# ProChecker: An Automated Security and Privacy Analysis Framework for 4G LTE Protocol Implementations

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

***Abstract*—Cellular protocol implementations must comply with the specifications, and the security and privacy requirements. These implementations, however, often deviate from the security and privacy requirements due to under specifications in cellular standards, inherent protocol complexities, and design flaws inducing logical vulnerabilities. Detecting such logical vulnerabilities in the complex and stateful 4G LTE protocol is challenging due to operational dependencies on internal-states, and intertwined complex protocol interactions among multiple participants. In this paper, we address these challenges and develop ProChecker which— (1) extracts a precise semantic model as a finite-state machine of the implementation by combining dynamic testing with static instrumentation, and (2) verifies the properties against the extracted model by combining a symbolic model checker and a cryptographic protocol verifier. We demonstrate the effectiveness of ProChecker by evaluating it on a closed-source and two of the most popular open-source 4G LTE control-plane protocol implementations with 62 properties. ProChecker unveiled 3 new protocol-specific logical attacks, 6 implementation issues, and detected 14 prior attacks. The impact of the attacks range from denial-of-service, broken integrity, encryption, and replay protection to privacy leakage.**

***Index Terms*—Cellular networks, Mobile and wireless security, Formal security models**

## I. Introduction

Implementations of cellular network protocols, such as 4G LTE and 5G NR, must adhere to the specified security and privacy requirements. Unfortunately, lack of secure-by-design approaches for these complex protocols often induces vulnerabilities in implementations with security and privacy repercussions. While memory corruption vulnerabilities (e.g., buffer overflows, use-after-free) can be detected without prior knowledge about the protocol utilizing memory sanitization techniques [1], detecting logical vulnerabilities (e.g., resetting the counter to break the replay protection of protocol messages) in large and complex protocol implementations is challenging since logical vulnerabilities do not have externally-discernible effects such as crashes or memory leaks. Instead, they require an in-depth semantic understanding of the protocol interactions and are thus primarily detected through manual analysis.

**Problem.** Recent work has demonstrated the effectiveness of formal verification in identifying logical vulnerabilities in 4G LTE [2] and 5G NR [3] protocols. Most of these proposals, however, primarily focus on developing a *standalone* security and privacy analysis framework for verifying *specifications* of protocols on a manually constructed simplified model, which is hardly an option for *commercial-scale* complex implementations. On detecting logical flaws of 4G LTE protocol implementations, previous approaches [4]–[9] have one or more limitations: (A) The analysis [4]–[8] is completely manual; (B) The analysis [9] performs stateless semi-automatic dynamic testing of the implementation but requires significant manual analysis and can only test few pre-defined properties. Even though such manual or semi-automated security analyses are effective to some extent, from a commercial vendor's point-of-view the use of different test infrastructures for separate functional and security testing is often expensive and leaves security testing at a low priority. To address these challenges, this paper aims at answering the following research question: *Is it possible to evaluate the security and privacy properties of a commercial-scale 4G LTE protocol implementation and integrate the evaluation with the mainstream functional testing framework to uncover logical vulnerabilities?*

**Challenge.** Prior work [2], [3], [10], [11] evaluating the design of cellular network protocols represents the high-level protocol interactions with finite state machines (FSMs) and evaluates the FSMs against desired security and privacy properties. Such approaches can also be naturally applied to the FSM's of 4G LTE protocol implementations. One major challenge in applying such model checking based formal verification to protocol implementations is, however, the automatic extraction of the FSM from the implementation. It is critical that the extracted model (represented by a FSM) is in bounds for the state-of-the-art model checking tools, contains semantic meaning, and is explicit enough to allow one to identify logical vulnerabilities. However, due to under-specifications in the standards, developers are free to design and implement some part of the protocol in their own way— with the only requirement of matching input/output behavior. Thus implementations of internal protocols structure most often deviate from the standards. This necessitates a sophisticated and automated model extraction technique to reverse-engineer a model from the implementation to properly verify properties on protocol implementations.

**Plausible approaches.** Conceptually, one can extract the model using one of the following two broad approaches: (1) static analysis, and (2) dynamic analysis. For a typical industrial implementation with pointers and function redirections, static analysis techniques are unable to meet the precision required to reason about both implementation soundness [12] and completeness. On the contrary, though dynamic analysis would appear to be effective because of its high precision, it fails to scale for production-level and large-size implementations, when executing all feasible paths and suffers from state space explosion. Nonetheless, existing popular dynamic extraction techniques such as active-automata learning [13], [14] are used to extract FSM's of the implementations of other protocols e.g., TLS, SSH in a black-box setting. However,

such approaches are prohibitively expensive as they require a significantly high time and number of queries to infer the target implementation's FSM. Moreover, the inferred FSM is not sufficiently large and semantically rich compared to that of the white-box settings. For the FSM extraction, our goal is, therefore, to achieve the accuracy of dynamic analysis without falling into state explosion [1] and utilize the white-box information to create a semantically rich model.

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

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

For model checking, like LTEInspector [2], we combine the reasoning powers of the symbolic model checker and a cryptographic protocol verifier to detect logical vulnerabilities that adhere to the cryptographic constructs of the protocol. The reason behind combining the model checker and cryptographic protocol verifier is to: (i) efficiently capture all the desired properties that we have observed; (ii) reason about rich temporal properties (e.g., safety, liveliness, correspondence) that could not be captured if one of them is solely used.

**Implementation.** We evaluate the effectiveness of ProChecker on a closed-source and two open-source (srsLTE [15] and OpenAirInterface [16]) 4G LTE implementations. We instantiate the model checking component of ProChecker with the nuXmv infinite-state model checker [17] and the ProVerif cryptographic protocol verifier [18]. For properties, we use the conformance test suite [19] suggested by the standard along with the properties which are implicit in the standard. The key properties and insights leveraged by ProChecker and the major procedures discussed here remain unchanged in the upcoming 5G deployment, making our framework directly applicable to 5G and securing upcoming generations.

**Contributions.** The paper has the following contributions:

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

**Responsible Disclosure.** We have reported the protocol vulnerability findings of ProChecker to GSMA through the coordinated vulnerability disclosure (CVD) program and are actively coordinating with GSMA regarding the issues. The CVD submission (CVD-20201-0043) has been awarded Mobile Security Hall of Fame status by GSMA [20]. We have also reported implementation issues to open-source 4G LTE protocol stack developers [15], [16].

## II. Background

We introduce relevant aspects of the 4G LTE protocol, logical vulnerabilities, and elaborate the key properties of cellular network protocols leveraged by ProChecker.

### A. LTE System Architecture

The 4G architecture can be divided into three components: (i) UE, (ii) E-UTRAN, and (iii) EPC. The "User Equipment" (UE) is a device (e.g., smartphone) and contains the modem that is essential for communication. The UE has the SIM card, which securely stores the unique international mobile subscriber identity (IMSI) number and associated keys for UE identification and authentication. For communication, each area is divided into hexagonal cells. Each cell is served by a single base station known as eNodeB. The eNodeB connects the UE to the core network. The network between the UE and the eNodeB and pairs of eNodeBs is the radio access network (E-UTRAN). Evolved packet core (EPC) is a mesh of interconnected services and is divided into several components. The Mobile Management Entity (MME), the most important to our discussion, manages attach, detach, and other important procedures of the UE's in a particular hexagonal cell.

