A Systematic Way to LTE Testing

Muhammad Taqi Raza
Computer Science Department
University of California, Los Angeles
taqi@cs.ucla.edu

Songwu Lu
Computer Science Department
University of California, Los Angeles
slu@cs.ucla.edu

ABSTRACT

LTE test cases are standardized by 3GPP. They must be executed on every LTE-capable device model before commercial release. In this work, we examine the LTE testing practices in terms of completeness and efficiency. We discover that the standardized tests are incomplete in that a number of test cases related to multiple protocol interactions are missing. Our analysis also shows that, the isolated treatment of test cases, but not from the system perspective, incurs repetitive executions of test operations, thus resulting in testing inefficiencies. We thus make a case for a paradigm shift from ad hoc testing to a methodical approach to LTE testing. We follow a few guidelines from the LTE standards and propose an algorithmic approach to systematic testing. In the process, we address various challenges, provide complete list of test cases, and present the related algorithms. Our evaluation shows that, by eliminating repetitive operations, our new scheme reduces up to 70% of LTE testing steps. We also find 87 new, yet valid test cases that are not defined by the LTE standards.

ACM Reference format:
Muhammad Taqi Raza and Songwu Lu. 2019. A Systematic Way to LTE Testing. In *Proceedings of The 25th Annual International Conference on Mobile Computing and Networking, Los Cabos, Mexico, October 21–25, 2019 (MobiCom '19),* 15 pages.
https://doi.org/10.1145/3300061.3300134

1 INTRODUCTION

The 4G LTE conformance tests ensure that the device and the network comply to established procedures for their control- and data-plane functionalities. These test cases are standardized by 3GPP. They are to validate the device implementation by different vendors to conform to LTE standard specifications. Despite its importance, LTE testing remains largely unaddressed by the research community.

In this paper, we look into LTE testing in terms of efficiency and completeness. Our study yields two findings. First, the current conformance testing focuses on single-protocol test cases, but largely ignores interactions among protocols. This leads to incomplete LTE testing. Second, the execution of each test case is isolated from others, resulting in repetitive operations across multiple test cases and test inefficiency.

To address both issues, we take an algorithmic view on LTE testing. We seek to achieve three goals. One is to embrace multi-protocol interactions in the testing design to ensure completeness. The second is to treat all test cases in a coherent framework, and eliminate redundant test operations among these cases to improve efficiency. We further seek to propose new algorithms to facilitate LTE testing.

To ensure test completeness, we formulate the problem of finding test cases on multiple protocol interactions. The key idea is to leverage the message exchanges between the device and the network for the LTE protocols. Note that the device and the network exchange a few messages when executing a test case. The device consequently traverses states of one or more LTE protocol finite-state machines (FSMs). By examining the output messages of the device, we can infer whether the device has properly traversed the corresponding states of FSMs. This premise greatly reduces our effort to generate test cases. Each test case is thus represented as an output message combinations that the device can generate. To provide a complete list of test cases, we are required to generate *all* possible combinations of these output messages. For n output messages, there are 2^n possible test cases, which are practically infeasible to analyze. Consequently, we seek to find all those output message combinations that the device will *never* produce, which maps into a problem of finding all don't cares output values for output message combinations. We traverse device protocol FSMs in the reverse order (from output state towards input state) to find these don't care outputs. This is challenging especially when each protocol FSM has many states to traverse (given all those states related to configurations, timing and functionalities). This motivates us to reduce the number of states at device protocol FSMs.

We thus exploit results from the FSM reduction and minimization algorithms in finite automaton. We propose two novel algorithms that minimize deterministic finite automaton (DFA) states only using the LTE domain knowledge and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MobiCom '19, October 21–25, 2019, Los Cabos, Mexico
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6169-9/19/10…$15.00
https://doi.org/10.1145/3300061.3300134

skip non-deterministic finite automaton (NFA) states minimization (which is an NP complete problem). Once we obtain compact representations of device protocol FSMs, we can quickly find the don't cares from output message combinations and produce a complete set of test cases without enumerating all device output message combinations.

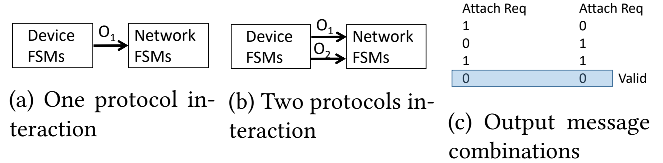
To improve LTE testing efficiency, we propose a new algorithm that eliminates repetitive operations for a test case. Our algorithm is based on the common graph data structure for all test cases. The graph nodes record the output parameters of a particular step in a test case, thus maintaining execution history in the graph nodes for all steps in the test case. Those test cases that are executed later do not repeat the same steps that have been completed by previous test cases. Instead, they reuse the stored output parameters while skipping the execution of those steps. However, storing output parameters for all steps of all test cases can be memory intensive and computationally infeasible. We address this issue by maintaining multiple graph data structures mapped to different test scenarios. A graph data structure is shared among test cases belonging to the same scenario.

To assess our proposed scheme, we have implemented 3GPP test cases together with our algorithms. We create the device FSMs and their representation as finite automaton. We also generate complete test cases by excluding don't care device outputs. Our evaluation yields 87 new test cases, where 60 test scenarios were not described in LTE testing standards. Moreover, our algorithm executes 43%, 11%, 70% and 50% fewer steps for *Attach*, *Detach*, *Tracking Area Update (TAU)*, and *Service Request* functionalities, when compared with the common practice for test case execution.

2 LTE TESTING

We now briefly review the importance of LTE testing, its limitations, and applications beyond LTE.

Importance of LTE testing LTE testing is a key factor to make the LTE technology a great success. Operators and network vendors not only require the manufactured device (called User Equipments (UEs)) to follow the standards, but also require each operating in the LTE network to be standard compliant. 3GPP defines how the phone and the network should behave in each operation scenario. These operational settings are formalized in the LTE protocol conformance testing specification [1], which thus specifies a number of test cases to validate LTE protocol operations. These test cases ensure that all LTE protocols (e.g., radio resource control, mobility management, session management, connectivity management, transport and tunneling protocols) work correctly in every operational situation. These test cases are validated through message exchanges between the device and a Network Simulator (NS). Both the device and the NS



(a) One protocol interaction (b) Two protocols interaction (c) Output message combinations

Figure 1: LTE test execution scenarios between device and network

implement their Finite State Machines (FSMs), and ensure the correctness of each test case via the expected state transitions. Figure 1a illustrates a test case execution scenario, in which the device FSM generates an output message (O_1) and the NS consumes the message to verify its correctness.

Current LTE Testing Practices and Limitations The current LTE Testing practices are rather ad hoc. They do not follow a systematic approach to testing, nor ensure completeness. Our study shows that, although the LTE standards discuss LTE testing in a given operational situation as well as abnormal device behavior, these test cases do not cover every possible test scenario. This is mainly because the current practice has focused on a single protocol execution. When cases for multiple protocol executions arise, not *all* interactions among these multiple protocols are tested. Figure 1b illustates the case when two output messages (O_1 and O_2) are fed to NS, which validates the correctness of their interactions. Take the LTE *Attach Request* procedure as an example. In this procedure, two EMM (EPS Mobility Management) protocols interact through the message of Attach Request. It generates 4 possible output message combinations (as shown in Figure 1c). (0 , 1) and (1 , 0) denote the absence of a message from one protocol, thus making it the test case of a single protocol interaction[1]. (1 , 1) represents the message from both protocols, indicating a case of two protocol interactions. (0 , 0) is a test case that corresponds to the absence of messages. The absence of Attach Request message triggers timeout that may affect the interacting protocols[2].

We further discover that, LTE testing treats each test case in the isolated manner based on conformance to LTE test standards. A test case does not share its execution information with a later-executed test case in the series of tests. This leads to inefficiencies with repetitive execution of test steps for a new test case to bring device into certain state (e.g., idle or connected state etc.). For example, almost all LTE *Attach* related test cases require the device to be switched *off* and then *on*, so that the device can initiate an *Attach* test case. However, such repetitive *power off/on* steps execution can be avoided if the next test case uses the knowledge from the previous test case that has successfully completed these steps.

[1]For example, device protocol generates one EMM message to NS.

