Stateful Analysis and Fuzzing of Commercial Baseband Firmware

Ali Ranjbar
Penn State University
aranjbar@psu.edu

Tianchang Yang
Penn State University
tzy5088@psu.edu

Kai Tu
Penn State University
kjt5562@psu.edu

Saaman Khalilollahi
Penn State University
saaman1377@gmail.com

Syed Rafiul Hussain
Penn State University
hussain1@psu.edu

***Abstract*—Baseband firmware plays a critical role in cellular communication, yet its proprietary, closed-source nature and complex, stateful processing logic make systematic security testing challenging. Existing methods often fail to account for the interdependencies between baseband tasks and the statefulness of input processing logic, limiting their scope and effectiveness. We present LORIS, a stateful fuzz testing framework designed to explore and analyze baseband firmware implementations effectively. We employ *iterative symbolic analysis* to progressively identify state variables and the predicates over them that define different protocol states, while alleviating the state explosion problem. It enables LORIS to perform targeted exploration and fuzzing of program regions with high potential for vulnerabilities. We evaluated LORIS across 5 commercial devices from two major vendors, covering both 4G Long-Term Evolution (LTE) and 5G New Radio (NR), demonstrating its broad applicability. Our testing revealed 7 new vulnerabilities exploitable by over-the-air attackers, potentially leading to baseband crashes, remote code execution, and denial of service.**

1. Introduction

The widespread deployment of cellular networks provides mobile connectivity for a wide range of user devices, including smartphones, IoT devices, and other smart technologies. Critical to this cellular connectivity is the baseband modem, which is responsible for processing signals and managing data flow between devices and the network infrastructure. Despite its importance, baseband security is often overlooked due to its closed nature and operating under licensed spectrum. To make matters worse, as an embedded system, the baseband modem is typically written in memory-unsafe languages, such as C or C++. This exposes the modem to memory corruption vulnerabilities, such as buffer overflows and use-after-free, which the attackers can exploit to perform remote code execution and denial-of-service attacks. Adding to this, cellular communication protocols [1], [2], [3], [4] are highly complex, comprising numerous protocol states and the complexity leads to memory errors that are often only exploitable in a stateful manner. Since baseband firmware is directly exposed to cellular network communications, attackers can exploit such vulnerabilities using malicious over-the-air (OTA) messages [5], [6], [7]. Recognizing these challenges, we focus on *systematically analyzing memory-safety violations in baseband firmware when exposed to unexpected or malicious OTA messages.*

Due to basebands' proprietary and closed-source nature, previous security analysis on basebands mostly resort to black-box OTA testing [8], [9]. However, this approach is prohibitively inefficient, taking $\sim$1 minute to execute just one test case over-the-air. More importantly, OTA testing can only observe input/output behaviors and cannot introspect basebands' internal states. As a result, existing OTA testing-based approaches fail to identify memory corruption vulnerabilities, their root causes, or potential security implications. To avoid the challenges of OTA testing, analysis of firmware implementations relies heavily on reverse engineering and emulation [5], [6], [7], [10], [11], [12], [13], [14]. Commercial firmware implementations are, however, highly complex, comprising numerous internal tasks and peripherals (typically close to 100 tasks in 4G basebands and over 150 in 5G), each having varying expected message formats and performing different functions like processing OTA messages, managing hardware drivers, and retrieving other critical information. These tasks have intricate interdependencies and complex interactions with peripherals that complicate emulation efforts. Even if emulation is feasible, testing a specific protocol while running all tasks simultaneously fails to capture state and task interactions, ultimately missing stateful bugs that only arise under specific conditions.

Previous Work. Due to these challenges, prior attempts to test baseband security have been limited in scope, overlooking complex program states and requiring extensive manual setup and harnessing, as summarized in Table 1. FirmWire [10] emulates the baseband as a whole while disabling some components (e.g., SIM task) and injects unexpected messages through manually-created harnesses. However, this approach overlooks the stateful nature of the baseband, resulting in low coverage, and its reliance on manual harnessing limits its extensibility. BaseSAFE [10] tests individual functions one at a time, demanding substantial effort to harness the correct parameters and construct valid input formats. This approach cannot detect flaws that arise across multiple functions, such as errors in data passed from decoding functions to subsequent processing stages. Lastly, BaseComp [15] uses static analysis to exclusively test integrity protection functions, but it is limited in scope and does not apply to other tasks.

TABLE 1. COMPARISON WITH EXISTING BASEBAND SECURITY WORK

Approach	Full Task	State-Aware	Manual Effort	Target Layer
FirmWire [10]	✓	✗	TI, SH	RRC (4G), SM, CC (2G)
BaseSAFE [12]	✗	✗	TI, FH	NAS, RRC (4G)
BaseComp [15]	✗	✗	TI, FH	NAS (4G)
LORIS	✓	✓	TI	NAS (4G, 5G)

M: MediaTek, *S*: Samsung

TI: task identification, *SH*: state harnessing, *FH*: function harnessing

Methodology. To address these limitations, we design and implement LORIS, a stateful and grammar-aware fuzzing framework designed to thoroughly analyze baseband firmware. We observe that tasks within baseband firmware are generally organized by the specific cellular protocol they handle. Thus, LORIS focuses on a single task at a time, specifically, the task responsible for processing the targeted OTA messages, while accounting for this task's complex internal states and its interdependencies.

LORIS identifies *state variables*, which store state information for the target task, by observing variables that persist across message processing iterations and follow the *use-before-define* principle. Essentially, LORIS considers non-local variables within the message processing loop as potential state variables if they are *assigned* within the loop (i.e., not constants or configuration values), and are *used* in an iteration either before or without being *defined* (i.e., assigned) in that iteration. This usage pattern indicates that the variable's value either persists from a previous iteration or is assigned by another task. These state variables reflect both internal state maintained by the task (e.g., the outcome of prior message handling) and results from interdependent tasks (e.g., outputs from other message handlers or peripheral interactions). Their values often govern branching decisions, enabling or restricting access to specific code regions based on runtime conditions, such as protocol state (e.g., whether a security context is established) or peripheral status (e.g., a SIM card read completed). LORIS treats these variables as effective proxies for the task's state.

LORIS then uses symbolic analysis to extract *state preconditions* that these state variables must satisfy for an OTA input to reach different code regions. This enables LORIS to configure various program states prior to injecting fuzzing inputs, facilitating the effective exploration of various program regions to significantly improve code coverage. To manage the hundreds of state variables in a task and effectively mitigate the state explosion problem during symbolic execution, we develop a novel *iterative symbolic execution* approach. In each iteration, LORIS analyzes the regions it can explore with the current symbolic values while also adding new state variables as symbolic to gradually open up more execution paths. This iterative approach enables LORIS to collect state preconditions for existing symbolic state variables while discovering additional state variables in newly accessible paths. Additionally, it allows LORIS to reuse prior analysis results on branches unaffected by the newly introduced symbolic values to progressively build results. LORIS employs a ranking mechanism to prioritize state variables that are computationally less expensive and likely to unlock more code regions in each iteration.

After iterative symbolic execution, LORIS generates concrete instantiation values for the state variables that satisfy the identified state preconditions. These values are stored as memory snapshots, setting the task's state prior to fuzzing. To explore the task effectively in each state, LORIS employs a grammar-aware input generator capable of producing specification-compliant messages.

Findings. We evaluated LORIS on 5 proprietary, closed-source commercial baseband binaries from Samsung and MediaTek, supporting both 4G LTE and 5G NR. These basebands are used in devices from two smartphone vendors, Google and Samsung. LORIS uncovered 8 previously unknown vulnerabilities, 7 of which are exploitable by OTA messages. These vulnerabilities may lead to sensitive data leakage, remote code execution, and denial-of-service attacks, impacting user connectivity and potentially compromising user privacy. We successfully reproduced all exploitable vulnerabilities on real devices via OTA messages.

Responsible Disclosures. We reported all found vulnerabilities to the respective vendors. All 8 vulnerabilities were confirmed, with 2 rated as high severity and 1 as critical.

Open-Sourcing. We make LORIS publically available at https://github.com/SyNSec-den/Loris.

Contributions. In summary, our main contributions are:

- We develop LORIS, a state-aware analysis framework tailored for effective testing of baseband firmware.
- We design and implement a novel *iterative symbolic analysis* to detect state variables and extract their preconditions to guide LORIS's testing while mitigating the state explosion problem.
- We are the first to support reverse engineering and emulation of commercial 5G basebands.
- We evaluate LORIS on 5 commercial devices from 2 vendors, covering both 4G LTE and 5G NR, demonstrating LORIS's broad applicability and effectiveness. LORIS uncovered 7 new exploitable issues that may affect user connectivity and compromise user privacy.

2. Background

Baseband Architecture. Besides the application processors that run the mobile operating system, modern smartphones also have dedicated baseband processors (BPs) responsible for handling cellular communication with network base stations. These BPs facilitate essential wireless connectivity, cellular communication, and data transmission. To ensure predictable behavior and meet the strict real-time requirements of cellular communication, the baseband processor operates using a real-time operating system (RTOS) as its firmware. For conciseness, we may refer to baseband firmware simply as "baseband" throughout this paper. The RTOS runs various cellular tasks, each implementing a never-ending loop to continuously receive and process new messages as long as the system is active. For example, Samsung's RTOS ShannonOS implements the Non-Access Stratum (NAS) stack in an `NASOT` task. While OTA messages processed by the baseband follow cellular specifications, the

proprietary nature of baseband means that details of its internal task implementation and inter-task messaging formats are typically undisclosed, requiring reverse engineering the firmware to understand its operation and assess the security of cellular communication systems [5], [6], [16].

Baseband Emulation. Emulation enables the execution of binaries compiled for different architectures, making it valuable for analyzing proprietary baseband firmware. Emulating a BP facilitates the execution of its firmware outside the physical device, allowing controlled access to its core functionality in a virtualized environment. Prior research has demonstrated that emulation is highly effective for security testing of embedded firmware [10], [12], [17], [18], [19]. Unlike OTA testing, which requires physical device interaction, emulation enables direct introspection of the firmware's internal state, allowing for detailed monitoring, debugging, and analysis of security issues as they emerge. This capability supports in-depth analysis of discovered vulnerabilities and insights into firmware behavior, providing a more comprehensive approach to security testing. QEMU [20] is an open-source platform that supports full-system emulation of various architectures and hardware peripherals, and is widely used in baseband research [10], [12], [17], [18].

3. Threat Model & Motivation

3.1. Threat Model

We consider Dolev-Yao style adversaries who can inject, drop, eavesdrop, or modify messages using fake base stations [21], Man-in-the-Middle (MitM) relays [22], or signal injectors [23]. These adversaries overpower legitimate base stations with stronger signals, luring baseband to connect to them or injecting malformed signals into the communications. Additionally, as 5G networks become more widespread across diverse use cases, the increase in smaller providers, disaggregated RAN systems [24], and home network devices (e.g., home base stations) introduces additional risks. These emerging systems and devices often lack stringent security guarantees or even carry backdoors and malicious intent, further exposing connected user equipment to potential threats [25], [26]. Consequently, we consider both pre-authentication messages and those that are integrity-protected and/or authenticated, addressing potential risks from both malicious and legitimate sources.

Scope of Testing. LORIS can be used to test different layers' OTA messages in both 4G and 5G. This work focuses on the NAS layer's messages exchanged between the core network and UE. The NAS protocols and messages are critical for the overall security and session management in cellular networks, with complex message structures and numerous protocol states that make thorough exploration challenging. We focus on identifying and mitigating memory corruption vulnerabilities, such as buffer overflows and use-after-free errors. We do not aim to detect logical errors such as authentication or access control violations.

3.2. Motivating Example

Baseband firmware implementations consist of multiple tasks running concurrently, each responsible for handling different internal or OTA messages. To illustrate the motivation behind LORIS's design, we present a previously unknown vulnerability LORIS detected in a Pixel Exynos baseband firmware (V4 in Table 4). Consider the simplified code example of a single task responsible for processing 5G NR NAS protocol messages, extracted through reverse-engineering, as shown in Listing 1. Before entering the main processing loop, the task undergoes initialization (line 4 in Listing 1), where several variables that control the processing of the received messages are defined and initialized. These variables include `MmProc` and `MmAS`, which are later used on lines 17 and 18, respectively. `MmProc` represents the current state of the 5GS Mobility Management (5GMM) procedure, the protocol used for the registration, deregistration, mobility, and security between the baseband and AMF (Application and Mobility Function) in the core network. `MmAS` denotes the status of the Access-Stratum (AS) connection between the baseband and base station managed through Radio Resource Control (RRC) layer messages. Both variables are initialized to zero, indicating the initial states of no active procedure or connection. We identify these variables as state variables because they capture the task's protocol state, and their values directly influence the subsequent handling of messages. Other variables may also initialize during this phase, such as `msg_type` which temporarily stores the type of the received message, and function pointers for decoding function dispatchers (line 34).