### B. NAS Layer Procedures

When a UE reboots, it tries to connect to the nearby eNodeB with the highest signal strength. After the connection establishment with the eNodeB, the UE starts the attach procedure by sending the `attach_request` to the MME through the established connection. The attach procedure then goes through two important phases. First, for verifying the authenticity of both the UE and MME a challenge-response authentication procedure is completed through the `authentication_request/response` messages. Second, the security algorithm is negotiated through the `security_mode_command/complete` messages. After these two procedures, the attach procedure completes; all the subsequent messages are encrypted and integrity protected and the MME

assigns a globally unique temporary identifier (GUTI) to the UE to limit the exposure of IMSI. Other than this the GUTI reallocation procedure, the paging procedure, and the tracking area update procedure are used to change the GUTI of a user's device, provide service, and update the user's tracking area simultaneously (see Figure 1).

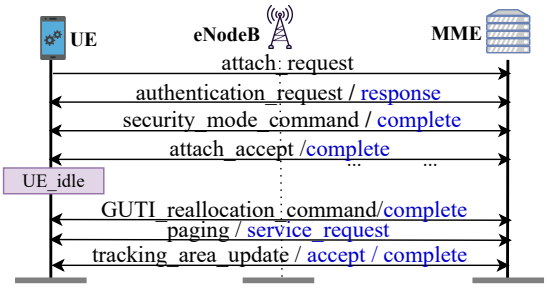



**Fig. 1:** Overview of NAS Layer Procedures

### C. Logical Vulnerabilities

Logical vulnerabilities are issues that force the protocol to deviate from (i.e., yield a trace that violates) basic security (confidentiality, integrity, availability) and privacy guarantees without having an externally-discernible effect such as crash or memory corruption. The deviations can be attributed to protocol level design-flaws, underspecifications in the standards, and implementation mismatch. For instance, underspecifications and inadequate checks in replay protection induce logical vulnerabilities in 4G/5G protocols enabling an adversary to force a user to use the same session keys [4] (also known as *key-reinstallation* attack) and reset the replay protection counters [3]. Note that all these previously uncovered issues have been identified manually or from manually derived models.

### D. Properties of LTE Protocol Implementation

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

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

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

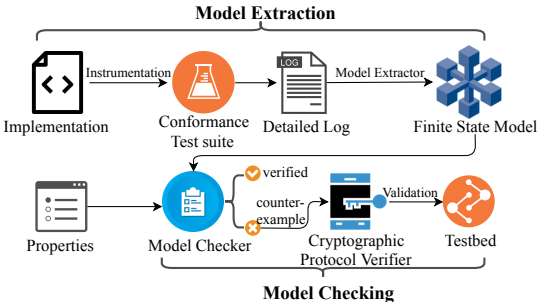



**Fig. 2:** Architecture of ProChecker

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

## III. Overview of ProChecker

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

### A. Threat Model

We consider a Dolev-Yao adversary model [21] in which the communication channel between the client and the server is subject to the following adversary actions: arbitrary packet dropping, injection, or modification while impersonating a legitimate participant. In this model, the adversary adheres to cryptographic assumptions, i.e., it can decrypt a packet only if it has the keys.

### B. Protocol Finite State Machine

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

### C. Challenges

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

**C1:** *(Colossal codebase).* The main challenge is the sheer scale of the complex 4G LTE protocol implementation; in the case of industrial implementations with legacy codes this problem becomes intractable. For such implementations, both

static and dynamic analysis results in imprecision [12] or state space explosion. The models generated from exclusively applying different static or dynamic analysis techniques either contain low-level intricacies of intra- and inter-procedural interactions and thus result in scalability issues.

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

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

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

### D. Insights on Addressing Challenges

For addressing C1 and C2, ProChecker does not rely directly on the codebase to extract the FSM of the implementation; rather it leverages the execution logs of the protocol interaction. The logs are captured from the execution of conformance test cases provided by the protocol standard body and/or the code manufacturers. 3GPP, the 4G LTE standard body, provides conformance test suites for protocol implementation verification [22]. Also, commercial vendors have their own functional testing infrastructure and code coverage information. To address C3 we automatically instrument the codebase to inscribe necessary information in the execution logs to create a FSM with semantic information. For C4 we only extract interactions of a particular layer from the execution logs and utilize the state and protocol message names from the standards [19].

### E. High Level Description of ProChecker

ProChecker comprises of two components: (i) Model extraction; and (ii) Model checking (see Figure 2). For inferring a 4G LTE implementation's FSM, the model extraction leverages the *testing logs* generated from the functional conformance test suite. It is, therefore, important to provide a detailed execution log enriched with a sufficient-level of semantic information to the model extractor. For this, our simple source-code level instrumentor automatically instruments the code to dump the values of global and local variables. Note that, our instrumentor does not require any knowledge about the implementations, such as control-flow, program-dependency or call graphs. The information-rich log is then passed to the model extractor, which from the log extracts the specific state, condition, and actions of the FSM following a generic algorithm (see Algorithm 1). The algorithm utilizes the traits of a generic 4G LTE implementation (that holds for both commercial and open-source implementations) and state and protocol message names from the specification. Our extracted model abstracts out all cryptographic assumptions and for all encrypted/integrity-protected messages, the plain-text counterpart is extracted. For instance, a specific protocol message may always be encrypted and transmitted with integrity protection; our extracted model does not include that information and only includes the interaction as plain-text. for analysis.

For model checking, our approach is based on the counter-example-guided-abstraction-refinement principle (CEGAR) [23]. In the CEGAR framework, an initial abstract model, and property are passed to the verifier. If the abstract model results in erroneous (or "spurious") counterexamples, the model is revised to rule out the spurious counterexamples. This continues until the verification goes through or a realizable counterexample is found. Based on CEGAR, in ProChecker, (1) our extracted 4G LTE model abstracts out the cryptographic assumptions. (2) We then instrument that model with Dolev-Yao [21] adversarial assumptions and call it a threat-enhanced model. (3) The threat-enhanced model and properties to check are passed to a general-purpose symbolic model checker. Note that the model may generate a spurious counterexample due to the absence of cryptographic abstractions. (4) To resolve this, we use a *cryptographic protocol verifier*. If the protocol verifier confirms that all the steps in the counterexample adhere to the cryptographic assumptions, then the counterexample (alternatively, the attack) is reported by ProChecker. Otherwise, we refine the property to ensure this spurious counterexample is never generated again. Like the CEGAR framework, this loop continues until the verification completes or a realizable counterexample is found.

## IV. Detailed Design of ProChecker

We now dive deep into both the components of ProChecker.

