Invariant-Guided Logical Testing of Open RAN Controllers

Tianchang Yang, Ali Ranjbar, Gang Tan, Syed Rafiul Hussain
The Pennsylvania State University
{tzy5088, aranjbar, gtan, hussain1}@psu.edu

Abstract

Open RAN (O-RAN) represents a fundamental shift in mobile network architecture, advancing interoperability and flexibility through open interfaces and software-driven components. While enabling programmability and innovation, this shift also makes the logical correctness of O-RAN components essential for the secure and reliable operation of the network. However, validating O-RAN's semantic correctness remains challenging due to system complexity, implementation diversity, and the absence of explicit correctness oracles. We present InvaRAN, a systematic testing framework for detecting logical flaws in O-RAN implementations using dynamically inferred program invariants as *proxies* for expected behavior. To reduce false positives and focus on semantically meaningful behaviors, InvaRAN classifies invariants into *critical* and *non-critical* categories based on their impact on program logic. Beyond traditional template-based invariant inference approaches that infer only limited semantic relations, InvaRAN captures inter-variable correlations across execution traces to discover more expressive semantic linkage. We evaluate InvaRAN on both platform components and xApps of two production-grade O-RAN controllers. InvaRAN uncovers nine previously unknown issues, including seven logical and two memory vulnerabilities, demonstrating the effectiveness of invariant-guided testing in exposing subtle, specification-silent bugs in O-RAN systems.

1 Introduction

Mobile networks' transition to 5G disaggregates the traditionally monolithic Radio Access Network (RAN) into modular components, enabling operators to source network functions from multiple vendors and flexibly optimize RAN deployments. Building on this architecture, the Open Radio Access Network (O-RAN) Alliance [16] advances open, interoperable, and intelligent RAN designs. A central innovation is the RAN Intelligent Controller (RIC), which supports service management and orchestration (SMO), enabling third-party applications to dynamically control RAN behavior in response to network events. By standardizing interfaces between RAN components and the RIC, O-RAN reduces reliance on vendor-specific controllers and allows operators to deploy, replace, and evolve RIC applications from diverse providers. O-RAN's service-based architecture (SBA) further improves scalability and adaptability. With a network-wide view of connected RAN nodes, the RIC enables coordinated control for congestion mitigation, network slicing, and energy optimization through fine-grained scheduling and device coordination.

However, this open and software-centric design also introduces significant security and correctness challenges. Integrating components from multiple vendors and third-party applications increases the likelihood of configuration oversights, interoperability issues, and implementation flaws. These vulnerabilities expose the RIC to unexpected or malicious traffic, leading to crashes or logical errors. Such issues degrade performance, weaken security, and threaten overall network stability. The heterogeneous nature of O-RAN and the complexity of its interactions further amplify the risk, as even minor logical errors can propagate and impact end users. Existing software and protocol testing tools [3, 10, 40, 63, 76] are effective at detecting memory issues with observable consequences, such as crashes or out-of-bounds accesses, but cannot automatically detect logical errors, which lack universal oracles. Current logical error detection approaches, including manual test oracle construction [48, 49, 68], formal modeling [24, 28, 62], and differential testing [27, 44, 70], all face major limitations with O-RAN's complexity, heterogeneity, and lack of formal specifications. Manual oracle does not scale to the wide range of RIC components and applications; incompleteness and unavailability of specifications hinder formal modeling; and proprietary, heterogeneous RIC implementations [17, 22] limit differential testing, since semantically equivalent components are often unavailable.

To address this, we observe that while formally verifying the functional correctness of xApps or RIC components is difficult and not scalable, their expected behavior can be empirically approximated from executions under benign condi-

tions. Based on this, we design and implement InvaRAN, the first systematic framework to infer likely program invariants and use them to detect logical errors in O-RAN components. More precisely, InvaRAN automatically infers likely program invariants from regular executions and uses them as behavioral oracles to expose potential flaws. From execution traces generated by benign traffic, InvaRAN extracts stable patterns of variable relationships and system states that serve as proxies for expected behavior. These invariants define a behavioral baseline that InvaRAN subsequently validates through fuzz testing. Deviations from this baseline indicate semantic inconsistencies, allowing InvaRAN to uncover logical vulnerabilities without requiring formal specifications.

A key challenge of this approach lies in the quality of the inferred invariants. Not all invariant violations indicate logical errors, as some merely capture benign input patterns rather than meaningful program semantics. To address this, InvaRAN classifies invariants into *critical* and *non-critical* sets through program analysis. Invariants that come with preceding validations or influence significant downstream processing are treated as critical, as they are more likely to reveal flaws, while others are deprioritized to reduce false positives. Moreover, existing invariant inference tools rely on rigid, template-based rules that limit detection to simple relations within narrow program contexts. InvaRAN overcomes this limitation by augmenting template-based inference with a correlation-based approach that captures inter-variable relationships across execution traces. By identifying variables that consistently change together, InvaRAN infers higher-level semantic relationships that generalize beyond lexical scopes and scale across entire program executions.

Evaluation. We evaluate InvaRAN on two widely adopted O-RAN implementations, covering both platform components and xApps. InvaRAN uncovers 9 previously unknown issues (7 logical errors and 2 crashes), leading to component crashes, acceptance of falsified metrics, inconsistent internal states, and stealthy Denial-of-Service (DoS) conditions.