After the initialization, the task executes the main message-handling loop, where the task continuously waits for, receives, and processes incoming messages (lines 5- 10) in a loop. Upon receiving a NAS message, the task first performs several validation checks, e.g., checks the values of related state variables (lines 17–18). The task processes the radio message (line 19) only when the baseband has started the 5GMM procedure and the connection with a base station has been established (i.e., an AS connection exists). When processing the received message, the task first verifies the validity of the message in `CheckSecCompliance`, where several other state variables are involved (line 24). For example, it may validate the security header type based on the current status of the security context, which is stored in a state variable. Only after all checks are successful, the received radio message is decoded and further processed (line 25). The decoding function, `RadioMsgDecode` (lines 30-36), processes each information element (IE), i.e., the fields in the OTA message, by iterating through the elements and dispatching a corresponding decoder function for each type (i.e., a function pointer to the corresponding decoder).

The vulnerable `DecodeExtEmergNumList` function is invoked by the indirect call on line 34. It decodes the extended emergency number list IE [2], which contains local emergency numbers that callers can use to reach emergency services. The loop on lines 40-44 iterates through each emergency number information block in the input buffer,

```
1  // The entry function of NASOT task
2  void NasotMain() {
3    Task_Msg_t *msgPtr;
4    NasotInitialize(); // MmProc=0, MmAS=0, msg_type=0
5    do {
6      int err = pal_MsgReceiveMbx(NASOT_QID, &msgPtr);
7      if (!err)
8        ExtMsgHandler(msgPtr);
9      PostProcessMsg();
10   } while (true);
11 }
12
13 // Handles messages based on the message type
14 void ExtMsgHandler(Task_Msg_t *msgPtr) {
15   msg_type = msgPtr->group >> 8 & 0xff;
16   if (msg_type == RADIO_MSG)
17     if (MmProc != 5GMM_PROC_NULL &&
18         MmAS == 5GMM_IN_CONNECT)
19       ProcessRadioMsg(msgPtr->payload, msgPtr->plSize);
20   ...
21 }
22
23 void ProcessRadioMsg(byte *dedInfoNas, ushort size) {
24   if (CheckSecCompliance(dedInfoNas, size)) {
25     int err = RadioMsgDecode(dedInfoNas, size);
26     ... // further processing
27   }
28 }
29
30 int RadioMsgDecode(byte *msg, ushort size) {
31   byte typ = GetNrmmMsgType(msg);
32   IeCb *cb = GetNrmmMsgCodec(typ);  // IE Control Block
33   if (cb->decoder != 0) {
34     cb->decoder(msg, size);  // call
       DecodeExtEmergNumList by pointer
35   }
36 }
37
38 int DecodeExtEmergNumList(byte *buf, ushort size) {
39   byte idx = 0;
40   do {
41     EmergNumInfo *info = (EmergNumInfo *)(buf + idx);
42     idx = idx + info.length + 1;  // int overflow
43     ...  // save and process info block
44   } while(idx < size)
45 }
```

Listing 1. Simplified Code for the NASOT Task in Pixel Baseband

buf, using the length of each block to locate the offset to the next block. The length of each block is stored in its first octet, and since the specification does not enforce bounds on this field, its value can range from 0 to 255. However, on line 42, a missing bounds check on the length field leads to an integer overflow vulnerability that results in an infinite loop. For example, if the length of the first block is 255, the 8-bit integer idx overflows. The overflow results in the value of idx wrapping back to 0 after the calculation, causing the decoder function to be trapped in an infinite loop.

In summary, uncovering this vulnerability requires the following capabilities. *(1) Identify the relevant task:* Locate the specific task responsible for processing OTA messages within the baseband's complex task interactions (§4.1); *(2) Understand task dependencies and state space:* Comprehend the impact of dependent tasks, such as MmProc and MmAS, and their state space to ensure the test input is accepted and reaches the vulnerable logic (§4.2); *(3) Resolve function pointers:* Resolve indirect calls to the vulnerable decoding function (§4.3); *(4) Craft syntactically and semantically correct inputs:* Generate inputs that follow both syntactic and semantic requirements to pass initial decoding checks (§7); *(5) Introduce appropriate mutation in the triggering field:* Produce the specific length field value required to trigger the vulnerability (§7).

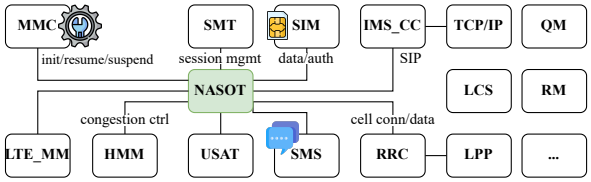


Figure 1. Dependencies of the NASOT Task in a Samsung Baseband

4. Challenges of Designing LORIS

§3.2 demonstrates that to effectively uncover vulnerabilities in a baseband firmware, LORIS has to navigate through several challenges. We detail a few critical ones below.

4.1. C1: Target Task Identification & Isolation

For ease of exposition, the motivating example presented in Listing 1 shows the simplified version of a single NASOT task in baseband firmware. In reality, each baseband firmware contains over 100 tasks, each handling various functionalities such as processing OTA messages, managing connections, and interacting with hardware drivers. For instance, as shown in Figure 1, the operation of the NASOT task relies on numerous other tasks that manage various messages and protocols (e.g., RRC, SMS), or interact with hardware drivers (e.g., SIM). Testing the entire baseband as a whole presents significant challenges. Many tasks, especially those interfacing with peripherals, cannot be fully emulated without manually implementing support for the missing functionalities. Creating stubs that mimic the expected functionalities of these tasks is primarily manual and hence time-consuming, error-prone, and unscalable, as each task may further rely on other tasks, creating an extensive chain of dependencies. As a result, prior approaches [10] often disable these tasks during testing, which prevents accurate reproduction of the behavior of dependent tasks. This leads to incomplete exploration of task states and the inability to uncover deeply rooted stateful bugs. Moreover, without the ability to generate test inputs for all tasks or resolve inter-task dependencies, previous efforts suffer from poor code coverage and are unable to thoroughly analyze the complex interactions that occur within the baseband.

To address these challenges, we isolate and test individual tasks instead of emulating the entire baseband, while still accounting for task dependencies (§ 4.2). This task-level isolation narrows the testing scope and improves efficiency by eliminating unnecessary dependencies, reducing execution time, and avoiding irrelevant message handling. Testing all tasks individually, however, is impractical because some tasks serve only internal functionalities and must use vendor-specific message formats. The proprietary nature of baseband firmware means that there are no publicly available technical documents that specify the task divisions or the expected message formats. Consequently, identifying

each task's message format requires labor-intensive reverse engineering efforts, which we cannot apply to all tasks.

To address this challenge and find the proper task to isolate and test, we observe that baseband tasks are generally organized according to the specific cellular protocol layer they handle, with each protocol layer associated with a particular task. Leveraging this insight, LORIS narrows its focus to testing only those tasks responsible for processing the targeted OTA messages. This approach simplifies task identification and ensures that any detected vulnerabilities are exploitable through externally controlled messages. This aligns with the threat model discussed in §3.1, where attackers are limited to manipulating OTA messages. Furthermore, this method enables us to refer to well-defined cellular specifications [2], [3] to generate grammatically valid radio messages as test inputs, eliminating the need to reverse engineer internal message formats from baseband binaries.

4.2. C2: State and Dependency Management

As demonstrated in the motivating example, baseband tasks are typically highly stateful, posing significant challenges for effective testing. The processing path of any received messages by a task depends on the values of state variables, i.e., variables whose values are initialized before the message is handled. These values may be assigned from baseband configurations or boot-time initialization, the results of previously processed messages within the same task, or through the processing of other dependent tasks. For example, in Listing 1, the branch conditions on lines 17 and 18, and the validation on line 24 all depend on values of different state variables. If any of these branch conditions are not satisfied (i.e., the values of the corresponding state variables are not appropriately assigned before the test input is received), the vulnerable site on line 42 is not reachable.

To navigate these different states and explore various execution paths and code regions, existing stateful testing efforts [8], [9] typically rely on recording sequences of inputs that prepare the program to a particular state, i.e., input sequences that lead to the assignment of appropriate values to these state variables, before sending the test input. However, applying this strategy to emulated baseband firmware is impractical. In baseband systems, message processing is distributed across multiple interdependent tasks, which must each be in specific states for the sequence to succeed. As discussed in § 4.1, many of these tasks cannot be emulated, leading to unresolved state dependencies. To work around this, prior baseband fuzzing efforts [10] depend on extensive manual reverse engineering, harnessing, and annotation to manually set the task state before launching fuzzing tests. This manual approach only enables exploration of a limited subset of states and does not scale. As a result, achieving high coverage and uncovering deep, state-dependent vulnerabilities remains a significant challenge that prior efforts are unable to effectively address.

To address this challenge, we observe that directly assigning specific values to a target task's state variables effectively simulates different protocol states and enables controlled exploration of desired execution paths. In essence, these state variables act as preconditions that gate access to different program regions during message processing. Leveraging this insight, LORIS eliminates the need for the fuzzer to generate complex input sequences or for researchers to manually identify and satisfy preconditions. Instead, LORIS directly identifies and sets the values of relevant state variables, configuring the baseband into the desired state before dispatching test inputs. This approach also naturally handles task dependencies: We find that dependent tasks execute asynchronously without blocking the target task, with their outcomes reflected in the state variables used by the target task. For example, in Listing 1, line 18 references the `MmAS` state variable, which represents the AS connectivity state managed by the `RRC` task that `NASOT` depends on. The dependent task updates such state variables through inter-task communication, which the target task monitors and reacts to these updates. By directly setting these state variables, LORIS simulates the effects of dependent tasks' behaviors without actually executing them.

4.3. C3: Effective State Analysis

Symbolic execution is a powerful approach for analyzing how different values of state variables influence control flow by reasoning the branching conditions that govern access to different program regions, revealing the specific preconditions (i.e., constraints) state variables must satisfy to reach those paths. However, several challenges complicate the application of symbolic execution in baseband. (1) Accurately identifying true state variables in complex basebands is non-trivial. In the motivating example, some variables, such as `msg_type` (used on line 15), are defined outside the loop's scope but do not represent a persistent state, as they are reassigned in each iteration by the input message. Consequently, instantiating these variables prior to testing is ineffective, as their values are overwritten during execution. Analyzing such variables unnecessarily consumes computational resources. Identifying true state variables requires tracing paths from the message reception point to each variable's usage to verify that the variable retains its value without being reassigned along those paths. (2) Each baseband task contains more than 100 true state variables, and setting all these as symbolic values would immediately trigger the state explosion problem. This exponential growth in symbolic states makes the analysis infeasible in terms of memory, time, and computation requirements. (3) Baseband tasks extensively use pointers for function dispatch and structure references (e.g., the dispatcher on line 35 in Listing 1). Symbolic execution alone cannot effectively resolve these indirect references, making it difficult to accurately explore execution paths dependent on pointer values.

To address these challenges, LORIS employs a two-stage analysis approach. In the first stage (§6.1), LORIS performs dynamic execution up to the entry of the main message processing loop (e.g., up to line 6 in Listing 1). This dynamic execution serves two primary purposes: (1) it dynamically executes task initialization functions, so that

when symbolic execution begins, all pointer references are resolved. (2) LORIS hooks into all memory accesses during this phase to identify non-local variables defined outside the loop scope. While not all identified variables will be true state variables, this approach narrows down the set of variables to track subsequently. In the second stage (§6.2), LORIS performs *iterative symbolic analysis*. It progressively identifies true state variables, expands the set of symbolic state variables, and explores new code regions with each iteration. This incremental approach allows LORIS to reuse previously computed results, skipping branches unaffected by the newly added symbolic variable. Additionally, the iterative process enables LORIS to return meaningful results intermediately without analyzing all state variables, and instead prioritizing those with the most impact on execution paths. To further manage the state explosion problem, LORIS incorporates additional techniques, discussed in §6.4.