### A. Model extraction

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

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

**(2)** *Code instrumentation*. The challenge for code instrumentation is to add information to the log with minimal implementation knowledge. To address this challenge, our code instrumentation prints only the values of global variables, local variables and function entrance/entry points in the log for each function. The value of global variables on the entry and exit for each function is used to detect state transitions, whereas the output

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

**(3)** *Dissecting the log to detect relevant states, conditions and actions.* The log created through code-instrumentation and conformance test suite contains all the global, local variable values, and function entrance indications that are executed/accessed during the test case execution. The next challenge is to use this information with minimal implementation knowledge to extract the FSM. We leverage key insights from the 4G LTE protocol and their implementations for this step. As the 4G LTE protocol follows an event-driven paradigm, we can dissect the log into blocks based on each incoming message to the protocol. After the packet is received by the implementation, it is passed to the corresponding *incoming message handler* designated for unpacking, decrypting, sanity checking (e.g., packet type and well-formedness), and validation of cryptographic primitives (e.g., message authentication code or MAC). Depending on which checks are passed, the internal state of the protocol is changed accordingly and the control moves to the corresponding *outgoing message handler* designated for taking the responsive action. Depending on the results of checks performed by the incoming message handler and protocol context, the receiver may take an action, i.e., send a response packet (*accept* or *reject* based on the validation results) to the other communicating entity (UE/MME) or take no action at all (referred to in our FSM as null_action). For example, whenever an authentication challenge is received, it is passed to the incoming message handler for processing authentication challenges. Upon completion of sanity checking and internal state transition, the authentication completion message is sent as a response from the outgoing message handler. Since the condition variables used in the sanity checking are local variables, we obtain their values from the information-rich log containing values of all the local variables declared and defined in the corresponding message handler. In a similar vein, we extract the current and next state information from the inscribed global state variables in the log. Based on the incoming message handler (from the function entrance indication in the log) and the outgoing message handler execution, we extract the type of message received i.e., the condition and sent i.e., the action of the FSM.

**(4)** *Mapping protocol specific variables to implementation.* To map the 4G LTE protocol/standard specific state variables, incoming and outgoing messages, and condition variables to the myriad of implementation-specific variables in the log, we leverage the following intuitions: (1) 4G LTE state names defined in the standards [19] are directly used in the implementations to ensure interoperability. Therefore, by simply knowing the name of each state defined in the standards, we can detect the corresponding state represented with global variables. (2) Similarly, incoming/outgoing message names defined in the protocol specification are indirectly used in the implementation as function signatures. For industrial implementations, the same signature is followed consistently throughout the implementation and even for the open-sourced implementations, consistent signatures have been used. The consistency aides tractability, efficient portability, and interoperability. For instance, a sample signature is to prepend `send_/recv_` (based on whether the protocol message is incoming or outgoing) as a prefix before the actual protocol message name. Instances of this signature can be `send_authentication_request`, `recv_authentication_response`. Leveraging this insight, we use the function entrance information to extract both the type of message received and sent during protocol interaction and represent them as conditions and actions in a transition of the FSM. The algorithm for model extraction from the log is shown in Algorithm 1. The algorithm takes the generated Log, state, and incoming/outgoing message signatures as inputs and outputs the $FSM$. First, the log is divided into a block based on the incoming message signature that caused the protocol interaction. The block is then scanned line by line to extract states ($FSM.\mathcal{S}$), conditions ($FSM.\Sigma$), and actions ($FSM.\Gamma$) [line (4-18)]. Intuitively, the first extracted state of the block

**Algorithm 1:** ProChecker Model Extractor

```
   Data: Log, state_signatures, incoming_signatures, outgoing_signatures
   Result: FSM(Σ, Γ, S, s_0, T)
1  while end of Log not reached do
2      B ← DivideBlock(Log, incoming_signatures)
3      for each line L ∈ B do
4          if L contains any s ∈ state_signatures then
5              append s to FSM.S
6              if s is the first state_signature ∈ B then
7                  s_in ← s
8              end
9              else
10                 s_out ← s
11             end
12         end
13         else if L contains any σ ∈ incoming_signatures then
14             append σ to conditions set FSM.Σ
15         end
16         else if L contains any γ ∈ outgoing_signatures then
17             append γ to conditions set FSM.Γ
18         end
19     end
20     if γ is empty then
21         γ ← null_action
22     end
23     append transition tuple (s_in, s_out, σ, γ) to FSM.T
24     remove B from Log
25 end
```

is denoted as the incoming state and the second one as the outgoing state [line (6-11)]. As already discussed, there might be the case when the incoming message does not trigger any action for the protocol (due to failed validation); in that case, the action is denoted as null_action [line (20-21)]. At the end of the extraction, the tuple $(s_{in}, s_{out}, \sigma, \gamma)$ is added to $FSM.\mathcal{T}$ to keep track of the transition relation system.

### B. Model checking

Our approach combines a symbolic model checker (MC) and a cryptographic protocol verifier (CPV). As the 4G LTE protocol can be considered as a set of communicating FSM's, we model each communication between two FSM's, for instance, the communication between the UE and MME as, $\mathsf{UE}^{\mu}$ and $\mathsf{MME}^{\mu}$, with two uni-directional channels; one from $\mathsf{UE}^{\mu}$ to $\mathsf{MME}^{\mu}$ and another from $\mathsf{MME}^{\mu}$ to $\mathsf{UE}^{\mu}$. The choice of using two unidirectional channels instead of a single bidirectional channel provides more flexibility (e.g., one direction of the public channels to be adversary controlled whereas the other to be reliable) in reasoning about specific scenarios and filtering spurious counterexamples. From the extracted models $\mathsf{UE}^{\mu}$ and $\mathsf{MME}^{\mu}$ and including the two unidirectional channels, we enhance the model to include a Dolev-Yao-Style adversary and create a threat instrumented model $\mathsf{IMP}^{\mu}$. We then use a general-purpose model checker [17] and a property to check whether the model satisfies the property. If the model satisfies the property, we adjudicate the property to be verified on the model. If, however, a counterexample is generated, there can be two possibilities: (a) the implementation model violates the property; (b) due to the abstraction of cryptographic-constructs, a spurious counterexample was generated. To prevent spurious counterexamples we run steps of the counterexample to a symbolic CPV. If the CPV confirms that all steps conform to the cryptographic assumptions, the counterexample can be considered valid. If the CPV adjudicates one of the steps taken by the adversary to be infeasible, we refine the property to ensure that the adversary does not exercise offending action in the future iterations of the verification. The verification loop continues until either the property is satisfied or a realizable counterexample is found.

## V. Running Example

To illustrate our model extraction approach, we walk through a simplified example code (see Figure 3) of the attach procedure for a device in 4G LTE UE. The code is abstracted to include only the protocol interactions of Non-Access Stratum (NAS) layer of the cellular stack. The same algorithm can be utilized for other protocol layers. For our running example, we focus on the code of a UE for the final phase of the attach procedure, i.e., the protocol interaction through `attach_accept`/`attach_complete`. For ease of exposition, we assume that the simplified implementation contains three functions (see Figure 3). $air\_msg\_handler$ takes a message from the MME, parses the message, identifies its type, and passes it to the corresponding handler associated with it. For our example, the incoming message is an `attach_accept` and it is thus routed to $recv\_attach\_accept$. The first task in any implementation of $recv\_attach\_accept$ is to check whether the message contains a valid MAC. If the MAC check is passed, the control is transferred to the respective outgoing message handler that sends an appropriate response– which in our case is $send\_attach\_complete$. In this example code snippet, our instrumentation tool automatically includes few print statements that inscribe all global and local variables values, and function entrance information (see Figure 3, the instrumented lines are shown in blue) when relevant test cases are executed. For instance, consider a simple test case: "When a properly formatted `attach_accept` message with appropriate MAC is sent to the UE, the UE responds with an `attach_complete`". As the test case gets executed with the instrumented code, we get a detailed log (see Figure 3(d)).