Contributions. We make the following main contributions:

- We design and implement InvaRAN, the first systematic logical error detection framework for O-RAN components, using dynamically inferred invariants as behavioral oracles.
- We introduce critical invariant classification to reduce false positives and focus on semantically meaningful violations.
- We propose a scalable correlation-based invariant inference approach that captures inter-variable semantic relationships beyond template-based methods.
- We evaluate InvaRAN on two O-RAN platforms, uncovering 9 new logical and crashing issues, resulting in falsified metrics, inconsistent states, and stealthy DoS.

2 Background

Open RAN (O-RAN). The O-RAN Alliance [16] standardizes a multi-supplier 5G and future-G O-RAN architecture (Figure 1) where equipment from various suppliers can be easily combined to form the RAN. O-RAN adopts a service-based design, where functions are implemented as cloud-native microservices that communicate over network traffic. Each microservice can be independently developed, deployed, and scaled on general-purpose servers. Beyond RAN disaggregation, O-RAN introduces two RAN Intelligent Controllers (RICs). The *Near-Real-Time RIC (Near-RT RIC)* manages and optimizes the RAN in near real-time (10*ms*-1*s*) [59]. Its functionality is provided by modular *xApps*, often developed by third parties or open-sourced, targeting tasks such as load balancing, interference management, and QoS control. An O-RAN RIC can govern multiple RAN nodes (also called E2 nodes) from different operators. Each RAN node communicates with the Near-RT RIC through the *E2 interface*. *E2 Termination (E2T)* is E2's endpoint in the Near-RT RIC. To support xApps, the Near-RT RIC includes *platforming components* such as the Shared Data Layer (SDL) and E2T. The *Non-Real-Time RIC (Non-RT RIC)* is part of the Service Management and Orchestration (SMO) framework and handles strategic functions on timescales above 1s, such as policy control, network planning, and long-term RAN management.

Available implementations. Two widely used open-source O-RAN systems support commercial deployments [12, 18]: the O-RAN Software Community (O-RAN-SC) [17], developed by O-RAN Alliance [16] with Linux Foundation [15], and SD-RAN [22], by Open Networking Foundation [20]. Both systems are cloud-native, with components packaged as Docker images [9] and deployed in Kubernetes [13].

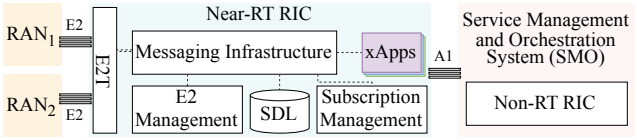

Figure 1: O-RAN RIC architecture

Likely program invariant inference. Formally, let C denote a target program, which is composed of a set of program points $\mathcal{P} = \{\rho_1, \rho_2, \dots\}$, where each $\rho \in \mathcal{P}$ represents a specific basic block in C. The execution of C may reach each program point ρ zero, one, or multiple times depending on the input and execution path. We refer to each dynamic occurrence of ρ, i.e., each time the program executes that basic block, as an *invocation* of ρ. Invocations, therefore, represent the runtime instances of program-point execution within a trace. Existing dynamic invariant inference tools [29] infer *likely local invariants* by observing variable values across the execution of C. Let τ denote an execution trace, capturing the runtime values of program variables at each invocation of program points throughout the recorded execution of C. Let I be the set of invariants inferred from the trace. Each invariant $i \in I$ is a logical predicate defined over the runtime values of one or more variables observed at a specific program point

$\rho \in \mathcal{P}$. A possible invariant predicate i^ρ at ρ is:

$$i^\rho : (v_1^\rho, v_2^\rho, \ldots, v_n^\rho) \mapsto \{true,\ false\},$$

where each v_j^ρ is a program variable observed at program point ρ. Note that variables are treated as lexically scoped to ρ, meaning that the same code variable appearing at different program points (e.g., v^ρ and $v^{\rho'}$ for $\rho \neq \rho'$) are considered distinct during invariant inference. We say i^ρ is a *likely invariant* in trace τ if, for all dynamic invocations ρ_k of ρ observed in τ, the predicate evaluates to true when instantiated with the corresponding runtime values:

$$\forall \text{ invocation } \rho_k \text{ of } \rho \in \tau, \quad i^\rho(v_1^{\bar{\rho}_k}, \ldots, v_n^{\bar{\rho}_k}) = true,$$

where $v_j^{\bar{\rho}_k}$ is the runtime value of variable v_j^ρ during the k-th invocation. If a single invocation produces values that falsify i^ρ, the predicate is no longer considered a likely invariant. These invariants are called *likely* because they are empirically inferred from a finite trace and do not come with formal guarantees of soundness (i.e., valid for all possible inputs) or completeness (i.e., capturing all behavioral properties).

3 Threat Model & Overview

Threat model. In O-RAN's highly interoperable environment, the RIC must interact with RAN components from multiple vendors and operators, each with distinct implementations and security practices. This heterogeneity exposes the RIC to unexpected, malformed, or malicious inputs from RAN nodes [76]. In addition, adversaries may introduce a compromised or malicious xApp into the RIC by exploiting supply-chain vulnerabilities [23, 51] or hijacking the onboarding process [21]. Once deployed, such an xApp can also generate adversarial inputs.