[2]Such interactions can be between the same protocol (e.g., two EMM message exchanges) or between two different protocols (e.g., EMM and ESM (EPS Session Management) protocols).

We also looked into commercially available testing solutions provided by two leading LTE test equipment vendors [2], Anritsu [3] and Anite [4]. Their testing data sheets [5][6] and demos [7] show that, their test equipments do not make any changes to the implementation guide provided by 3GPP [8]. They treat each test case individually and only execute those test cases that are provided by the 3GPP. Their approach to testing is inefficient as they isolate test cases and do not transfer the previous test case execution knowledge to the followup one. We further conclude that, LTE chipset manufacturers, device vendors, and network operators perform test cases similarly to how test equipment vendors execute them.

The above observations show that, the testing community is focused on telecommunication standards and textbook testing design. We make a case for a paradigm shift to a system design approach from computer science when looking into the LTE testing problem. We aim to design and develop a testing system that treats individual test cases as the building blocks in a coherent testing system framework, along with algorithms to improve efficiency and ensure completeness.

Testing beyond 4G LTE Researchers who are developing systems for the next-generation wireless networks often face the challenge of verifying their designs. Testing plays a crucial role in making their design practical and deployable in reality. Through testing, researchers ensure their systems to adapt properly and quickly under scenarios that are both common and corner-case usage settings. Although we focus on 4G LTE testing in this paper, our methodology is generic and applicable to other technologies. Two examples are 5G New Radio (NR) testing and cellular Internet of Things (CIoT) testing. mmWave with 5G NR will introduce new test challenges where devices are using transceivers with integrated phased array antennas. These challenges include repeatability, configuration, and coverage, as well as testing accuracy, test time, and cost. These challenges can be addressed through our proposed testing methodology. Similarly, 5G cellular IoT solutions, including LTE-M and NB-IoT, require extensive testing before their deployment. Their test cases are related to network integration, coverage, battery consumption, and more. Our testing methodology can apply to these emerging technologies.

3 TEST COMPLETENESS

We next identify limitations of the current practice on providing a complete list of test cases, and present our approach.

3.1 Limitations and Challenges

We take different test cases as a sequence of message exchanges, instead of a particular test scenario. The LTE test case procedure involves an exchange of messages between the device and the network. The sequence of these messages would inform whether the test case has passed or failed. For example, in the device *Attach* test case, the device and the network exchange a number of messages related to Radio Resource Control (RRC), Security Mode, Authentication, and Packet Data Network (PDN or IP) connectivity. We view the LTE testing as the interactions between the device and the network. If *all* such interactions (*in any order*) are successful, we have completed *all* tests; otherwise not. To provide a complete list of cases, we seek to generate a list of tests that include *all* possible combinations of these messages. Given n device output messages, there are 2^n possible tests, which are practically infeasible to analyze. The issue is to obtain the complete list of test cases without enumerations.

3.2 Our approach

We seek to validate LTE protocol interactions to ensure that all combinations of protocol messages are assessed for correctness.

Testing as protocols interaction The purpose of device state transitions and their interactions is to generate a message for the network simulator. Hence, LTE testing looks into message exchanges between the device and the network. We can thus reduce testing efforts on device protocol FSMs by simply looking at the output message (O_1) that the device FSMs have produced, as shown in Figure 1a. If the message produced by the device is the one that the NS is expecting, then the internal FSM state transitions and interactions at the device were correct. If the NS has received an unexpected message from the device, we debug those states that have produced that output message, instead of traversing all states of the device FSM(s). Therefore, by checking device output values, the state transitions can be found, through which a test case can be defined.

Don't care outputs We provide a complete set of test cases that explore *all* possible output message combinations produced by the device FSMs, when two protocols interact between the device and NS. For n possible output messages produced by two protocols running at the device, one is required to test 2^n message combinations.

We reduce the output message combinations for two protocol interactions. We do not generate test cases for those combinations that were never produced by the device FSMs. We annotate these combinations as don't care outputs. To find such outputs, we traverse the device FSMs in the reverse order and check whether the corresponding output is valid or not. Finding don't care outputs, however, is practically infeasible when the device FSMs have too many states.

Compressing device FSMs We can quickly find the don't care outputs if we skip a few states in the device protocol FSM. We can even skip visiting a portion of FSM that might

have constraints on message combinations. This motivates us to compress device FSMs by merging states from FSMs.

Finite Automaton To compress FSMs, we model these FSMs as a finite automaton. Similar to FSMs in finite automaton, we have a start state, a final state, a finite set of states, a set of transitions, and a transition function. Some states are deterministic while others are not. If for each pair of states and possible transitions, there is a unique next state (as specified by the transition function), then the finite automaton is deterministic, i.e., Deterministic Finite Automaton (DFA); otherwise, the finite automaton is non-deterministic, i.e., Non-deterministic Finite Automaton (NFA).

Converting NFA to DFA Our goal is to reduce the number of states and to let FSMs run in polynomial time. We find that reducing the number of NFA states (by removing unnecessary states) is shown to be NP complete [9]. Moreover, running time for an NFA is $O(n^2 m)$ compared to $O(m)$ in case of DFA, where n is the number of states, and m is the number of *identical* transition conditions [10][11]. This is because NFA has n possible next states compared to DFA, which has only 1 path to next state for a given transition. This motivates us to convert NFA into DFA.

3.3 LTE Testing as Finite Automata

Overview of our solution We model the LTE testing as the problem of finding don't care output combinations in Finite Automata. All possible output combinations *minus* don't care outputs are the complete list of test cases. To find don't care outputs efficiently (i.e., the running time of FSM interactions remains linear), we first convert NFA into DFA. We propose a novel algorithm that reduces FSMs states by converting the selected NFA states to DFA. We further minimize those FSMs states through the DFA minimization procedure. We can do so because two or more DFA states can be equivalent, where these states migrate to the same next states upon the same transition condition. We merge these equivalent states and get compact representations for the LTE protocol FSMs. To merge as many equivalent states as possible, we introduce a new definition of states equivalence. Using the LTE domain knowledge, we argue that a number of FSM states have LTE timing and protocol constraints. These states are equivalent and merged because in reality they never occur together. In this regard, we propose a novel DFA minimization algorithm that converts non-equivalent states to equivalent states and merges these states.

3.3.1 Reducing FSM States

We first convert the NFA states to the DFA states to reduce the total number of states in FSMs.

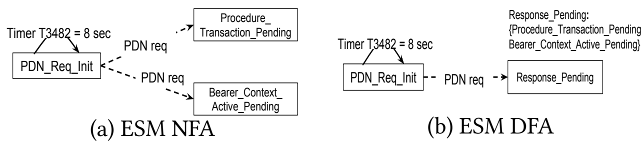
Inefficiencies in NFA to DFA conversion The Robins and Scott algorithm [12] is the best known scheme to convert NFA to DFA [13][14]. Such a conversion is made through power set construction. The DFA is obtained through a state set 2^n, the power set of n, containing all subsets of the original NFA state set n. The exponential number of DFA states are due to the degree of non-determinism of the current state. The current state can transition to a number of next states for n possible transitions. Through power set construction – which is practically inefficient – all possible states are recorded. It has been shown [15][16] that the number of states in DFA dramatically increases when more than one transition conditions are considered. It has been proved that the maximum number of states in DFA reaches $2^{\frac{n}{2}}$, $2^{\frac{2n}{3}}$, $2^{\frac{3n}{4}}$, and 2^n when the number of transitions are 2, 4, 8, and n, respectively.

Algorithm 1 Selected NFA states to DFA states conversion

```
1: input: = {nstates, trans_cond, trans_func, curr_state, next_state}
2: Call procedure Reverse-Edges()
3: procedure NFA-TO-DFA-PROCEDURE
4:     while nstates are not visited do
5:         if curr_state transitions to two or more next_state then
6:             if trans_func takes only one trans_cond then
7:                 for all next_states do
8:                     dfa_state ∪ next_state
9:                 divert edge arrows from next_states to dfa_state
10:                add edge from dfa_state to curr_state
11:                add trans_cond to the edges
12: Call procedure Reverse-Edges()
13: procedure REVERSE-EDGES
14:     if curr_state transitions next_state(s) on trans_cond then
15:         for all next_state(s) do
16:             swap (curr_state, next_state(s))
17:             reverse edges arrows
18:             keep trans_cond and trans_func
```

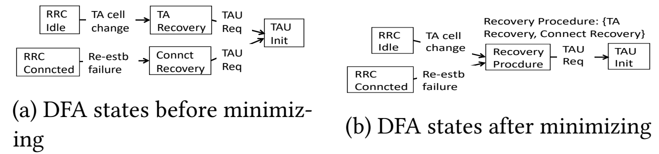
Converting selected NFA states to DFA We next propose NFA to DFA conversion (Algorithm 1) that converts only those selected NFA states to DFA and does not generate power set of NFA states. In our algorithm, we focus on two aspects: (1) visiting the states from the output (in reverse), and (2) converting those NFA states to DFA with *only one transition condition* (which is common in LTE, as we will show later).

