereg_accept null_action

Table 9: Tested Symbols