We consider both RAN nodes and xApps as potential sources of misbehavior, whether through implementation flaws or intentional attacks, without considering collusion among multiple adversaries. Prior work has assumed a similar threat model but primarily examined misbehaving RAN nodes [76]. We extend this model to also include malicious xApps. This threat model aligns with threats discussed in O-RAN's technical documents (e.g., under threats T-O-RAN-05, T-E2-01, T-E2-03, T-NEAR-RT-01, T-NEAR-RT-02, and T-xAPP-01 in O-RAN Alliance's risk assessment [56]). O-RAN's security study on Near-RT RIC and xApps [57] highlights both misbehaving RAN nodes and malicious xApps as critical risks, emphasizing that: (1) the Near-RT RIC must not assume that data received from the RAN is valid or trustworthy, and (2) sufficient access control and isolation must be enforced to prevent a misbehaving xApp from compromising benign components or other xApps. Exploitation under this threat model can lead to logical errors (e.g., accepting malformed inputs, access control violations) as well as memory-safety vulnerabilities. While prior efforts have focused mainly on memory corruption [76], our goal is to uncover logical flaws that threaten the integrity and security of the RIC components and applications.

Finally, we assume access to the source code of the tested components. This assumption is reasonable given O-RAN's open design and open-source RAN implementations, with growing support from industry [17, 22] and government initiatives [8]. We also discuss potential extensions of InvaRAN to gray-box (i.e., binary-only) targets in §7.

3.1 Motivation & Key Insights of InvaRAN

Motivating example. To highlight the challenges of detecting logical errors and motivate the methodology used by InvaRAN, consider the input reception and processing logic in Listing 1, simplified from an SD-RAN xApp [19], which optimizes User Equipment (UE, e.g., smartphones) handovers by analyzing radio conditions, cell types, and QoS profiles of connected UEs reported from E2 nodes (i.e., RAN nodes). The function `listenIndChan` (line 1) continuously receives and processes messages delivered via active E2 subscriptions. When a metric report is received, it is delegated to `handlePeriodicReport` for further processing in a separate thread (line 6). This function extracts the reported cell ID and UE ID (lines 14-15), retrieves the UE information stored in the xApp (line 18), and proceeds with subsequent processing, including the potential handover decisions.

```
1  func listenIndChan(ctx context.Context) {
2    for indMsg := range IndChannel {
3      switch x := indMsg.(type) {
4      case *IndMessageFormat1:
5        if indMsg.TriggerType == TYPE_PERIODIC {
6          go handlePeriodicReport(ctx, indMsg.Header,
       indMsg.Message)
7        }
8      ... // other cases
9      }
10   }
11 }
12
13 func handlePeriodicReport(ctx context.Context, hdr
       *IndHeaderFormat1, msg *IndMessageFormat1) {
14   cgi := GetCGIFromIndicationHeader(hdr)
15   ueId := msg.GetUeId()
16   mu.Lock()
17   defer mu.Unlock()
18   ueData := GetUe(ctx, ueIdString)
19   if ueData == nil {
20     ueData = CreateUe(ctx, ueIdString)
21     c.AttachUe(ctx, ueData, cgi, cgiObject)
22   } else if ueData.CGIString != cgi {
23     return // persistent message rejection
24   }
25
26   rsrpServing, rsrpNeighbors :=
       GetRsrpFromMeasReport(ctx,
       GetNciFromCellGlobalID(hdr.GetCgi()),
       msg.MeasReport) // measurement item accepted without
       validating legitimacy of the cell
27   if err := validateRsrp(rsrpServing); err != nil {
28     log.Errorf("invalid serv rsrp: %v", rsrpServing)
29     return
30   }
31   ... // other validation and handling using the
       extracted reported information
32 }
```

Listing 1: Simplified vulnerability in an SD-RAN xApp [19]

This implementation contains two critical logical vulnerabilities that InvaRAN detected: (1) ***Persistent message rejection:*** The condition in line 22 may create a sink state where all subsequent messages from the UE are ignored if

the UE's recorded cell identifier (`CGIString`) in the xApp does not match the reported `cgi`. This means that if a UE's state becomes inconsistent (e.g., due to an untracked handover or transient desynchronization), it will be permanently ignored by the xApp. As a result, valid measurement reports are repeatedly discarded without any corrective mechanism, leading to a denial-of-service (DoS) scenario for the affected UE. (2) ***Lack of cell validation.*** After line 25, the xApp processes reported measurement items without verifying whether the reported cell IDs are legitimate. This missing check introduces a security risk, as any RAN node can submit falsified reports. A malicious RAN node could exploit this weakness by injecting fake measurement data for cells it does not control, potentially leading to incorrect network optimizations, causing network starvation in specific areas or artificially increasing loads on targeted cells, degrading overall service quality, and potentially enabling targeted manipulation of optimization decisions. More details about potential attacks exploiting these issues are discussed in §6.1.