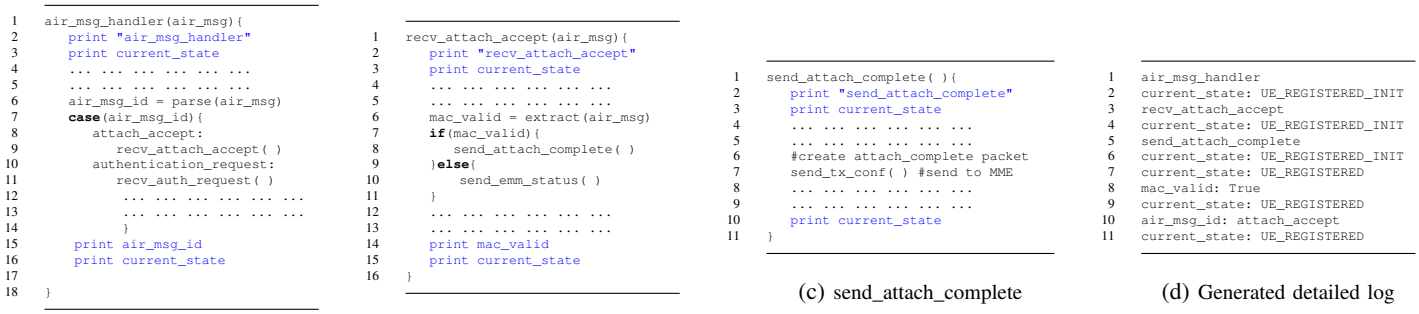
Now the task of the model extractor is to build the FSM from the log. For building the FSM, we need to extract four specific pieces of information from the log: (1) incoming state, (2) outgoing state, (3) conditions, and (4) actions. in our example, line 3 of the log (Figure 3(d)) indicates that the control has moved to $recv\_attach\_accept$ handler, which essentially means that the condition for this transition is the incoming `attach_accept` message. Down the trace, line 8 indicates that the MAC for the message is computed as valid. Note that the initial state for this transition is extracted from line 6 and identified as `UE_REGISTERED_INIT`. The final state is extracted from line 9 as before completing the specific test case the state transitions to `UE_REGISTERED` state. Line 5 manifests that an `attach_complete` message was sent by the device in response to this particular test case. This example shows the effectiveness of our approach in building a FSM of an implementation without requiring detailed knowledge about the source code. In a practical case, the generated log will contain information about multiple rounds of interaction between the UE and the MME. In that case, the log can be divided into blocks based on the incoming message signature names. From the blocks, a similar strategy can be applied to extract the entire state machine.

## VI. Implementation

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

**Formal property gathering.** The set of properties that ProChecker aims to check includes authenticity (e.g., disallowing impersonation attacks), availability(e.g., preventing denial of service attacks), integrity(e.g., restricting unauthorized messages), privacy of user's sensitive information (e.g., preventing location data, activity profiling, and preserving users soft identity), and replay protection. (e.g., restricting reception of the same message more than once). We identify and extract the precise and formal security goals from the informal and high-level descriptions given in the conformance test suites [22] and technical specification documents [19] provided by 3GPP and translate them into properties. We extracted, formalized, and verified a total of 62 properties among them 25 are related to privacy and 37 related to security.