For (1), we swap the current state with the next state (Step 16) and reverse the edges, while keeping transition condition and function unmodified (Steps 17-18). Thereafter, the FSM can be traversed in reverse and the initial FSM state can be reached that has generated final output value. For (2), our algorithm processes only those states that have exactly one transition condition to the next state. We first check whether the current state is indeed an NFA, i.e., it has more than one next states on a given transition condition (Step 5). Note that, when the current state has only one next state, the current state is already deterministic and the algorithm moves to the next state (Step 4). When the first *if* condition (Step 5) yields true, the algorithm checks whether the state transitions are carried through one transition condition or not (Step 6). If this condition is satisfied, NFA to DFA conversion begins. In



(a) ESM NFA (b) ESM DFA

Figure 2: One-transition NFA states are converted into DFA



(a) DFA states before minimizing (b) DFA states after minimizing

Figure 3: Two equivalent DFA states are combined

such a conversion, our algorithm merges all next states for the current state and creates one DFA state, which is a set of merged states (Step 8). Thereafter, it changes the edges such that all incoming and outgoing edges of all merged next states are diverted to the newly created DFA state, and a transition edge is inserted between the DFA state and the current state (Steps 9-11). Finally, we reverse the edges from the current state to the next state(s) before we take further actions on DFA states. Note that this is an important step before we further reduce DFA states (section 3.3.2). If we do not reverse the edges, reducing DFA will guide us NFA back again. This has been shown in Brzozowski's algorithm [17], where DFA minimization converts the input DFA into an NFA by reversing all its arrows and exchanging the roles of the current and next states.

States with one transition condition are common in LTE We next show that one transition condition states are common in LTE; they are related to LTE timers and functionalities. The most common examples are timers that handle *reject* conditions. The standards mandate that, if the device request is *rejected* for certain number of times, the current state should migrate to a particular next state (with the *reject* transition condition); otherwise, the current state should move (with the same *reject* transition condition) to another different next state. Similarly, most LTE functionalities also have NFA states with exact one-transition condition. The transition condition remains the same for different actions, such as cell search, camping on cell, multiple or single bearer request, and priority related features. For example, if the device serving the cell state meets certain threshold value, it should move to the *intra-freq measurement state*; otherwise, it migrates to the *inter-frequency measurement state*. The transition condition for both states is *measurement*.

Discussion Our algorithm processes those states with only one transition condition to the next state. It does not construct the power set of states. The complexity of our algorithm is linear where the *while()* loop (Step 4) iterates over a limited number of states (the finite set of states is provided as an input to the algorithm). The *for()* loop (Step 7) also iterates over a constrained number of next states, because our algorithm executes this step only when the current state moves to the next states upon single transition condition. Note that, the number of next states can be found by looking at the number of arrows coming to the current state (note that we are looking at arrows but not tails, as we are processing in reverse).

Example We now provide an example where our algorithm converts an NFA to a DFA, as shown in Figure 2. Two different tests (case numbers 10.5.1 and 10.5.1b in [1]) move from the current state of *PDN_req_init* into two different states *Procedure_Transaction_Pending* and *Bearer_Context_ Active_Pending* states, respectively, using one transition condition *"PDN req"*. Both cases are requesting PDN from the network. The first case requests an additional PDN for its uplink (UL) data, whereas the second test requests an additional PDN but using NAS signaling low priority. In 10.5.1b, the device establishes a dedicated radio bearer associated with the default EPS bearer context, before sending an additional PDN connectivity request. This is why the device moves to the *Bearer_Context_Active_Pending* state.

3.3.2 Minimizing FSM States

In DFA minimization, two or more equivalent DFA states are merged and represented by one state. Two or more states are said to be equivalent, if these states migrate to the same next state upon the same transition condition.

DFA minimization overview The Hopcroft algorithm for DFA minimization [18] is the best known solution to minimizing a DFA [19][20] with its complexity of *O(nlogn)*. The key idea is to partition the states when two states are not equivalent. At first, all states are placed into one partition and thereafter the partition in refined. The states that are not equivalent are removed from the partition, whereas the equivalent states are merged. The key ingredient of the algorithm lies in how partitioning is done. The trick is to not partition on already visited transition conditions until the partition is further split. In that case, the algorithm only checks one of the two new partitions.

DFA minimization of mixed NFA-DFA FSM We propose a new algorithm for DFA minimization, but use the partitioning procedure from the Hopcroft algorithm. Unlike the Hopcroft algorithm that works on DFA FSM only, our algorithm reduces FSM which is a mix of NFA-DFA. Indeed, like many other FSMs, LTE protocol FSMs are the hybrid of NFA and DFA states, where we have also converted few NFA states into DFA states (algorithm 1). From the Hopcroft algorithm, we find that, as the number of equivalent states increases, the number of states in FSM decreases (as the equivalent sates are merged). Although we cannot increase the states equivalence in FSM, we can obtain similar results by merging two or more states that have constraints on each

other. We use our LTE domain knowledge and introduce protocol states and timing constraints.

Protocol state constraints In LTE protocol FSMs, some states have constraints on others. A few of such constraints have been described in the LTE NAS standard specification (Figure 5.1.3.2.2.7.1: EMM main states in the UE) [21]. For example, when the device is at the idle state and plans to send/receive voice/data packets, it initiates the *Service Request (SR)* procedure and moves to the *Service_Request_Init* state. While the *SR* procedure is ongoing, if the device changes its location and performs the *Tracking Area Update (TAU)* procedure, it cannot do so. This is because the *TAU* procedure can only be initiated from the device state of *EMM_Registered*. We can say that all *TAU* and *SR* related states have constraints on each other, where the device cannot be at both states concurrently and we can merge these states. As a result, our modified FSMs will never produce those output values that capture *TAU* and *SR* interactions (which are don't care outputs). In short, we list all such protocol constraints and merge them.

Timing constraints: Similar to protocol state constraints, there are timing constraints between states. For example, to initiate the *normal TAU* request message, the device must have moved to a different LTE base station cell (that is, its location must have been changed). Further, the LTE base station can only correspond to one tracking area, because *System Information Block-1 (SIB1)* broadcasts only one tracking area code for a cell. Therefore, one cell cannot belong to two different tracking areas. This example illustrates the timing constraint situation where the *normal TAU* procedure can only be initiated after the device has roamed to a different cell in a different tracking area.

Algorithm 2 Minimizing DFA states

```
1: input: = {nstates, trans_cond, trans_func, curr_state, next_state}
2: procedure MINIMIZE-DFA
3:     p_1 is sub-partition 1; p_2 is sub-partition 2
4:     constraint, constraint vector
5:     while no further partitions can be done do
6:         parition_1, all set of states in FSM; parition_2, ∅
7:         for each element p of parition_1 do
8:             if p is non-deterministic then
9:                 split(p, parition_1)
10:            else if (p → other partitions) OR (p ∉ constraint) then
11:                {p_1,p_2} = split(p, parition_1)
12:                if p belongs to {p_1,p_2} and p ≠ ∅ then
13:                    parition_2 = p
14:                    if curr_state == p then
15:                        curr_state_min = p
16:                    else if next_state == p then
17:                        next_state_min = p
18:                    update trans_cond for p
```

Algorithm We now explain our algorithm of minimizing DFA states, as described in Algorithm 2. At the start, there is only one partition that contains all states in an FSM (Line 6). Like the Hopcroft DFA minimization algorithm, our scheme iteratively reduces partitions by removing non-identical states. However, at each iteration, it only takes action when the state is deterministic (Line 10); otherwise, it splits the partition and removes the non-deterministic state (Lines 8-9). Before acting on deterministic states, it ensures that (1) for state p, there is no equivalent state, i.e., state p does not belong to the current partition ($p \rightarrow$ other partitions); and (2) there is no state in the current partition that has constraints with state p (i.e. $p \notin constraint$). If both (1) and (2) are false, it implies that the current state has equivalent or constrained state in the partition and *for()* loop (Line 7) moves to the next iteration. If either (1) or (2) is true, state p is moved out of the partition (Line 11) and becomes a different partition (Line 13). Once the algorithm moves p out of the partition, it checks whether p belongs to the current state or the next state and updates accordingly (Lines 14-17). Note that it is possible that the merged states may not be fully connected to other states. Therefore, at each iteration, we make sure that all incoming and outgoing transition arrows of the merged states are updated accordingly (Line 18).