5. Design Overview

Figure 2 presents the overall architecture and workflow of LORIS. LORIS tests a single target task $\mathcal{T}$ at a time. $\mathcal{T}$ is a 2-tuple $\mathcal{T} = (\mathcal{I}, \mathcal{L})$, where $\mathcal{I}$ denotes the task initialization code, a sequence of instructions $\langle i_j, \ldots, i_k \rangle$ (e.g., line 4 in Listing 1), and $\mathcal{L}$ denotes the main input-reception and processing loop, a sequence of instructions $\langle i_m, \ldots, i_n \rangle$ that continuously receives and handles incoming messages (e.g., lines 5–10 and the corresponding invoked functions). The task $\mathcal{T}$ that LORIS targets must receive some OTA messages $\mathcal{M}$, which serves as test inputs for vulnerability analysis.

LORIS consists of three components: Analyzer, Fuzzer, and Emulator. Analyzer identifies the state variables of the target baseband task and analyzes the corresponding state preconditions (i.e., the constraints on these variables) that must be satisfied to reach different code regions. These constraints are then passed to a Satisfiability Modulo Theory (SMT) solver [27], which computes concrete values for the state variables that satisfy the constraints. The resulting state instantiations are shared with Fuzzer, which sets the corresponding state variable values before dispatching test inputs, effectively configuring the task into different desired protocol states, enabling the test inputs to penetrate different code regions only accessible under specific states. Additionally, Analyzer gathers path constraints on input messages, specifying the conditions that inputs must meet to drive deeper execution within each state. These constraints are integrated into Fuzzer's grammar-aware input generation, enabling it to selectively craft messages that achieve deeper exploration. Supporting both Analyzer and Fuzzer is LORIS Emulator, which prepares the baseband task for symbolic execution for state variable analysis or concrete execution for fuzzing. Analyzer extends a state-of-the-art baseband emulator [10], adding support for new processors and peripherals to enable the *first* emulation capable of handling 5G basebands with their new CPU architectures and major design changes comparing to their 4G predecessors (§8).

Analyzer. The goal of Analyzer is to determine various states $\mathcal{S}$ in which Fuzzer can deliver input $\mathcal{M}$ to the target task $\mathcal{T}$ to effectively explore vulnerabilities within the main input-reception and processing loop $\mathcal{L}$. Formally, we define a *state snapshot* $s \in \mathcal{S}$ as a concrete instantiation of state variables $\mathcal{V}$ that satisfies a set of *state preconditions* Π_s. Π_s represents a set of branch conditions that state variables need to satisfy and dictate the execution flow to specific code regions in $\mathcal{L}$. Formally,

$$\Pi_s = \bigwedge \pi_i(v_x, \ldots, v_y), \quad \text{where } v_x, \ldots, v_y \in \mathcal{V}$$

where π_i denotes an individual branch condition involving one or more variables $v_x, \ldots, v_y \in \mathcal{V}$. Each π_i is a constraint on the values of these variables that must hold true for the program execution to follow the desired path toward the target code region in $\mathcal{L}$. The set of all such conditions Π_s forms a *necessary state preconditions* for reaching certain regions of interest in $\mathcal{L}$. By identifying these state preconditions, Analyzer can determine the required assignments of $\mathcal{V}$, forming a state s such that when $\mathcal{M}$ is received, the execution flows through the intended branches.

$$s : \hat{\mathcal{V}}_{\Pi_s} = \{\hat{v} \in \mathcal{D}(\mathcal{V}) \mid \Pi_s(\hat{v})\}$$

where $\hat{\mathcal{V}}_{\Pi_s}$ is the instantiation of state variables satisfying a set of preconditions Π_s, and $\mathcal{D}(\mathcal{V})$ is the domain of possible assignments to state variables. Here, $\hat{v}$ is a valuation (i.e., assignment) over $\mathcal{V}$ that satisfies all constraints in Π_s.

Iterative Symbolic Execution. To identify the set of state variables $\mathcal{V}$ and the corresponding preconditions Π required to reach different code regions, while mitigating the state explosion problem, we develop a novel *iterative symbolic execution* (§6.2). This approach performs multiple complete iterations of symbolic execution of the target task, incrementally treating additional state variables as symbolic, thereby expanding the symbolic state space and enabling the exploration of code paths that were previously unreachable due to non-symbolic (concrete) branch conditions. As new code regions are explored, additional state variables influencing control flow are discovered and incorporated into subsequent iterations. Specifically, during iteration i, the symbolic execution engine treats a newly identified state variable $v_k \in \mathcal{V}$, discovered in a prior iteration j as symbolic, and re-executes the task starting from the entry point of the message processing loop $\mathcal{L}$. Previously discovered symbolic paths are reused when the newly added symbolic variable does not influence branching decisions. This reuse, combined with the iterative nature of the approach, enables a gradual expansion of the symbolic state space, avoiding the need to symbolically evaluate all branches in a single run, which would quickly lead to path explosion. The iterative process continues until state explosion occurs, even with the applied mitigation techniques and the incremental nature of the analysis (detailed in §6.4). Importantly, this design allows Analyzer to yield meaningful results even when terminated early, as partial sets of symbolic state variables and their conditions are often sufficient to expose valuable execution paths that are unreachable using traditional, non-iterative symbolic execution techniques.

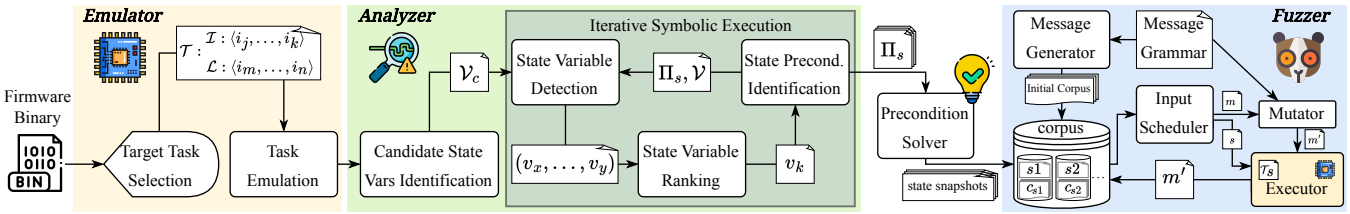


Figure 2. Architecture of LORIS

Checkpoint-Based Path Pruning. To further mitigate the state explosion problem, we introduce two novel techniques (§ 6.4): (1) first, we design *checkpoint-based path pruning* that prioritizes paths that are likely to lead to deeper and more meaningful execution when a state explosion is detected to balance the effort on exploration width and depth. (2) We also replace commonly used functions that are prone to state explosion with *emulated procedures* based on our domain knowledge from observed patterns in emulation.

Fuzzer (§7). During fuzzing, Fuzzer maintains a separate corpus c_s for each state snapshot $s \in \mathcal{S}$, where each corpus stores inputs that have triggered new code coverage when executed under that specific state snapshot. The mutator generates grammar-aware mutations enforcing both syntactical (i.e., structural) and semantical (i.e., value) constraints. For each fuzzing iteration, LORIS selects a state snapshot s, samples a message $m \in c_s$, applies mutations to obtain a new input m', sets $\mathcal{V}$ to concrete values specified in s to achieve a state-instantiated task $\mathcal{T}_s$, and sends m' for execution in $\mathcal{T}_s$. If m' leads to new code coverage, it is added to the corresponding corpus c_s. Moreover, we leverage the insight that an input m' leading to new covered code in state s may also result in new coverage in other states. Thus, whenever a new input m' is added to c_s, LORIS also tests m' under all other snapshots $s' \in \mathcal{S}$. If m' triggers new coverage in $\mathcal{T}_{s'}$, m' is likewise added to $c_{s'}$. This cross-state sharing reduces redundant effort across states and alleviates the problem of states dividing the focus of the fuzzer since discoveries of one state are reused across other states.

6. Identifying State Preconditions

As demonstrated in the motivating example (§3) and discussed in §4.2, baseband tasks are highly stateful. To efficiently explore the diverse behaviors of a stateful target baseband task, LORIS requires identifying state variables representing different protocol states and analyzing their impact on the reachability of various code regions in the task. To achieve this, we introduce Analyzer, which employs a novel dynamic state exploration technique that operates in two stages, candidate state variable identification (§6.1), and iterative symbolic analysis (§6.2).

6.1. Candidate State Variables Identification

One key characteristic of state variables is that their values persist across iterations of the task's message processing loop (e.g., lines 5–10 in Listing 1), which requires them to be non-local to the loop's scope. However, not all non-local variables qualify as state variables. Specifically, if a non-local variable is reassigned at the beginning of each iteration and used only after reassignment, it does not carry meaningful state information. In such cases, the variable's value prior to the current iteration has no influence on the program's execution path. For example, in Listing 1, the variable `msg_type` is updated with the type of each received message before it is used. As a result, its value does not persist across iterations and does not qualify as a state variable. Additionally, a variable that is never updated after its initialization is not a state variable, as its value is unaffected by OTA message inputs (e.g., the function pointer `cb->decoder` in Listing 1). To accurately identify true state variables, we formally define them as follows.

Definition 6.1 (State Variable). A variable v used within the main input-reception and processing loop $\mathcal{L}$ is a *state variable* (i.e., $v \in \mathcal{V}$) if it satisfies the following conditions: (1) v is a non-local variable to $\mathcal{L}$ (i.e., v is also defined in task initialization code $\mathcal{I}$); (2) v is updated within $\mathcal{L}$, i.e., there exists a write operation $w(v) \in \mathcal{L}$; and (3) for at least one execution path p from the reception of an input to the first read operation $r(v)$ in the current iteration, there exists no preceding write operation $w(v)$. Formally, v is a state variable iff $\exists p, r(v) : v \in \mathcal{I} \wedge \exists w(v) \in \mathcal{L} \wedge$ no $w(v)$ precedes $r(v)$ along p.

Since our focus is on how state variables influence the reachability of different code regions, we consider only those variables used in branch conditions that alter the task's control flow. A potential approach to automatically identify these state variables is through static data flow analysis [28] using dataflow and control-flow dependency graphs. However, existing static, binary-only analysis techniques struggle with resolving indirect calls and pointers, which are common in baseband implementations.

To address these challenges, Analyzer first performs dynamic execution of the baseband firmware until the target task reaches the input-waiting stage (e.g., line 6 in Listing 1). This process effectively executes all task initialization code $\mathcal{I}$. During this phase, Analyzer tracks all variable definitions from the entry point of the task's main function up to the start of the input-processing loop $\mathcal{L}$. It hooks into all memory operations, recording both read and write accesses to variables stored on the stack, heap, and global variables. Once these definitions are collected, Analyzer performs static analysis [29] to eliminate variables that are never referenced within $\mathcal{L}$ by analyzing their cross-references. The

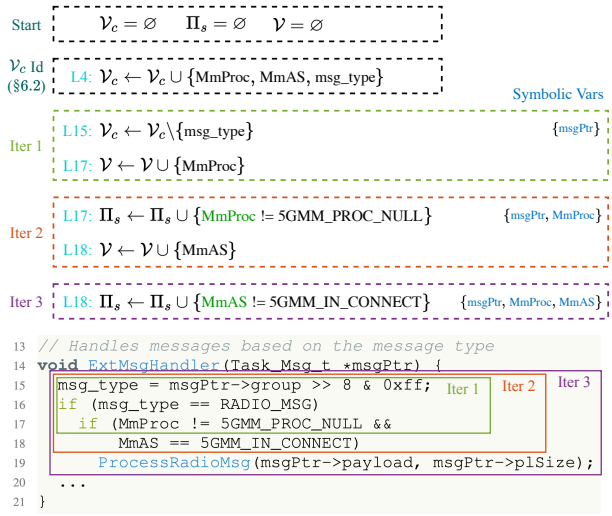
remaining variables are marked as *candidate state variables*, and their concrete values at the end of $\mathcal{I}$ are saved for use as concrete values for non-symbolic variables in the next stage. This dynamic execution phase also resolves relevant pointers and indirect call targets, allowing for accurate path exploration during symbolic execution. In a nutshell, this approach effectively narrows down the variables to track and enables the resolution of indirect pointers before symbolic execution. Note that candidate state variables may include variables that are not true state variables (e.g., `msg_type` in Listing 1) and LORIS resolves those in the next stage, i.e., iterative symbolic execution.

6.2. Iterative Symbolic Analysis

In symbolic analysis, variables can be treated as symbolic, allowing them to represent any value within their type domain. During analysis, Analyzer tracks how these symbolic variables affect task execution flow by collecting path constraints depending on these variables across all encountered branches. By resolving these collected constraints, Analyzer can determine the precise values required for state variables to follow specific execution paths when processing the input OTA message. However, Analyzer faces two key challenges. First, the candidate state variables identified in the previous stage are not necessarily true state variables. Analyzing non-state variables provides no meaningful insights, as these variables will be reassigned during execution, regardless of the initial value set pre-testing. Second, each task in the baseband firmware can contain more than 100 true state variables, and thousands of candidate state variables, making it infeasible to analyze them all as symbolic, as it would require a prohibitive amount of memory and computational resources due to the state explosion problem, so the analysis could not finish.

To address these challenges, Analyzer employs an *iterative symbolic execution* approach that progressively identifies more true state variables, prioritizes the most impactful ones to analyze, and gradually builds state preconditions required for these state variables to reach different code regions within the task. In the first iteration, only the input message is symbolic, and all other variables retain their concrete initialization values, as no true state variables have yet been identified. In this initial pass, Analyzer can only explore paths reachable by these concrete values, but it identifies true state variables along these paths. In each subsequent iteration, Analyzer introduces one additional state variable as a symbolic value based on a priority ranking (§6.3). This allows Analyzer to evaluate both directions of the branches influenced by the added symbolic variable, expanding its coverage and finding new state variables as it explores previously unvisited paths.

Specifically, symbolic execution begins at the OTA message reception point in the target task, where a potentially malformed or malicious message is received (e.g., line 6 in Listing 1). At each iteration, Analyzer applies depth-first search (DFS) with backtracking to systematically explore all possible execution paths enabled by the current symbolic



Each box marks the portion of ExtMsgHandler function explored during each iteration.

Figure 3. Illustration of Iterative Symbolic Analysis on Listing 1

variables. If a candidate variable is reassigned along a particular path, Analyzer temporarily excludes it from the candidate list for that path, re-adding it upon backtracking when examining other paths. This redefinition check ensures that only variables retaining their values across loop iterations are considered potential state variables.

For each encountered branch, Analyzer takes different actions depending on whether the branch condition involves a non-candidate variable, a non-symbolic candidate variable, or an already-symbolic variable. If the branch condition depends on a variable not in the candidate list and not symbolic, the branch is considered to be not affected by states and is resolved using the concrete value of the variable. If the branch condition involves a candidate state variable that is currently non-symbolic, Analyzer marks the variable as a true state variable. In the current iteration, it proceeds only along the path dictated by the variable's concrete initialization. In future iterations, once the variable becomes symbolic, Analyzer explores both directions of the branch. If the branch condition depends on an already-symbolic variable (either an input field or a symbolic state variable), Analyzer explores both paths. For each path, it records the corresponding preconditions, and upon completing one path, backtracks to explore the other. To improve efficiency and mitigate state explosion, Analyzer reuses results from previous iterations when both the current and all subsequent branches are unaffected by the newly added symbolic state variable. In such cases, it performs an early return, avoiding redundant exploration. However, if the current branch is unaffected but a future branch along the path depends on the newly added symbolic variable, Analyzer fast-forwards execution to that branch, reusing previous results along the unaffected segments. Once it reaches the impacted branch, it resumes symbolic analysis to explore the newly enabled code regions.

Illustration of Iterative Symbolic Execution. We demon-

strate the iterative symbolic execution process by showing how it operates on the motivating example in Listing 1 to identify state variables $\mathcal{V}$ and determine the preconditions Π_s to set state s that enable test inputs to reach the vulnerable `DecodeExtEmergNumList` function. Without this process, `DecodeExtEmergNumList` is not reachable by test inputs due to unmet checks with state variables, such as those on lines 17 and 18. Figure 3 illustrates this process. Before symbolic execution begins, Analyzer identifies `MmProc`, `MmAS`, and `msg_type` (line 4) as candidate state variables ($\mathcal{V}_c$), as these are defined outside $\mathcal{L}$. In the first iteration, only the input message (pointed to by `msgPtr`) is treated as symbolic. Symbolic execution proceeds to the `ExtMsgHandler` function, where `msg_type` is assigned from the input message (line 15). Since `msg_type` is reassigned before any read operation, it is removed from the candidate state variable list on this path. Execution then reaches the branch condition on line 16, which depends on `msg_type`. Given that `msg_type` is symbolic (derived from the input message), Analyzer explores both branches. We follow the path where `msg_type` is a radio message type, though symbolic execution also covers the other path. The execution continues to line 17, where `MmProc` is used in a branch condition. As `MmProc` is read without prior assignment, it is confirmed as a true state variable, and Analyzer records it along with statistics (e.g., it controls one branch at a depth of 2, since it follows a symbolic branch on line 16). In this iteration, since `MmProc` is not symbolic, it retains its initialized value of 0 (representing `5GMM_PROC_NULL`), which does not satisfy the branch condition, so execution cannot proceed down this path. Analyzer then explores the other path, ultimately terminating without reaching the vulnerable `DecodeExtEmergNumList` function.

After the first iteration, Analyzer ranks the identified state variables and selects one to make symbolic. Suppose `MmProc` is chosen. In the second iteration, both the input message and `MmProc` are symbolic. Analyzer follows the same process, navigating the branch on line 16 by reusing results from the previous iteration. Now, since `MmProc` is symbolic, Analyzer explores both directions for the branch on line 17, reaching the path leading to line 18 and recording the precondition `MmProc != 5GMM_PROC_NULL`. At line 18, `MmAS` is used in a branch condition, and Analyzer records it as another state variable. Since the concrete value of `MmAS` does not satisfy this condition, execution cannot reach line 19. In a subsequent iteration (iter 3), Analyzer selects `MmAS` as symbolic, allowing exploration of both directions of the branch on line 18. This enables access to line 19, with the recorded preconditions `MmProc != 5GMM_PROC_NULL` and `MmAS == 5GMM_IN_CONNECT`. By following this workflow, Analyzer ultimately discovers all preconditions required to reach the `RadioMsgDecode` function. Within the `RadioMsgDecode` function, Analyzer successfully resolves the function pointer, as it was initialized during task setup, allowing Analyzer to reach the vulnerable `DecodeExtEmergNumList` function and collect required preconditions for inputs to reach this vulnerable point.

6.3. State Variable Ranking

As each task in the baseband firmware can contain hundreds of state variables, even employing the iterative symbolic execution, analyzing all identified state variables as symbolic values would lead to severe path explosion. To address this, Analyzer uses a ranking mechanism that considers key characteristics of identified state variables, aiming to prioritize symbolic analysis on variables that are less likely to lead to path explosion (i.e., require fewer resources to analyze) and provide the most insight (i.e., enabling the discovery of more code regions).

To assign priority, Analyzer computes a ranking score for each state variable using a weighted formula that considers three factors: the variable's size, the depth of the shallowest branch it controls, and the number of branches it influences. This information is collected dynamically during the iterative symbolic analysis process described in §6.2. The ranking system gives preference to variables with smaller sizes, as they typically have smaller domains and are less likely to cause path explosion. Next, the analysis considers the depth of the shallowest branch the variable controls, prioritizing variables associated with shallower branches. These branches tend to occur earlier in execution and are more likely to enable broader code exploration. Lastly, the number of branches depending on the variable is considered, since while analyzing variables that control many branches offer greater coverage potential, they may also introduce higher computational cost. The size ranking divides state variables into four tiers, each assigned a size score $S(v)$. $S(v) = 3$ variables include boolean and enum values, which have a small fixed set of possible values. Enum values are identified in the baseband binary as variables whose assignments are all constants and none of the assignments come from variables or computation results. $S(v) = 2$ contains one-byte values, which are typically flag values. $S(v) = 1$ includes two- to four-byte values, generally representing integers or floats. $S(v) = 0$ variables are pointers and values larger than four bytes, such as strings or byte arrays. These variables are excluded from selection due to their complexity and high computation cost.

The ranking score $R(v)$ for each state variable v is calculated as $R(v) = S(v) \times W_s + \frac{1}{D(v)} \times W_d + B(v) \times W_b$, where $S(v)$ is the size score, $D(v)$ is the depth of the shallowest branch it controls (inverted to give preference to shallower depths), and $B(v)$ is the number of branches influenced by the variable. W_s, W_d, and W_b are corresponding weights that prioritize size over branch depth and branch influence. Before each iteration of symbolic analysis, the variables are ranked, and the highest-ranked state variable is selected to be added to the symbolic set in the upcoming iteration.

6.4. Further Alleviating State Explosion

LORIS also applies several novel techniques other than iterative symbolic analysis to further alleviate the state ex-

plosion issue during its symbolic execution.

Checkpoint-Based Path Pruning. As Analyzer's iterative symbolic execution progresses, both the number of symbolic state variables and the number of discovered paths grow, gradually increasing the risk of state explosion. To address this, when encountering state explosions, LORIS shifts its focus from exploring a wide array of paths to concentrating on fewer paths that may lead to meaningful discoveries. We observe that each baseband task contains key program points that implement critical functionalities, and paths flowing through these points often lead to deeper and more meaningful processing. For example, the `RadioMsgDecode` in Listing 1 decodes input messages, directing those that pass through to further processing logic, making these paths more valuable for exploration. We mark such key points as *checkpoints* and, when state explosion risks arise, redirect symbolic execution to only paths that include checkpoints.

Initially, Analyzer freely explores all available paths to capture a broad view of potential execution paths. However, it monitors for signs of state explosion, defined by any of the following criteria: *(a)* Analyzing a single path from the starting point (message reception) to the endpoint (next input-waiting) exceeds a time limit (set to 1 minute in our setup). This time limit is intentionally small, as symbolically executing a single path behaves similarly to a concrete execution since it follows only one side of each branch. Based on our empirical observations, symbolically executing each path typically requires around 10 seconds, making the threshold reasonable to catch overly complex paths while maintaining efficiency. *(b)* An entire analysis iteration surpasses a total time budget (6 hours in our setup). *(c)* The analysis requires more memory than available.

When state explosion is detected, Analyzer narrows its focus to paths that contain checkpoints, pruning those that do not and reallocating resources to dive deeper into these high-value paths that are more likely to reveal vulnerabilities. This checkpoint-based pruning balances breadth and depth in symbolic execution, enabling initial broad path discovery while refocusing on deeper analysis along selected paths when broader exploration is no longer feasible. The iterative symbolic execution terminates once checkpoint-focused exploration also results in state explosion, at which point Analyzer returns all collected sets of preconditions.

Simulated Procedures. In testing, we observe that some utility functions are used across different program contexts but are highly prone to path explosion, for instance, memory operations like `memcpy`, `memset`, and `memcmp`, as well as utilities like `hexdump`. These functions typically contain branches dependent on buffer size, which leads to a rapid increase in the number of paths when the buffer size is symbolic. For instance, as shown in the simplified version of `memcpy` in Listing 2, if the input size `num` is symbolic, the function must evaluate a branch at each iteration to either exit or continue the loop, forcing Analyzer to explore `num` distinct paths. Furthermore, compiler-optimized versions of `memcpy` can introduce over 10 more branches each iteration due to optimizations like loop unrolling and architecture-specific memory handling, which amplifies path explosion.