**Codebases.** For the closed-source implementation, the com-



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

}
```

(a) air_msg_handler

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

(b) recv_attach_accept

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

(c) send_attach_complete

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

(d) Generated detailed log

**Fig. 3:** Instrumented generic example implementation (instrumented lines in the code are colored as blue)

plete size of the codebase is around 80 GB (including all testing infrastructure and legacy support). We integrate ProChecker with the mainstream functional testing framework of the implementation. For the open-source implementations we use the two most popular ones, srsLTE [15] and OpenAir-Interface(OAI) [16]. All the codebases are written in C++.

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

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

**Model extractor.** We implement the model extractor in Python 2.7 with around 1000 lines of code. We leveraged the protocol state names directly from the standards [19] as the implementations use identical names. We mapped the incoming/outgoing message signatures, sanity checking variable names following the incoming/outgoing message names from the standards, and a manual inspection of the source files of the NAS layer. For future generations, this mapping can be documented with minimal effort while designing the implementation, thus eliminating this one-time manual intervention altogether. For the largest log from the closed-source implementation, it takes our model extractor around 5 minutes to analyze the log and generate the semantic model.

**Adversarial model instrumentor.** The adversarial model instrumentor takes two FSM's— $\mathsf{UE}^{\mu}$ and $\mathsf{MME}^{\mu}$ for UE and MME as input and returns another model $\mathsf{IMP}^{\mu}$ which is an extension of $\mathsf{UE}^{\mu}$ and $\mathsf{MME}^{\mu}$ containing explicit adversarial influence. Given two public communication channels– $c_1$ from $\mathsf{UE}^{\mu}$ to $\mathsf{MME}^{\mu}$, and $c_2$ from from $\mathsf{MME}^{\mu}$ to $\mathsf{UE}^{\mu}$, our ProChecker incorporates adversarial capabilities into $\mathsf{UE}^{\mu}$ and $\mathsf{MME}^{\mu}$ and thus combine them all to build a new threat-instrumented abstract model $\mathsf{IMP}^{\mu}$. For instrumenting threat to a given transition, the adversary non-deterministically decides either to drop/pass/change the message. We have developed a model generator that takes as input the state machine of the protocol written in Graphviz-like language and outputs a SMV [17] description of the model. For our implementation, we extracted the models of the UE by using our proposed model extraction module. We, however, did not have access to the commercial/closed-sourced implementation of a core network and thus used the open-source core network's FSM manually constructed by Hussain et al. [2], which precisely served our purpose as we were interested in identifying vulnerabilities on the UE side. But it is evident that, given the implementation and the test cases, this approach can also be applied to the core network's implementation.

**Model checker (MC).** To model check $\mathsf{IMP}^{\mu}$, we use NuXmv [17]. A major challenge in formal verification is scalability; the model checker may not be able to terminate when the model is large. We want to report that it was indeed possible to run a model checker on the model extracted from an industry-level large codebase. This is because of our semantic model-extraction based on high-level protocol interactions from the log and abstracting out low-level details of implementation.

**Cryptographic protocol verifier (CPV).** The counterexample generated from MC is fed to the ProVerif [18] CPV to determine its validity. For each adversary action in the model checker provided as a counterexample, we query the CPV to check its feasibility. If all adversarial actions can be proven feasible, then the counterexample is presented as a feasible attack and tested on the testbed. Otherwise an invariant is added to the property ruling out the infeasible adversarial action to refine the property. The verification loop between MC and CPV is continued until either the property is satisfied

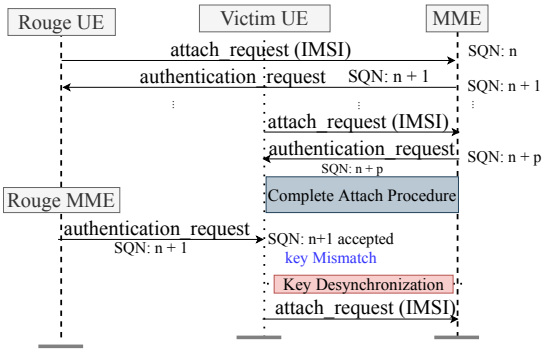



**Fig. 4:** Service disruption using `authentication_request`

by the model or a realizable counterexample is found.

**Testbed.** We build a testbed using low-cost software-defined radios and open-source LTE software stack, srsLTE [15], totaling to a cost of around USD $4000. After verifying the counterexample, we manually analyze the counterexample to determine whether it is due to the implementation deviating from the protocol specification or the deviation is due to underspecification, design-flaw in the standards.

## VII. Evaluation and Findings

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

- **RQ1. Logical Vulnerability Detection:** How effective are the extracted models of ProChecker? Is it possible to reason about security and privacy properties to detect logical vulnerabilities with the models?
- **RQ2. Model Comparison:** How expressive is the automatically extracted FSM of ProChecker compared to the state-of-the-art model [2] for formal verification?
- **RQ3. Scalability:** Is the generated model scalable with the COTS model checking tools?

### A. RQ1. Logical Vulnerability Detection

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

*1) New counterexamples*

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

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

**Detection and attack description.** We uncover this attack by model checking IMP$^{\mu}$ with respect to the property: "*If the UE is in the registered initiated state, it will get authenticated with an authentication sequence number (`SQN`) which is greater than the previously accepted `SQN`*". We observe a counterexample in IMP$^{\mu}$ where the UE accepts a `SQN` value smaller than the current value. We validate the capability of the adversary to fabricate `attach_request` message to generate legitimate `authentication_request` by the MME using ProVerif. The steps of the attack are shown in Figure 4. The adversary using a malicious UE sends `attach_request` message to the MME to capture `authentication_request` message to be used later in the attack. At the time of the attack, the attacker replays such captured `authentication_request` to the victim UE. Due to the specific design of the generation and verification of `authentication_request` message's `SQN` (which we describe in detail in the next section), the victim UE accepts and processes this stale `authentication_request` message, and regenerates all session keys causing a key desynchronization between the UE and the legitimate MME. As a result, the UE will also keep discarding actual packets from the legitimate MME until the connection is dropped and the authentication procedure is invoked from scratch by the legitimate MME. By analyzing the traces of real operational networks, we uncover that it is possible to use previously captured `authentication_request` messages–which are days old to carry out such an attack. To extend the attack impact, the adversary can keep using previously captured `authentication_request` messages and replay them to the victim UE recurrently.

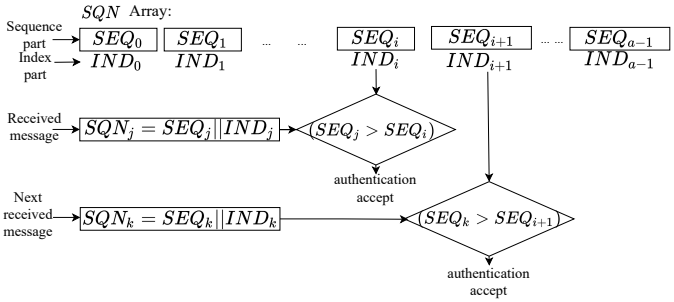



**Fig. 5:** Sequence number handling in USIM

**Vulnerability.** The `SQN` generation and verification for `authentication_request` message is performed through a complex scheme defined in TS 33.102 [24] Annex C. The `SQN` is divided into two concatenated parts SQN = SEQ||IND, namely the *sequence* (`SEQ`) and the *index* (`IND`). To generate a fresh `SQN`, the core network increments both `IND` and `SEQ`, concatenates them together and sends to the UE. For verification on the UE side, the USIM keeps track of a `SQN_array` of $a = 2^{\text{IND}}$ items, where each item is a `SQN` as shown in Figure 5. Whenever a new $\text{SQN}_\text{j} = \text{SEQ}_\text{j}||\text{IND}_\text{j}$ $(0 \leq \text{IND}_\text{j} \leq (\text{a}-1))$ in an `authentication_request` is received by the UE, the USIM in the UE looks up its `SQN_array`, checks if $\text{IND}_\text{i} == \text{IND}_\text{j}$, and based on that it retrieves the corresponding $\text{SEQ}_\text{i}$ value from the `SQN_array`. After comparing this saved $\text{SEQ}_\text{i}$ with the received $\text{SEQ}_\text{j}$, the UE either accepts or rejects the `SQN` number and the `authentication_request` message. In case $\text{SEQ}_\text{j} \leq \text{SEQ}_\text{i}$, the USIM generates an authentication synchronization failure message using the highest previously accepted `SQN` anywhere

in the SQN_array. Due to this design, the UE allows out-of-order SQN values.

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

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