Discussion Our algorithm does not incur additional complexity compared with the Hopcroft algorithm, which determines the state to be deterministic or non-deterministic in one step (Line 8). It simply checks that the current state must not migrate to more than one next states for a given transition condition. This implies that, the state is NFA without even checking the next states. The step of splitting on non-deterministic states can be viewed as the state having no equivalence behavior. Therefore, the splitting procedure (Line 9) does not add extra complexity. The step of checking the protocol state and timing constraints requires to know whether the current state being partitioned is part of the constraints or not. If it is part of a set of constraints, the algorithm checks whether the current partition contains those states or not. We make these two steps efficient by first logging all such constraints as a constraint vector. The vector is of fixed length and the number of constraints are not large (because these constraints are related to overall LTE functionalities, but not specific to states or timers). Therefore, checking this constraint can always be done in polynomial time. Furthermore, we avoid creating extra steps by merging constraint conditions with partition conditions.

Example We now show an example of three LTE test cases where our algorithm minimizes the equivalent DFA states. Three cases (test case number 9.2.3.1, 9.2.3.1.9a, and 8.5.1.4 in [1]) share part of the common procedure. As shown in Figure 3, these tests perform the *TAU* procedure but under different triggering conditions. In test case 9.2.3.1, when the device moves to a different cell in a different tracking area, it initiates *TAU*. However, in test cases 9.2.3.1.9a and 8.5.1.4, the device fails to recover from the radio link failures and needs to perform connectivity recovery. Once recovered, the

TAU procedure is performed. Hence, we can minimize these DFAs by merging *TA Recovery* and *Connect Recovery* states (shown in Figure 3b).

Table 1: Summary of test cases – our procedure finds new legitimate test cases

Protocol Interaction	Procedure defined by 3GPP	Tests cases defined by 3GPP	Procedure *not* defined by 3GPP	Total *missing* test cases by 3GPP
ECM[3] and ECM	141	118	18	14
ECM and EMM	14	10	7	11
EMM and EMM	161	142	29	54
ESM and ESM	25	22	6	8
Total	**341**	**292**	**60**	**87**

3.4 Proof of Completeness

We prove the completeness by contradiction. The intuition is based on contradicting the number of possibilities and reasoning that our test cases cover them all.

Proof by contradiction We consider two cases and contradict them to show the completeness.

Case 1: Assume that a practical input value was not tested. It is true that we do not test for don't care output values, but these values never happen in reality. We ignore all those input values I that the FSM will never produce (called don't care outputs X). Skipping test cases with don't care values will not lead to missing test cases (i.e., incompleteness scenarios) because in reality there are no such FSM transitions. We test all *practical* combinations of output values of a FSM defined by the 3GPP standards. C is a specification domain that includes all FSM states S that work with the finite set of inputs, I. That is, $C = S \times I$. Furthermore, we traverse the FSM in reverse for a given output value O (Algorithm 1), and we always trace back to the given input value I. In other words, the output leads us to the deterministic FSM and all states can be traversed. This shows that there is not any practical value that was not tested on a completely defined FSM. Hence, our argument contradicts the assumption that we can miss any practical input value that drives the FSM state(s).

Case 2: Assume that certain protocol interactions are missing. Missing those protocol interactions that never occur in reality is equivalent to not testing the case that never occurs. Recall that we consider two protocol interactions. Therefore, for every output value, there are two possibilities for the other output (where the other protocol output exists or not). In other words, we have four possible combinations (these are not four values). Because all such possibilities are within the 3GPP standardized device behavior, we can test all valid combinations (that an FSM can generate). Hence, it is not true that we could miss certain protocols interactions.

By contradicting both cases, we prove that the test cases we generate are complete.

[3] EPS Connection Management (ECM) involve signaling connection that is made up of two parts: an RRC connection and an S1_MME connection. In this paper, RRC test cases are part of ECM procedure.

3.5 Analysis

Once we have reduced the device-side FSMs and identify don't care output values, we can generate the *complete* test cases for LTE protocol interactions. We next provide analysis on test completeness, but defer to Section 5 for discussion on our implementations.

Our procedure generates 30% more test cases compared with the test cases defined by 3GPP. Our result is summarized in Table 1. We also find that, 60 protocol interactions are not specified by the 3GPP protocol standards. Table 2 shows the list of 10 new test cases and identifies new vulnerabilities in the 3GPP standard. The remaining 74 test cases expose relatively less serious issues, such as delay in service access, procedure repetition, downgrade to lower-priority cell, temporary loop between the device states (i.e., idle and connected states), two procedures temporarily blocking each others, etc. Now we provide brief analysis on three novel vulnerabilities discovered by our test cases (other than those discussed in Table 2).

Integrity and ciphering is not enforced We discover that, the device can skip the RRC ciphering and integrity protection even if both are enabled at the network. Such a scenario has been shown in Figure 4, where the *Attach Request* message is forwarded to MME (Mobility Management Entity) (Step 4) before the RRC security procedure starts (Step 6). If the device sends the *Security Mode Failure* message to eNodeB (i.e., the LTE base station), the 3GPP standard allows the device to communicate with the network without any protection, whereas the *Attach* procedure is allowed to complete. We verify this with the LTE RRC standard (Section 5.3.4 Initial security activation procedure in 3GPP TS 36.331 [22]). The standard mandates that, after sending the *Security Mode Failure* message, the UE shall "continue using the configuration used *prior* to the reception of the *Security Mode Command* message, i.e., neither applies integrity protection nor enables ciphering."

To address this issue, we create a test case that makes 5 retries on the *Security Mode Command* message, upon receiving the *Security Mode Failure* message from the device. Once receiving the 5th *Security Mode Failure* message, the eNodeB bars the device from camping on its cell for 60 seconds in our test case.

Sending data without RRC security success We find that, the device can send uplink data even if it has failed to complete the *RRC* security procedure, as shown in Figure 5. The device sends the *Attach Request* message as piggybacked with the *RRC Connection Complete* message and migrates to the *RRC-Connected* state. In the *RRC-Connected* state, the eNodeB sends its *Security Mode Command* to the device (Step 6). However, before receiving the *Security Mode Response* from the device, eNodeB establishes its Signaling Radio Bearer 2

Table 2: Summary of novel findings. These use cases are not defined in 3GPP testing standard and potential vulnerabilities remain untested.

Issue	Protocols	Problem	Root Cause	Impact	3GPP test case exists?	Standard discusses issue?
Detaching victims	ECM – EMM	Device can send non-integrity protected Detach with cause *power off*	The standard allows certain types of messages can be sent as non-integrity protected	Adversary can let victim device detach.	No	No
Service provisioning	ECM – EMM	Local EPS bearer context is deactivated without ESM signaling	The device fails to establish user plane radio bearers when ECM process at network is delayed	EPS bearer context deactivated.	No	No
Skipping integrity	ECM – EMM	TAU message without integrity protection is accepted	TAU due to an inter-system change in idle mode is accepted by the MME even without integrity protection	The device reports wrong location to network.	No	No
Privacy leakage	EMM – ECM	After 4 retries from MME, the GUTI reallocation procedure stops	MME does not mark the device which has failed to perform GUTI reallocation procedure as vulnerable.	Using old GUTI compromises user location.	No	No
Null integrity	EMM – EMM	2nd attach is processed by MME whose IE differs from 1st attach	The device capability related information element (IE) in the Attach Req differs from the ones received earlier	Device attaches as non-integrity protected.	No	No
Barring to Attach	EMM – EMM	Processing Attach request without receiving Identity Response	The MME processes the Attach Request while waiting on Identity Response message from UE	Sending Attach instead of Identity Req bars UE.	No	No
Inconsistent states	EMM – EMM	Device proceeds Detach procedure whereas MME proceeds TAU	Before the detach request is received at UE, UE initiates TAU procedure. MME aborts detach and proceeds TAU	MME and Device states are inconsistent.	No	Yes, TS 24.301, 5.5.2.3.5 e
TAU is blocked	ESM – ECM	PDN procedure is blocked by RRC reconfiguration (doing TAU)	Bearer Modification and RRC-Reconfiguration with TAI change collide. TAU is blocked by earlier procedure	UE will end up keeping invalid tracking area.	No	No
Unauthorized connection	ECM – ECM	UE keeps radio connection for rejected RRC request	When the user identities are not found at EPC, the RRC req is rejected but UE remains camped on eNodeB cell	Connecting eNodeB with expired USIM cards.	No	Yes, TS 36.413, 8.3.3
Deadlock	ESM – ESM	UE and network both initiates Dedicated Bearer Procedures	Both UE and network has received/sent Activate Dedicated Bearer Request and enter into undefined behavior	Network and UE wait on each other request.	No	No

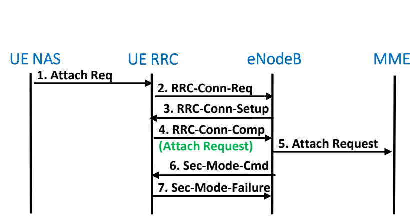


Figure 4: RRC integrity and ciphering is not applied

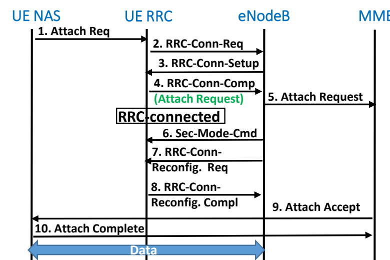


Figure 5: Data transmission starts without activating RRC security

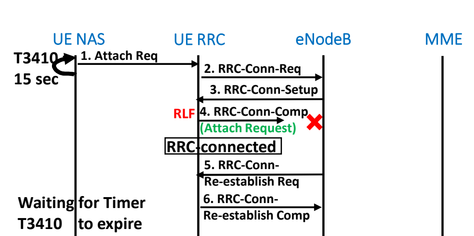


Figure 6: Re-Attach Request is delayed due to Radio Link Failure

(SRB) for the device's uplink/downlink data by performing the *RRC Connection Reconfiguration* procedure (Steps 7-8). Meanwhile, the *Attach* procedure completes (Steps 9-10); whereas the device does not generate the security mode response at all. We find that, the device is able to send its uplink data even if the RRC security procedure did not conclude. This is indeed a loophole in the standard (see Section 5.3.5.3 Reception of an RRC Connection Reconfiguration procedure in 3GPP TS 36.331 [22] and Note 3). It has been stated that, "if the *RRC Connection Reconfiguration* message includes the establishment of radio bearers other than SRB1, the UE may start using these radio bearers immediately, i.e., there is no need to wait for an outstanding acknowledgment of the *Security Mode Complete* message."

Note that eNodeB does not have any timer linked to the security mode command and cannot resend the *Security Mode Command* message, if the response to the previous request is not made. To address this vulnerability, we add a test case to ensure the device has sent the security mode response (i.e., complete or reject).

Re-Attach Request is delayed In this issue, the device registration procedure is delayed up to 15 seconds (the default value for timer T3410). Figure 6 shows that, although the eNodeB has failed to receive the *RRC Connection Complete* message piggybacked in *Attach Request*, the device enters the *RRC Connected* state. Such a failure of message arises because of the Radio Link Failure (RLF). Upon RLF, the device recovers its radio connectivity by performing the *RRC Connection Reestablishment* procedure (Steps 5-6), but does not resend the *Attach Request* message. Therefore, the NAS layer at the device times out for the Attach Request message and resends the request. One can argue that the *RRC Connection Complete* message sent over SRB1 will be recovered by the Radio Link Control Acknowledgement procedure (RLC ACK). However, the RLC procedure recovers the bit errors or retransmission failures over the wireless link, and does not recover the failure because of the device being out of sync with the eNodeB cell (RLF scenario). Moreover, the RLC ACK mode has small timer value (45 milliseconds as the default value [22]) and cannot recover the failure when the radio recovery procedure takes too long.

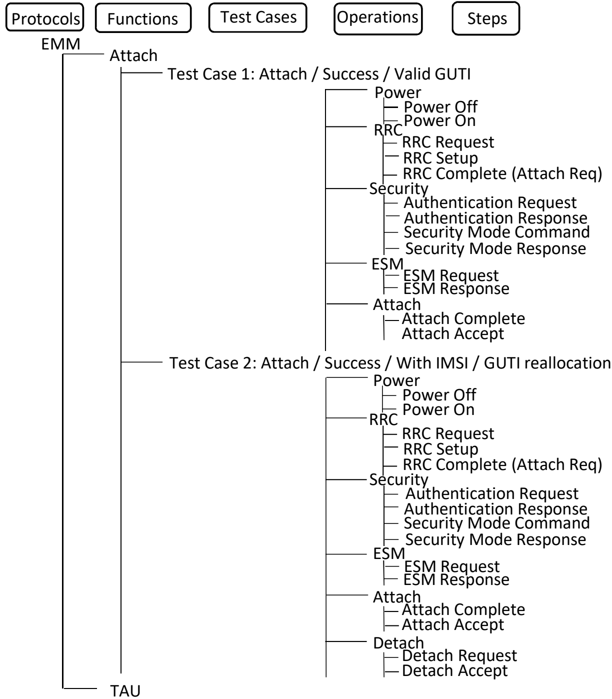


Figure 7: Most of the test case steps are repetitive among test functions. Each LTE protocol (EMM protocol in the Figure) tests a number of test cases (test case 1 and test case 2) that belong to particular LTE function (LTE Attach function). These test cases perform a number of operations (power cycle, RRC, security, etc.). These operations further execute a number of steps (RRC Reques, RRC Setup, RRC Complete etc.)

To address this issue, we create a test case where the EMM layer requests the RRC layer to notify its piggybacked request. If RRC is not able to deliver the piggybacked message within a couple of seconds, the EMM layer will resend the packet.

4 LTE TESTING EFFICIENCY

We next identify inefficiencies when running LTE test cases, and propose a graph based test case execution methodology to improve the efficiency.

4.1 Limitations and Challenges

Testing limitations and challenges arise during scheduling and execution.

Test cases scheduling Protocol conformance tests verify whether a device complies with the LTE protocol specifications or not. Each protocol supports a number of functions that logically separate one protocol from another. For example, the main functions of the *Radio Resource Layer (RRC)* protocol are to establish, configure, maintain and terminate the device's wireless connectivity with the LTE base station. Similarly, the *EPS Mobility Management (EMM)* protocol supports functions related to device mobility, including device registration, authentication and security, location update, and deregistration with the network. Each protocol function is further divided into multiple test cases, each of which

Operations	Steps	Time (sec.msec)
Power	Power off	00.00
	Power on	34.002
RRC	RRC Connection Req	34.009
	RRC Connection setup	34.104
	RRC Connection complete	34.147
Security	Authentication Request	34.150
	Authentication Response	34.285
	Security Mode Command	34.288
	Security Mode Complete	34.428
ESM	ESM Information Request	34.459
	ESM Information Response	34.770
Attach	Attach Accept	35.455
	Attach Complete	35.486

Table 3: Time taken for each step in one of LTE Attach test case

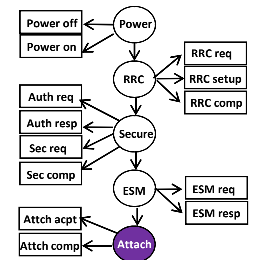


Figure 8: Graph data structure of test case execution

supports operations that execute a number of steps. Figure 7 shows an example of the EMM protocol's *Attach* function that has several test cases. Each case performs several operations that finally execute test steps. In the current practice, all tests from a particular function must complete before tests from other functions could start. However, we find that, many functions are independent of each other, their preconditions are different, and do not overlap with each other. Take the example of tests that validate two different functions (i.e., device *Attach* and *Tracking Area Update (TAU)*) from the same protocol EMM. The *Attach* function is responsible for device registration with the network, whereas the *TAU* function updates the new location to the network upon location updates. To initiate the *Attach* related cases in EMM, the device must be *deregistered*; whereas to initiate the *TAU* related tests, the device must be *registered and its location should have been changed*. We argue that two independent functions can be executed in parallel (such as *Attach* and *TAU* functions in EMM), not currently supported by 3GPP (**inefficiency 1**).