Limitations of existing approaches. Detecting such logical errors is difficult for existing methods:

L1: Lack of apparent failures. Unlike memory safety violations (e.g., buffer overflows), which typically result in crashes or are detectable via sanitizers [4], these issues are subtle and often unnoticed for years. Issue #1 results in the premature rejection of valid inputs without an obvious failure, while Issue #2 does not produce any immediate, discernible effect. The xApp accepts and processes falsified report metrics without triggering alerts. Detecting such inconsistencies requires the semantic understanding of system behavior, which existing fuzzing techniques [3, 10, 40, 63] are not designed to capture.

L2: Limitations of specification-derived oracles. Most previous approaches aiming at detecting logical vulnerabilities in various domains [48, 49, 68] construct properties or test cases from specifications, an effort often requiring extensive manual effort. However, these approaches can only detect vulnerabilities they are explicitly designed to uncover, and cannot generalize across different implementations or new functionalities. For instance, Issue #1 results from an implementation-specific sink state, which is not explicitly described in specifications and is difficult to detect without an in-depth understanding of the implementation. Additionally, while O-RAN specifications provide broad functional guidelines, they often lack security-specific properties that could serve as direct logical oracles. For example, for Issue #2, the traffic steering xApp specification states that "*the E2 measurements are necessary for inference and prediction in the Near-RT RIC as the driver for decisions*" [60], emphasizing the importance of correct and trusted measurement reports. However, specifications do not explicitly require validation of the correct association between a RAN node and its reported cells. As a result, even manually constructed oracles would likely fail to detect Issue #2. Finally, detecting such logical inconsistencies requires a deep semantic understanding of variable meanings within an implementation. One must invest significant effort in interpreting technical documents and analyzing code to infer the correct relationships between variables, which is time-consuming, error-prone, and impractical for large-scale systems.

L3: Ineffectiveness of differential testing. To address L2, some logical error detection approaches instead rely on differential testing [27, 44, 70], which identifies discrepancies between different implementations of the same functionality. However, differential testing is only effective when components perform identical functionalities, which is not the case for O-RAN RIC components. First, xApps, developed and deployed by various third-party vendors, lack standardized specifications or well-defined properties governing their expected behaviors since they are meant to perform diverse, customizable functionalities. While O-RAN defines general use cases such as traffic steering, resource optimization, and energy efficiency [60], it does not mandate specific behaviors for xApps. Differential testing is also ineffective for O-RAN's platforming components (e.g., E2T). While O-RAN's technical specifications define high-level operational expectations of these components, they do not standardize implementation details or internal communication messages. As a result, different implementations vary significantly. For example, the SD-RAN [22] consolidates multiple logical components, such as E2T, E2 management, and subscription management, into a unified E2T microservice. In contrast, the O-RAN-SC [17] separates these components into separate programs. They also use different internal communication methods, e.g., gRPC for SD-RAN, and a customized message router for O-RAN-SC with varying message formats and exchange patterns.

Key insights of InvaRAN. As demonstrated, detecting logical errors in O-RAN RIC components is challenging due to several key factors: logical errors often lack consistent, apparent consequences (e.g., crashes or assertion failures), platforming components exhibit diverse internal architectures and message formats, and xApps lack standardized specifications to define their expected behavior. As a result, building a unified testing framework that generalizes across heterogeneous implementations is challenging. InvaRAN sidesteps these challenges by shifting the focus away from implementation-specific assumptions or predefined correctness properties. Instead, it builds an empirical observation of expected behavior by analyzing patterns (i.e., likely invariants) in benign execution traces collected in a controlled environment. The central idea of InvaRAN is to *identify discrepancies between the empirically observed benign behaviors and the broader set of behaviors permitted by the code, by detecting violations of the inferred invariants.* Such violations can reflect a variety of logical issues that attackers could exploit, including semantically inconsistent behavior (e.g., Issue #1 in Listing 1), overly permissive logic (e.g., Issue #2), or missing validation checks. For example, in Issue #2, benign traces show that a

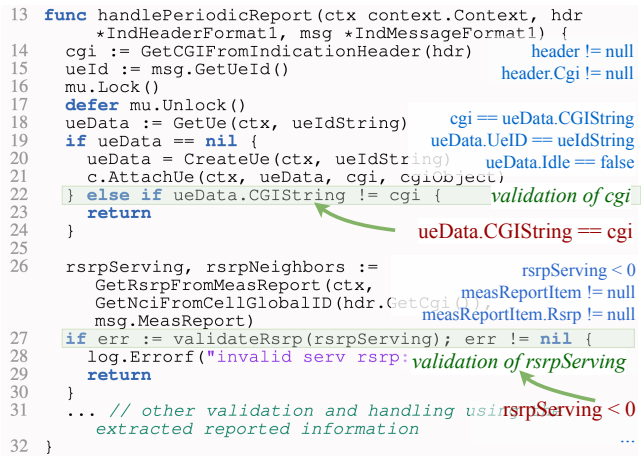

Figure 2: Sample invariants inferred from Listing 1

Blue texts: invariants inferred at corresponding program basic blocks they appear in; Red: invariants classified as critical (§4.1); Arrows: validation checks leading to the critical classification; Green regions: input validation.

RAN node consistently reports metrics only for its assigned cell IDs (detailed in §4.2). If, during testing, a mutated input causes the same node to report metrics for an unassigned cell and the xApp accepts it, the deviation from previously observed behavior indicates the presence of the issue.