Interestingly, in TS 33.102 [24] Annex C 2.2, there is a freshness limit ($L$) on the range of old accepted SQN. If the difference between the saved sequence part $SEQ_i$ at index $IND_i$ and the received sequence part, $SEQ_j$ is $SEQ_j - SEQ_i > L$, it will be rejected. However, the use of such a range is completely optional and the value is undefined for both 4G and 5G. The specification states – "*The use of such a limit is optional. The choice of a value for the parameter L affects only the USIM. It has no impact on the choice of other parameters and it is entirely up to the operator, depending on his security policy.* ***Therefore no particular value is suggested here***". Apparently, being optional and unspecified none of the major vendors are implementing such a check, paving the way to the acceptance of old sequence numbers.

**(P2) Linkability attack using `authentication_response`:** This attack can enable an adversary to track the presence of a user's device in a particular cell area violating the user's privacy by exploiting the different responses of authentication_request message. From the counterexample of the previous attack (**P1**), we identified that the UE accepts previously captured stale authentication_request with a SQN value smaller than the current accepted value. We utilize ProVerif's capability to reason about observational equivalence and pose the query: "*is it possible to distinguish two UE's based on their responses to an* authentication_request*?*", to identify this attack. The first phase of the attack to capture authentication_request is similar to **P1** (see Figure 4).

For the next step, the attacker with a malicious base station connects to all the UEs in a particular cell area and replays the captured authentication_request to all of them. The victim UE will accept this message and respond with authentication_response whereas all the other UEs in the cell will respond with MAC failure due to integrity check failure (see Figure 6) identifying and tracking the presence of the victim user. This attack is inspired by the linkability attack using auth_sync_failure shown in 3G [25] with the caveat that the distinction between different messages and a different vulnerability is utilized by the attacker for the two attacks. In this attack, the attacker differentiates between an out-of-order accepted authentication_request and a synchronization failure auth_sync_failure message to detect the presence of a user, whereas in the 3G attack two failure messages (auth_sync_failure and auth_MAC_failure) are utilized for the attack. The root cause of this attack is same as **P1**.

**Impact on 5G.** The generation and verification scheme of

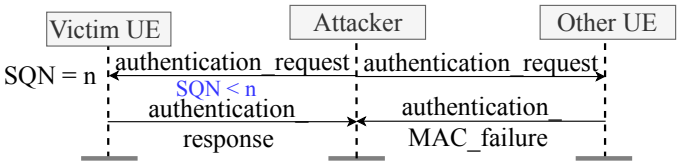



**Fig. 6:** Linkability using authentication_response

sequence number (SQN) in authentication_request message described in **P1** (Section VII-A) is exactly the same in the 5G specifications, thus making the 5G rollout directly vulnerable to **P1** and **P2** attacks.

**(P3) Selective security procedure denial:** In this attack, the adversary exploits the underspecification of the sequence number checking to prevent important security procedures entirely causing severe security and privacy issues.

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

relevant packets until the security procedure is abandoned by the MME (in most of the times it is tried 4-5 times). This forces the victim UE and the core network to keep using previous security contexts or the temporary identifier GUTI.

**Vulnerability.** Such kind of selective service denial attack is possible because of the underspecification of the standards. In TS 24.301 [19] it is specified that "*Replay protection must assure that one and the same NAS message is not accepted twice by the receiver. Specifically, for a given NAS security context, a given NAS COUNT value shall be accepted at most one time and only if message integrity verifies correctly.*" However, the case where an adversary is dropping packets surreptitiously is not handled in the specification. Due to the higher sequence number being the only satisfying condition and the inadequate check on the sequence numbers of two consecutive packets, it is possible to carry out this attack without detection.

**Impact.** The impact of this attack can be catastrophic as it affects multiple crucial security and privacy-preserving procedures. For instance, when the MME assigns a new GUTI to the UE with `GUTI_reallocation_command` message, the adversary can drop the message without being detected by the UE/MME and thus can induce both parties to use the same GUTI for a longer time than expected. The consequence of such packet dropping on the GUTI reallocation procedure is critical because of the following specification - TS 24.301 [19]: "*The GUTI reallocation procedure is supervised by the timer T3450. The network shall, on the first expiry of timer T3450, reset and restart timer T3450 and shall retransmit the* `GUTI_reallocation_command`*. This retransmission is repeated four times, i.e. on the fifth expiry of timer T3450, the network shall abort the reallocation procedure*". This implies that an adversary can surreptitiously drop five consecutive `GUTI_reallocation_command` messages and prevent the procedure entirely. After the five tries, the MME thus aborts the procedure and both MME and UE will keep using the previous GUTI. Since frequent updates of GUTI are mandated by the standard to prevent user tracking, this attack forces the GUTI reallocation to fail and thus enables the adversary to track the victim for a long period of time. Similar implications also apply to the *security mode command* procedure, where it is also possible to surreptitiously prevent the UE and MME from re-negotiating the keys. Such kind of selective procedure denial enables the adversary to force a device to reuse the same GUTI or session keys for an elongated time period and thus to track the victim device easily.

**Impact on 5G.** For the vulnerability and attack described here, the same procedures exist with the same design in 5G [27] as well, making it vulnerable to selective security procedure denial attack. Moreover, 5G introduces some new procedures which are also vulnerable to this attack. For instance, in TS 24.501 [27] the *5G Configuration Update Procedure* it is stated–"*The network shall, on the first expiry of the timer T3555, retransmit the* `configuration_update_command` *message and shall reset and start timer T3555. This retransmission is repeated four times, i.e. on the fifth expiry of timer T3555, the procedure shall be aborted*" making it possible to drop five messages, deny the procedure entirely and force the use of the same 5G-GUTI. Similar to this attack on 4G LTE, the adversary exploiting the vulnerability in the 5G network can track the user for long periods of time.

**(I1) Broken replay protection with all protected messages:** As discussed in the previous attack (P3), the UE should never accept any replayed packet after the security context has been established. We, however, found that both srsUE [15] and OAI [16] implementations allow the adversary to replay packets. We tested the FSMs of these implementations with the replay protection property and observed that OAI accepts only the last message when replayed, whereas srsUE accepts any replayed messages and resets the downlink counter with the counter value given in the replayed packet.

**(I2) Broken integrity, confidentiality with all protected messages:** The standard [19] also specifies a primitive property that a UE must not accept any plain-text messages after the security context is established. However, while testing the FSM of OAI with this property, we found a counterexample where the UE accepts plain-text messages with *plain-NAS (0x0)* as the packet header after the security context is established. We validate the counterexample in the testbed and indeed the OAI implementation accepts all security-protected messages in plain-text and un-cyphered after establishing the security context, effectively breaking integrity and confidentially protection of the protocol implementation.

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

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

*2) Proving previous attacks:*

Along with detecting new logical issues, ProChecker is able to automatically identify 14 design-level logical vulnerabilities uncovered by previous works [2], [6], [25], [26] (see Table I). These vulnerabilities were previously identified through manual inspection or from models that were manually derived.

### *B. RQ2. Model Comparison*

To evaluate the expressiveness of ProChecker's automatically extracted models we compare the extracted model from

| Attack | Property Type | Implication | Vulnerability Type | srsLTE [15] | OAI [16] |
|---|---|---|---|---|---|
| New Attacks | | | | | |
| (P1) Service disruption using authentication_request | Security | Service disruption | Standards | ● | ● |
| (P2) Linkability using authentication_response (Inspired by [25]) | Privacy | Location privacy leakage | Standards | ● | ● |
| (P3) Selective service dropping | Security | Surreptitious service disruption | Standards | ● | ● |
| (I1) Broken replay protection with all protected messages | Security | Broken replay protection | Implementation | ● | ● |
| (I2) Broken integrity, confidentiality with all protected messages | Security-Privacy | Integrity, encryption broken | Implementation | ○ | ● |
| (I3) Counter-reset with replayed authentication_request | Security | Breaks replay protection | Implementation | ● | ○ |
| (I4) Security bypass with reject messages | Security | Security bypass | Implementation | ● | ○ |
| (I5) Privacy leakage with identity_request | Privacy | IMSI leaking | Implementation | ○ | ● |
| (I6) Linkability with security_mode_command | Privacy | Location tracking | Implementation | ● | ● |
| Previous Attacks | | | | | |
| Authentication sync. failure [2] | Security | Denial of Service | Standards | ● | ● |
| Stealthy kicking-off [2] | Security | Detaching victim surreptitiously | Standards | ● | ● |
| Panic attack [2] | Security | Creating artificial chaos | Standards | ● | ● |
| Linkability using TMSI_reallocation [26] | Privacy | Location privacy leak | Standards | – | – |
| Linkability using IMSI to GUTI using paging_request [25] | Privacy | Location privacy leak | Standards | ● | ● |
| Linkability using auth_sync_failure [25] | Privacy | Location privacy leak | Standards | ● | ● |
| Authentication relay [2] | Security-Privacy | DoS, location history poisoning | Standards | ● | ● |
| Numb Attack [2] | Security | Prolonged DoS, batter depletion | Standards | ● | ● |
| Downgrade using tracking_area_reject [6] | Security | DoS | Standards | – | – |
| Denial of all services [6] | Security | DoS | Standards | ● | ● |
| Paging hijacking [2] | Security | Stealthy DoS, panic | Standards | ● | ● |
| Detach/Downgrade [2] | Security | DoS, battery depletion | Standards | ● | ● |
| Service Denial [2] | Security | DoS | Standards | ● | ● |
| Linkability (GUTI/TMSI) [2] | Privacy | Location Tracking | Standards | ● | ● |

**TABLE I:** Summary of ProChecker's findings. ● yes, ○ no, – not implemented


(a) LTEInspector (b) ProChecker (i) Mapping transitions with new conditions

(a) LTEInspector (b) ProChecker (ii) Mapping transitions with new states

**Fig. 7:** Transition refinement between ProChecker and LTEInspector

the closed-source implementation to the closest available 4G LTE model from LTEInspector [2]. We compare the extracted model from the closed-source implementation because it implements all the procedures and has a complete conformance test suite. For the comparison we first introduce a notion of refinement for FSMs and use it to compare the models.

**Refinement.** Let, $\mathsf{M_1}^\mu = (\Sigma_1, \Gamma_1, \mathcal{S}_1, s_{0_1}, \mathcal{T}_1)$ and $\mathsf{M_2}^\mu = (\Sigma_2, \Gamma_2, \mathcal{S}_2, s_{0_2}, \mathcal{T}_2)$ be two protocol FSM's. We say that $\mathsf{M_2}^\mu$ is a refinement of $\mathsf{M_1}^\mu$ if the following properties hold: (1) The set of states $\mathcal{S}_1$ is a subset of the set of states $\mathcal{S}_2$. For this property to be true, for each state, $s \in \mathcal{S}_1$, there is an one-to-one mapping to a state $s' \in \mathcal{S}_2$. (2) The sets of conditions $\Sigma_2$ and actions $\Gamma_2$ are a strict supersets of $\Sigma_1$ and $\Gamma_1$ respectively, containing new constraints on the transitions. (3) The transitions in set $\mathcal{T}_1$ can be mapped to transitions in $\mathcal{T}_2$. For each transition $t_i = (s_{i_{in}}, s_{i_{out}}, \sigma_i, \gamma_i) \in \mathcal{T}_1$ there can be several cases: (i) $t_i$ can be directly mapped onto a transition in $\mathcal{T}_2$; (ii) $t_i$ can be mapped to a transition $t_j = (s_{i_{in}}, s_{i_{out}}, \sigma_j, \gamma_j) \in \mathcal{T}_2$, where $t_j$ has the same incoming and outgoing state as $t_i$. However the condition, $\sigma_j$ has the form $\sigma_i \wedge \phi$, where $\phi$ is a new condition defined in $\Sigma_2$, thus making the condition of $t_j$ stricter than $t_i$; (iii) $t_i$ can be mapped onto multiple transitions (based on the new states) in $\mathcal{T}_2$. The transition $t_i$ from $s_{i_{in}}$ to $s_{i_{out}}$ can go through multiple new intermediate states such as $s_{i_i}$, generating transitions of the form $t_{i1} = (s_{i_{in}}, s_{i_i}, \sigma_{i1}, \gamma_{i1})$ and $t_{i2} = (s_{i_i}, s_{i_{out}}, \sigma_{i2}, \gamma_{i2})$, which can be mapped onto transitions in $\mathcal{T}_2$. The mapped transitions in $\mathcal{T}_2$ contain all the previous conditions and actions on $t_i$ together with new conditions and actions defined on $t_{i1}$ and $t_{i2}$.

**Comparison of the models.** We now show that the model

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

**TABLE II:** Common properties of ProChecker and LTEInspector

of the closed-source implementation extracted by ProChecker ($\mathsf{Pro}^\mu$) is a refinement of the model of LTEInspector ($\mathsf{LTE}^\mu$). First, the majority of the states in the set $\mathcal{S}_{LTE}$ of $\mathsf{LTE}^\mu$ can be directly mapped onto the states in the set $\mathcal{S}_{Pro}$ of $\mathsf{Pro}^\mu$. States in $\mathcal{S}_{LTE}$ that do not have a direct mapping in $\mathcal{S}_{Pro}$ (such as *ue_registered* and *ue_deregistered*) can be mapped onto the set of sub-states of the respective states. This mapping from states to sub-states is done following the standards [19]. Specifically, due to the automated extraction of FSM by ProChecker, *it was possible to extract sub-states of several procedures*; manually generating such sub-states would be severely cumbersome. Second, the condition $\Sigma_{Pro}$ and action $\Gamma_{Pro}$ sets of $\mathsf{Pro}^\mu$ are strict supersets of $\Sigma_{LTE}$ and $\Gamma_{LTE}$ of $\mathsf{LTE}^\mu$, respectively. Furthermore, as *data and packet payload information* were also extracted in $\mathsf{Pro}^\mu$, *new constraints* (such as sequence numbers and back-off counters) and new actions are included in $\Sigma_{Pro}$ and $\Gamma_{Pro}$. Finally, some transitions follow one-to-one mapping between $\mathsf{Pro}^\mu$ and $\mathsf{LTE}^\mu$. Others can be mapped based on new states or new conditions following our definition of refinement. The transitions defining new conditions impose stricter constraints (using predicates) that are based on the data and packet payload. For instance, the transition $t_{LTE} \in$

$\mathcal{T}_{LTE}$ presents a change of state from *ue_register_initiated* to *ue_registered* for the condition `security_mode_command` and action `security_mode_complete`. In $t_{Pro} \in \mathcal{T}_{Pro}$, $t_{LTE}$ is mapped to $t_{Pro}$, which is the refined version of $t_{LTE}$ where both states and actions remain the same but the condition has the form `security_mode_command` and `ue_sequence_number=0`, and it is thus stricter. These example transitions for both LTEInspector and ProChecker are shown in Figure 7(i). The rest of the transitions in $\mathcal{T}_{LTE}$ can also be mapped onto transitions on $\mathcal{T}_{Pro}$ based on *new states*. To illustrate this, let us consider the transition from *ue_dereg_initiated* to *ue_deregistered* having the condition `detach_request` and action `detach_accept` in $\mathcal{T}_{LTE}$. In $\mathcal{T}_{Pro}$ a new intermediate state is introduced *ue_dereg_attach_needed* and the transition is broken into two, introducing new conditions (shown in Figure 7 (ii)). Pro$^{\mu}$ is, therefore, a refinement of LTE$^{\mu}$ and considers more procedures and critical aspects, including transitions based on data and packet payload.

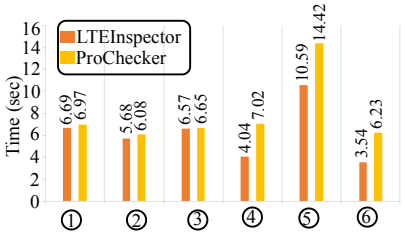



**Fig. 8:** Execution time of the common properties used in ProChecker and LTEInspector. Properties are numbered according to Table II.

### C. RQ3. Scalability

We take our largest and most detailed extracted model– from the closed-source implementation, and record the execution times for verifying the properties common to the LTEInspector model (see Table II). We used a laptop with an Intel i7-3750QCM CPU and 32 GB DDR3 RAM. The results in Figure 8 show that the time required by ProChecker for each property is only a fraction higher than LTEInspector. This result also signifies the scalability of our framework since it can run a COTS model checker on the automatically extracted model from an implementation with negligible overhead.

## VIII. Related Work

**Formal Verification of Cellular Networks and Other Protocols.** Approaches using formal verification either rely on manually extracted models from specifications [2], [3], [11], [28], require models in formally-verifiable languages [29], [30], or require a reference implementation of the protocol in a custom language [31], [32]. These approaches are not scalable for commercial protocol implementations [2], [3], [11], [28]–[32]. Model-checking has also been applied to verify properties of protocols [33]–[35]. But these approaches are unable to reason about properties that depend on protocol events. Execution-based model checking approaches [36], [37] do not require an explicit model but are prone to state-space explosion.

**FSM extraction.** There are approaches that infer protocol specifications as a model from traces [38], from network traces [39]–[41], or using program analysis [42]. However, the FSM's extracted through such approaches represent discernible external interactions of the protocol (e.g., the sequence of exchanged messages) and do not contain enough semantic meaning to reason about security and privacy properties. In a black-box setting active-learning [13] has been used to extract the FSM of a system. However, the extracted FSM does not have a proper indication of states and in our white-box setup, we have a lot more information to utilize. Symbolic execution has also been used to generate formally analyzable models of protocol implementations. Aizatulin et al. [43] combined symbolic execution with proof techniques for extracting a ProVerif model from implementations in C. However, their technique is limited to protocols without branching.

**Security of 4G LTE.** Kim et al. [9] design a stateless dynamic testing framework with pre-generated test cases and discovered several vulnerabilities. However, their tool is semi-automatic, requires manual analysis, and only reasons about some specific properties. Whereas ProChecker can automatically extract the FSM, reason about any security and privacy property, and easily scales to the future generation and new releases of cellular network implementations such as 5G. Rupprecht et al. [44] found that missing integrity allows the redirection to a malicious website by an active Man-in-the-Middle attack. In the area of identifying logical flaws in implementation or specification, previous approaches use manual inspection [4]–[8], [25], [26] or use manually extracted models [2].

## IX. Discussion and Limitations

**Completeness of our model.** Model completeness depends on the coverage of test suites. In our experiments, we managed to extract state machines detailed enough to reason about protocol aspects critical for security. For commercial 4G LTE implementations, having a complete conformance testing suite is a must; therefore this automatically corresponds to high coverage. We also showed that in the case of open-source implementations, it is possible to add some procedure-specific test cases and extract a formal model. Even though ProChecker's ability to extract details is limited by the coverage of the test suite, its true strength lies in its efficiency. For a given test environment, it adds negligible resource overhead to extract a state machine. As the test suite grows in coverage, ProChecker can generate increasingly detailed FSMs.

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

**FSM for both communicating parties.** Since we did not have access to the source code for the core network, we had to use an open-source standard model derived by the research community. For protocols, such as Wi-Fi and Bluetooth, where

both communicating parties of the protocol are implemented by the same vendor, the same methodology can be applied.

**ProChecker for 5G implementations.** The design requirements of ProChecker for analyzing 5G implementations are similar to that of 4G (i.e., properly defined protocols states, protocol message names [27], conformance test case suite [45]). Therefore, this framework can easily be adapted to evaluate any 5G implementations. More precisely, since ProChecker works with a very minimal overhead with the existing testing infrastructure, it can be easily adapted to verify the security and privacy properties of the 5G protocol implementations from the get-go.

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

**Threat to validity.** We used automatically extracted models for UE FSMs. Due to the low coverage of test suites and manually constructed MME FSMs extracted from the 3GPP standard, the counterexamples derived from the model may not completely reflect the behavior of real operational networks. Thus inaccuracies in the model may induce false positives in commercial networks, although, we have not observed any such behavior. Due to ethical considerations, we validate the attacks in a custom-built network, which may not faithfully capture the operation network behavior.

## X. Conclusion and Future Work

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

## Acknowledgement

We thank the anonymous reviewers for their feedback and suggestions. We also thank GSMA for supporting the CVD process and the developers of open-source implementations (srsLTE and OAI) for their efforts in implementing and maintaining all the tested open-source implementations.

## References

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

[2] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino, "Lteinspector: A systematic approach for adversarial testing of 4g lte," in *Network and Distributed Systems Security (NDSS) Symposium*, 2018.

[3] S. R. Hussain, M. Echeverria, I. Karim, O. Chowdhury, and E. Bertino, "5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol," in *CCS '19*.

[4] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper, "Call me maybe: Eavesdropping encrypted LTE calls with revolte," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[5] M. Chlosta, D. Rupprecht, T. Holz, and C. Pöpper, "Lte security disabled: Misconfiguration in commercial networks," in *WiSec '19*, 2019.

[6] A. Shaik, R. Borgaonkar, N. Asokan, V. Niemi, and J.-P. Seifert, "Practical attacks against privacy and availability in 4g/lte mobile communication systems," 2015.

[7] C.-Y. Li, G.-H. Tu, C. Peng, Z. Yuan, Y. Li, S. Lu, and X. Wang, "Insecurity of voice solution volte in lte mobile networks," in *CCS '15*, 2015.

[8] H. Kim, D. Kim, M. Kwon, H. Han, Y. Jang, D. Han, T. Kim, and Y. Kim, "Breaking and fixing volte: Exploiting hidden data channels and mis-implementations," in *CCS '15*.

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

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

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

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

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

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

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

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

[17] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 334–342.

[18] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, June 2001, pp. 82–96.

[19] *3GPP. Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3 Specification 3GPP TS 24.301 version 12.8.0 Release 12.*, [Online]. Available: http://www.3gpp.org/dynareport/24301.htm.

[20] *GSMA Mobile Security Hall of Fame*, https://www.gsma.com/security/gsma-mobile-security-hall-of-fame/.

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

[22] *3GPP Conformance testing*, https://www.3gpp.org/technologies/keywords-acronyms/108-conformance-testing-ue.

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

[24] *Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); 3G security; Security architecture (3GPP TS 33.102 version 11.5.1 Release 11)*, .

[25] M. Arapinis, L. Mancini, E. Ritter, M. Ryan, N. Golde, K. Redon, and R. Borgaonkar, "New privacy issues in mobile telephony: Fix and verification," in *CCS '12*.

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

[27] *5G; Non-Access-Stratum (NAS) protocol for 5G System (5GS); Stage 3 (3GPP TS 24.501 version 15.0.0 Release 15) )*, https://www.etsi.org/deliver/etsi_ts/124500_124599/124501/15.00.00_60/ts_124501v150000p.pdf.

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

[29] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: Proving practical distributed systems," in *SOSP '15*.

[30] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *PLDI '15*.

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

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

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

[34] M. Canini, D. Venzano, P. Perešíni, D. Kostiundefined, and J. Rexford, "A nice way to test openflow applications," in *NSDI'12*.

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

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

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

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

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

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

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

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

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

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

[45] *5G; User Equipment (UE) conformance specification; Part 1: Protocol (3GPP TS 38.523-1 version 15.3.0 Release 15).*, http://www.3gpp.org/ftp//Specs/archive/32_series/32.899/32899-f10.zip.