Given that these utility functions perform well-defined tasks, we do not need to analyze their internal implementations in detail. Instead, Analyzer substitutes such functions with simplified equivalents to alleviate path explosion. This approach offers two main benefits. First, running a simulated procedure bypasses the need to simulate the actual code, improving execution speed even when not performing symbolic analysis. Second, when performing symbolic analysis, Analyzer applies heuristics to simplify these functions' behaviors. For example, if `num` is symbolic when encountering `memcpy`, Analyzer constrains `num` to its maximum concrete value, which simulates the effect of running the loop to its upper limit without introducing additional branches. This simplified analysis still provides sufficient information since the goal is to validate the function behavior up to a maximum bound rather than explore every possible intermediate state, especially for well-understood utility functions. Consequently, symbolically analyzing every possible `num` value in `memcpy` provides little additional insight and only slows down analysis. This mechanism allows Analyzer to significantly improve symbolic execution speed and reduce unnecessary path forking, focusing instead on meaningful aspects of the program's execution.

```
1 void memcpy(char *dst, char *src, size_t num) {
2   for (size_t i = 0; i < num; ++i)
3     *dst++ = *src++;
4 }
```

Listing 2. Example `memcpy` Procedure Causing State Explosion

Additionally, for functions that do not influence execution flow, such as `hexdump`, Analyzer replaces them with stubs that return immediately without performing any operations. These stubs eliminate irrelevant computations, streamlining the analysis process. We also apply manual optimizations to frequently invoked functions based on our domain knowledge. For example, if a function returns the current SIM card index in a dual-SIM baseband, we return a constant value, as the specific index does not impact input processing logic. To identify candidates for function stubbing, we collect functions that take a long time to complete symbolic execution and are frequently invoked. We then manually implement appropriate stubs and apply pattern matching [10] to automatically locate and replace these functions during analysis. Finally, we also set a maximum symbolic execution time limit to detect complex functions. If a function exceeds this time limit, it is replaced with a no-op stub. Although this approach may produce incomplete results, it ensures that Analyzer can bypass complex functions without becoming stalled. The allotted time, t_a, for each function call is calculated as $t_a = \frac{max}{\sqrt{d+1}}$, where max denotes the maximum time limit, and d is the depth of the function in the call stack, with the main input-handling loop at depth $d = 0$. This allocation favors functions earlier in the call path, as these functions have control over a greater number of potential branches and code regions.

7. Stateful Fuzzing

The overall workflow of Fuzzer is presented in Figure 2 and summarized in §5. Below we detail a few novel techniques LORIS Fuzzer incorporates.

Grammar-Aware Input Mutation. For a message m in corpus, Fuzzer applies grammar-aware input mutation to generate a new OTA test message m' that adheres to both *syntactic* and *semantic* constraints defined by the corresponding cellular specifications [2], [3]. Fuzzer employs context-free grammar (CFG) to capture the structural requirements of message formats. However, due to its inherent limitations, CFG alone cannot express inter-field dependencies, such as the length tags for value fields. As a result, previous approaches often neglect semantic constraints [30], [31], or, when they address them, rely on SMT solvers to generate concrete values [32], which is highly time-consuming and stalls the fuzzing process. To mitigate this challenge, we analyze the message specifications in the technical documents, and observe that only limited semantic requirements are used for cellular OTA messages, allowing us to design domain-specific semantic annotations on the CFG to enforce these requirements. While Fuzzer employs a novel approach for grammar-aware input mutation, it does not directly relate to our main contribution of handling state and dependencies in baseband. Therefore, we provide additional details on the limitations of prior efforts, our method, motivating examples, all supported semantic annotations, and a description of both grammar-aware and havoc mutations in Appendix A.

Feedback. During fuzzing, code coverage is tracked through emulator instrumentation, capturing both branch (edge) coverage and coarse branch-taken hit counts.

Testing Oracle. Existing fuzzers [33], [34] typically detect memory corruption issues through observable program behaviors, such as crashes and execution timeouts. While stack-based overflows often lead to crashes by overwriting return addresses and causing program counter (PC) faults, many memory corruptions (e.g., heap overflows) can occur without any visible behaviors. Most memory sanitizers [35], [36], [37], [38] are only applicable at compilation time, making them unsuitable for our binary-only analysis. To address this limitation, we replace the baseband heap management API with a customized emulated heap manager that provides memory sanitization and detects vulnerabilities such as double-free, use-after-free, and heap-based out-of-bounds access during memory allocation and deallocation.

Allocation: The heap manager allocates a memory buffer twice the requested size; for an allocation request of S bytes, it allocates $2S$ bytes and returns a pointer offset by $\frac{S}{2}$ bytes from the start of the allocated region. This configuration allows us to monitor accesses to the leading and trailing $\frac{S}{2}$-byte buffers with callbacks to detect both buffer overflows and underflows. ***Deallocation:*** Upon the first request to free a previously allocated buffer, the heap manager adds the buffer's address to a list of deallocated regions to enable double-free detection. It hooks any access to the deallocated region to detect use-after-free vulnerabilities. To crash the emulation upon detecting memory corruption, we set the program counter to the address of the Data Abort exception handler, which Fuzzer interprets as a crash.

8. Implementation

Table 2 summarizes the implementation language and lines of code (LoC) for each component in LORIS. Specifically, LORIS Analyzer (§6) is implemented in Python using the *angr* binary analysis platform [39]. LORIS Fuzzer (§7) is developed in Rust employing the LibAFL library [40]. LORIS Grammar parser is defined in *pest* [41], a general-purpose parser that uses expression parsing grammars. We extracted message definitions of NAS EMM and ESM messages from 3GPP documentations [2], [3], and manually expressed message grammars in LORIS's customized CFG. LORIS Emulator extends FirmWire [10], the state-of-the-art baseband emulator built on QEMU [20]. FirmWire supports emulation of several Exynos basebands, but it does not support any 5G basebands. Among the baseband we tested (presented in Table 3), FirmWire only supports Galaxy S10. We extend FirmWire to be the first to support recent Exynos 5G basebands by (1) supporting the Coretex-A processor newly used in Exynos 5G basebands (2) creating a new vendor plugin for Pixel basebands (3) adding support for new peripherals such as the interrupt controller and timers. We conduct reverse engineering on top of Ghidra [29], a software reverse engineering suite of tools developed by NSA. We developed several Ghidra scripts to collect code coverage feedback, and semi-automate root cause analysis of vulnerabilities.

TABLE 2. IMPLEMENTATION DETAILS OF LORIS'S COMPONENTS

Component	Language	LoC	Component	Language	LoC
Analyzer	Python	2892	LORIS CFG	Pest	111
Fuzzer	Rust	6084	Msg Grammar	LORIS	694
Emulator	Python/C	5078	Harnessing	C	234
Ghidra Scripts	Python	770	-	-	-

9. Evaluation

We evaluate the effectiveness of LORIS by answering the following research questions:

- How effective is Loris in discovering vulnerabilities in commercial baseband? (§9.1)
- How does LORIS compare to existing tools? (§9.2.1)
- How effective are LORIS's ranking and iterative symbolic execution strategies in addressing state explosion during the analysis of state variables? (§9.2.2)

Baseband Targets. As summarized in Table 3, we evaluate 5 baseband firmware images from Samsung and Pixel smartphones. These images span two major baseband manufacturers, Samsung and MediaTek, and cover two instruction set architectures, ARM and MIPS, respectively.

Experiment Setup & Statistics. We perform all our evaluations on a machine with an Intel Xeon Gold 6448H @ 4.1GHz CPUs (64 physical and 128 logical cores) and

TABLE 3. SMARTPHONES EVALUATED FOR EACH PROTOCOL

Protocol	Phone Model	Firmware Image	Instruction Set	Build Date
LTE NAS	Galaxy S10	G973FXXSHHWI1	ARM	Oct'23
	Galaxy S20	G981BXXSKHXEA	ARM	May'24
	Galaxy S21 5G	G991BXXSCGXF5	ARM	Jul'24
	Pixel 6	oriole-ap2a	ARM	Sep'24
	Galaxy A41	A415FXXS8DXD1	MIPS	May'24
5G NAS	Galaxy S21 5G	G991BXXSCGXF5	ARM	Jul'24
	Pixel 6	oriole-ap2a	ARM	Sep'24

1007GB RAM running Ubuntu 22.04 with Linux kernel 5.15.0. Analyzer was allowed to utilize all available memory during its analysis, while each fuzzing instance was restricted to a maximum of 512MB of RAM. During the evaluation, Analyzer identified an average of 65,000 *candidate state variables* of 96KB size per target task. *Iterative symbolic execution* was performed on each task for 24 hours, completing approximately 10 iterations. On average, Analyzer identified around 100 true state variables, with 9 variables selected as symbolic across the 10 iterations. State explosion was detected after approximately 8 iterations, at which point *checkpoint pruning* was applied to guide exploration. Additionally, we designed simulated procedures for 5 distinct functions prone to state explosion, which LORIS replaced on average of 12 occurrences of these functions in each task. Analyzer identified on average 300 sets of state preconditions per task. The corresponding state snapshots, derived by solving these preconditions, were used to initialize Fuzzer. Fuzzing was conducted for 24 hours across 10 independent runs for each task.

9.1. Discovered Vulnerabilities

9.1.1. Previously Unknown Vulnerabilities. Table 4 summarizes the new vulnerabilities LORIS uncovers. In total, LORIS uncovers 10 issues (I1–I10) in both LTE and 5G baseband firmware, leading to 8 new vulnerabilities (V1-V8), 7 of which are exploitable OTA (V1-V6 & V8). We have validated all 7 exploitable vulnerabilities using OTA messages on commercial devices and reported them to the respective vendors. All reported vulnerabilities have been acknowledged, with severity assessments included in Table 4. Five of the vulnerabilities (V1, V3, V5, V6, V8) have already been patched and assigned CVEs (CVE-2024-52924, CVE-2024-52923, CVE-2025-27891, CVE-2025-26784, CVE-2025-26785, respectively). Note that the listed impacted UE models in Table 4 are not exhaustive and only show the models tested during our evaluation. Our responsible disclosure process has enabled vendors to systematically identify and address all affected baseband versions and UE models. Due to space constraints, we describe only a few of these vulnerabilities in detail below.

V1. LORIS discovers a vulnerability in Exynos basebands when decoding the *operator-defined access category definitions* information element (IE), which is used in both *registration accept* and *configuration update command* messages. This IE can include various information fields, including the *DNN (Data Network Name)*, which is formatted as a length-value (LV) field. Listing 3 illustrates the vulnerability within the decoder function for this IE. In this function, on line 3, a buffer is allocated to store the decoded bytes, and an `uint8` variable, `ieOffset`, is initialized to track the number of processed bytes. The function then enters a loop to process each field within the IE, up to `ieLength` (also an `uint8`), which is the total length of the IE (lines 4-17). In each iteration, it reads the field type (line 5). If the field is a DNN field, the decoder reads the length of the field (line 8) and, based on this length, copies the field's value into the allocated buffer (line 10). After processing each field, the decoder advances `ieBuf` to the next unread value (line 11) and increments `ieOffset` (line 12). For fields other than DNN, it continuously reads the next byte (lines 13-15).

```
1  // "\x01" * 144 + "\x00\x00\x00\x6d" + payload
2  void decode(uint8_t *ieBuf, uint8_t ieLength) {
3    uint8_t local[144][10], ieOffset = 0;
4    for (i = 0; ieOffset < ieLength - 1; ++i) {
5      uint8_t ieTyp = ieBuf[0]; // type of IE field
6      local[i][0] = ieTyp;
7      if (ieTyp == 0) { // a DNN field
8        local[i][1] = ieBuf[3]; // length field
9        // stack overflow
10       memcpy(&local[i][2], ieBuf + 4, ieBuf[3]);
11       ieBuf += ieBuf[3] + 4; // increment buf pointer
12       ieOffset += ieBuf[3] + 3;  // int overflow
13     } else {
14       ieBuf += 1;
15       ieOffset += 1;
16     }
17   }
18 }
```

Listing 3. Simplified Code Demonstrating Vulnearbility V1

However, the `uint8 ieOffset` is vulnerable to an integer overflow, which can cause the length check on line 4 to always evaluate as true, resulting in a stack overflow that can be exploited by OTA messages. This overflow enables an attacker to write to any location on the stack. Consider the following crafted input: `"\x01"` repeated 144 times, followed by `"\x00\x00\x00\x6d"` and an arbitrary custom payload. The first 144 `"\x01"` bytes fill the allocated buffer, `local`, exhausting its 144 indices (line 6). The decoder then reads the type as 0 (line 7) and interprets `\x6d` as the length of the next field. It then attempts to copy `\x6d` bytes from the IE buffer into the already-filled `local` buffer, causing an overflow of the `local` buffer (line 10). The `ieOffset` also overflows on line 12, as $144 + 0x6d + 3 = 256$, wrapping `ieOffset` back to 0 for the `uint8` variable. Since the length check on line 4 is based on `ieOffset` and `ieLength`, the overflowed `ieOffset` allows the loop to continue processing further bytes from the OTA message, enabling the attacker to write arbitrary-length content onto the stack. Even without triggering an integer overflow, a message such as `"\x01"` repeated 145 times would still cause a stack overflow. However, in that case, the attacker would have limited control over the overflow length, constrained by `ieLength`'s maximum value of 255. The integer overflow grants the attacker significantly more control, allowing them to write arbitrary-length content to the stack by circumventing the length limit.

Attack Impact & Vendor Response. The stack overflow can lead to crashes or remote code execution (RCE). We are able

TABLE 4. SUMMARY OF PREVIOUSLY UNKNOWN VULNERABILITIES DISCOVERED BY LORIS
(* represents any IE in the message is vulnerable, - indicates no direct exploit for this vulnerability)

#	Protocol	Message Type	Vulnerable IE	Exploited Issues	Impacted UE Model	Severity Rating		Impact
						Samsung	Google	
V1 (§9.1)	5G NR	Registration accept & Configuration update command	Operator-defined access category definitions	(I1) Integer Truncation (I2) Stack overflow	Galaxy S21	Medium	-	DoS, RCE
V2 (§9.1)	5G NR	Registration accept & Configuration update command	Operator-defined access category definitions	(I1) Integer Truncation (I3) Heap overflow	Pixel 6	-	High	DoS, RCE
V3	5G NR	DL NAS transport	Payload container	(I4) Assertion failure	Galaxy S21, Pixel 6	Medium	Moderate	DoS
V4 (§3.2)	5G NR	Registration accept	Emergency number list	(I5) Integer overflow	Pixel 6	-	Low	DoS
V5	5G NR	DL NAS transport	UE parameters update	(I6) Heap overflow	Galaxy S21, Pixel 6	Medium	Low	DoS
V6	5G NR	DL NAS transport	Steering of Roaming	(I7) Heap overflow	Galaxy S21, Pixel 6	High	High	DoS
V7	LTE	Security mode command	*	(I8) Integer overflow (I9) Heap overflow	Galaxy S10, S20, and S21, Pixel 6	-	-	-
V8	5G NR	DL NAS transport	SMS	(I10) Heap overflow	Galaxy S21, Pixel 6	High	Critical	DoS

to reproduce the stack overflow in a commercial Galaxy S21 5G (detailed in §9.1.2), leading to a forced reboot of the phone. To mitigate stack overflows, the baseband processor used in Galaxy S21 5G implements a stack canary mechanism that helps prevent stack corruption from affecting other parts of the system. This mechanism places a 4-byte canary value at the top of a function's stack frame during the function's prologue, which is then checked for integrity just before the function returns (in the epilogue). The canary value is updated with each baseband boot, ensuring its unpredictability. If the stored canary value is altered, indicating a stack overflow, the execution is immediately aborted. For an attacker to achieve RCE via this stack overflow, they would need to bypass the stack canary mechanism, or the baseband simply crashes, preventing further exploitation. An information leak or a more realistic scenario, a memory corruption resulting in write-what-where conditions can bypass this canary mechanism. We observed that the canary value is loaded from a memory address that is updated with a random value only during the boot process, the canary offset is fixed in the stack frame of all functions, and the canary check is a simple integer comparison (no complex functionality is involved). With the absence of Address Space Layout Randomization (ASLR), the stored canary value can be manipulated or leaked by chaining with other vulnerabilities to achieve RCE. As we focus on vulnerability detection rather than developing exploitations, bypassing this mechanism is beyond the scope of this work. We reported the stack overflow vulnerability to Samsung, who confirmed its existence.

V2. A similar integer overflow is also identified in the Pixel 6 baseband. This baseband dynamically allocates a buffer to store decoded OTA messages, avoiding stack overflow. However, the dynamically allocated buffer is stored within a fixed-length heap buffer in a `NrmmRegContext` class object. Listing 4 shows a structure containing this class instance. Any overflow within `reg_ctx` field can lead to an intra-chunk heap overflow, overwriting adjacent fields. This vulnerability can also result in severe exploits, as the neighboring fields (`proc_ctx` and `security_ctx`) utilize virtual tables and store sensitive information about 5GMM procedure and security contexts.

```
struct cn::mm::NrmmContext {
    ...
    NrmmRegContext reg_ctx;
    NrmmProcedureContext proc_ctx;
    NrmmSecurityContext security_ctx;
    ...
};
```

Listing 4. Structure vulnerable to heap overflow (Vulnerability V2), where an overflow in `reg_ctx` can corrupt adjacent fields

V4. This vulnerability is detailed in § 3.2, where an integer overflow in the decoder causes the function to enter an infinite loop, causing DoS.

V8. This vulnerability is triggered by providing an overly long SMS payload within the payload container IE of a *Downlink NAS Transport* message with *multiple payload container types*. The decoder allocates a fixed-size buffer (0x61c bytes) to hold the SMS content, but it fails to validate the SMS field length against the allocated buffer size. As a result, the decoder could copy data beyond the bounds of the heap buffer, overwriting adjacent memory. This heap corruption is detected by Samsung's heap sanitizer newly introduced in their 5G basebands, which aborts execution and triggers a crash. We verified that this vulnerability is specific to the handling of multiple payload containers, and does not occur when the SMS payload is directly embedded within a single container in the *DL NAS Transport* message.

9.1.2. Over-the-Air Exploit Validation. To demonstrate the practical impact of the vulnerabilities discovered through emulated fuzzing, we validated all detected exploitable vulnerabilities (V1–V6, V8) over-the-air (OTA) on real devices. We used a USRP B210 software-defined radio (SDR) as gNodeB with a modified Open5GS [42] as the core network to transmit the attacking NAS message payloads identified by LORIS. We followed the standard registration process to send the attacking *registration accept* messages (V1, V2, V4). For the *DL NAS transport* message (V3, V5, V6, V8), we configured an access point name (APN) on the phone, prompting it to send a PDU session establishment request. The SDR then responded with the attacking *DL NAS transport* message. To verify the exploit is successful, we send several *security mode command* messages after

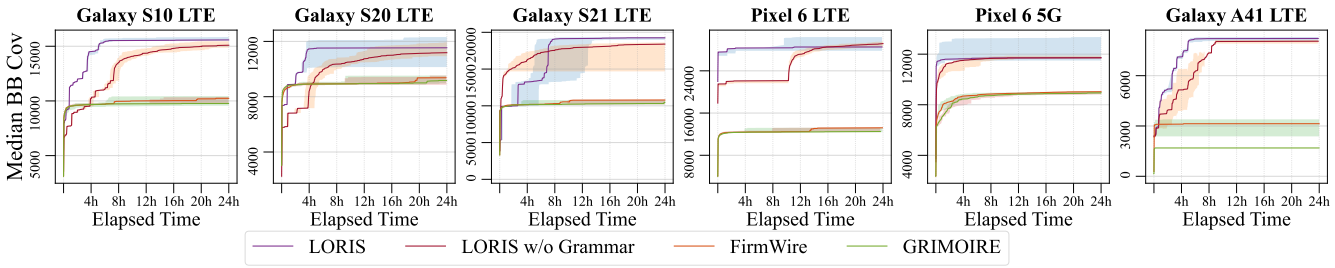


Figure 4. Discovered Basic Blocks (BB) Over Time for Fuzzing NAS Layer Tasks Across Different Phone Models & Protocols

the malformed attack message. Under normal operations, these commands should guarantee a response, and a lack of responses confirms that the device has become unresponsive.

Results. We successfully reproduced all 7 discovered exploitable vulnerabilities on either the Galaxy S21 5G or Pixel 6. On both devices, crashes were indicated by signal loss (i.e., disconnection from the network with *no service* indication). On the Galaxy S21, triggering a crash while *Upload Mode* was enabled resulted in a forced reboot and the generation of a core dump. On the Pixel 6, we enabled core dump generation using a user-debug build of the Android ROM. Subsequent analysis of the collected core dumps validated our findings.

9.1.3. Known Vulnerability Discovery. To further validate LORIS's bug-finding capabilities, we evaluated it on an older version of the Galaxy S21 baseband (G991BXXU5CVF3). LORIS identified 3 heap overflow vulnerabilities that have since been patched in recent versions. According to Samsung's CVE program [43], 6 heap overflow vulnerabilities in total have been reported for this baseband model. However, due to the lack of available information regarding the vulnerability details, firmware versions, or specific tasks, we could not determine how many of these reports correspond to the issues in this task or firmware version. We also examined two prior studies with overlapping scopes [12], [44], each reporting one vulnerability in NAS message handling. When applied to the vulnerable basebands analyzed in these studies, Analyzer identified the state preconditions needed to reach the vulnerable code regions in 5 and 11 iterations of symbolic analysis, respectively. Fuzzer then generated inputs to trigger these vulnerabilities within 24 minutes and 2.5 hours, respectively. Notably, among all prior reports, only BaseSAFE [12] employed automated methods for vulnerability discovery, while others likely relied on manual efforts. In contrast, LORIS fully automated the process, demonstrating its efficiency and effectiveness.

9.1.4. Vulnerability Discovery Accuracy. Like any testing approach, LORIS may miss vulnerabilities due to time and resource constraints limiting code coverage, or failing to generate test cases triggering vulnerabilities for highly restrictive conditions. However, we have not observed any false negatives in our testing. Regarding false positives, LORIS focuses on analyzing and testing individual baseband tasks in isolation. As a result, some discovered vulnerabilities may not be remotely exploitable in practice if mitigated by lower-layer validations. For example, V7 (Table 4) triggers a crash in the `SAEL3` task, but is rendered non-exploitable over-the-air due to checks in the Layer 2. While Samsung has confirmed the underlying issue, they also agree that no OTA exploit is possible. Aside from this case, we have not encountered other false positives in our evaluation.

TABLE 5. VULNERABILITY DISCOVERY TIMINGS
Fac. indicates the time factor by which LORIS is faster than comparison. No time is reported when comparison fails to discover a vulnerability.

#	Fuzzer	Min.	Max.	Mean	Fac.
V1	LORIS	0h-9m-33s	0h-15m-57s	0h-12m-0s	-
	FirmWire	0h-33m-45s	1h-3m-11s	0h-48m-35s	4
	no grammar	0h-8m-43s	0h-23m-5s	0h-15m-48s	1.8
V2	LORIS	0h-0m-29s	0h-7m-43s	0h-5m-52s	-
	FirmWire	0h-24m-34s	1h-54m-54s	0h-53m-57s	9.2
	Grimoire	1h-4m-30s	1h-10m-48s	1h-7m-33s	11.5
	no grammar	0h-44m-30s	0h-53m-29s	0h-49m-37s	8.5
V3	LORIS	0h-6m-18s	1h-24m-29s	0h-42m-55s	-
	FirmWire	1h-3m-39s	1h-42m-20s	1h-11m-7s	1.65
	Grimoire	0h-17m-18s	6h-17m-41s	3h-12m-51s	4.5
	no grammar	0h-24m-22s	1h-15m-25s	0h-49m-1s	1.14
V4	LORIS	0h-9m-27s	1h-7m-53s	0h-53m-4s	-
	no grammar	1h-31m-46s	4h-13m-5s	3h-10m-51s	3.6
V5	LORIS	0h-2m-46s	0h-5m-5s	0h-3m-16s	-
	no grammar	0h-4m-5s	0h-56m-46s	0h-10m-1s	3
V6	LORIS	0h-0m-34s	0h-4m-3s	0h-0m-42s	-
	no grammar	0h-12m-38s	0h-31m-7s	0h-15m-16s	21.8
V8	LORIS	0h-1m-19s	0h-4m-51s	0h-2m-24s	-
	no grammar	0h-7m-39s	0h-16m-17s	0h-10m-27s	4.35

no grammar represents LORIS without employing grammar-aware mutation

9.2. Benchmark Comparisons & Ablation Studies

9.2.1. Benchmark Comparisons. To evaluate the effectiveness of LORIS against other approaches, we adapt existing state-of-the-art tools for comparison, specifically FirmWire [10] and Grimoire [45]. We do not include BaseComp [15], as it uses a static approach focused solely on finding logical bugs in integrity protection functions, nor BaseSAFE [12], which requires manual setup for each function it tests. LORIS is designed to eliminate these manual harnessing efforts. We present some qualitative comparisons against these tools in Table 1. Additionally, we note that vulnerability V4 LORIS discovered involves interactions across multiple functions to trigger, a scenario that BaseSAFE's function-specific approach cannot detect even with extensive manual setup due to its inherent limitations.

FirmWire [10] applies AFL++ [33] to emulated basebands while managing state and task dependencies manually. AFL++ is a state-of-the-art evolutionary fuzzer that employs

only havoc mutations on binary inputs, lacking any awareness of the input's underlying structure.

Grimoire [45] is a coverage-guided fuzzer that generates new inputs by dynamically inferring the grammar structure of the expected input format. It supports mutation operations including splicing, recursive replacement, string substitution, and random deletion on the inferred input structure.

Grammar Ablation Studies. Additionally, to evaluate the impact of LORIS's grammar-aware mutator on fuzzing effectiveness, we conduct an ablation study in which LORIS uses AFL++'s havoc mutators in place of its grammar-aware mutator (i.e., LORIS w/o Grammar), while still leveraging the state analysis results produced by Analyzer.

Manual Harnessing. We use the same initial corpus for all tools, including LORIS, and ensure a fair comparison by allocating the same computational resources and a fixed testing duration of 24 hours for each tool. However, testing OTA NAS messages on the baseband firmware without state information leads to immediate rejection of the test inputs due to the incorrect message flow. As a result, both FirmWire and Grimoire struggle to achieve significant coverage without the state insights that LORIS gains from its Analyzer. To ensure that inputs generated by these two fuzzers reach at least the initial decoding logic, we manually reverse-engineered each target firmware to identify a specific sequence of four NAS messages that allow subsequent messages to pass through decoding. This additional manual effort must be repeated for each tested baseband, taking hours for experts familiar with the firmware (e.g., 2 hours for us when we evaluate Galaxy S21). This manual setup mirrors Analyzer's automated analysis through iterative symbolic execution to determine the necessary preconditions that set the task to the required state. In 4 iterations, Analyzer successfully identifies the relevant state variables and their preconditions to achieve this same state. Additionally, further iterations allow Analyzer to uncover additional state variable values that enable deeper code exploration.

Comparison Result. Figure 4 illustrates the basic block (BB) coverage achieved by different tools when fuzzing the NAS message handling task across various phone models on both LTE and 5G. Each plot shows data from 10 independent 24-hour runs, with the line representing the median BB coverage and the shaded area indicating the range (upper and lower bounds) across these runs. As shown, LORIS consistently outperforms both FirmWire and Grimoire, which perform similarly to each other. Notably, LORIS achieves nearly double the coverage for most tested phone models. LORIS with AFL++ mutator achieves similar or slightly lower coverage as LORIS although its coverage growth is significantly slower. Table 5 summarizes the number of vulnerabilities detected by each tool and the time taken to identify them. LORIS not only identifies the highest number of vulnerabilities (more than twice as many as other tools) but also does so in the shortest amount of time. LORIS without grammar finds all vulnerabilities but slower.

9.2.2. Ablation Study on State Analysis. To assess the impact of different techniques employed by Analyzer in mitigating state explosion, we conduct an ablation study on state variable symbolic analysis under two scenarios: (1) Analyzer randomly selects discovered state variables to make symbolic instead of using the ranking system (i.e., no ranking); (2) Analyzer does not perform iterative analysis at all, making all candidate state variables symbolic simultaneously (i.e., no iterative). The experiments were conducted for 8 hours, and the results are presented in Figure 5. We measure the number of completed paths that successfully reach the decoding function (e.g., line 30 in Listing 1), as inputs that fail to reach decoding are rejected and have no impact on the task. The results indicate that LORIS without iterative symbolic execution suffers from immediate state explosion and fails to identify any meaningful paths. LORIS without ranking is also significantly less effective at discovering meaningful paths and experiences more frequent state explosion due to having more active paths at any time.

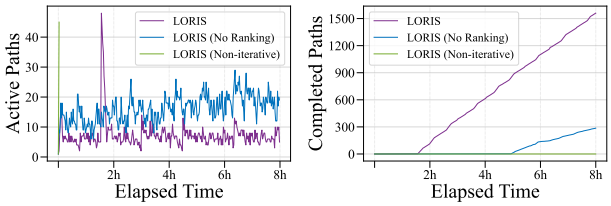


Figure 5. Number of Completed Paths and Active Paths Over Time

10. Discussions

Required Manual Effort. The proprietary and closed-source nature of baseband firmware inherently requires some manual analysis. LORIS requires reverse engineering to identify the target task and the data structures containing the OTA messages for delivery to the target task. However, LORIS does not require reverse engineering of OTA message structures, as these are well-defined in cellular specifications (§4.1). In our workflow, approximately 10 hours of manual effort is required to prepare a new baseband for testing, or less than 1 hour when details of the target task are available from previous research [10]. LORIS's grammar-aware mutator also requires manual grammar extraction, which takes approximately 20 hours in our workflow to extract NAS message definitions from the corresponding technical specifications. However, this is a one-time effort per protocol and can be reused to test all baseband implementations. To support future research, we will open-source the extracted grammars. Moreover, as shown in Figure 4, LORIS remains effective even without the grammar-aware mutator. Finally, Extending LORIS's emulation capabilities to support new basebands (e.g., 5G basebands) also requires manual engineering effort.

While LORIS significantly outperforms existing approaches (as shown in Figure 4), this type of manual effort is common to all methods aimed at baseband firmware testing. Notably, LORIS avoids the need for manual fuzz harnessing, which is a common requirement in previous work. For example, prior approaches [10], [12] require analysts to manually configure tasks to reach message reception states

or to build function-level fuzzing harnesses (§9.2). In contrast, LORIS automates the analysis of state preconditions necessary for message acceptance and processing (§6), enabling deeper and automated exploration of program logic.

Extensibility of LORIS. The core concept of LORIS is adaptable to other baseband architectures and general firmware that employ a task-like design to continuously handle incoming messages, enabling it to resolve the complex state and dependencies of these tasks. We demonstrate this extensibility by evaluating LORIS on two different baseband architectures, Samsung's ARM-based Shannon basebands and MediaTek's MIPS-based basebands. Adapting LORIS to MediaTek's MIPS-based baseband required reverse engineering to find the target task and locate the task's message-reception entry point, which serves as the starting point for Analyzer. Once identified, Analyzer automatically infers state preconditions, which are then used by Fuzzer with minimal additional manual intervention. Supporting MIPS-based basebands also required extending LORIS's emulation and symbolic execution capabilities. Specifically, to enable symbolic execution for the MIPS16e2 instruction set used in MTK basebands, we extended the MIPS Pcode definitions in angr. Because angr's default intermediate representation (VEX) lacks MIPS support, we adopted the experimental UberEnginePcode backend, which integrates Ghidra's Pcode IR. The MIPS16e2 extension required a one-time manual effort of approximately 6 hours and successfully enabled symbolic execution on MTK firmware.

Analyzer Accuracy. Analyzer may encounter false negatives if it fails to identify a relevant state variable, potentially causing Fuzzer to stall at early branches that depend on the missed variable. Such misses can result from the incompleteness of static analysis [46], where some write operations may not be captured, leading to incorrect classification of a variable as a constant rather than a candidate state variable. However, we did not observe such cases in our evaluation. Additionally, some correctly identified state variables may remain unexplored during symbolic execution due to resource constraints. In such cases, subsequent fuzzing mutations may still enable the exploration of paths determined by these variables. On the other hand, all state variables identified according to Definition 6.1 are valid, and we did not encounter any false positives in our experiments.

11. Related Work

Cellular Protocol Security. Several previous efforts analyze various security aspects of cellular communications [9], [47], [48], [49]. 5GBaseChecker [47] applies black-box automata learning and differential testing for analyzing the security of 5G baseband control plane protocols. DoLTEst [9] performs negative tests on LTE RRC and NAS message handling functions in commercial UEs. However, these black-box and OTA-based approaches are slow and rely on logs from UEs to diagnose problems. These logs may not be detailed enough or available at all on proprietary basebands. In comparison, LORIS performs testing directly on baseband firmware emulations which enables detailed analysis and provides accurate introspection to triage vulnerabilities.

Firmware Security. Previous efforts on general firmware security [50], [51], [52], [53] either depend on manual harnessing, unscalable static analysis or symbolic analysis, or could not reason the complex states at all. Mulliner's research [11] pioneered cellular baseband security analysis using Sulley [54] fuzzing framework to generate invalid SMS messages and delivered them through AT command from application processor to modem. BaseSAFE [12] conducts testing on isolated functions, whereas FirmWire [10] employs full-system emulation and testing of baseband. None of these previous efforts can effectively reason the state and dependencies in baseband. In comparison, LORIS performs symbolic exploration and stateful fuzzing, and its approach is extensible to other binary-only stateful testing.

Stateful Analysis. Several stateful analysis approaches are proposed for protocol testing [55], [56], [57], [58]. Among them, Frankenstein [55] extracts memory snapshots from physical devices to initialize state variables. However, its code-injection approach for snapshot extraction is infeasible in proprietary basebands. Additionally, it can only capture limited snapshots, and cannot test states not captured. LORIS assigns arbitrary values to state variables, enabling broader state space exploration and code coverage. Ijon [56] and SandPuppy [57] rely on source code, leading to unsatisfactory results in binary-only basebands. Ferry [58] can only track state variables derived from inputs, failing to capture inter-task dependencies in basebands (Section 4.2). LORIS presents a precise definition (Definition 6.1) and employs an iterative approach to accurately identify and analyze state variables. On the other hand, Ijon and SandPuppy use state variables as fuzzing feedback, but their random, unreliable fuzzing cannot systematically reach different states. Ferry symbolically analyzes state variables to derive input constraints, but it quickly encounters state explosion in complex targets (Figure 5). Even Ferry acknowledges limitations in complex environments (Section 3.1 of [58]). Loris employs iterative symbolic execution to comprehensively analyze state dependencies while mitigating state explosion.

12. Conclusion

We present LORIS, a novel framework for stateful, dependency-aware analysis and fuzz testing of baseband firmware. By employing an iterative symbolic analysis, LORIS progressively uncovers state preconditions while effectively mitigating the path-explosion problem, enabling it to explore various code regions and achieve high coverage. Evaluations on 5 commercial devices from 2 major vendors, spanning both 4G LTE and 5G NR protocols, revealed 8 new vulnerabilities, with potential impacts including remote code execution and denial of service when exploited over-the-air. Comparisons with state-of-the-art tools highlight LORIS's ability to mitigate state explosion, achieve better coverage, and uncover more vulnerabilities.

Acknowledgements

We thank the anonymous reviewers and the shepherd for their feedback and suggestions. We also thank the corresponding developers for cooperating with us during our responsible disclosure. This work has been supported by the NSF under grants 2145631, and 2215017, the Defense Advanced Research Projects Agency (DARPA) under contract number D22AP00148, and the NSF and Office of the Under Secretary of Defense—Research and Engineering, ITE 2326898 and 2515378, as part of the NSF Convergence Accelerator Track G: Securely Operating Through 5G Infrastructure Program.

References

[1] 3GPP, "IP multimedia call control protocol based on Session Initiation Protocol (SIP) and Session Description Protocol (SDP); Stage 3," 2022.

[2] ——, "TS 24.301; Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3," 2022.

[3] ——, "Non-Access-Stratum (NAS) protocol for 5G System (5GS); Stage 3," 2022.

[4] ——, "Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification," 2022.

[5] A. Cama, "A walk with shannon: A walkthrough of a pwn2own baseband exploit." Insomni'hack, 2018. [Online]. Available: https://www.insomnihack.ch/conference-2018/#acez

[6] D. K. Nico Golde, "Breaking band." RECON, 2016. [Online]. Available: https://recon.cx/2016/talks/Breaking-Band.html

[7] R.-P. Weinmann, "Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks," in *6th USENIX Workshop on Offensive Technologies (WOOT 12)*. USENIX Association, 2012.

[8] H. Kim, J. Lee, E. Lee, and Y. Kim, "Touching the untouchables: Dynamic security analysis of the lte control plane," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1153–1168.

[9] C. Park, S. Bae, B. Oh, J. Lee, E. Lee, I. Yun, and Y. Kim, "DoLTEst: In-depth downlink negative testing framework for LTE devices," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1325–1342.

[10] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, and K. R. B. Butler, "FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware," in *Symposium on Network and Distributed System Security (NDSS)* , 2022.

[11] C. M. Collin Mulliner, "Fuzzing the phone in your phone." Black Hat, 2009. [Online]. Available: https://www.blackhat.com/html/bh-usa-09/bh-usa-09-speakers.html

[12] D. Maier, L. Seidel, and S. Park, "Basesafe: baseband sanitized fuzzing through emulation," in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '20. Association for Computing Machinery, 2020.

[13] N. Golde, "There's life in the old dog yet: Tearing new holes into intel/iphone cellular modems," 2018, accessed: 2024-06-13. [Online]. Available: https://comsecuris.com/blog/posts/theres_life_in_the_old_dog_yet_tearing_new_holes_into_inteliphone_cellular_modems/

[14] "All your baseband are belong to us." DeepSec IDSC, 2010. [Online]. Available: https://deepsec.net/archive/2010.deepsec.net/

[15] E. Kim, M. W. Baek, C. Park, D. Kim, Y. Kim, and I. Yun, "BASECOMP: A comparative analysis for integrity protection in cellular baseband software," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, 2023.

[16] G. Delugré, "Reverse-engineering a qualcomm baseband." Chaos Computer Club (CCC), 2011. [Online]. Available: https://media.ccc.de/v/28c3-4735-en-reverse_engineering_a_qualcomm_baseband

[17] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz, "Hoedur: embedded firmware fuzzing using multi-stream inputs," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.

[18] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson, "Sok: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21. New York, NY, USA: Association for Computing Machinery, 2021.

[19] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Comput. Surv.*, vol. 54, no. 1, jan 2021.

[20] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

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

[22] D. Rupprecht, K. Jansen, and C. Pöpper, "Putting LTE security functions to the test: A framework to evaluate implementation correctness," in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, Aug. 2016.

[23] G. Lee, J. Lee, J. Lee, Y. Im, M. Hollingsworth, E. Wustrow, D. Grunwald, and S. Ha, "This is your president speaking: Spoofing alerts in 4g lte networks," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '19. Association for Computing Machinery, 2019.

[24] T. Yang, S. M. M. Rashid, A. Ranjbar, G. Tan, and S. R. Hussain, "ORANalyst: Systematic testing framework for open RAN implementations," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1921–1938.

[25] L. Coppolino, V. D'Alessandro, S. D'Antonio, L. Levy, and L. Romano, "My smart home is under attack," in *2015 IEEE 18th International Conference on Computational Science and Engineering*, 2015.

[26] L. Janzen, L. Becker, C. Wiesenäcker, and M. Hollick, "Oh no, my RAN! breaking into an O-RAN 5g indoor base station," in *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 101–115.

[27] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[28] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1544–1561.

[29] NSA, "Ghidra software reverse engineering framework," 2024. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[30] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for deep bugs with grammars," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[31] P. Srivastava and M. Payer, "Gramatron: effective grammar-aware fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021.

[32] D. Steinhöfel and A. Zeller, "Input invariants," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 583–594.

[33] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, 2020.

[34] R. Swiecki, "Honggfuzz—coverage-guided mutational fuzzer," 2021. [Online]. Available: https://github.com/google/honggfuzz

[35] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318.

[36] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 62–71.

[37] "UndefinedBehaviorSanitizer." [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[38] E. Stepanov and K. Serebryany, "Memorysanitizer: Fast detector of uninitialized memory use in c++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[39] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[40] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, ser. CCS '22. ACM, November 2022.

[41] D. Tiselice, "pest," https://github.com/pest-parser/pest, 2022.

[42] "Open5gs." [Online]. Available: https://github.com/open5gs/open5gs

[43] Samsung, "Product security update." [Online]. Available: https://semiconductor.samsung.com/support/quality-support/product-security-updates/

[44] D. Komaromy, "Cve-2023-21517," 2023. [Online]. Available: https://labs.taszk.io/blog/post/85_ss_esm_bof/

[45] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "GRIMOIRE: Synthesizing structure while fuzzing," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1985–2002.

[46] A. Møller and M. I. Schwartzbach, "Static program analysis," https://cs.au.dk/~amoeller/spa/, 2018.

[47] K. Tu, A. A. Ishtiaq, S. M. M. Rashid, Y. Dong, W. Wang, T. Wu, and S. R. Hussain, "Logic gone astray: A security analysis framework for the control plane protocols of 5g basebands," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 3063–3080.

[48] S. R. Hussain, I. Karim, A. A. Ishtiaq, O. Chowdhury, and E. Bertino, "Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4g lte cellular devices," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. Association for Computing Machinery, 2021.

[49] M. Akon, T. Yang, Y. Dong, and S. R. Hussain, "Formal analysis of access control mechanism of 5g core network," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. Association for Computing Machinery, 2023.

[50] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Krügel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *Network and Distributed System Security Symposium*, 2015.

[51] D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," 01 2016.

[52] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-Scale analysis of the security of embedded firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 95–110.

[53] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 463–478.

[54] R. Sears, "Sulley," https://github.com/OpenRCE/sulley, 2014.

[55] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 19–36.

[56] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1597–1612.

[57] V. Paliath, E. Trickel, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "Sandpuppy: Deep-state fuzzing guided by automatic detection of state-representative variables," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, F. Maggi, M. Egele, M. Payer, and M. Carminati, Eds. Cham: Springer Nature Switzerland, 2024, pp. 227–250.

[58] S. Zhou, Z. Yang, D. Qiao, P. Liu, M. Yang, Z. Wang, and C. Wu, "Ferry: State-Aware symbolic execution for exploring State-Dependent program paths," in *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Aug. 2022.

[59] M. Zalewski., "Afl—coverage-guided mutational fuzzer." 2021. [Online]. Available: https://github.com/google/AFL

Appendix A. Grammar-Aware Input Generation/Mutation

OTA messages processed by baseband tasks follow structured input formats specified by 3GPP standards, each with distinct encoding schemes. NAS messages follows TLV (Type-Length-Value) encoding [2], including both syntactic (i.e., structural) and semantic (i.e., value constraint) requirements. Traditional fuzzers [34], [59], however, do not account for these structured formats and instead rely solely on random byte mutations (i.e., havoc mutations) to generate inputs. As a result, most generated inputs fail to conform to the expected structures and are immediately rejected. Consequently, structure-unaware fuzzers fail to explore deeper parts of the message-handling process and cannot adequately test the subtle input validation logic of these protocols.

To address these limitations, previous work [30], [31] has employed context-free grammars (CFGs) to specify the syntactic structure of input formats. CFGs effectively capture structural rules, such as optional fields, choices, lists, repeating elements, and recursive fields. However, they cannot maintain semantic context across fields, for instance, CFG cannot express a TLV structure, where the *L* (length) field should match the actual length of the subsequent *V* (value) field. To overcome this limitation, some systems [32] incorporate SMT solvers on top of CFGs to evaluate semantic constraints, ensuring that relationships between fields are enforced. For example, Listing 5 defines `UeSecCap_T4LV`, an LV field with a length field `L1` and a value field `UeSecCapValue` (line 1). The value field can randomly take one of three lengths (16, 31, or 38 bits), selected by the random integer variable `c` (lines 2-5). To generate a valid LV field, an SMT solver enforces constraints ensuring `L1` matches the actual length of `UeSecCapValue` and that `c` is within the range of possible choices (line 6).

While SMT solvers support complex semantic validation, they introduce significant computational overhead, reducing fuzzing throughput and limiting scalability.

```
1 UeSecCap_T4LV = Concat(L1, UeSecCapValue)
2 L1 = BitVec(8); c = Int()
3 UeSecCapValue = If(c == 0,
4     BitVec(16),
5     IF(c == 1, BitVec(31), BitVec(38)))
6 solve([Length(ueSecCapvalue) == L1, c < 3])
```

Listing 5. LV Grammar Requiring SMT Solver

Our Approach. To address these limitations, we conduct a comprehensive analysis of message definitions and specifications in the technical documents [2], [3] for the NAS OTA messages we aim to test. We observe that the semantic constraints for these messages in the 3GPP standards are limited, allowing us to design a custom grammar with augmented semantic annotations tailored to this domain.

LORIS first defines syntactic rules using a CFG-like structure to capture field layouts, choices, optional elements, and other structural attributes of OTA messages. It then augments these rules with semantic annotations to enforce field dependencies and inter-field constraints without requiring computationally intensive solvers. This allows us to generate valid OTA messages by incorporating context directly into the generation process. For example, consider Listing 6, which illustrates LORIS's grammar when generating the same `UeSecCapValue` format presented in Listing 5. This grammar contains two components, the CFG definition specifying the message's syntactic structure (lines 2-4) and a semantic annotation that enforces a constraint on the generated message (line 1). The CFG includes two concatenation operators: $\sim$ for left-associative concatenation and < for right-associative concatenation. The rule on line 2 generates `UeSecCapValue` first, followed by `L1`. On line 4, the optional field operator `?` on the `BYTE` and `BITS7` primitives generates random values of one of three possible lengths for `UeSecCapValue`. After generating a syntactically valid `UeSecCapValue`, the semantic annotation enforces to assign the length of `UeSecCapValue` to `L1` (line 1), ensuring consistency between the length and value fields without requiring an SMT solver, thereby improving efficiency in valid message generation.

```
1 #[L1.value = UeSecCapValue.length]
2 UeSecCap_T4LV = { L1 < UeSecCapValue }
3 UeSecCapValue = {
4     BYTE ~ BYTE ~ (BYTE ~ BITS7 ~ BITS7?)?}
```

Listing 6. LORIS's Grammar Avoiding SMT Solving

Supported Semantic Annotations. To meet the message requirements outlined in cellular specifications [2], [3], Fuzzer extends its grammar model with semantic annotations providing *properties* and *operations* that account for inter-field dependencies, length constraints, and specific value requirements. The supported semantic properties include:

- `length`: specifies inter-field length constraints, ensuring that length tags are assigned by the actual byte count of their respective fields.
- `value`: assigns specific values to fields, either directly or through calculated constraints.
- `reps`: specifies the number of repetitions for a given field, which is used when a field or subfield can appear multiple times within a message structure (e.g., a list).

To modify these properties, Fuzzer employs semantic operations that allow precise control over field values:

- `Arithmetic Operations` (+, −): Used to increment or decrement values as required by the message specification. For instance, if a field length must account for header bytes, Fuzzer uses these operations to adjust the length value accordingly.
- `Logical Operations` (`AND`, `OR`, `XOR`, `RSHIFT`, `LSHIFT`): Used to enforce bit-level constraints. For example, if a technical document specifies that a specific bit within a field must always be set to 0, Fuzzer can use an `AND` operation with the appropriate bitmask.
- `Assignment Operation` (=) for direct value setting, enabling fields to take on specific values or modified values based on conditions derived from the protocol's semantic requirements.

Corpus Generation. Using the defined grammar, LORIS generates a diverse set of initial inputs to cover each specified message type. For each type, the generation algorithm employs a tree-based structure to explore all branching variations, ensuring coverage of possible formats.

Input Mutation Strategies. LORIS applies several mutation strategies for its test input generation. ***Grammar-aware random mutation.*** LORIS randomly selects a field from the message and regenerates it using the defined grammar. ***Grammar-aware splice mutation.*** This strategy combines elements from two messages to create a new valid test input. It randomly selects two messages of the same type (i.e., generated by the same grammar), identifies a random cut point within a field of the first input, and finds a compatible splice point in the second message based on the grammar. LORIS then joins the left portion of the first input with the right portion of the second, creating a new input, and ensuring the semantic annotations of the corresponding grammar are still satisfied. ***Probabilistic havoc mutation.*** LORIS also applies a havoc mutation strategy that introduces random changes at the raw-message level, without regard to grammar or structure, to create slight deviations from the defined grammar to test the decoding logic. Specifically, LORIS performs bit flips, arithmetic mutations, and insertion/deletion operations on initially grammatically valid inputs (or corpus mutated by previous havoc mutations). This mutation is probabilistically applied with a configurable probability (set to 0.3 in our tests) during each mutation round. Since havoc mutations break the syntax of the test input, any further grammar-aware mutations (i.e., the first two strategies) are no longer applicable to that message. For each mutation round, LORIS probabilistically applies one of the three mutation strategies up to a configurable number of times (set to 8 in our tests), enabling the generation of test inputs that range from slight modifications of the initial message to significant deviations.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

LORIS is a new cellular baseband fuzzer, which makes various improvements compared to prior work to uncover new vulnerabilities. It explores the state required to reach specific parsing functionality in an efficient way, enabling it to cover code that previous fuzzers could not reach with similar inputs. LORIS will be open-sourced, enabling future researchers to verify and built upon its exceptional results.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) Real-world impact: It uncovers new security vulnerabilities affecting real devices.
2) Strong experimental results: It outperforms state-of-the-art fuzzers like FirmWire and Grimoire.
3) Technical analysis: Explanation of techniques used, including state variable detection, precondition inference, and grammar-aware fuzzing.

B.4. Noteworthy Concerns

Exploitability: The authors described how the vulnerabilities can be exploited but did not create a proof of concept exploit.