Test case execution Protocol conformance tests certify that each protocol function is executed under all usable conditions. To test each condition, 3GPP defines a new test case that ensures correctness in terms of the signaling flow and the content of each message. However, all such tests execute repetitive operations. For example, the *Attach* function is validated for two types of device identities (i.e., GUTI and IMSI). These are test conditions for which two separate *Attach* tests are defined (test cases 1 and 2 in Figure 7). Both cases validate the *Attach* operation when "Valid GUTI" is passed as a test condition, or "IMSI/GUTI reallocation" is set as a test condition. However, we see both cases perform a number of repetitive operations (e.g., *Power*, *RRC*, *Security*, and *ESM*) that are not related to conditions. As a result, case 2 (Figure 7) executes (i.e. *exec()*) 63% redundant steps which are already performed by case 1 (Figure 7) (**inefficiency 2**).

4.2 LTE Testing as a Graph Data Structure

We improve efficiency via a graph data structure approach.

Overview of our solution We abstract test execution as a graph data structure, where graph non-leaf vertices represent protocol operations and leaf vertices denote the steps that protocol operations take, as shown in Figure 8. We seek to reduce the execution of those steps that are common among different tests of a protocol function (**mitigating inefficiency 2**). We let all tests of a given protocol function to execute one-by-one on the single graph data structure. As an operation step runs, we record its execution information in memory, such as its output parameters. We can thus find whether non-leaf vertex was previously visited or not. If it was visited before, we retrieve the output of that operation steps from memory and move to the next operation without executing these steps. Note that, we are not skipping any step, rather to bypass those executions by retrieving their logged output parameters. Furthermore, we change the current practice by allowing test cases from different protocol functions to execute in parallel (**alleviating inefficiency 1**).

Savings To quantify the efficiency of our approach, we execute the LTE *Attach* related test case over the Anritsu test simulator [23]. Table 3 shows the time taken by each step during the LTE *Attach*. The overall test takes 35 seconds and 486 milliseconds to execute. If we eliminate those repetitive steps on device power off/on during 56 Attach related test cases, we can save about *31 minutes*. If we further avoid repeating executions of RRC, Security and ESM related steps, our test duration decreases from 1.486 seconds to merely 0.031 seconds for each of the followup *Attach* test case. Hence, our approach can decrease the Attach related tests from 33 minutes to 37.222 seconds, 53.66X execution time savings. In this paper, we discuss test efficiency in terms of steps; the lower the number of steps a test executes, the faster it runs in wall clock time.

Contributions We make two contributions. First, to improve test efficiency, we minimize executions by reusing the results from those completed cases. Second, we enable concurrency between different cases and execute concurrent tests in parallel to speed up.

4.2.1 Optimizing test cases

In the proposed scheme, we first initialize a graph that represents all tests of a protocol function. We then perform graph traversal and eliminate reexecuting common operation steps.

Graph initialization Graph initialization is performed at the start of testing. This procedure maps the test case to a graph. Two operations are performed in graph initialization.

- Create the graph non-leaf vertices. Each non-leaf vertex denotes an operation.
- Add leaf vertices to a non-leaf vertex. Each leaf vertex represents a step of an operation.

Since the graph representation is to represent all test cases of a protocol function, repetition of common graph vertices is avoided as early as in the initialization phase. If a specific test operation performs an extra step (e.g., *RRC Connection Release*, in addition to three RRC steps – *RRC Connection Request*, *RRC Connection Setup*, and *RRC Connection Complete*), this step must be added with the original operation (*RRC*) rather than creating a new *RRC* vertex with just one step. Such an approach provides a compact representation for the graph data structure, thus reducing memory utilization and computation.

Graph traversal Graph traversal starts as soon as the first test of a protocol function runs. For the first case, all leaf and non-leaf vertices are executed. During the process, each leaf vertex reports its execution parameters to the associated non-leaf vertex. The non-leaf vertex *marks* itself as visited and stores reported parameters.

For each non-leaf vertex i, we store an array of the vertices (its leaf vertices) adjacent to it. Each array element refers to a character array that contains all parameters produced during the execution. After the execution of first test, the second case of the same protocol function starts. The second test is probably executing most operations as the first one. However, one or two operations are required to be re-executed because the test condition has changed. Therefore, simply looking at the *mark* field to learn whether the operation was previously executed or not is misleading. For example, *Attach with valid GUTI* and *Attach with IMSI* are two different tests but perform mostly the same operations (*RRC*, *Security*, *ESM*, and *Attach* operations, as shown in Figure 7). They only differ in the *Attach* operation. To address this issue, we leverage our domain knowledge and identify those operation(s) pertaining to the test case. To this end, we examine test conformance requirements that describe why the test is executed and which protocol operations will be impacted. For example, both *Attach with valid GUTI* and *Attach with IMSI* require that the *Attach* operation be validated under two different conditions (i.e., GUTI and IMSI). Therefore, even though the *Attach* operation is *marked* by the first test as executed, we have to re-execute the operation. We thus modify our implementation by first creating a hash value of the test condition, and then associating all those leaf vertices of the operation under the test condition with that hash value. For each non-leaf vertex i, the array of the vertices (its leaf vertices) adjacent to it are represented by the hash value of the test condition.

Suppose that *Attach with valid GUTI* was the first test case that completes its execution. Now the second test, say *Attach with IMSI*, starts its execution. We know that the hash value of the condition (IMSI) maps to the *Attach* operation. Therefore, the running test retrieves the parameters for *RRC*, *Security*, and *ESM* operations steps from memory, instead

of executing these steps, and move to the *Attach* operation. At the *Attach* operation, the test searches whether the hash value for the condition (IMSI) exists or not. Since it does not exist, the test will execute all steps in the *Attach* operation and proceeds to the next operation.

Note that, storing the leaf vertices and their parameters with different hash conditions may grow the consumed memory size. To address this, we limit the number of tests that can be traversed over a single graph. We can do so since only a limited number of independent test cases exist within a protocol function. For example, the *Attach* function supports attach-related cases under three different scenarios (i.e., *attach/success*, *attach/failure*, and *attach/abnormal*). They have 17, 23, and 11 test cases, respectively [1]. Consequently, the graph data structure for the *Attach* function only needs to store parameters associated with 23 different conditions.

Parallelizing test cases We further parallelize mutually exclusive test cases to optimize the overall efficiency. The mutually exclusive test cases are those testing different scenarios but not sharing testing conditions, variables, and results with each other. We take a two-step procedure: (1) enabling test case concurrency, and (2) running concurrent tests in parallel. The first step is relatively simple before test cases are executed. We feed the complete list of test cases to our program, which uses the *switch()* statement to differentiate tests based on their three settings (i.e., success, failure, and abnormal). The test cases for the same setting are grouped together. In the second step, we execute tests in three different settings in parallel. We take the process-oriented programming approach, based on ideas derived from CSP (Communicating Sequential Processes) [24]. Each process is a separate piece of code independent from others. Such a programming approach does not incur race hazards involving shared data, scheduling corner cases, and deadlocks.

5 IMPLEMENTATION

Our implementation includes 3GPP test cases, our proposed algorithms, and creation of FSMs and their representation as finite automaton, the complete set of test cases by excluding don't care device outputs. Our implementation is highly modular for better code re-use.

Test cases and their execution For 3GPP test cases, we implement RRC, EMM and ESM cases as described by the 3GPP specification (36.523-1 [1] section series 8, 9 and 10). We prototype each test case by following the procedure described in the specification and run the test as a message sequence between the device and the network. The device and NS are two processes running on the Linux machine. The device generates a message for NS, where NS produces the response by following the standards. If the device does not receive a response from NS (which is the typical case in our protocol interaction testing), we mark it as vulnerable/missing case and manually confirm it with the 3GPP standards. Before each test starts, our program takes a set of configurations defined from a number of *config* files as required by the test case. We create different *config* files for different purposes. For example, *preamble.config*, *cells.config*, *timers.config*, *ie.config*, denote test start states, cell related config, timers and their values, and information elements with device capabilities, respectively.

Finite-state machine Each test case is executed such that the execution is captured as a transition between different states. Such an implementation choice is important to represent test cases as a finite automaton and generate new test cases for inter-protocol communications. The current states, next states, and transition conditions are *enum* type values. For each state, the set of valid state transitions are stored as a *multimap*. The transition function is an action that the current state performs and migrates to the next state. The action is basically a callback function that informs which steps should be performed by the device and for which next state (*ActionCallbackFunc callback = stateTransition[std::pair(current_State, next_State)]*). Using this logic, we can easily represent our FSM as a finite automaton. In the finite automaton, the current state is allowed to have more than one transition (DFA or NFA). We modify the Hopfcroft algorithm code provided by Antti Valmari [25], and define contradictory states as equivalent states.

Protocol Interaction Protocol interaction is represented as test case collisions and their interactions (with/without delay), where each test represents the same or two different protocols. Such interactions have to be tested on each valid output value (in any sequence). To implement this, the messages that a case produces during testing, as well as their corresponding responses, are placed in the queue. The execution of this test case is what the 3GPP testing standard mostly assesses (i.e., single protocol interaction). To find protocol interaction vulnerabilities, we let messages from two protocols interact with the NS, and observe their behavior. Each found vulnerability was manually verified with the 3GPP protocol specifications. For each vulnerability, we propose a new test case that describes the procedure (with the detailed steps) and the fix (the expected behavior).

6 EVALUATION

6.1 Completeness

To evaluate test completeness, we seek to find how many states our algorithm reduces when compared with the best-known finite automaton algorithms. We also quantify how many new cases we can find without enumerating all possible output sequences (results are in Table 1 of Section 3.5).For algorithm comparisons, we vary the total number of NFA

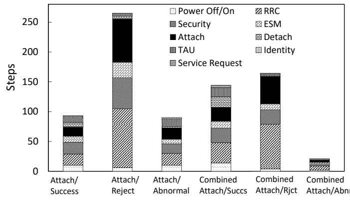


Figure 9: No. of steps in Attach func.

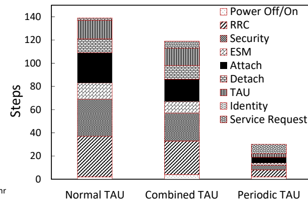


Figure 10: No. of steps in TAU func.

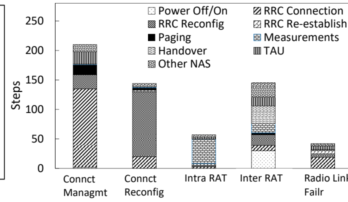


Figure 11: No. of steps in RRC func.

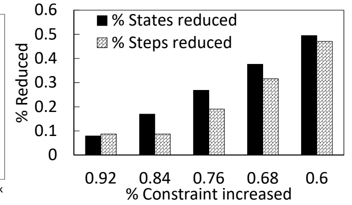


Figure 12: No. of constraint vs No. of steps

states from 100 to 500, and assume half of them have 1 transition condition. In Section 3.3.1, we show that most NFA states have only 1 transition condition in LTE. Table 4 compares our algorithm with the Robins and Scott algorithm on NFA to DFA conversation. The Robins and Scott algorithm converts all NFA states to DFA. The number of steps and states are exponential to the number of NFA states (to be converted). After power set conversion, the algorithm converges by removing unreachable states and the optimal number of states can be obtained. However, the power set conversion at the first step is the major bottleneck. On the other hand, our algorithm only converts those NFA states that have 1 transition condition. We can reduce 1/3 of these NFA states on average. Our algorithm makes NFA to DFA conversion linear through selective state conversion at the cost of 10% more states.

Table 5 compares the Hopcroft DFA minimization algorithm with our proposed scheme. The Hopcroft algorithm does not work on NFA-DFA mixed FSMs. We thus show the DFA states only in Table 5. The Hopcroft algorithm cannot benefit from constraints between different states. In contrast, our algorithm reduces more states by considering such constraints. We find that, we reduce more states by adding fewer constraints in the FSMs (as depicted by Figure 12). Table 5 shows that, we can reduce 26% states by adding just 20% constraints. Moreover, our algorithm performs only 10% more steps than the Hopcroft algorithm, where it skips NFA minimization (an NP complete problem) and merges those states with constraints (treating them as equivalent states).

Table 4: Our algorithm comparison with Robin and Scott algorithm

States			Robin and Scott Algorithm			Proposed Algorithm		
NFA (Total)	1 Trans NFA	DFA (Total)	Steps	States	Optimal States	Steps	Reduced DFA	Total States
100	50	100	2^{200}	2^{200}	167	600	34	184
200	100	200	2^{400}	2^{400}	334	1381	67	367
300	150	300	2^{600}	2^{600}	500	2230	100	550
400	200	400	2^{800}	2^{800}	667	3123	134	734
500	250	500	2^{1000}	2^{1000}	834	4049	167	917

6.2 Efficiency

We measure the efficiency of a test as the number of steps it executes. The rationale is that, every test consists of a number of test operations where every operation is executing a number of steps. Because these steps execute sequentially (one after the other) in a case, their execution time contributes to the testing time. We show that, we can efficiently execute LTE test cases. Table 6 shows the test execution comparison between the ad hoc testing (used by the LTE standard) and our system testing (through our solution). First, we see that our approach executes 43%, 11%, 70% and 50% fewer steps for the *Attach*, *Detach*, *TAU*, and *Service Request* functions, respectively. This is because the graph data structure shares a test execution knowledge with other tests. We did not save much in the *Detach* function, because these tests either require the device to reboot or the USIM to be removed. Therefore, our graph data structure cannot hold the information on previous tests and the steps in the new test have to be re-executed. The savings from the *Attach* function are because our algorithm does not always execute the reboot-device steps, unless explicitly mentioned. In the *TAU* function, our algorithm executes 34 *TAU* related steps compared with 375 steps in the 3GPP standard. We find that, most TAU steps (including the same tracking area code and other configurations) are repetitive. Their execution can be avoided by simply retrieving the execution outcome from the memory.

Table 5: Our algorithm comparison with Hopcroft algorithm

States			Hopcroft Algorithm		Proposed Algorithm	
DFA (Total)	Equivalent	Constraints	Steps	States	Steps	States
100	50	20	400	75	440	56
200	100	40	921	150	1013	111
300	150	60	1487	225	1635	166
400	200	80	2082	300	2290	221
500	250	100	2699	375	2969	276

Figures 9, 10, and 11 show the number of steps taken at different test cases in the *Attach*, *TAU* and *RRC* functions, respectively. We can see that, the *Attach* function can be split into 6 independent test scenarios, thus enabling their parallel executions. Similarly, the *TAU* and *RRC* functions can be split into 3 and 5 test scenarios, respectively, which can also run in parallel. The *Detach* and *Service Request* functions (not shown in the Figure) do not allow for any parallelism. Note that, in the *Detach* function, "UE initiated" and "network initiated" tests cannot be separated, because the device needs to be physically rebooted. However, there is not much gain, even if we can parallelize these functions. This is because

the *Detach* and *service request* functions have only 12 and 10 tests, respectively, compared with *51*, *56* and *114* cases for the *Attach*, *TAU* and *RRC* functions.

Table 6: Comparison of number of steps without (original in test case standard) and with our optimization approach (based on graph data structure)

	Steps Before Optimization				Steps After Optimization			
Operation	**Attach**	**Detach**	**TAU**	**SR**	**Attach**	**Detach**	**TAU**	**SR**
Power Off/On	118	13	20	10	46	12	8	10
RRC	465	54	186	94	254	46	70	35
Security	216	40	164	28	140	32	60	16
ESM	100	14	54	12	68	14	26	8
Attach Req	195	13	68	16	65	11	15	7
Attach Success	130	18	70	28	68	16	34	8
Attach Reject	50	1	1	NA	47	1	1	NA
Detach	42	36	24	16	36	36	24	16
TAU	52	15	375	10	47	12	34	6
Identity Req/Resp	2	2	NA	NA	2	2	0	NA
Service Req/Resp	8	2	20	26	4	2	16	14
Total	**1378**	**208**	**982**	**240**	**777**	**184**	**288**	**120**

7 RELATED WORK

The LTE testing is still a relatively unaddressed topic in the research community, particularly for its efficiency and completeness. Some early studies [26][27][28] have looked into cellular protocol interactions from the performance standpoint, whereas others [29][30] have examined practical attacks over LTE. Other prior work [31][32][34] has discussed the importance of performance or vulnerability related tests for LTE. In contrast, we focus on the LTE testing in terms of complete test cases and efficient test execution. Therefore, we address a different and bigger problem scope to certain extent.

There are also prior efforts on wireless or network related testing. They include network protocol testing [35][36][37], testing via model checkers [38][39][40], and test cases generation by learning queries [41] and finite state machines [42][43]. Specifically, [35] tackles runtime wireless protocol validations by sniffing wireless transmission first and adding nondeterministic transitions later to incorporate uncertainty. The presented technique does not address the NP completeness in search and instead uses heuristics to limit the search. [36] and [37] discuss the model based approach to NFV testing and network fault detection, respectively. Both model the network nodes as FSMs and generate test traffic for FSM executions. However, [36] and [37] do not provide complete list of test cases, nor discuss the efficiency of their approaches. [38] and [39] verify the state-space exploration. They require either constrained metrics as the input or going through all system states. In this paper, we show that finding all possible inputs is practically not feasible for LTE testing. [40] uses model checking to find TCP implementation bugs, whereas, while our work does not aim to find implementation bugs.

On general test design, early work [41] learns test cases through learner and teacher interactions. In contrast, we do not require the device to learn through interacting with the network. Our approach finds valid input values through device FSM transitions. [42][43] address the nondeterminism of FSMs by keeping the input alphabet small, whereas our scheme does not constrain the input alphabet.

8 CONCLUSION

In this paper, we present the first algorithmic approach to LTE testing. Our scheme can offer the complete list of test cases with multiple protocols interactions, and excludes those cases whose corresponding output message combinations are not generated in protocols interactions. We also optimize LTE test cases by eliminating re-execution of repetitive steps among distinctive test cases. Our future work will further look into test scenarios being developed for the upcoming 5G technology, coexistence with other solutions such as 4G and WiFi. We believe that our proposed approach is also conceptually applicable to these new technologies.

ACKNOWLEDGEMENT

We thank our shepherd Dr. Sunghyun Choi and the anonymous reviewers for their insightful comments. This work was partly funded by NSF grants 1423576 and 1526985.

REFERENCES

[1] 3GPP TS 36.523-1: Protocol conformance specification, 2018.
[2] Anite maintains LTE conformance testing lead. http://www.anite.com/businesses/handset-testing/news/anite-maintains-lte-conformance-testing-lead.
[3] Anritsu conformance test systems. https://www.anritsu.com/en-US/test-measurement/mobile-wireless-communications/conformance-test-systems.
[4] Anite conformance test systems. http://www.anite.com/businesses/handset-testing/our-products.
[5] Anritsu: How conformance tests are carried out. http://dl.cdn-anritsu.com/en-en/test-measurement/files/Product-Introductions/Product-Introduction/me7873la-el1100.pdf.
[6] Anite test documents. http://www.keysight.com/en/pd-2372474-pn-E7515A/uxm-wireless-test-set?pm=PL&nid=-33762.1078013&cc=US&lc=eng.
[7] UE demonstration of conformance testing – Anite. http://www.anite.com/businesses/handset-testing/our-products.
[8] 3GPP implementation conformance statement. http://www.etsi.org/deliver/etsi_ts/136500_136599/13652302/11.03.00_60/ts_13652302v110300p.pdf.

[9] H. Gruber and M. Holzer. Computational Complexity of NFA Minimization for Finite and Unary Languages. *LATA*, 8:261–272, 2007.

[10] Y.-H. E. Yang and V. K. Prasanna. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In *IEEE Infocom 2011*.

[11] M. Sipser. Chapter 1: Regular languages. *Introduction to the Theory of Computation*, pages 31–90, 1998.

[12] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[13] M. Sipser. *Theorem 1.19 in Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[14] F. R. Moore. On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. *IEEE Transactions on computers*, 100(10):1211–1214, 1971.

[15] K. Salomaa and S. Yu. NFA to DFA transformation for finite languages. In *International Workshop on Implementing Automata*, 1996.

[16] R. Mandl. Precise bounds associated with the subset construction on various classes of nondeterministic finite automata. In *Inf & Sys Sciences*, 1973.

[17] J.-M. Champarnaud, A. Khorsi, and T. Paranthoën. Split and join for minimizing: Brzozowski's algorithm. *Stringology*, 2002:96–104, 2002.

[18] Hopcroft. An/n log n algo for minimiz. states in kf in ite automaton. 1971.

[19] M. Almeida, N. Moreira, and R. Reis. On the performance of automata minimization algorithms. *Logic and Theory of Algorithms*, page 3, 2007.

[20] Gómez and et al. DFA minimization: from Brzozowski to Hopcroft. 2013.

[21] 3GPP. TS24.301: Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3, Jun. 2018.

[22] 3GPP. TS36.331: Radio Resource Control (RRC), 2018.

[23] Anritsu: verification and test for deployment of LTE. http://www.chinacom.tw/ngn2010/pdf/ngn/Anritsu.pdf.

[24] Easy Concurrency for C++. https://www.cs.kent.ac.uk/projects/ofa/c++csp/.

[25] DFA minimization using Hopcroft alogirithm: C++ implementation. http://www.cs.tut.fi/~ava/DFA_minimizer.cc.

[26] G.-H. Tu, Y. Li, C. Peng, C.-Y. Li, H. Wang, and S. Lu. Control-Plane Protocol Interactions in Cellular Networks. In *ACM SIGCOMM*, August 2014.

[27] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An in-depth study of LTE: effect of network protocol and application behavior on performance. In *ACM SIGCOMM*, 2013.

[28] C.-H. Chuang and et al. Performance study for HARQ–ARQ interaction of LTE. *Wireless Communications and Mobile Comp.*, 10(11):1459–1469, 2010.

[29] Hussain, Syed Rafiul and Chowdhury, Omar and Mehnaz, Shagufta and Bertino, Elisa. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[30] A. Shaik and et al. Practical attacks against privacy and availability in 4G/LTE mobile communication systems. 2015.

[31] R. Subramanian, K. Sandrasegaran, and X. Kong. Benchmarking of real-time LTE network in dynamic environment. In *IEEE APCC*, 2016.

[32] H. Zhao and H. Jiang. LTE-M system performance of integrated services based on field test results. In *IEEE IMCEC*, 2016.

[33] B. Cui, S. Feng, Q. Xiao, and M. Li. Detection of LTE Protocol Based on Format Fuzz. In *IEEE BWCCA*, 2015.

[34] Gerasimenko and et al. Energy and delay analysis of LTE-advanced RACH performance under MTC overload. In *IEEE Globecom Workshops*, 2012.

[35] Shi, Jinghao and Lahiri, Shuvendu K and Chandra, Ranveer and Challen, Geoffrey. Wireless protocol validation under uncertainty. In *International Conference on Runtime Verification*. Springer, 2016.

[36] Fayaz, Seyed Kaveh and Yu, Tianlong and Tobioka, Yoshiaki and Chaki, Sagar and Sekar, Vyas. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *NSDI*, 2016.

[37] Lee, David and Netravali, Arun N and Sabnani, Krishan K and Sugla, Binay and John, Ajita. Passive testing and applications to network management. In *International Conference on Network Protocols*. IEEE, 1997.

[38] Godefroid, and Patrice. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997.

[39] Khurshid, Sarfraz and Păsăreanu, Corina S and Visser, Willem. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003.

[40] Fiterău-Broştean, Paul and Janssen, Ramon and Vaandrager, Frits. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification*. Springer, 2016.

[41] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. In *IEEE Information and Computation*, 1987.

[42] T.S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 1978.

[43] Petrenko, and Alexandre. Toward testing from finite state machines with symbolic inputs and outputs. *Software & Systems Modeling*, 2017.