To uncover such issues, InvaRAN first captures the normal operating behavior of the component under test (CUT) using only benign inputs generated through standard interactions with unmodified RAN components. We assume that execution under this environment is free of malicious behavior and reflects legitimate program semantics. During this phase, InvaRAN records the values of variables at different program points (i.e., in each basic block). From these traces, it infers invariant properties, i.e., recurring patterns that characterize benign behavior. These invariants serve as *proxies* for the program's operational expectations. For example, an array may always maintain a fixed length, a variable may never be null, or a static relationship may exist between a RAN node and its managed cell list. Figure 2 demonstrates likely invariants inferred from the motivating example (Listing 1). We summarize the types of invariants InvaRAN targets and the corresponding classes of potential errors they help reveal in Table 1. InvaRAN then performs dynamic fuzz testing with mutated inputs to explore alternative execution paths. If a mutated input causes a violation of any previously learned invariants, the behavior is flagged as suspicious and subjected to further triage. By relying on learned invariants rather than predefined oracles, InvaRAN adapts to diverse implementations and xApps without requiring manual modeling of expected behaviors. This allows it to generalize across components, accommodate functional variability, and scale to complex RIC systems where specifications are incomplete or lack implementation-level requirements.

Table 1: Invariant types and issues they could discover

Local invariants inferred within a single function or program point (inherent limitations of existing tools [29]); *Global* invariants capture relationships that span lexical scopes.

Invariant Type	Potential Logical Errors	Scope	Example
Exact value	Unexpected state modifications, incorrect value updates	Local	`UeData.RrcState = "CONNECTED"`
Range constraints	Invalid input acceptance, improper boundary checks	Local	`measItem.Rsrp < -50`
Relations (binary and ternary invariants)	Unexpected code execution, missing cross-variable checks, incorrect permission checks	Local	`plmnId = mcc || mnc`
Consistent correlation (§4.2)	Missing checks, failure to enforce expected relationships between variables	Global	`e2NodeID ↔ cgi`

Limitations of static and LLM-based inference. Besides dynamically inferred invariants, prior works also explored inferring invariants through static analysis [25, 77] or using large language models (LLMs) [32, 61, 64, 73]. However, these methods often suffer from inaccuracy due to the inherent unsoundness and incompleteness of static analysis, and the tendency of LLMs to hallucinate or miss invariants. More fundamentally, source code analysis-based approaches infer invariants (e.g., pre-conditions or post-conditions of functions) that hold across *all* executions, describing what the code permits rather than how the system should behave. For buggy code, these invariants capture the buggy behavior itself rather than the intended semantics, and thus cannot serve as reliable oracles. These invariants are useful for program verification when formal specifications are available, but less applicable in systems like O-RAN, where specifications are incomplete or absent. Instead, InvaRAN learns invariants from benign runtime traces, capturing how the system behaves under normal operation. This enables InvaRAN to highlight deviations that are syntactically allowed by the code but not observed from benign executions, exposing logic flaws through this slight semantic gap. Finally, O-RAN's codebase spans multiple distributed components that communicate via network traffic, making the holistic system logic difficult to analyze statically. This architectural fragmentation creates implicit cross-component dependencies that are hard to detect, link, or reason about, leading to inaccuracies in static analysis and LLM-based techniques [77].

3.2 Challenges of InvaRAN

We summarize a few critical challenges InvaRAN faces in realizing its high-level idea.

C1: Minimizing false positives in logical bug detection. Not all invariant violations indicate true logical errors. Even when using large volumes of execution traces and extensive sampling, inferred invariants represent patterns observed under benign inputs and do not necessarily capture the component's intended or semantically meaningful behavior. As a result,

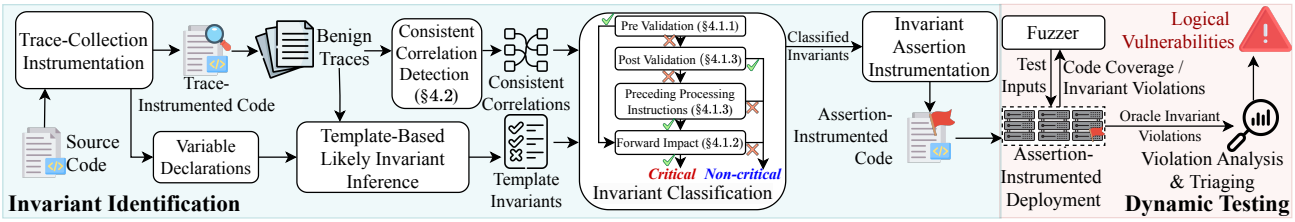

Figure 3: Workflow of InvaRAN, divided into two stages: *invariant identification* and *dynamic testing*

some invariants may be violated trivially by malformed inputs, without indicating any real flaw in the implementation. For example, consider the handling of RSRP (Reference Signal Received Power), a measurement representing the signal strength between a UE and its surrounding cells. RSRP values are negative, with higher (less negative) values indicating stronger signal quality. In a well-designed implementation, a positive RSRP value should be flagged as invalid and discarded early. Consider the RSRP processing code in lines 26–30 of Listing 1. Invariants derived from benign executions may record that RSRP values are always negative at both the retrieval point (line 26) and after the validation check (line 31). However, if a fuzzed input contains a positive (and thus invalid) RSRP value, it will trivially violate the pre-validation invariant at line 26. This does not reflect a logical error as long as the component validates and discards the input (line 27). A logical flaw is only indicated if the invalid RSRP is mistakenly accepted and propagated beyond the validation check (e.g., violating the invariant at line 30). Distinguishing between spurious violations and genuine logical bugs is challenging. O-RAN components process a variety of inputs, execute diverse code paths, and apply validation checks at different stages of processing. Some fields are checked immediately upon message receipt, while others may be validated only before being used. Furthermore, the structure and location of validation logic differ across implementations, making it difficult to determine when a field is validated.

C2: Limited scope of invariants. Existing invariant inference tools [29] are constrained by static, predefined expression templates. These tools infer invariants by matching observed variable values against a fixed set of candidate patterns (e.g., linear arithmetic relationships, bounds, or equality checks). While many of these tools are extensible, allowing users to define new templates, they remain fundamentally limited: (1) their fixed set of templates may miss a large number of possible program behaviors or semantic relationships, for example, implicit correlations such as value coupling (where changes in one variable consistently trigger changes in another) or many-to-many correlations (where sets of input values correlate with sets of output values in complex patterns, such as the relationship between cells and the UEs connected to them), and (2) due to scalability concerns, they typically restrict analysis to a narrow lexical scope, comparing variables only at the same program point or within a function scope, thus failing to capture relationships that span across function boundaries or between semantically related but syntactically distant variables. As a result, these tools cannot detect higher order semantic invariants. For example, detecting Issue #2 in Listing 1 requires capturing a consistent mapping between a RAN node ID and its associated cell IDs. This is difficult for two reasons: (1) the relationship is a map from a string (the E2 node ID) to a list of strings (cell IDs), which lies outside the expressiveness of standard scalar templates (e.g., numeric bounds), and (2) the relevant variables are extracted in different functions: the node ID is parsed in one function (omitted), while the cell ID appears in another (line 14). Tools limited to intra-procedural analysis cannot infer their association.

C3: Preserving context in message-driven and concurrent execution. Invariant inference tools are traditionally applied to simple, sequential programs that handle a single input type at a time. In contrast, O-RAN components process diverse message types (e.g., setup, update, report) and operate in highly concurrent environments, where multiple threads handle different messages simultaneously. These characteristics introduce challenges for accurate invariant inference. Existing trace collection methods [6, 7] typically infer invariants at the function level by analyzing variable behavior across repeated invocations. However, they treat the execution trace as a flat sequence, lacking awareness of both message boundaries and thread contexts. This leads to two key limitations: (1) In concurrent settings, trace entries from multiple threads can become interleaved, obscuring message attribution. (2) They may miss invariants that hold only within the scope of a specific message type. For example, a variable might remain stable during the processing of report messages but differ between reports and setup messages. Without distinguishing message types, such invariants are overlooked.

3.3 Approach Overview

To address the challenges outlined in §3.2, we design and implement InvaRAN. InvaRAN operates in two main stages: invariant identification and dynamic testing, as illustrated in Figure 3. In the *invariant identification* stage, we deploy the component under test (CUT) together with the necessary O-RAN components in a controlled environment populated with multiple RANs configured to induce diverse execution behaviors. In our setup, we instantiate 10 simulated RAN nodes

with heterogeneous coverage sizes, handover parameters, and E2 reporting settings. Each RAN node is associated with between 1 to 5 cells, and each cell manages on average 15 UEs with randomized mobility patterns to emulate varying radio conditions and traffic loads. This controlled deployment generates traces covering both steady-state operations and benign error-handling behaviors (e.g., transient disconnects, delays, and non-malicious configuration inconsistencies), while excluding any intentionally adversarial or malformed inputs in this stage. The deployment operates for 24 hours, during which we record all message exchanges generated under these benign conditions, along with the internal variable values observed within CUT. We choose a 24-hour window to capture a broad spectrum of normal operational behaviors, including periodic events and other infrequent interactions that may not surface in shorter runs. Note that benign execution cannot fully enumerate all semantically valid behaviors. As a result, the collected traces provide an empirical approximation rather than an exhaustive enumeration of expected program behavior. Invariants inferred from these traces, therefore, reflect common-case semantics but may miss benign behaviors, leading to potential false positives in vulnerability detection, or overlook unexercised execution paths, leading to false negatives. We evaluate the impact of this limitation in §6.2.1.

From the collected traces, InvaRAN infers likely program invariants. To address Challenge C1, InvaRAN classifies inferred invariants into critical and non-critical subsets to prioritize those most likely to reflect meaningful program logic (§4.1). It identifies invariants that appear in semantically meaningful *input processing logic* as *critical*, by detecting preceding input validation checks and analyzing instruction-level heuristics. InvaRAN uses critical invariants as bug oracles. It uses other invariants only as feedback signals to guide dynamic testing. This filtering reduces false positives by excluding invariants that are unlikely to correspond to true logical violations. To address challenge C2, InvaRAN augments template-based invariant inference with a correlation analysis that captures complex relationships (e.g., many-to-many mappings) between variables across traces (§4.2). This enables the discovery of semantic properties (e.g., node-to-cell mappings) that are not expressible using traditional scalar invariant templates. To address challenge C3, InvaRAN performs context-aware trace attribution by instrumenting thread creation and message dispatch logic. This allows it to maintain thread hierarchies and accurately associate trace segments with specific input messages and sessions. As a result, invariants can be inferred at component-, message-, or session-level granularity. We leave details on trace attribution in Appendix B. During *dynamic testing*, InvaRAN instruments the CUT with assertions of all inferred invariants and performs feedback-guided fuzz testing. Inputs violating critical invariants are flagged as potential bugs for further triage, while violations of non-critical ones provide feedback to guide continued fuzzing.

4 Approach Details

4.1 Invariant Classification and Prioritization

Analyzing violations of every invariant indiscriminately would result in a high volume of false positives, wasting manual analysis efforts (Challenge C1 in §3.2). To address this, InvaRAN distinguishes between invariants based on their likelihood of revealing true logical errors. It classifies them as either *critical*, which are more likely to reflect meaningful program semantics and serve as bug oracles, or *non-critical*, which are less reliable as indicators of bugs but still useful for guiding input mutation during fuzzing. This classification is informed by the invariant's program context, including its control-flow location and data-/control-dependencies relative to input-derived fields. We observed that in typical input-driven systems like O-RAN components, inputs pass through a processing pipeline that includes decoding, validation (i.e., input sanitization), and downstream program logic/behavior such as state updates or control decisions. During decoding and validation, malformed or semantically incorrect inputs are filtered out. In contrast, values that pass through these stages influence downstream behavior and are assumed to be meaningful to the processing logic. Following this observation, InvaRAN leverages this insight: invariants inferred *after* validation/sanitization checks are more likely to characterize trusted internal states, implicit/explicit assumptions/requirements, or semantic behavior of the program. These invariants describe the relationships among values that are not only accepted by the system but also actively used in the system's decision-making, making them strong candidates for identifying logic bugs. Conversely, invariants inferred *before* or *during* validation/sanitization mostly only capture syntactical input properties (e.g., formatting patterns) and are prone to trivial violations by malformed inputs, which are often safely discarded during validation. For example, in Listing 1, an invariant over the RSRP value inferred after line 27 (a validation check) may guide handover decisions. A violation of this invariant could signal a genuine bug in the control logic. In contrast, an invariant inferred at line 26 before RSRP validation is more likely to be broken by malformed inputs that are subsequently rejected at line 27, and thus does not reflect a real logic error.

However, identifying whether an invariant appears after a validation check is non-trivial, since such checks are often distributed throughout the component, implemented across multiple functions, and invoked separately along different execution paths. Some components (e.g., E2T) centralize validation early in the message-processing pipeline, while others (e.g., xApps) perform inlined or field-specific validation at the point of use. This variation makes it infeasible to identify validation logic using simple syntactic information. To address this, we introduce an efficient static analysis that heuristically approximates validation checks. A decision tree illustrating

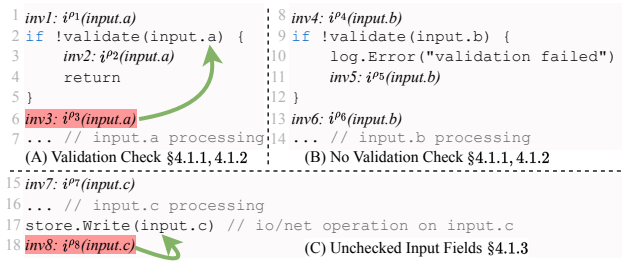

Figure 4: Example scenarios considered in §4.1

Red highlights invariants classified as critical, and arrows point to the specific instructions leading to the classification.

the classification is presented in Figure 3. Figure 4 demonstrates scenarios INVARAN considers and invariants classified as critical in each scenario. §4.1.1 to §4.1.3 incrementally present the core ideas and techniques, and §4.1.4 combines the discussion and formalizes the classification criteria.

4.1.1 Branch identification for sanitization checks

We first observe that validation checks take the form of branching decisions. That is, the program inspects certain input fields (e.g., line 27 in Listing 1) and, based on their values, either continues processing the input (e.g., line 31) or rejects the message without further processing (e.g., line 29). Therefore, tracking branching conditions that (1) depend on the same input field as used in the invariant (i.e., via dataflow), and (2) influence the reachability of the invariant site (i.e., the invariant site is control dependent on the branch instruction) provides a good approximation of whether the field has been validated prior to reaching the invariant. For example, consider an invariant involving the `rsrp` field. This analysis distinguishes between operations on `rsrp` that occur before line 27 (i.e., prior to validation) and those after line 31 (i.e., post-validation), allowing us to identify line 27 as the relevant validation check. In contrast, if a branch condition does not alter the control flow leading to the invariant site (i.e., all paths lead to the site regardless of the branch outcome), it likely reflects either general processing logic or non-critical checks (e.g., the logging-only branch in line 10 of Figure 4). Such branches are excluded from INVARAN's validation inference. Overall, this strategy indicates that input variables used at the invariant site have been subjected to some validation logic that influences the control flow.

4.1.2 Forward impact analysis

After an upstream sanitization check is found, INVARAN identifies invariants that influence meaningful downstream processing and marks them as critical. To do so, it performs a lightweight forward impact analysis to determine whether an inferred invariant lies on a path that leads to further computation rather than early message rejection. For example, in Listing 1, the invariant on `rsrp` at line 28 depends on the validation at line 27 but immediately results in rejection, and thus is not critical. In contrast, invariants on paths leading to continued processing (e.g., line 31) are considered impactful.

To assess this, INVARAN begins at the invariant's program point ρ and computes a bounded forward program slice S that traverses both data and control dependencies. The slice terminates under either of two conditions: (i) a fixed traversal hop limit h is reached to ensure scalability, or (ii) control returns to the main message-processing loop (i.e., the receipt of the next input). Concretely, this analysis performs a breadth-first search (BFS) traversal over the control-flow graph (CFG) starting from the basic block containing ρ. During traversal, INVARAN identifies *computation instructions*, which are instructions that indicate downstream (i.e., in the forward slice) computation behaviors by tracking: (1) arithmetic and logical operations, recognized from SSA instruction types (e.g., binary or unary operations), and (2) I/O or network operations, identified via heuristic analysis of function calls, for example, access to standard libraries (e.g., `net`) or file operations. If the number of processing sinks in S exceeds a small heuristic threshold θ, the invariant is classified as critical, reflecting its influence on downstream logic. This analysis is designed to be lightweight by using only instruction-type classification and a bounded BFS traversal, avoiding expensive whole-program analysis. Note that the presence of a larger number of downstream processing instructions does not directly imply a more impactful invariant. In fact, it may indicate that more validation checks are still present downstream. Therefore, INVARAN foremost ensures the existence of upstream validation checks (i.e., checks found in §4.1.1) and uses the processing threshold only to confirm that the validated invariant meaningfully influences downstream behavior, rather than being rejected.

4.1.3 Processing of unchecked input fields

While identifying input-related sanitization checks (§4.1.1) helps determine whether an invariant is preceded by explicit validation logic, it only applies to fields that are directly validated within the component. However, O-RAN messages often contain dozens to hundreds of fields, many of which are not explicitly checked. Some requirements are implicitly enforced through message schema (e.g., value ranges or length constraints), meaning only syntactically valid messages are accepted. Yet other requirements, such as the validity of identifiers or the semantic correctness of values, cannot be encoded and thus may go unchecked. As a result, many fields may be used in processing logic without any validation checks (i.e., scenario C in Figure 4). The absence of checks may also lead to logical errors caused by missing validations. To detect these, INVARAN applies a fallback analysis when no preceding control-altering branches are found for an invariant.

In this case, INVARAN performs a forward analysis (sim-

ilar to §4.1.2) to check whether validation of the input field occurs downstream of the invariant site. It searches for branch conditions derived from the same input field. If such a branch is found, and one branch path leads to meaningful processing (i.e., satisfies §4.1.2) while the other leads to rejection (i.e., fails §4.1.2), InvaRAN infers the presence of a downstream validation. Because the invariant precedes this sanitization check, it is classified as non-critical. To improve coverage, this analysis uses a larger hop limit h' than the h applied in §4.1.2, but due to its higher cost, it is only invoked when no earlier validation branches are found in §4.1.1.

If no downstream validation is detected, InvaRAN determines that the corresponding input field is never explicitly checked. Yet, if such a field is directly used in meaningful processing logic, its misuse is still highly likely to cause incorrect behavior, and invariants over that field should be treated as critical. To capture this, InvaRAN examines the preceding context of the invariant's variables to distinguish between validation and processing logic. An invariant is deemed critical if its backward slice reveals strong *processing cues*, such as I/O or network operations (e.g., line 17 in Figure 4) that typically indicate reporting, state updates, or control-plane communication. Because such operations rarely appear in validation routines, they provide reliable evidence that the invariant belongs to processing logic. This heuristic is a stricter variant of the computation-instruction analysis described in §4.1.2. The backward slicing is used only when no sanitization is found, conservatively flagging unchecked invariants that influence critical processing. General arithmetic and logical operations are excluded, as they occur in both validation and processing code and are thus less discriminative. Overall, this analysis allows InvaRAN to identify invariants on unchecked input fields that nonetheless participate in critical processing paths, where violations could compromise program semantics.

4.1.4 Formalizing critical invariants classification

Figure 3 summarizes the above heuristics as a decision workflow. For each invariant variable, InvaRAN first checks whether the invariant site is gated by an upstream sanitization branch whose condition depends on corresponding input-derived value (§4.1.1). If so, it further checks whether the invariant influences downstream processing (§4.1.2); such invariants are considered *critical*, while reject-only ones are classified as *non-critical*. If no upstream validation exists, InvaRAN searches for a downstream validation (§4.1.3); invariants appearing before such checks are non-critical. Finally, when neither pre- nor post-validation is present, invariants are flagged as critical only if the input-derived value participates in semantic processing, capturing unchecked but semantically meaningful fields. Below, we formalize this workflow.

Following the notations in §2, let $i^{\rho} \in I$ be an invariant observed at program point ρ, involving one or more program variables. If i^{ρ} involves multiple variables, each variable is analyzed independently using the criteria below, and i^{ρ} is classified as *critical* only if *all* involved variables satisfy the corresponding conditions (i.e., logical AND). