State Machine Mutation-based Testing Framework for Wireless Communication Protocols

Syed Md Mukit Rashid
The Pennsylvania State University
University Park, PA, United States
szr5848@psu.edu

Tianwei Wu
The Pennsylvania State University
University Park, PA, United States
tvw5452@psu.edu

Kai Tu
The Pennsylvania State University
University Park, PA, United States
kjt5562@psu.edu

Abdullah Al Ishtiaq
The Pennsylvania State University
University Park, PA, United States
abdullah.ishtiaq@psu.edu

Ridwanul Hasan Tanvir
The Pennsylvania State University
University Park, PA, United States
rpt5409@psu.edu

Yilu Dong
The Pennsylvania State University
University Park, PA, United States
yiludong@psu.edu

Omar Chowdhury
Stony Brook University
Stony Brook, NY, United States
omar@cs.stonybrook.edu

Syed Rafiul Hussain
The Pennsylvania State University
University Park, PA, United States
hussain1@psu.edu

ABSTRACT

This paper proposes `Proteus`, a protocol state machine, property-guided, and budget-aware automated testing approach for discovering logical vulnerabilities in wireless protocol implementations. `Proteus` maintains its budget awareness by generating test cases (*i.e.*, each being a sequence of protocol messages) that are not only *meaningful* (*i.e.*, the test case mostly follows the desirable protocol flow except for some controlled deviations) but also have a high probability of violating the desirable properties. To demonstrate its effectiveness, we evaluated `Proteus` in two different protocol implementations, namely 4G LTE and BLE, across 23 consumer devices (11 for 4G LTE and 12 for BLE). `Proteus` discovered 25 unique issues, including 112 instances. Affected vendors have positively acknowledged 14 vulnerabilities through 5 CVEs.

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**.

KEYWORDS

Property Guided Testing, Finite State Machine, 4G LTE, Bluetooth

ACM Reference Format:
Syed Md Mukit Rashid, Tianwei Wu, Kai Tu, Abdullah Al Ishtiaq, Ridwanul Hasan Tanvir, Yilu Dong, Omar Chowdhury, and Syed Rafiul Hussain. 2024. State Machine Mutation-based Testing Framework for Wireless Communication Protocols. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3658644.3690312

1 INTRODUCTION

Due to the pervasiveness and broadcast-based over-the-air (OTA) communication, wireless protocols (*e.g.*, BLE, LTE, Wi-Fi) are attractive targets for attackers, especially because vulnerabilities in them can be used as a stepping stone by adversaries to launch attacks against applications relying on them. Software testing has proven to be the most effective and dominant approach for ensuring the correctness of wireless protocol implementation by uncovering bugs in the pre-deployment stages. This paper focuses on designing, developing and evaluating an automated testing approach called `Proteus` for uncovering *logical/semantic vulnerabilities* in wireless protocol implementations [? ? ? ? ? ? ?]. The class of logical bugs we focus on is only those that induce the implementation to deviate from the intended design *without* necessarily causing a crash. Such logical bugs are not only challenging to discover but also often have severe security and privacy implications (e.g., authentication bypass) and go undiscovered during pre-deployment stage testing. These classes of bugs are the focus of this paper.

Any testing approaches focusing on discovering such logical bugs in commercial-off-the-shelf (COTS) wireless protocol implementations must be aware of the following salient aspects. ① COTS implementations are typically closed-source and expose only their input-output interfaces, making them only amenable to black-box testing. ② Wireless protocols are often stateful. Consequently, triggering a vulnerability amounts to injecting the right protocol packet type and payload *only when* the protocol is in a specific internal protocol state. In addition, driving the implementation to the bug-triggering state may require a long sequence of protocol packets. ③ When executing a test case, the target device's protocol under test has to be in the initial state. After running each test case, the device thus has to be reset. This results in a very high and fixed amortized cost of running a single test case over-the-air (e.g., ~1.5 minute for BLE), limiting the number of test cases executable within a given time *budget.* Given a fixed testing budget, it is therefore crucial that the test cases used are not wasted and indeed meaningful, that is, they have a high probability of exercising the bug-inducing behavior in the implementations.

Existing research efforts that specialize in black-box testing of wireless communication protocol implementations can be categorized into the following high-level categories: **(A)** *Manual analysis or fixed test case-based approaches* [? ? ? ?]; **(B)** *Reverse engineering-based approaches* [? ? ? ? ? ?]; **(C)** *State machine learning-based approaches* [? ? ? ? ?]. Approaches in categories (A) and (B) are either unscalable due to manual effort or ineffective in identifying intricate bugs in complex and stateful protocols that require long execution packet traces to be exercised. Category (C) approaches can address both aspects ① and ②, but fail to address ③. Among the category (C) approaches the ones that rely on automata learning [? ?] need to run a large number of over-the-air (OTA) queries to obtain the protocol state machine before testing can commence, hence their testing-budget agnosticism.

Furthermore, many of the above approaches rely on *differential testing*, in which diverse implementations-under-test (IUT) are used as cross-checking *test oracles* to find logical vulnerabilities. However, these oracles are inherently unfaithful as test oracles because they all can suffer from the same logical vulnerability. Finally, most fuzzing techniques [? ? ?] only *implicitly* consider a testing budget. They aim to effectively use the given testing budget in terms of the number of vulnerabilities discovered by attempting to minimize the execution time of each test case while also taking guidance from some rich forms of coverage information (e.g., code coverage). However, in a testing setup like ours, where the cost of running each test case cannot be substantially minimized and the availability of coverage information is limited, the existing philosophy of decreasing the execution time of each test case cannot be effectively adopted. *In summary, none of the current approaches suit our testing setup and thus warrant a new testing philosophy.*

We address the above limitations by designing an adaptive testing-budget-aware, stateful, black box testing approach called `Proteus` for COTS wireless protocol implementations. Concretely, `Proteus`'s testing philosophy takes advantage of the following three observations. ❶ Since COTS devices generally pass through a quality assurance stage where they are tested for conformance and interoperability, undiscovered bugs are more likely to occur when implementations subtly deviate from rare but good protocol flows. As such, a test case is *meaningful* if it *mostly* adheres to a desired protocol message sequence (*e.g.*, sending protocol messages only after the initial connection request message), except for some controlled perturbations. ❷ Any testing-budget-aware approach will be effective only if it reduces test case wastage by generating *only* meaningful test cases that have a high probability of triggering logical vulnerabilities within the given budget. ❸ The number of possible meaningful test cases depends on the amount of perturbation and the maximum length we allow. We can adapt an approach to maximize vulnerability discovery within a given testing budget if they can be explicitly controlled.

`Proteus` takes three inputs for generating test cases, namely, a guiding protocol state machine $\mathcal{M}$, a set of desirable security and privacy properties Φ expressed in past-time linear temporal logic formulae, and a testing budget β. `Proteus` relies on $\mathcal{M}$, either derived from the standard [? ?] or extracted from an implementation [? ?], to ensure that the generated test cases are indeed meaningful. `Proteus` generates test cases through controlled perturbation over $\mathcal{M}$, thus *mostly* capturing good protocol behavior (realization of observation ❶). Similarly, any property φ of the protocol under test guides `Proteus` to only generate test cases likely to trigger violation of φ (realization of observation ❷). The set of properties Φ serves not only as a faithful test oracle but also liberates `Proteus` from needing to have access to diverse implementations of the same protocol to test a single protocol implementation. Furthermore, the number of test cases to be generated by `Proteus` is *explicitly* controlled by the testing budget parameter β. It achieves budget-awareness by controlling the maximum size of a test case (*i.e.*, the length of the message sequence) and the number of perturbations to be applied in a good protocol flow for generating test cases (realization of observation ❸). In addition, as discussed in ❶, the number of mutations one needs to consider (given by β) need not be large. This observation is corroborated by our findings, where most vulnerabilities are discovered by considering only **2** mutations of $\mathcal{M}$.

Conceptually, `Proteus`' design is inspired by mutation-based testing, where the guiding protocol state machine $\mathcal{M}$ is mutated to obtain another state machine $\mathcal{M}^*$ such that $\mathcal{M}^*$ violates φ. If the implementation under test is equivalent to $\mathcal{M}^*$, we can automatically obtain the vulnerability's root cause by tracking the mutation applied to $\mathcal{M}$ for obtaining $\mathcal{M}^*$. `Proteus` realizes this conceptual design through a novel three-stage approach. First, `Proteus` uses a novel algorithm to automatically synthesize test case templates that are guaranteed to violate φ. Second, `Proteus` instantiates these test case templates using a dynamic programming algorithm for the given $\mathcal{M}$, φ and β (*i.e.*, the maximum length of the message sequence and the number of allowed mutations from $\mathcal{M}$). Finally, `Proteus` efficiently schedules, concretizes, and executes the instantiated test cases OTA. `Proteus` analyzes the output generated by the implementation to discover logical vulnerabilities.

To demonstrate the efficacy of `Proteus`, we have tested several protocol implementations of two popular wireless communication protocols: 4G LTE cellular network and Bluetooth Low Energy (BLE). For 4G LTE, we have tested 11 devices and identified 10 issues, including 3 new ones. For BLE, we have tested 12 devices and identified 15 issues, including 7 new ones.

In summary, this paper makes the following contributions.

- We developed `Proteus`, which is an efficient, a black box, protocol state machine and property-guided, budget-aware automated testing approach for discovering logical vulnerabilities in wireless protocol implementations.
- We developed two novel algorithms that cooperatively generate meaningful test case templates that are likely to violate a given set of security properties under the guidance of a state machine.
- We developed an effective dispatcher that efficiently schedules test cases for maximizing the number of property violations within the allocated testing budget, issues OTA test cases to the devices and analyzes the output to find logical vulnerabilities.
- We evaluated `Proteus` on LTE and BLE to determine its efficacy. It identified 3 new issues in 11 LTE implementations and 7 new issues testing on 12 BLE implementations.

Responsible disclosure. We have responsibly reported all of our new findings to the affected vendors. For BLE, the vendors acknowledged 11 issues and assigned 3 CVEs; for LTE, the vendors acknowledged 3 issues with 2 CVEs. The source code of `Proteus` is available at [?].

2 PRELIMINARIES AND NOTATIONS

Protocol state machine (PSM). A protocol model or state machine (PSM) $\mathcal{M}$ is a tuple $\langle Q, \Sigma, \Lambda, q_{init}, \mathcal{R}\rangle$, in which Q is a non-empty set of states, Σ is the non-empty finite set of *input* symbols, Λ is the non-empty finite set of *output* symbols, $q_{init} \subseteq Q$ is the initial state, and $\mathcal{R}$ is the transition relation $\mathcal{R} \subseteq Q \times \Sigma \times \Lambda \times Q$. Given a transition $\langle q_1, \alpha, \gamma, q_2\rangle \in \mathcal{R}$, it signifies that if the protocol is in state $q_1 \in Q$ and receives the input symbol α, then it will transition to state q_2 and will generate the output symbol γ. We consider this reception of input α and generation of output γ as the *observation* of the transition from q_1 to q_2.

The input and output alphabets Σ and Λ in our description are intentionally left to be abstract. In our context, Σ can be viewed as the cross-product of all the input message types and predicates over the input message fields. Similarly, Λ can be defined as the cross-product of all the output message types and predicates over them. For example, the input symbol SECURITY_MODE_COMMAND: INTEGRITY == 1 *&* EIA == 1 *&* SECURITY_HEADER == 3 denotes the SECURITY_MODE_COMMAND message, with the predicate INTEGRITY == 0 denoting the message is not integrity protected, SECURITY_HEADER == 3 denoting the security header field value of the message is set to 3, and EIA == 1 denoting the EIA algorithm field is set to 1.

Trace. A trace π is a sequence of the form $[\alpha_1/\gamma_1, \alpha_2/\gamma_2, \ldots, \alpha_k/\gamma_k]$ where $\alpha_i \in \Sigma$ and $\gamma_i \in \Lambda$. In our context, π signifies a protocol execution in which the protocol implementation is fed the input symbols $[\alpha_1, \alpha_2, \ldots, \alpha_k]$, and it generates the output symbols $[\gamma_1, \gamma_2, \ldots, \gamma_k]$. For instance, consider the trace π^* in BLE [SCAN_REQ/ SCAN_RESP, CON_REQ/ NULL_ACTION, VERSION_REQ: EXT_LL==0/ VERSION_RESP, PAIR_REQ: SC==1/ PAIR_RESP, KEY_EXCHANGE / KEY_ RESPONSE]. In π^*, SCAN_REQ has been first sent to the IUT that responds with SCAN_RESP. Then, CONNECTION_ REQUEST is sent, to which the response is NULL_ACTION (i.e., the IUT does not generate an output). After that, the response to VERSION_REQUEST:EXT_LL == 0 is VERSION_REQUEST:EXT_LL == 0, and so on. The length of trace π is denoted by $|\pi|$. The i^{th} element of π is denoted with π_i (where $0 \leq i < |\pi|$). "$\cdot$" represents the concatenation of two traces. As an example, $[\alpha/\gamma] \cdot \pi^1$ denotes a trace obtained by prepending the trace with a single observation α/γ to trace π^1.

3 MOTIVATION OF PROTEUS

We first review the unique challenges of testing COTS wireless protocol implementations and then use a running example to motivate `Proteus`'s design choices.

Closed-source implementations. As COTS wireless devices tend to be closed-source, any testing approach relying on some level of access to the source code (white box or gray box) is inapplicable, warranting a black box testing approach which `Proteus` follows.

Need for a faithful test oracle. Crashing behavior can serve as a protocol-agnostic universal symptom for memory-related bugs. In contrast, semantic or logical bugs require a faithful test oracle to accurately adjudicate the correct behavior for a given test case. `Proteus` uses a set of security properties collected from RFCs/specifications as faithful oracles. If a property is violated, `Proteus` directly concludes that it is a logical vulnerability.

Protocol statefulness. Statefulness of wireless protocols introduces the following challenges: (1) Analysis mechanisms must be aware of the underlying protocol state. (2) The protocol behavior depends on both the current state and the current protocol packet. (3) Triggering a bug may require driving the target protocol implementation in a particular protocol state. This may require sending a long sequence of messages to the IUT (to reach the bug-triggering state) before we can inject the bug-triggering packet. (4) Due to 2 and 3, the universe of test cases is potentially infinite. A naive sampling of this infinite test case universe is unlikely to be effective for a given testing budget. `Proteus`, therefore, uses guidance from both the PSM and properties to efficiently sample meaningful test cases that will likely trigger a vulnerability.

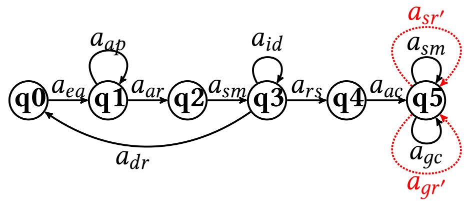


Figure 1: A partial LTE protocol state machine used as guiding PSM for running example. The transition labels are presented in Table 1, with red transitions indicating mutations.

High cost of running a test case. Because of protocol statefulness, any previous test case may drive the protocol to an unknown state for which the test oracle does not know the correct behavior. If a protocol does not have a known *homing sequence* [?], one has to reset the device's state machine either using a sequence of OTA messages or reboot the device. Also, after sending each protocol message in a test case, the tester has to wait a certain time to check whether the message induces a device crash. The resets and device responsiveness checks substantially increase the amortized cost of executing a single test case. As an example, it takes up to ~1.5 minutes in BLE and ~1 minute in LTE to run a test case containing ~6 − 8 messages. The high execution cost significantly slows the overall testing speed and allows testing only a limited number of test cases within a given time budget. Failing to explore more protocol behavior within a specific time results in fewer bug detections.

Testing budget awareness. Traditional fuzzing approaches for general-purpose systems *implicitly* consider a *testing budget.* They try to accelerate the discovery of security vulnerabilities through different mechanisms such as enhancing code coverage [? ? ?], optimizing the search space of test messages [?], adopting efficient scheduling mechanisms [?], increasing testing speed [?], or improving discovery of execution paths [?]. However, these approaches have no explicit mechanism to adapt their test case generation according to the available testing budget. In contrast, `Proteus` *explicitly* respects the testing budget. It leverages a guiding PSM and input properties and explicitly controls the mutation amount and length of the test cases to generate meaningful test cases that can be executed within the available time budget.

3.1 Running Example

To justify the approach taken by `Proteus`, we consider a running example using 4G LTE.

Guiding PSM. The partial PSM used in this example, as shown in Figure 1, is extracted from the LTE NAS layer protocol specification. Table 1 explains the transition labels of the guiding PSM. Some errors are introduced intentionally (red transitions) in guiding PSM.

Table 1: Transition labels (*i.e.*, input and output symbols) in the guiding PSM.

Symbol	Description
a_{ea}	ENABLE_S1 / ATTACH_REQUEST
a_{ar}	AUTHENTICATION_REQUEST / AUTHENTICATION_RESPONSE
a_{sm}	SECURITY_MODE_COMMAND / SECURITY_MODE_COMPLETE
a_{id}	IDENTITY_REQUEST: INTEGRITY == 1 & IDENTITY_TYPE == 1/ IDENTITY_RESPONSE
a_{dr}	DETACH_REQUEST / DETACH_ACCEPT
a_{rs}	RRC_SECURITY_MODE_COMMAND / RRC_SECURITY_MODE_COMPLETE
a_{ac}	ATTACH_ACCEPT / ATTACH_COMPLETE
a_{gc}	GUTI_REALLOCATION_COMMAND / GUTI_REALLOCATION_COMPLETE
$a_{gr'}$	GUTI_REALLOCATION_COMMAND: REPLAY == 1 / GUTI_REALLOCATION_COMPLETE
$a_{sr'}$	SECURITY_MODE_COMMAND: REPLAY == 1 / SECURITY_MODE_COMPLETE
a_{ap}	ATTACH_ACCEPT: INTEGRITY == 0 & CIPHER == 0 & SECURITY_ HEADER _TYPE == 0/ NULL_ACTION
$a_{ap'}$	ATTACH_ACCEPT: INTEGRITY == 0 & CIPHER == 0 & SECURITY_ HEADER _TYPE == 4/ ATTACH_COMPLETE

Table 2: Sequences (test traces) considered for the running example in Figure 1.

ID	Sequence
S0	$a_{ea}, a_{ar}, a_{sm}, a_{id}$
S1	$a_{ea}, a_{ar}, a_{sm}, \boldsymbol{a_{id'}}$
S2	$a_{ea}, a_{ar}, a_{sm}, \boldsymbol{a_{dr}}, a_{ac}, a_{gc}, a_{gr'}$
S3	$a_{ea}, a_{ar}, a_{sm}, a_{rs}, a_{ac}, a_{gc}, \boldsymbol{a_{sr'}}, \boldsymbol{a_{gr'}}$

Desirable property. The security property of interest in this example is the following. ϕ_g : "*After successfully completing the attach procedure, a replayed GUTI Reallocation Command message should not be accepted.*" This property prevents an attacker from performing linkability attacks by violating the freshness of a device's ephemeral identity (*i.e.*, GUTI). Violating this property enables an adversary to launch an attack in which they send a previously captured GUTI_REALLOCATION_COMMAND message intended for the victim to all devices in a cell. If the victim device violates the above property and is present in that cell, it will respond positively, whereas others will just respond with a rejection. In this way, the adversary can test the presence of a victim in a cell.

3.2 Benefit of Having PSM and Properties

Before explaining the advantage of having access to both a guiding PSM and desirable security properties for generating meaningful test cases within a testing budget, we first explain why any approach having access to just one of these is unlikely to be as effective.

Why is guiding PSM not enough? Consider a PSM-guided testing approach that mutates a trace adhering to the protocol flow to generate a test case. However, this approach does not guarantee that the mutated test case will violate ϕ_g; wasting a test case. As an example, consider a good protocol flow sampled from the PSM denoted as **S0** in Table 2. Suppose we mutate **S0** by adding a mutation on the IDENTITY_REQUEST message to generate the test case **S1** in Table 2. **S1** clearly does not violate the property as it does not even include the GUTI_REALLOCATION_COMMAND message, let alone its replay.

Why are properties not enough? Access to properties, however, ensures that such blatantly vacuous test cases like **S1** are not generated. Considering the property ϕ_g, one can automatically generate *test skeletons* that are guaranteed to violate it. One such test skeleton or template (expressed as a regular expression for ease of exposition) is $\sigma_g = (.)^* \mathrm{a}_{ar} (.)^* \mathrm{a}_{sm} (.)^* \mathrm{a}_{ac} (.)^* \mathrm{a}_{gc} (.)^* \mathrm{a}_{gr'}$ where wildcard characters signify that they can be replaced with 0 or more occurrences of any input symbols, and still result in a ϕ_g-violating test. σ_g captures the necessary messages to violate ϕ_g, with wildcard characters allowing other protocol messages. Suppose we instantiate σ_g by replacing wildcard characters with arbitrary concrete symbols. Without any knowledge of a typical good protocol flow, we may obtain instantiated test case **S2** (see Table 2) which violates ϕ_g. However, any practical protocol implementation would most likely reject this trace since a_{dr} will discard the NAS security context and the subsequent messages of **S2** will be dropped. Also, a successful RRC security context a_{rs} is not performed, preventing the completion of the attach procedure. Such blind instantiations of wildcard characters to generate a test case will likely result in meaningless test cases that the IUT will reject, wasting test cases.

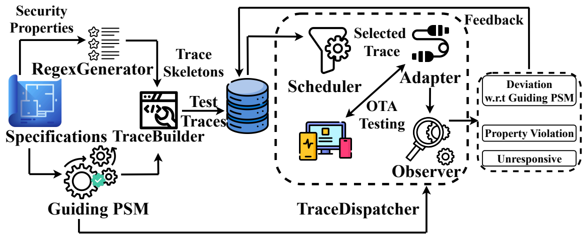


Figure 2: Overview of `Proteus`.

Advantage of `Proteus`' approach. `Proteus` takes advantage of the strengths of both PSM-guided and property-guided approaches. The main insight `Proteus` uses is that after a trace skeleton like σ_g is created, it does not blindly instantiate the wildcard characters and instead relies on the PSM for guidance on instantiating these free choices. When instantiating these open choices, it does not necessarily follow the PSM exactly but also includes controlled deviations. For example, using the PSM, `Proteus` would instantiate σ_g and generate a test case **S3**, which not only violates ϕ_g but also have a higher chance of being accepted by a buggy implementation as it closely follows a good protocol flow.

4 DESIGN OVERVIEW AND CHALLENGES

4.1 Proteus Overview

Given a set of desired properties $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$, a (potentially, standard-prescribed) guiding PSM $\mathcal{M}$ of a protocol P, a testing budget β, and an implementation $\mathcal{I}_P$ of the protocol P under test (i.e., IUT), `Proteus` aims to identify execution traces π^c of $\mathcal{I}_P$ such that it falsifies one or more desired properties $\phi_i \in \Phi$ while receiving guidance from $\mathcal{M}$ and Φ in time proportional to β. β is of the form $\langle \lambda, \mu \rangle$ in which λ denotes the maximum length budget (*i.e.*, the number of input symbols in a test case) and μ represents the mutation budget (*i.e.*, the maximum number of places a test case can deviate from a good protocol flow). Figure 2 presents the high-level overview of `Proteus`. The full pseudocode of `Proteus`'s test generation is presented in Algorithm 1 in Appendix. Conceptually, for each property $\phi_i \in \Phi$, `Proteus` carries out the following steps.

(1) Test skeleton generation. `Proteus` uses its RegExGenerator to generate a test skeleton denoted as a regular expression (RE) for ϕ_i. This test skeleton is an abstract execution/trace π^a of the protocol under test guaranteed to violate ϕ_i. Some positions of π^a have specific input symbols, whereas others have wildcard characters.

One can instantiate wildcards with input symbols and still violate ϕ_i. Based on β, Proteus will generate multiple such test skeletons.

(2) Instantiating test skeletons. In this step, Proteus instantiates the abstract trace π^a, especially in unrestricted positions (represented with wildcard characters) with specific input symbols under the guidance of $\mathcal{M}$ while respecting the budget β. Based on β, Proteus will generate multiple instantiated test cases π^c, containing input symbols in all positions with no wildcard characters, and store them in *TestTraceSet.* Guidance from $\mathcal{M}$ is crucial to ensure that any generated instantiated trace is indeed meaningful.

(3) Dispatching test cases. Proteus then takes each instantiated π^c, schedules it, and then prepares OTA protocol packets for each input symbol. It then sends the resulting concrete protocol packets contained in this concretized test case OTA to $\mathcal{I}_P$.

(4) Vulnerability detection and reporting. If $\mathcal{I}_P$ accepts the concretized test case executed OTA, then it signifies a vulnerability. In such a case, the test case is stored, and a vulnerability is reported.

4.2 Challenges and Insights

Realizing Proteus' approach requires tackling the following challenges. We address the challenges with the outlined novel insights.

❏ **Challenge C1: Synthesizing test skeletons from properties.** Proteus takes the desired properties as past linear temporal logic (PLTL) formulae. We, therefore, first require to generate test skeletons that are guaranteed to violate a property. Also, we need a succinct representation for test skeletons.

Insight I1. We use regular expressions (REs) as our succinct intermediate representation of a property-violating test skeleton. RE is not only expressive enough to capture the temporal ordering in a PLTL formula but also can represent logical dependencies. This leads to the following research question: *how does one generate π^a in regular expression format from a PLTL formula automatically?*

To answer this, we observe that a PLTL expression contains logical (e.g., AND operator) and temporal (e.g., SINCE operator) operators. Logical operators of the PLTL formula express constraints over its operands for any π^a expressing the violation of the PLTL expression ϕ. In contrast, temporal operators express constraints over their operands in a range of positions in π^a. An abstract syntax tree (AST) effectively expresses the relation between an operator and its operands, and its leaves are propositions required to obtain the input symbol or wildcard characters in π^a. Proteus parses the AST of the PLTL formula and leverages the PLTL operator semantics to non-deterministically generate a trace skeleton π^a.

❏ **Challenge C2: Instantiating abstract test skeletons using guiding PSM.** The next challenge is to instantiate the abstract test skeleton π^a to a test case π^c while taking guidance from the PSM. We assume that the guiding PSM satisfies all security properties. In contrast, property-violating traces will likely be present when implementations subtly deviate from the standard. As such, we opt to perform *mutations* on the guiding PSM and generate traces to instantiate the abstract test skeleton.

However, if we perform too many mutations on the guiding PSM $\mathcal{M}$, we will generate traces that substantially deviate from the standard, which are unlikely to trigger any vulnerabilities. In contrast, if we perform too few mutations, we may miss some vulnerabilities that require more mutations to identify (e.g., generating a test trace that detects the GUTI reallocation replay attack in LTE requires two mutations). Thus, we need to vary the amount of mutation μ within a range of values.

Similarly, generating arbitrarily long traces negatively impacts testing. Long traces likely repeat the same states and transitions, making it unlikely to uncover new vulnerabilities. Even worse, excessively long traces takes significantly more time to test OTA, ultimately wasting our testing budget. In contrast, we need our test traces to be at least as long as the number of literals in the trace skeleton π^a to generate a trace that satisfies π^a. We also require a slightly longer trace to visit extra states and uncover potential property violations. Thus, similar to mutations, we need to vary the length of the generated traces λ within a range of values.

Finally, for a given test skeleton π^a, the challenge is to design an automatic approach to generate instantiated traces π^c that would (1) satisfy the abstract test skeleton π^a, (2) align with the guiding PSM $\mathcal{I}_P$, (3) ensure that each generated instantiated trace π^c require at most μ mutations on $\mathcal{M}$, and (4) maintain a maximum length of λ. For example, consider the guiding PSM in Figure 1 and the property ϕ_g provided in Section 3. A trace skeleton that violates ϕ_g is σ_g, and **S3** (shown in Table 2) is a test trace that satisfies σ_g. **S3** aligns with $\mathcal{M}$ within 2 mutations (i.e., it would satisfy any requirement of $\mu >= 2$). The mutations are marked bold in Table 2. Also, **S3** is of length 8 and thus satisfies any length requirement of $\lambda >= 8$.

Insight I2. We observe that we can solve the problem of generating test cases described above by combining *overlapping subproblems* originating from destination states q_n ($q_n \in \mathcal{M}$) of the outgoing transitions of a particular state q_c. The solution of the subproblems will produce traces by using mutations wherever necessary to satisfy some suffix of π^a starting from state q_n. By prepending an observation (depending on π^a) to these traces, we can generate traces originating from q_c that satisfy π^a within the specified mutation and length budget. For example, in Figure 1 if we have that trace $\pi_k = a_{ar}a_{sm}a_{sm}a_{rs}a_{ac}a_{gc}a_{sm'}a_{gr'}$ that satisfies σ_g from state q_1, and append observation a_{ea} (obtained from the transition from q_0 to q_1) to this trace, we will obtain trace **S3** shown in Table 2. **S3** satisfies σ_g from q_0 and requires the same amount of mutation but has one length more than π_k. Also, consider two subproblems involving two transitions with the same observation and destination state but different source states. We want to satisfy the same trace skeleton with the same budget. Then, any trace obtained from the first subproblem will also be a solution for the second subproblem. Thus, the subproblems exhibit *overlapping* characteristics and Proteus adopts a dynamic programming-based solution to generate test cases.

❏ **Challenge C3**: **Arbitrary mutations miss logical vulnerabilities.** As discussed in challenge **C2**, one must perform mutations on the PSM to increase the chance of triggering security property violations. To mutate a PSM, we must select a transition and alter it, i.e., its input, output, or destination state. At any protocol state, randomly selecting a transition to mutate is less likely to violate the given property since it would not be property-driven. Similarly, even after selecting a transition to mutate, randomly altering the input message or output message of the transition at the bit/byte level, like traditional fuzzers [**? ?**], would also be inefficient since it does not consider the semantic meaning of the message fields (i.e., byte offset and boundary of each message field), the given security

Table 3: Simple scheduling example for insight I4.

Test case ID	Properties Violated	Average States Covered	No. of Deviations Covered	Selection Order
π^1	ϕ_1	5	2	1
π^2	ϕ_1	5	1	x
π^3	ϕ_1	5	0	x
π^4	ϕ_2	3	2	2

property, and the current protocol state. Although grammar-guided fuzzers [?] offer semantic meaning-aware mutations, they do not consider the property being tested and the current state of the protocol while selecting a message to mutate. Such mutation schemes are less likely to generate mutated messages that violate a given security property. Therefore, an effective mutation scheme should (i) consider the security property being tested while selecting a transition and (ii) consider the semantic meaning of a message, the given property, and the current protocol state while mutating a selected message (i.e., have a clever choice to fill the wildcard characters of the test skeleton π^a).

Insight I3. To address challenge **C3**, we consider selecting a transition to mutate $\mathcal{M}$ with a goal to generate traces satisfying test skeleton π^a. Such a mutation scheme is *property driven* since we aim to satisfy π^a, which signifies the violation of its corresponding security property. Also, while mutating any transition in $\mathcal{M}$, we consider mutating a transition's observation or destination state. Mutating the observation may uncover improper handling of prohibited messages. In contrast, mutating the destination state may uncover certain bypass attacks since these attacks represent skipping certain intermediate states to reach a secure state in protocol registration/authentication procedures. Moreover, to perform a mutation over the input message, we consider performing operations on it that are semantic aware (e.g., understanding field boundaries of any message field), state aware (e.g., setting field values within or outside a defined range according to the current protocol state) and also property driven (e.g., creating a plaintext version of a message to test at a security context established state, considering the property that plaintext messages should be dropped at such a state). Consequently, our mutation scheme is more likely to uncover a violation of the given security property than other existing works.

❑ **Challenge C4: Arbitrarily scheduling the properties and test traces to test OTA will lead to inefficiency.** Once we generate the set of test traces for all given properties, one can randomly schedule test traces to execute over-the-air (OTA). However, this approach is inefficient as it may fail to uncover vulnerabilities or adequately explore the search space within the given testing budget.

Insight I4. To address challenge **C4**, `Proteus` adopts an efficient scheduling mechanism to uncover vulnerabilities faster. In each testing iteration, `Proteus` first selects a property ϕ from the property set Φ to test and then a trace π^c to test ϕ. `Proteus` prioritizes scheduling the properties whose generated traces cover more states in the guiding PSM, increasing the likelihood of vulnerability detection. After selecting a property ϕ, `Proteus` selects a trace π^c, prioritizing based on the frequency of use of π^c, the number of already identified deviations π^c covers, and the number of instances of π^c rendered the target unresponsive. To avoid testing redundant traces, `Proteus` does not test any further traces of a property whose violation has already been detected.

As a simple example of our scheduling scheme, consider four traces, their corresponding property, the average number of states covered by the traces associated with the property, the number of already identified deviations covered by the trace, and their selection order in Table 3. Since traces associated with ϕ_1 cover more states on average than ϕ_2, `Proteus` first selects ϕ_1 to test. Among the three traces associated with ϕ_1, suppose all other factors are the same, but trace π^1 covers more transitions where `Proteus` observed a deviation from $\mathcal{M}$ in previous iterations. `Proteus` then selects π^1 to test OTA. Now suppose π^1 identified a violation of property ϕ_1. Then, `Proteus` discards testing all traces associated with ϕ_1 (i.e., π^2 and π^3), and in the next iteration selects π^4 to test.

5 RegExGenerator: CONSTRUCTING TEST SKELETONS FROM SECURITY PROPERTIES

`Proteus` first constructs test skeleton(s) for each property $\phi \in \Phi$ expressed in PLTL formula and uses the skeletons represented in regular expressions to generate test cases. To automatically construct test skeletons, we have developed RegExGenerator that takes a PLTL (past linear temporal logic) formula as input and produces REs violating the PLTL formula as output. To construct such an RE σ_i, RegExGenerator leverages the PLTL formula's Abstract Syntax Tree (AST) and traverses it in pre-order. Nodes in this AST correspond to PLTL operators (e.g., Since S, Yesterday Y), while leaves represent observations (α/γ). For instance, Figure 3 presents the AST for the PLTL formula $\phi = \boxed{a_{sm} \implies !a_{id'}\ S\ a_{dr}}$.

To create a violating regular expression for a PLTL formula ϕ, at each internal (non-leaf) node n_i during AST traversal, RegexGenerator needs to satisfy or negate the sub-formula ϕ^{n_i} rooted at n_i based on the semantic meaning of the operators involved in ϕ^{n_i}. For this, RegExGenerator recursively satisfies or negates the expression at n_i's child nodes according to the semantic meaning of the operator at n_i. While traversing a leaf node n_l of the AST, RegExGenerator places a literal or a kleene star element on σ according to the requirement (satisfaction or negation) at n_l. For example, if the operator is a "*since*" as in $\boxed{!a_{id'}\ S\ a_{dr}}$ in Figure 3, RegExGenerator attempts to first avoid satisfying the right subtree a_{dr} by placing $\neg(a_{dr})^*$, and then violate the left subtree by placing $a_{id'}$ after $\neg(a_{dr})^*$. Thus it will find $\neg(a_{dr})^* a_{id'}$ violating the "*since*" operator. Again, for the "*implies*" operator, RegExGenerator attempts to first satisfy the left subtree (obtaining RE $(.)^* a_{sm}$) and then violate the right subtree (obtaining RE $\neg(a_{dr})^* a_{id'}$). Finally, to construct an RE representing the violation of the given PLTL formula, the operator/operand at the root node of the AST must be negated. The final RE σ_v violating ϕ will be $\boxed{(.)^* a_{sm} \neg(a_{dr})^* a_{id'}}$. RegExGenerator generates multiple different violating expressions. It also checks if a previously generated RE already covers the expression. If so, it discards the new RE.

6 TraceBuilder: GENERATING TEST CASES FROM TEST SKELETONS AND GUIDING PSM

Given a test skeleton π^a representing violations of a security property $\phi \in \Phi$, `Proteus` uses a guiding PSM $\mathcal{M}$ and mutates $\mathcal{M}$ to efficiently generate meaningful test cases π^c with the shape of π^a.

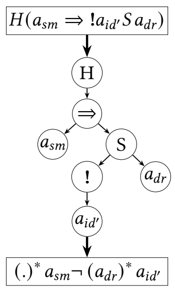

$H(a_{sm} \Rightarrow !a_{id'} \, S \, a_{dr})$
H
$\Rightarrow$
a_{sm}
S
!
a_{dr}
$a_{id'}$
$(.)^* \, a_{sm} \neg \, (a_{dr})^* \, a_{id'}$

Figure 3: Example AST of a PLTL formula.

Note that the guiding PSM used by Proteus can be abstract compared to PSMs extracted from target device implementations. As such, they are substantially smaller than PSMs learned from commercial devices (§8). One can also use PSMs directly obtained from RFCs (e.g., [?]) as a guiding PSM. Since `Proteus` does not require detailed PSMs, it alleviates the prohibitively high time required to learn a detailed PSM from target implementations. We now discuss our PSM mutation strategies and then present our test case generation mechanism.

6.1 Mutating a PSM

As discussed in challenge **C2** in §4, since the guiding PSM is assumed to satisfy all security properties, we need to mutate the guiding PSM $\mathcal{M}$ to generate traces satisfying a test skeleton π^a, which signifies the violation of a security property. `Proteus` selects a transition $\mathcal{R}(q_c, \alpha, \gamma, q_n)$ to mutate while generating instantiated traces satisfying a test skeleton π^a (described in §6.2). For mutations, `Proteus` essentially performs two types of operations on a selected transition $\mathcal{R}$, as discussed below.

❏ **Mutation kind M1: Mutating the observation of a transition.** In this case, `Proteus` alters the observation α/γ of $\mathcal{R}$, i.e., alters either the input α or the output γ or both based on the input and output in trace skeleton π^a when the execution reaches at q_c.

At any state q_c, if the trace skeleton π^a to satisfy requires input α' and output γ' but the guiding PSM $\mathcal{M}$ does not have α'/γ'at q_c, to ensure the generated trace conforms with π^a, we mutate the transition with α' at q_c in $\mathcal{M}$ to obtain input α' and output γ'. For example, in Figure 1, suppose at state q_5 if we require output GUTI_ REALLOCATION _COMPLETE for input GUTI_ REALLOCATION _COMMAND: REPLAY ==1 according to the test skeleton. However, since it is not prescribed at state q_5, we mutate the transition with observation a_{gc} (with input GUTI_ REALLOCATION _COMMAND) to obtain input GUTI_ REALLOCATION _COMMAND: REPLAY == 1 and output GUTI_ REALLOCATION _COMPLETE. This mutation strategy is required to generate traces satisfying π^a.

On the other hand, if π^a has a wildcard character at state q_c, `Proteus` instantiates the wildcard character with a mutated version of an input message α to detect any deviation from the PSM that can lead to a security property violation. To increase the likelihood of a mutated message being accepted by a practical protocol implementation, `Proteus` considers the semantic meaning of the input message. It performs one of the six semantic operations. We summarize these operations with examples in Table 4. These operations consider the semantics of the message fields, their defined values and ranges according to the protocol specification, and overall message semantics (e.g., plaintext or replayed version). As an

Table 4: List of possible semantic operations performed on an input message for mutation type M1. HOP is a 5-bit field in CONNECTION_ REQUEST message in BLE whose value is defined to be between 5 to 16 according to BLE specifications. ATTACH_ACCEPT and SECURITY_MODE_COMMAND are messages in 4G LTE.

Operation	Description	Example
OP1	Change value of a field to a defined value in range	CONNECTION_ REQUEST: HOP == 5
OP2	Change value of a field to prohibited value/value outside of defined range	CONNECTION_ REQUEST: HOP == 20
OP3	Change value of a field to one of its boundary values (i.e., setting all bits of the field to 0 or 1)	CONNECTION_ REQUEST: HOP == 31
OP4	Send the plaintext version of any integrity protected and/or ciphered message	ATTACH_ACCEPT: INTEGRITY == 0 & CIPHER == 0
OP5	Combination of applying OP1, OP2, OP3 and OP4 multiple times	ATTACH_ACCEPT: INTEGRITY == 0 & CIPHER == 0 & SECURITY_HEADER_TYPE == 15
OP6	Replay a previously captured version of the message	SECURITY_MODE_COMMAND: REPLAY == 1

example, in Figure 4(i), consider a mutation of the transition with observation a_{ap} (with input ATTACH_ ACCEPT: INTEGRITY == 0 & CIPHER == 0 & SECURITY_ HEADER _ TYPE == 0, marked blue) at state q_1. If we perform **OP2** operation and change the value of the SECURITY_ HEADER _ TYPE field to a prohibited value 4, we obtain the mutated message a'_{ap} with input ATTACH_ ACCEPT: INTEGRITY == 0 & CIPHER == 0 & SECURITY_ HEADER _ TYPE == 4. If the target accepts this mutated message, we obtain ATTACH_COMPLETE as a response, which deviates from the guiding PSM.

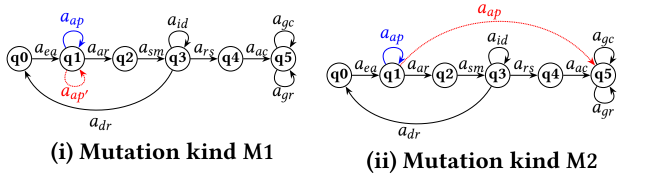


Figure 4: Two kinds of mutations M1 and M2. The transition labels are presented in Table 1.

❏ **Mutation kind M2: Mutating the destination state of a transition.** In this case, `Proteus` alters the destination state q_n of a selected transition $\mathcal{R}$ $(q_c, \alpha, \gamma, q_n)$. The goal of this kind of mutation is to test whether the input message α at state q_c leads to a state q_m different than what is expected by the guiding PSM (in this case q_n), which in turn can lead to an observable deviation or security property violation. This kind of mutation enables `Proteus` to identify certain bypass attacks, e.g., detect whether any intermediate step can be bypassed to reach a secure state. To illustrate, in Figure 4(ii), suppose we perform mutation over the transition with observation $a_{ac'}$ at state q_1 (marked blue). We can perform a mutation over the next state q_2 of the transition by changing the next state of the transition to q_5. This mutation would generate an instantiated trace that tests whether the target incorrectly reaches state q_5 if a plaintext ATTACH_ACCEPT is fed at q_1.

6.2 Instantiated Trace Generation

Given a test skeleton π^a representing violation of a property $\phi \in \Phi$, a guiding PSM $\mathcal{M}$, a mutation budget μ and a length budget λ, we leverage insight **I2** and solve the problem of generating instantiated traces by combining the solutions of overlapping subproblems. For this, we formulate a dynamic programming problem as follows.

$\mathcal{G}(\mathcal{M}, \pi^{a}, q_c, \mu, \lambda)$: *At any state* $q_c \in Q$ *of a guiding PSM* $\mathcal{M}$, *we want to craft a set of instantiated traces* $\mathcal{T}$, *where each instantiated trace (i) satisfies the test skeleton* π^a, *(ii) is the outcome of a maximum of* μ *mutations applied on any trace in the guiding PSM* $\mathcal{M}$ *and (iii) is within a maximum length* λ.

Consider we are at state q_i of our guiding PSM $\mathcal{M}$ and we need to satisfy the j^{th} observation $l_j = (\alpha_j/\gamma_j)$ of π^a, i.e., the test skeleton up to input symbol l_{j-1} has been satisfied and the next wildcard character in π^a is l_k. Also, at q_i, consider the remaining mutation budget is μ_i, and the remaining length budget is λ_i. For this problem, we consider the following four cases, formulating subproblems and prepending suitable observations.

Case I. If there is a transition $\mathcal{R}(q_i, \alpha, \gamma, q_m)$ at state q_i whose observation satisfies l_j, TraceBuilder can leverage solutions from the destination state q_m that satisfies π^a from observation l_{j+1}, with mutation budget μ_i and length budget λ_i-1. TraceBuilder prepends l_j to the instantiated traces being generated from the subproblem. Since it places an observation (l_j) in the instantiated trace, it uses length budget λ_i-1 to formulate the subproblem.

Case II. If there is no transition at q_i satisfying l_j, TraceBuilder can consume a mutation to mutate the transition for α_j, the input message of observation l_j. TraceBuilder places observation l_j in the instantiated trace using mutation kind **M1**. Again, TraceBuilder leverages traces from the destination state q_m that satisfies π^a from observation l_{j+1}, with length budget λ_i-1 but using mutation budget μ_i-1 since it consumed a mutation to place l_j. TraceBuilder prepends l_j to the obtained instantiated traces from the subproblem.

Case III. If there is a transition $\mathcal{R}(q_i, \alpha, \gamma, q_m)$ at state q_i whose observation (α/γ) satisfies wildcard character element l_k, TraceBuilder leverages solutions from the destination state q_m that satisfies π^a from observation l_j with mutation budget μ_i and length budget λ_i-1. It prepends observation (α/γ) to the obtained instantiated traces from the subproblem.

Case IV. Finally, for any transition $\mathcal{R}(q_i, \alpha, \gamma, q_m)$ at state q_i, TraceBuilder can mutate the transition's observation and place a mutated observation $(\alpha/\gamma)_m$ using mutation kind **M1**. TraceBuilder can leverage solutions from the destination state q_m that satisfies π^a from observation l_j with length budget λ_i-1 and also mutation budget μ_i-1 since we perform a mutation. It prepends the mutated observation $(\alpha/\gamma)_m$ to the obtained instantiated traces from the subproblem. Note that we only place a *mutation marker* in this case. This marker would be resolved later by TraceDispatcher (§7). Note that we can generate multiple traces to test OTA by placing various mutated messages in place of the mutation marker.

Furthermore, for each case, TraceBuilder may also consider mutating the destination state of the selected transition (mutation kind **M2**). In that case, TraceBuilder mutates the destination state to another state q_{mut} and formulates the subproblem from state q_{mut} with mutation budget $\mu-1$. TraceBuilder terminates if either π^a is satisfied or runs out of mutation or length budget.

7 TraceDispatcher: TEST EXECUTION AND FLAW DETECTION

After TraceBuilder generates instantiated traces associated with each property in its property set Φ, the TraceDispatcher component of `Proteus` iteratively selects a property ϕ and then an instantiated test trace π^c generated from ϕ to test OTA. TraceDispatcher runs for t iterations, where t is proportional to β. TraceDispatcher consists of three components: a *scheduler*, an *adapter* and an *observer*.

Scheduler. In each testing iteration, the *scheduler* of TraceDispatcher first chooses a property $\phi \in \Phi$ to test using weighted random sampling. The scheduler determines the weight of each property by determining the average number of distinct states in the guiding PSM covered by all instantiated traces of ϕ. This scheme favors properties whose instantiated traces cover more states within the guiding PSM and implicitly prioritizes traces more likely to violate a security property within the input property set Φ.

Once a property ϕ is selected, the scheduler chooses an instantiated trace π^c to test from all traces associated with ϕ. Note that an instantiated trace may or may not have mutation markers (case IV in §6.2). The scheduler prioritizes selecting instantiated traces with mutation markers since they test mutated input messages, which are more likely to trigger property violations. Furthermore, if the scheduler chooses an instantiated trace with a mutation marker, it first prioritizes scheduling traces that mutate messages that were not mutated in previous iterations. Again, among the traces that have messages not mutated previously, the *scheduler* selects an instantiated trace based on three factors: (i) the frequency of selection (f) of π^c; (ii) the number of deviations covered by π^c (d); (iii) the number of instances where π^c rendered the IUT $\mathcal{I}_P$ unresponsive (u). The scheduler selects the instantiated trace with the minimum score ($p = f - d + u$), randomly choosing one in case of a tie. The scheduler prioritizes instantiated traces that trigger known deviations since they can lead $\mathcal{I}_P$ to an inconsistent state where `Proteus` is more likely to find property violations. Also, the scheduler is less inclined to traces that rendered $\mathcal{I}_P$ unresponsive since we cannot find further property violations from an unresponsive $\mathcal{I}_P$. The scheduler randomly performs one of the six operations defined in Table 4 to mutate a message with a mutation marker and obtain a mutated input message.

Adapter. The adapter takes an instantiated trace selected by the scheduler to test OTA. It translates the abstract input symbols into concrete messages and sends them to $\mathcal{I}_P$. Also, it records the response from $\mathcal{I}_P$, converts it back into an abstract symbol, and sends the response sequence back to the observer.

Observer. The observer analyzes the response from $\mathcal{I}_P$ to an instantiated trace and detects whether it violates any security properties. If there is any deviation from the guiding PSM, the observer automatically checks for property violation by checking the trace against the violating test skeletons (represented as REs) generated from the security properties. Additionally, to determine unresponsiveness, the *observer* identifies the final protocol state q_f according to the guiding PSM for each test trace. It then executes a message α_f OTA expected to elicit a valid output message γ_f. If $\mathcal{I}_P$ does not respond, it is deemed unresponsive.

8 EXPERIMENTS

We evaluate `Proteus` with the 4G LTE's NAS and RRC layer protocols (e.g., mobility management procedures) and BLE's SMP and Link Layer protocols [?] based on the following research questions:

- **RQ1.** How effective is `Proteus` in finding novel and known issues in LTE and BLE implementations (§9)?

- **RQ2.** How effective and efficient `Proteus` is compared to existing works (§10)?
- **RQ3.** How does `Proteus` perform with respect to generating test cases (§11)?

8.1 Experiment Setup For Testing

We combine LTE's NAS and RRC layers' protocols and construct a single guiding PSM consisting of 7 states with 86 transitions in total. For BLE, our guiding PSM has 15 states with 244 transitions in total. Compared to PSMs learned from implementations [? ?], the guiding PSMs in `Proteus` are substantially smaller. For example, an average-sized PSM learned from Pixel3A [?] reportedly had 21 states and 548 transitions, which is three times larger than `Proteus` guiding PSM for 4G LTE.

For LTE, we set up a base station using srsRAN and a USRP B210, and a core network using srsEPC. We run them using an Intel i7-8665U processor with 16GB RAM. For BLE, we use nRF52840 acting as a central to test peripheral devices. We have tested 11 LTE and 12 BLE devices. Note that we updated all target devices with the latest patches before performing testing. The details of our tested devices (including SoC model, vendor, and baseband), and the identified issues in each device are provided in Tables 12 and 13 in Appendix.

9 IDENTIFIED ISSUES

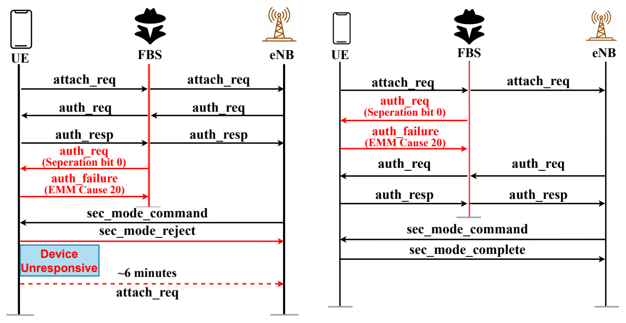
To answer **RQ1**, we have tested each device with 3000 OTA queries using `Proteus`. Our evaluation reveals that `Proteus` identifies 31 and 81 vulnerabilities in LTE and BLE, respectively, with 3 unique new issues in LTE and 7 unique new issues in BLE implementations. The identified issues resulted in five new CVEs, two bug bounties, and 9 acknowledgments from various vendors such as Google, Samsung, Qualcomm, and Microchip.

Tables 5 and 6 summarize the identified issues for 4G LTE and BLE, respectively. The tables show the number of distinct devices where each issue was identified, the impact of each attack, and any new CVE/acknowledgment obtained by `Proteus` for each attack. We discuss in detail some of our identified issues in 4G LTE and BLE in §9.1 and §9.2, respectively.

9.1 Identified Issues in LTE

Attacker model. Similar to prior works [? ? ?], we assume that a Dolev-Yao attacker [?] knows the victim's Cell Radio Network Temporary Identity (C-RNTI) using the victim's phone number [? ? ?] and then send malformed messages to the victim device using a fake base station (FBS) [? ?]. For attacks **L-E3** and **L-E4**, the attacker requires an adversary in addition to a FBS to capture and replay messages. The attacker can also use a Machine-in-the-Middle (MitM) relay [? ?] to exploit the vulnerabilities, as MitM relays are more powerful than FBS.

L-E8: AUTHENTICATION_REQUEST with separation bit 0 causes further security mode procedure failure. According to the 4G NAS specifications (TS 24.301, clause 5.4.2.6), a UE should send an AUTHENTICATION_FAILURE message with EMM cause # 26 upon receiving an AUTHENTICATION_REQUEST message with "separation bit" set to 0. However, after successful authentication, affected devices respond with an AUTHENTICATION_FAILURE message with EMM cause #20 in response



(a) After authentication. (b) Before authentication.

Figure 5: AUTHENTICATION_REQUEST with separation bit 0 causing further security mode procedure failure.

to such a message. The device then moves to "unauthenticated" state, leading to a subsequent failed security mode control. Figure 5(a) shows the attack steps where the attacker sends such a malicious message to the victim UE using an FBS. Note that if the attacker attempts to send a malformed AUTHENTICATION_REQUEST message before a successful authentication, the attack does not succeed (see Figure 5(b)). This signifies the stateful nature of the vulnerability that the existing works [? ?] cannot detect.

Impact. Upon receiving the malformed message, Nexus 6P and Samsung A71 remained unresponsive for over 6 minutes and then attempted to reconnect. The attacker can repeat the attack steps, leading to a prolonged denial-of-service (DoS) attack on the victim user. Qualcomm identified the issue as medium severity and assigned a CVE. Note that the DoS attacks identified by `Proteus` are different in nature from spectrum jamming [? ?], which are easily detectable, expensive, and unreliable. In contrast, our identified DoS attacks exploit logical vulnerabilities and allow targeting a single device with only one software-defined radio (SDR) and cannot be thwarted by jamming-specific defenses [?].

L-E9: Respond to AUTHENTICATION_REQUEST with header 3 with SECURITY_ MODE_REJECT. The affected devices incorrectly interpret a malformed AUTHENTICATION_REQUEST with security header 3 before a successful authentication as a SECURITY_MODE_COMMAND, and respond with SECURITY_ MODE_REJECT. However, in this scenario, the TS 24.301 [?] clause 7.5.1 suggests returning a EMM_STATUS message with cause #96 'invalid mandatory information'.

Impact. Since a correctly implemented device would drop this packet, an attacker can fingerprint the victim device up to the baseband manufacturer level and launch attacks by combining other known vulnerabilities at the baseband-chipset level. Huawei acknowledged this vulnerability as low severity.

L-E5: Accepts RRC_ SECURITY_ MODE_COMMAND with EIA0 IE. The affected devices respond with RRC_SECURITY_ MODE_ COMPLETE to the plaintext RRC_ SECURITY_ MODE_COMMAND with EIA0 message. This flaw leads to a remote privilege escalation attack with no additional execution privileges required. The attacker can bypass the RRC layer security activation by exploiting this vulnerability. The victim UE then accepts all plaintext RRC messages without integrity protection. `Proteus` detected this vulnerability in Pixel7. Google assigned a high-severity CVE to it. Although Rupprecht et al. [?] identified

Table 5: Vulnerabilities identified by `Proteus` for COTS LTE devices. NAS-SC: NAS security context establishment, AS-SC: AS security context establishment. ○ : known vulnerability found on devices previously confirmed to have the attack, ◑ : known vulnerability found on new device not previously reported to have the vulnerability, ● : previously unknown vulnerability. E-Exploitable, I-Interoperability, O-Other issue.

Issue	Description	Category	#Vuln. Instance	Impact	Newly Obtained CVEs/ Acknowledgements
L-E1	Accepts plaintext IDENTITY_REQUEST After NAS-SC [? ?]	◑	3	Location Tracking	Marked as duplicate by Samsung [1]
L-E2	Accepts plaintext AUTHENTICATION_REQUEST after NAS-SC [? ?]	◑	4	Location Tracking, DoS	Marked as duplicate by Samsung
L-E3	Accepts replayed SECURITY_MODE_COMMAND after NAS-SC [? ?]	○◑	4	Location Tracking	Marked as duplicate by Unisoc
L-E4	Accepts replayed GUTI_REALLOCATION_COMMAND after attach procedure [?]	◑	2	Location Tracking	Marked as duplicate by Unisoc
L-E5	Accepts RRC_ SECURITY_ MODE_COMMAND with EIA0 IE after NAS-SC [? ? ?]	◑	1	Security Bypass	High Severity CVE from Google
L-E6	RRC_ SECURITY_ MODE_COMMAND with EIA0 IE after NAS-SC causes unresponsiveness [?]	○◑	4	DoS	-
L-E7	Accepts plaintext COUNTER_CHECK before AS-SC [?]	◑	4	Fingerprinting	Marked as duplicate by Unisoc
L-E8	AUTHENTICATION_REQUEST with separation bit 0 cause further security mode procedure failure	●	4	DoS	Medium Severity CVE from Qualcomm
L-E9	Respond to AUTHENTICATION_REQUEST with header 3 with SECURITY_ MODE_REJECT	●	1	Fingerprinting	Acknowledged by Huawei
L-O1	Respond to IDENTITY_ REQUEST (TMSI) with IDENTITY_ RESPONSE (IMSI) before NAS-SC	●	4	Fingerprinting	-

a similar issue in older devices, `Proteus` detected **L-E5** in new devices not previously reported to have the vulnerability.

9.2 Identified Issues in BLE

Attacker model. We also assume the Dolev-Yao attacker model for BLE. Similar to previous works [? ? ?], the attacker acts as a malicious central and can intercept, replay, modify, or drop packets. The attacker only knows the public information of the target peripheral (e.g., Bluetooth name, address, protocol version number, and capabilities) and does not require knowledge regarding any secret keys shared between the target peripheral and any other device. For issues **B-E1, B-E2, B-E4, B-I1, B-O1** to **B-O3**, the attacker can directly establish a connection with the victim peripheral to launch the attacks. For all the other issues, the attacker must inject malicious traffic to the target peripheral [?] when the target pairs with another device for the first time.

B-E9: Accepts two continuous PAIRING_REQUEST leading to DoS. The affected peripherals cannot complete the pairing procedure when a malicious central sends two consecutive PAIRING_REQUEST messages. On the other hand, Microchip's BLE device responds to both PAIRING_REQUEST with PAIRING_RESPONSE. For other devices not having this vulnerability, the second PAIRING_REQUEST is ignored, and the pairing procedure proceeds as usual.

Impact. This attack leads to DoS, where the victim cannot complete the pairing process. If the central device does not have an auto-reconnect feature, the pairing procedure needs to be manually re-initiated, and even with auto-reconnect, the device needs to restart the entire pairing procedure again. Also, the vulnerability can be exploited to perform a manufacturer-level fingerprinting of vulnerable implementations. Microchip assigned a CVE with low severity to this issue.

B-E1, B-E2, B-E6, B-E7, B-E10 and B-E11. To exploit these vulnerabilities, i.e., **B-E6**, **B-E7**, **B-E9** to **B-E11**, the adversary assumptions are identical to those for **B-E9**. For **B-E6**, we observe that before the pairing process starts, the affected devices will respond to LENGTH_REQUEST with MaxRxOctets and MaxTxOctects field set to 1, and as a result, the subsequent pairing process cannot be completed. For **B-E7**, we observe that sending Data Physical Channel PDUs, such as MTU_REQUEST, LENGTH_REQUEST, VERSION_ REQUEST, after ENCRYPTION_REQUEST, can cause the peripheral to disconnect the link. For **B-E10**, we observe that the affected devices cannot finish pairing if it receives a ENC_PAUSE_REQ_PLAINTEXT after PUBLIC_KEY_EXCHANGE. For **B-E11**, an affected device disconnects after receiving a CHANNEL_MAP_REQUEST with the unchanged *Channel Map* field. For **B-E1** and **B-E2**, the device cannot recover by itself and requires manual reboot. For the other identified DoS attacks, the pairing procedure needs manual re-initiation. Also, sending the packets repeatedly can lead to prolonged DoS and battery depletion for each case.

B-E4: Accepts malformed CONNECTION_ REQUEST with increased data length field value. An implementation accepts CONNECTION_ REQUEST with an increased *Data Length* value. The CONNECTION_ REQUEST is extended to 247 bytes when *Data Length* field value is increased. Thus, after accepting this packet, the implementation may allocate more memory than required.

Impact. Since the vulnerable device accepts L2CAP packets with the wrong length, more bytes than expected are allocated in memory for an incorrectly implemented device. Sweyntooth [?] found a similar vulnerability for PAIRING_REQUEST packet, leading to memory leakage. Moreover, an attacker can fingerprint the device at the manufacturer's level, which may be further exploited by abusing other known firmware-level vulnerabilities. Samsung acknowledged this issue and provided a medium-severity CVE.

B-E3: Bypassing passkey-entry during legacy pairing. During the passkey-entry association method, a malicious central can bypass the passkey entry step by sending a SM_RANDOM message with a temporary key set to 0. This bypasses all MitM protection mechanisms inherent in the passkey entry method. BLEDiff [?] initially identified this issue in older devices. However, we identified this issue on several newer devices, including peripherals from Samsung, Google, Hisense, and Motorola (see Table 13 in Appendix). Samsung assigned a CVE with medium severity to this issue.

Other Issues. `Proteus` also identified several issues, such as accepting malformed CONNECTION_ REQUEST with a Hop field greater than the defined range (**B-O1**), accepting VERSION_ REQUEST message multiple times (**B-O2**) and accepting ENCRYPTION_REQUEST with non-zero EDIV and Rand field value (**B-O3**). The attacker can exploit these to fingerprint the affected device.

10 COMPARISON WITH EXISTING WORKS

To address **RQ2**, we compare `Proteus` with existing LTE and BLE testing frameworks. We first provide a qualitative comparison with respect to different metrics and then provide empirical comparisons with respect to the number of vulnerabilities detected, coverage, and cumulative vulnerability detection over time.

Table 6: Vulnerabilities identified by Proteus for COTS BLE devices. All interpretations are the same as for Table 5.

Issue	Description	Category	#Vuln. Instance	Impact	Newly Obtained CVEs/ Acknowledge-ments
B-E1	Stops advertising if CONNECTION_ REQUEST with Interval/channelMap field set to 0 is sent [? ?]	○◑	3	Crash	Acknowledged by MediaTek as low severity
B-E2	Stops advertising with plaintext ENCRYPTION_PAUSE_RESPONSE [?]	◑	2	Crash	Acknowledged by MediaTek & Samsung as low severity
B-E3	Bypassing passkey entry in legacy pairing [?]	○◑	10	Privacy/ Null Encryption	Medium severity CVE from Samsung
B-E4	Accepts malformed CONNECTION_ REQUEST message with increased length	●	8	Memory Leakage	Medium severity CVE from Samsung
B-E5	Accepts SM_CONFIRM with wrong values [?]	○	1	DoS	-
B-E6	LENGTH_REQUEST with MaxRxOctets and MaxTxOctets fields set to 1 cause DoS	●	2	DoS	Acknolwedged by MediaTek as low severity
B-E7	Unexpected Data Physical Channel PDU during encryption start procedure cause DoS	●	11	DoS	Acknolwedged by Samsung
B-E8	PUBLIC_KEY_EXCHANGE in legacy pairing leading to DoS [?]	○	1	DoS	Reported by Sweyntooth [?] and fixed in newer version
B-E9	Two continuous PAIRING_REQUEST leading to DoS	●	2	DoS	CVE from Microchip with low severity
B-E10	Plaintext ENCRYPTION_PAUSE_REQUEST cause DoS	●	4	DoS	Acknolwedged by MediaTek & Samsung as low severity
B-E11	CHANNEL_MAP_REQUEST with unchanged channelMap field cause DoS	●	6	DoS	-
B-I1	Issue with OOB Pairing Fails [?]	◑	8	Fingerprinting	-
B-O1	Accept CONNECTION_ REQUEST with Hop field > defined range	●	9	Fingerprinting	Acknowledged by MediaTek as a functional bug (Negligible Security Impact)
B-O2	Accept VERSION_ REQUEST multiple times [?]	◑	2	Fingerprinting	Acknowledged by Samsung
B-O3	Accept ENCRYPTION_REQUEST with non-zero EDIV and Rand [?]	◑	12	Fingerprinting	Acknowledged by MediaTek as a functional bug (Negligible Security Impact)

10.1 Qualitative Comparison

We compare Proteus with existing works with respect to different metrics as shown in Table 7. Among these works, DIKEUE [?], DoLTEst [?], Contester [?], BaseComp [?], BLEDiff [?] and BLE Blackbox Fuzzing [?] perform stateful testing. However, DIKEUE, BaseComp, BLEDiff, and Blackbox Fuzzing do not consider any specific properties corresponding to the protocols while generating queries. Although LTEFuzz [?], DoLTEst and Contester have property-guided generation, LTEFuzz is not stateful, and DoLTEst and Contester do not generate test cases dynamically. Also, SweynTooth [?] does not consider the guidance of properties while generating test cases.

Further, DoLTEst, LTEFuzz, and SweynTooth do not perform positive testing, and only DIKEUE, DoLTEst, Contester, BLEDiff, and Blackbox Fuzzing have the capabilities to identify interoperability issues. Finally, none of the works except Proteus consider efficient scheduling of test traces to maximize property violation detection within a fixed testing budget. Only Proteus simultaneously covers all these aspects of testing. In addition, it is the only dynamic framework that can provide control over the amount of test traces being generated and, hence, can be tuned to satisfy a time budget for testing.

Table 7: Comparison with existing testing approaches.

Approach	Dynamic	Stateful Testing	Property Focused Testing	Positive Test Cases	Time Budget Wise Tuning	Interoperability Issue Detection
DIKEUE [?]	✓	✓	✗	✓	✗	✓
LTEFuzz [?]	✓	✗	✓	✗	✗	✗
DoLTEst [?]	✗	✓	✓	✗	✗	✓
Contester [?]	✗	✓	✓	✓	✗	✓
Basecomp [?]	✗	✓	✗	✓	✗	✗
BLEDiff [?]	✓	✓	✗	✓	✗	✓
SweynTooth [?]	✓	✗	✗	✗	✗	✗
Blackbox Fuzzing [?]	✓	✓	✗	✓	✗	✓
Proteus	✓	✓	✓	✓	✓	✓

10.2 Total Number of Vulnerability Detection

We compare Proteus with DIKEUE [?], and DoLTEst [?] for LTE, and BLEDiff [?] and Sweyntooth [?] for BLE, with respect to the total number of identified vulnerabilities without considering any time budget. For each vulnerability, we determine whether the vulnerability is identifiable by– (i) only Proteus, (ii) only the baseline work, or (iii) both works. For DIKEUE [?] and BLEDiff [?], we used their provided alphabet.

Table 8: Detected vulnerability count by existing works.

Baseline	#Vuln. both Proteus and Baseline can Identify	#Vuln. only Proteus can Identify	#Vuln. only Baseline can Identify
DIKEUE	15	4	0
DoLTEst	25	5	1
BLEDiff	13	7	0
Sweyntooth	18	8	2

We present the results in Table 8. Our analysis reveals that Proteus can identify all vulnerabilities detected by the existing works except for one identified by DoLTEst [?]. To detect this particular vulnerability, DoLTEst manually crafts and sends two traces to $\mathcal{I}_P$: one with an IDENTITY_REQUEST message using security header 12 and another using security header 15, observing any differences in the target's response. In contrast, Proteus can automatically generate and reason about only one trace at a time, meaning it can only check if a single trace violates any security properties at once. Proteus is not designed to reason about hyper-properties which require simultaneous consideration of multiple traces.

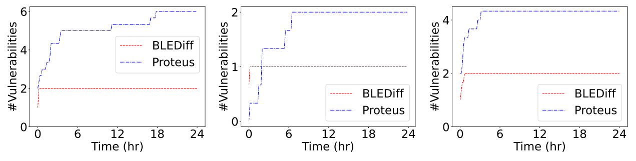
In contrast, even without considering a testing budget, DoLTEst [?] cannot detect replay vulnerabilities (e.g., **L-E3, L-E4**) and also stateful vulnerabilities where the vulnerability is triggered at a state not defined by it. For example, issue **L-E8** can only be triggered after authentication and before NAS security activation, but DoLTEst does not incorporate this state. Sweyntooth [?] cannot detect logical vulnerabilities because it lacks a test oracle to identify such issues. Furthermore, to detect all vulnerabilities identified by Proteus through an automata-based learning approach [? ?], we must incorporate a set of alphabet containing all symbols used by Proteus, i.e., all messages, message fields, and their mutations during PSM learning. Using this alphabet in a 4G LTE environment (111 symbols) with DIKEUE [?] would require 87,246 unique queries (~133 days if tested OTA) to learn the PSM of an implementation, taking over four months to detect all vulnerabilities Proteus identified with 3000 queries (2 days when tested OTA).

Additionally, as shown in Table 8, Proteus detects 4-5 new vulnerabilities in LTE and 7-8 new vulnerabilities in BLE compared to these prior works. Thus, Proteus is more effective in identifying vulnerabilities in the tested protocols.

10.3 Vulnerability Count Growth

We demonstrate Proteus 's efficiency in detecting vulnerabilities over time. This evaluation also demonstrates the quality of the generated traces by Proteus to detect vulnerabilities. We compare Proteus with the existing works in terms of cumulative vulnerability count on 3 devices from BLE and 2 devices on LTE. Further, since we focus on logical bugs, we consider the security properties as an oracle to detect vulnerabilities. Note that we did not observe any false positives (i.e., any reported property violation that does not result in exploitable vulnerability). The default value of the length budget parameter for TraceBuilder is $l + 1$, where l is the number of observations in any test skeleton for properties. Also, we set the default mutation budget to 2.

Cumulative vulnerability count on BLE devices. We compute the cumulative vulnerability count obtained over time on three BLE devices– Galaxy S6, Galaxy A22, and Oppo Reno7, with both Proteus and BLEDiff [?]. We consider BLEDiff as our baseline. However, BLEDiff learns the PSM of the target device first and then performs differential testing. Thus, to make a fair comparison, we consider the queries generated during learning as input queries to detect vulnerabilities and use the security properties obtained for Proteus as an oracle to determine whether any vulnerability is detected. We run Proteus and BLEDiff for 24 hours and count the number of detected vulnerabilities. We run the experiments for 3 iterations and present their average in Figure 6.

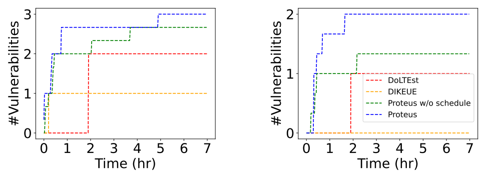


(a) Galaxy A22 (b) Galaxy S6 (c) Oppo Reno7

Figure 6: Cumulative vulnerability comparison in BLE.

Figure 6 shows that Proteus can detect more than 5 vulnerabilities on average within 24 hours, whereas the learning queries from BLEDiff can only detect a maximum of 2 vulnerabilities in the tested devices. Although Proteus, at first, falls behind BLEDiff in speed in S6, upon inspection, we find that the only vulnerability identified by BLEDiff is a trivial one, requiring a single message, and Proteus later finds more vulnerabilities, which BLEDiff cannot detect in 24 hours. In other cases, Proteus consistently performs better than BLEDiff. This demonstrates that Proteus is more efficient in detecting vulnerabilities than BLEDiff's automata learning-based approach, suggesting a better quality of the generated traces.

Cumulative vulnerability count on LTE devices. For LTE, we consider Pixel7 and Huawei P8 Lite and compute the cumulative vulnerabilities detected over time by both Proteus and the baseline works– DoLTEst [?], DIKEUE [?]. We run our experiment for ∼ 7 hours since all DoLTEst queries are executed within that period. Again, we run each test for 3 iterations and report the average count in Figure 7.



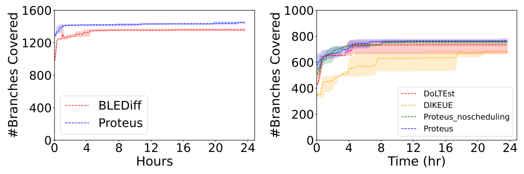
(a) Pixel 7 (b) Huawei P8 Lite

Figure 7: Cumulative vulnerability comparison in LTE.

Figure 7 shows that Proteus detects 3 and 2 vulnerabilities on average within 7 hours in Pixel7 and Huawei P8lite, respectively, whereas DoLTEst detects 2 and 1 on average. Moreover, the figure presents that Proteus detects those vulnerabilities faster than DoLTEst and DIKEUE, representing its superior efficiency.

10.4 Coverage Growth

This experiment aims to evaluate the efficacy of the generated traces with respect to coverage growth.



(a) BLE coverage (b) LTE coverage

Figure 8: Coverage growth over time.

Coverage growth in BLE devices. At first, we compare Proteus against BLEDiff [?] with respect to the number of branches covered over time while testing BLE implementations. Similar to §10.3, we consider the queries used for learning PSM of the target implementation as inputs to BLEDiff. Using BTStack v1.5.6.3 [?] as the target, we perform three 24-hour runs of both Proteus and BLEDiff to determine average branch coverage.

We present the results in Figure 8 (a), which shows that Proteus achieves ∼ 10% more coverage 3 times faster than BLEDiff.

Coverage growth in LTE devices. For LTE, we compare Proteus against DIKEUE [?] and DoLTEst [?]. We run each tool 3 times on srsUE [?], with each run lasting 24 hours, and report their average branch coverage over time. Figure 8(b) shows the coverage growth in LTE devices, where it is evident that Proteus achieves more branch coverage faster than both the baselines.

11 PERFORMANCE OF PROTEUS

Effectiveness of considering both security property and guiding PSM. We compare Proteus with two approaches described in §3: (i) *PSM only approach*, where we sample traces from our guiding PSM and perform mutations without any guidance (ii) *Property-only approach,* where we consider a RE representing the violation of the security property, and instantiate the RE using arbitrary observations independent of any guidance from a PSM. We use an RE representing the acceptance of a replayed GUTI_REALLOCATION_COMMAND attack

Table 9: Regular expression representing acceptance of a replayed guti_reallocation_command and security_mode_command

Property	Expression
σ_g	(enable_s1/ attach_request) (authentication_request/ authentication_response) $(.)^*$ (security_mode_command/ security_mode _complete) $(.)^*$ (rrc_ security_ mode_command/ rrc _ security _mode _complete) $(.)^*$ (attach_accept/ attach_complete) $(.)^*$ (guti_reallocation_command/ guti _ reallocation _ complete) $(.)^*$ (guti_reallocation_command: replay == 1/ guti _ reallocation _ complete)
σ_s	(enable_s1/ attach_request) (authentication_request/ authentication_response) $(.)^*$ (security_mode_command/ security_mode _complete) $(.)^*$ (security_mode_command:replay == 1/ security_mode _complete)

Table 10: No. of traces generated for varying mutation budget.

Property	Number of Observations	Number of Wildcards	Length Budget	Mutation Budget		
				1	2	3
σ_s	4	2	5	22	53	63
σ_g	7	5	8	49	121	166
σ_g	7	5	9	1441	5662	11560

after a successful registration procedure provided in Table 9. We test a HiSense F50+ device to check the number of queries it takes to detect the vulnerability using the two approaches and `Proteus`. We observe that `Proteus` and the property-only approach take 366 and 1690 queries to detect the vulnerability, respectively. The PSM-only approach cannot detect the vulnerability in 3000 queries. This signifies that combining the guidance from both PSM and a security property assists in quickly discovering the vulnerability.

Effect of varying mutation and length budget. We examine how the number of generated instantiated traces varies according to the length and mutation budget we set in TraceBuilder. We consider the two regular expressions σ_g and σ_s (Table 9), representing the acceptance of replayed guti_reallocation_command and security_mode_command, respectively, as text skeletons.

To observe the effect of varying the length budget, we set the mutation budget to 2 and varied the length budget from 4 to 10. The number of unique instantiated traces generated for both the test skeletons are presented in Table 11. We observe that the length budget has to have a minimum length equal to at least the number of literals present in the trace skeleton since, for a lower length budget, no sequence would be able to satisfy the given skeleton. On the other hand, the total number of unique traces generated increases as the length budget increases.

To observe the effect of varying the mutation budget, we vary it from 1 to 3 for both trace skeletons. We observe that if the length budget is constrained, e.g., a length budget of 5 with 4 literals in σ_s, then even with a higher mutation budget, we would not have many more test cases since there would be no space to inject any mutation. However, if the length budget is increased, the number of traces generated increases with the increase of the mutation budget. However, we empirically observed that a mutation budget of more than 2 did not yield any extra vulnerabilities.

Efficiency of scheduling approach. We compare with a version of `Proteus` that randomly schedules properties and instantiated traces (denoted as `Proteus` w/o scheduling). The experimental setup is the same as `Proteus`. Figure 8 and 7 show its coverage and cumulative vulnerability count over time. We observe that with our scheduling approach, we reach a higher coverage and detect vulnerabilities faster.

Table 11: Number of unique traces generated with various length budget values.

Property	# Literals	# Kleene Star	Length Budget (No. Of Unique Traces)						
			4	5	6	7	8	9	10
σ_s	4	2	1	62	2638	>20000	>20000	>20000	>20000
σ_g	7	5	0	0	0	1	121	5662	>20000

12 DISCUSSIONS

Scope of `Proteus`. In addition to logical bugs, `Proteus` can conceptually cause crashes of the COTS device under test due to memory bugs (*e.g.*, use-after-free). As `Proteus` operates in a black box setting, it cannot transparently observe inside the analyzed devices to discern crash bugs. As such, we consider such crash-inducing bugs to be outside `Proteus`'s scope.

Proteus's reliance on state machine and desired properties. As discussed before, `Proteus` relies on having access to the protocol's state machine and its desired security and privacy properties to generate meaningful test cases. One can consider this a limitation since one has to obtain both the state machine and properties of a protocol before testing of its implementation can commence. In our evaluation, we, however, demonstrate that access to this extra information is well worth the effort as it enables `Proteus` to generate high-quality test cases within a given testing budget.

Prior works applied formal verification techniques to evaluate the security and privacy of different protocol designs [**? ?**], where both protocol models and properties are manually constructed. For protocols where such efforts are underway, `Proteus` can take advantage of the constructed models and properties. Another approach currently gaining traction is using natural language processing (NLP) techniques and large language models to automatically extract protocol state machines from the standard specification text [**? ?**]. To evaluate the feasibility of applying such approaches, we used one such tool, Hermes [**?**], to automatically extract the protocol state machine of LTE from the standard specification. After comparing the extracted state machine with the hand-constructed one we use for our evaluation, we observe that the extracted state machine is missing a transition. Conceptually, `Proteus` can operate with a state machine that is not entirely accurate at the cost of some spurious test cases. As a thought exercise, we decided to use `Proteus` with the HERMES-extracted state machine to test using `Proteus` with the same security properties and experimental setup. We ran `Proteus` against Pixel 7 for 24 hours and identified all 3 detected issues identified by `Proteus` using an entirely correct PSM. Despite the missing transition in the automatically generated state machine, we observed that `Proteus` was able to identify all the vulnerabilities that it uncovered when using the hand-constructed state machine. This corroborates our hypothesis of `Proteus`'s loose reliance on the accuracy of the state machine.

Limitations of open-source adapter implementations impose constraints on testing. `Proteus`'s capabilities are constrained by the message types and predicates that we can implement in the *adapter* (§7) with open-source protocol stacks, which may support only a limited set of message types and predicates. Moreover, as a black-box testing method, `Proteus` can only observe the output messages of the IUT. Therefore, we only include transitions in our guiding PSM, for which the implementation should provide observable responses according to the specifications.

13 RELATED WORKS

Protocol testing. Prior works on testing protocol implementations include mutation-based fuzzing approaches [? ? ?]. However, they require significant time to reach diverse code paths and identify vulnerabilities. DY Fuzzing [?] mutates network packets considering a Dolev-Yao adversary in the communication channel and combines domain knowledge for effective protocol testing. However, since the mutation scheme of these works does not consider the security property being tested or the current protocol state, they are inefficient in finding vulnerabilities compared to `Proteus`. Fiterau-Brostean et al. [?] developed a method to first learn an automaton from protocol implementations and then compare it against a catalog of manually obtained bug patterns, represented as deterministic finite automata (DFA) to identify vulnerabilities. However, this approach requires learning the state machine from the implementation, which is costly (§8). It is also limited to identifying only known vulnerabilities. The new bugs found in previously untested targets are mostly instances of known bugs found in other devices earlier. Further, another direction of work employs differential testing to find exploitable and interoperability issues [? ?]. Because of fixed sets of packets, they cannot explore vulnerabilities that require diverse packet fields and corresponding values.

Security testing of cellular protocols. Previous studies investigating the security of LTE implementations have typically focused on either OTA testing [? ? ? ? ?] or the analysis of baseband firmware [? ? ? ? ?]. Park et al. [?] have developed a negative testing framework along with a set of extensive test cases to detect vulnerabilities in UE devices. However, these test cases are manually designed and statically generated instead of being generated dynamically depending on the implementations. Kim et al. [?] introduce a semi-automated approach for fuzzing the LTE control plane, which relies on some basic security properties. Researchers have also used NLP to generate test cases from cellular specifications [? ?].

Security testing of BLE protocols. Frankenstein [?] utilizes advanced firmware emulation techniques to fuzz firmware dumps, enabling the direct application of fuzzed input to a virtual modem. FirmXRay [?] proposes static binary analysis tools to find the security issues caused without running the firmware. On the other hand, ToothPicker [?] focuses on one platform, providing host fuzzing techniques for this platform. However, these works do not perform stateful fuzzing. BLESA [?] utilizes ProVerif [?] to perform formal verification of BLE protocols. InternalBlue [?] performs reverse engineering on multiple Bluetooth chipsets to test and find vulnerabilities. BLEScope [?] identifies misconfigured devices by analyzing the companion mobile apps. SweynTooth [?] offers a systematic testing framework to fuzz BLE implementations. However, none of these works consider stateful and property-focused testing. BLEDiff [?] develops an automated black-box protocol noncompliance testing framework that can uncover noncompliant behaviors in BLE implementations. However, their approach cannot control the number of test cases generated and be carried out within a time budget t.

14 CONCLUSION AND FUTURE WORK

We design `Proteus`, a black box, protocol state machine and property-guided, and budget-aware automated testing approach for discovering logical vulnerabilities in wireless protocol implementations. `Proteus` leverages guidance from a PSM and also security properties to efficiently generate traces that can detect more vulnerabilities using much less amount of OTA queries than existing works. In the future, we will use `Proteus` to analyze other wireless protocols.

ACKNOWLEDGEMENTS

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

A BRIEF DESCRIPTION OF 4G LTE AND BLE PROTOCOLS

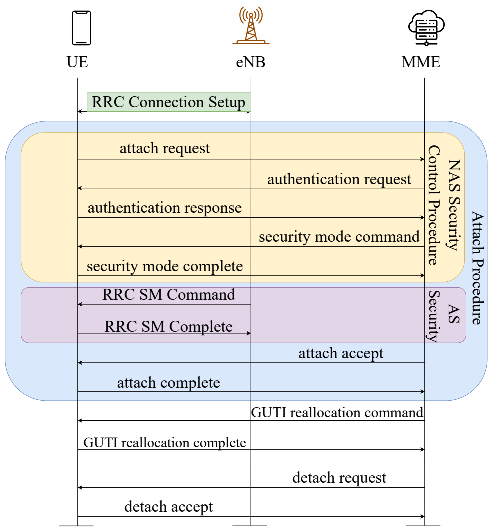


Figure 9: Regular flow of NAS and RRC messages in LTE protocol.

A.1 4G Long Term Evolution (LTE) Protocol

The 4G LTE is currently the most widely used cellular technology globally. The user devices on LTE are called User Equipment (UE), which includes identity information and cryptographic keys. On the network side, eNodeB acts as the base station that provides the radio connection. Moreover, the core network consists of several important components that provide different functionalities in LTE. Among them, the Mobility Management Entity (MME) is responsible for UE's attaching to the network.

LTE protocol stack, at the lowest level, has the Physical Layer. Sequentially, the next layers are the Medium Access Layer (MAC), Radio Link Control (RLC), Radio Resource Control (RRC), Packet Data Convergence Control (PDCP), and Non Access Stratum (NAS) layer, respectively. Among these layers, RRC and NAS are responsible for establishing radio connections, UE's attaching to the network and updating the location of the UE within the network. In this work, we primarily focus on the attach procedure.

The attach procedure starts with UE sending an ATTACH_REQUEST message to the network. Then, after proper authentication via the exchange of AUTHENTICATION_REQUEST and AUTHENTICATION_RESPONSE messages, the MME establishes the NAS security mode control procedure via the exchange of SECURITY_MODE_COMMAND and SECURITY_MODE_COMPLETE messages and finally completes the attachment via the exchange of ATTACH_ACCEPT and ATTACH_COMPLETE messages. In the RRC layer, AS security context messages are established via the exchange of RRC_SECURITY_ MODE_COMMAND and RRC_SECURITY_ MODE_ COMPLETE messages between the UE and the base station (eNB). Additionally, the MME can identify information from the UE through IDENTITY_REQUEST message, to which the UE should reply as per the specification, based on the state and identity type requested. After attachment, the MME also periodically changes the temporary identifier of the UE via the exchange of GUTI_REALLOCATION_COMMAND and GUTI_REALLOCATION_COMPLETE messages. A sequence diagram of the messages is given in Figure 9.

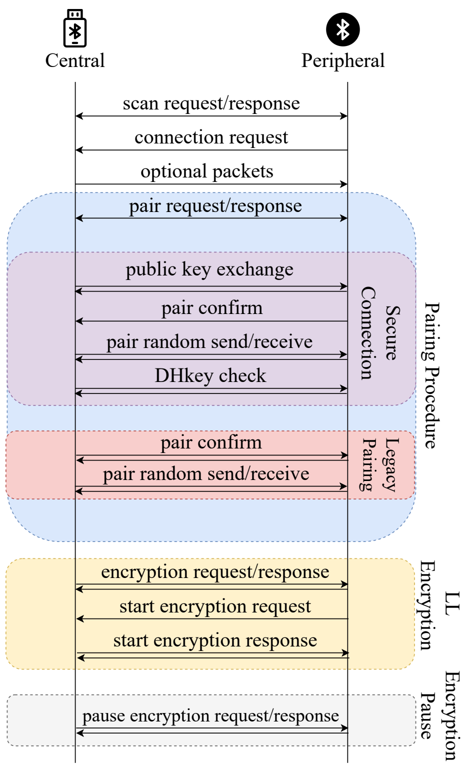


Figure 10: Regular flow of BLE protocol.

A.2 Bluetooth Low Energy (BLE) Protocol

For short-range communication, Bluetooth is one of the most popular technologies, being widely adopted in numerous devices, including smartphones, IoT devices, and peripherals. Among different Bluetooth technologies, BLE provides an energy-efficient communication protocol appropriate for low-cost, energy-restricted devices.

The BLE protocol stack has two distinct subsystems– *controller* and *host*. The controller subsystem includes the physical layer, the link layer (LL), and the host-controller interface. On the other hand, the host includes Logical Link Control and Adaptation Layer Protocol (L2CAP), Security Manager Protocol (SMP), Attribute Protocol (ATT), Generic Attribute Protocol (GATT), and Generic Access Protocol (GAP).

A BLE pairing and bonding process (As shown in Figure 10) starts by establishing the Link Layer connection via the SCAN_REQ, SCAN_RESP and CONNECTION_ REQUEST message between the central and peripheral devices and exchanging several optional packets (e.g., VERSION_ REQUEST, LENGTH_REQUEST, MTU_REQUEST message) to determine different connection parameters. After that, the pairing procedure takes place with PAIRING_REQUEST and PAIRING_RESPONSE messages. Depending on the type of pairing— legacy pairing or secure connections— different sets of messages are exchanged. For legacy pairing, SM_CONFIRM and SM_RANDOM messages are exchanged. For secure connections, in addition to these messages, PUBLIC_KEY_EXCHANGE and DH_KEY_CHECK

Table 12: List of tested LTE devices and identified vulnerabilities. *: The device is known to have the vulnerability.

Device Name	SoC Model	Baseband Version	Identified Vulnerabilities
Huawei P40 Pro	Kirin 990 5G	21C93B373S000C000	L-E9
Hisense F50+	Tiger T7510	5G_MODEM_20C_W21.12.3_P5	L-E3, L-E4
Galaxy S21	Exynos 2100	G991BXXU5CVF3	L-E1, L-E2, L-E7
Pixel 6	Google Tensor	g5123b-116954-230524-B-10194842	L-E1, L-E2, L-E7
Pixel 7	Google Tensor G2	g5300q-230626-230818-B-10679446	L-E2, L-E5, L-E7
Xperia 10 IV	Snapdragon 695	strait.gen-01223-04	L-E6, L-E8, L-O1
HTC One E9+	Helio X10	1.1506V24P22T34.2103.0805_AD5W	L-E1, L-E2
Nexus 6P	Snapdragon 810	angler-03.88	L-E3*, L-E6, L-E8, L-O1
Galaxy A71	Snapdragon 730	A715WVLU4DVI3	L-E6, L-E8, L-O1
Pixel 3a	Snapdragon 670	g670-00042-200421-B-6414611	L-E3*, L-E6*, L-E8, L-O1
Huawei P8 Lite	Kirin 620	22.300.09.00.00	L-E3, L-E4, L-E7

Table 13: List of tested BLE devices and identified vulnerabilities. *: The device is known to have the vulnerability.

Device Name	Vendor	BLE Version	Identified Vulnerabilities
DT100112	Microchip	4.2	B-E1*, B-E3*, B-E4, B-E5*, B-E7, B-E8*, B-E9, B-O1, B-O3
ESP32-C3	Espressif	5.0	B-E3*, B-E7, B-E10, B-E11, B-I1, B-O3
Hisense F50+	Hisense	4.2	B-E3, B-E4, B-E10, B-E11, B-O3
Galaxy S10	Samsung	5.0	B-E3, B-E4, B-E7, B-I1, B-O1, B-O3
Galaxy S6	Samsung	4.1	B-E1*, B-E3*, B-E4, B-E7, B-E11, B-O1, B-O3
Galaxy A22	Samsung	5.0	B-E2, B-E3, B-E4, B-E6, B-E7, B-E10, B-I1, B-O1, B-O2, B-O3
Pixel 6	Google	5.2	B-E4, B-E7, B-E11, B-I1, B-O1, B-O3
Pixel 7	Google	5.2	B-E3, B-E4, B-E7, B-E11, B-I1, B-O1, B-O3
Xperia 10 IV	Sony	5.1	B-E3, B-E7, B-O1, B-O3
OPPO Reno7 Pro 5G	OPPO	5.2	B-E1, B-E2, B-E3, B-E4, B-E6, B-E7, B-E10, B-E11, B-I1, B-O1, B-O2, B-O3
Motorola Edge+ (2022)	Motorola	5.2	B-E3, B-E7, B-I1, B-O1, B-O3
Laptop	Lenovo	BTstack 1.5.6.3 with BLE 5.2	B-E7, B-E9, B-I1, B-O3

Table 14: Glossary of symbols used.

Symbol	Meaning
$\mathcal{M}$	Guiding protocol state machine (PSM)
Q	The set of states in a PSM
Σ	The alphabet of input symbols
Λ	The alphabet of output symbols
q	A single protocol state. q_{init} denotes the initial protocol state, q_c denotes the current protocol state, q_n denotes the destination/next protocol state of a transition
$\mathcal{R}$	The set of transitions in a PSM
α	Input symbol of a transition in a PSM
γ	Output symbol of a transition in a PSM
π	A single trace/trace skeleton. π^a denotes an abstract trace/test skeleton, whereas π^c denotes an instantiated trace. π_i denotes the i^{th} element or observation of a trace.
ϕ	A single security property
Φ	A set of security properties
μ	Mutation budget (maximum number of perturbations to be applied in a good protocol flow)
λ	Length budget (maximum size of a test case or length of a message sequence)
β	Testing budget. Is a combination of (μ, λ)
l_j	j^{th} observation in the testing skeleton
$\mathcal{I}_P$	Target protocol implementation under test

messages are exchanged. Finally, the Link Layer encryption is established via the exchange of ENCRYPTION_REQUEST and ENCRYPTION_RESPONSE messages, and the devices' pairing and bonding are completed. The encryption can also be paused between two paired devices via PAUSE_ ENCRYPTION_REQUEST and PAUSE_ ENCRYPTION_RESPONSE messages.

B ALGORITHM FOR PROTEUS

Algorithm 1 Approach of `Proteus`

```
Input:
    $\mathcal{I}_P$ : Implementation under test
    $\mathcal{M}$: Guiding PSM
    $t$: Max number of traces that can be tested within testing budget
    $\Phi$: Set of desired properties to test
    $\mu$: Mutation Budget
    $\lambda$: Length Budget
Output: A list of traces followed by $\mathcal{I}_P$ violating any security property $\phi \in \Phi$
 1: procedure TESTPROTOCOL
 2:   $\Psi = \emptyset$                                  ▷ Violating Skeletons For All Properties
 3:   for each $\phi \in \Phi$ do
 4:     $\Psi = \Psi \cup$ GetSkeletons($\phi$)                              ▷ RegExGenerator
 5:   end for
 6:   $\mathcal{T}_{all} = \emptyset$                                      ▷ Set of all traces to test
 7:   for each $\sigma_v \in \Psi$ do
 8:     $\mathcal{T}_{all} = \mathcal{T}_{all} \cup$ GetTraceSet($\mathcal{M}$, $\sigma_v$, $q_{init}$, $\mu$, $\lambda$)          ▷ TraceBuilder
 9:   end for
10:   numTraceTested = 0
11:   while numTraceTested ≤ t do                                  ▷ TraceDispatcher
12:     $\gamma$ = SelectTraceToTest($\mathcal{T}_{all}$)                                ▷ Scheduler
13:     $\mathcal{R}$ = ExecuteTrace($\mathcal{M}$, $\mathcal{I}_P$, $\gamma$)       ▷ Execute Trace and Observe Response
14:     if $\gamma$ violates any security property $\phi$ then
15:       Report $\gamma$ violates security property $\phi$
16:     end if
17:     StoreFeedback($\gamma$, $\mathcal{R}$)
18:     numTraceTested = numTraceTested + 1
19:   end while
20: end procedure
```

C PROTEUS'S EFFECTIVENESS AND EFFICIENCY WITH INCORRECT PSM

We also test what would happen in case of incorrect behavior in the guiding PSM due to human errors during its construction. For this experiment, we intentionally altered our guiding PSM for LTE, and deleted the transition for SECURITY_MODE_COMMAND/ SECURITY_MODE_COMPLETE. We run with the same property discussed in the previous section (§11). We find that with mutation budget 1 and length budget 8 there is no trace generated, since now from the guiding PSM at least 2 mutations are required to satisfy σ_v, one to trigger the replayed GUTI_REALLOCATION_COMMAND message and one to place the SECURITY_MODE_COMMAND/ SECURITY_MODE_COMPLETE observation. Consequently, if we run the same example with mutation budget 3 and length budget 9, we get 1920 test traces which also contain a trace that generates the counterexample that is accepted by a practical protocol implementation with the vulnerability. Without the error, only 121 traces are generated with mutation budget 2 and length budget 8 that contains the vulnerability. Thus we conclude that even with the introduction of the error `Proteus` can detect the vulnerability, but the error affects its efficiency and we have to run TraceBuilder with a higher mutation and length budget to detect it.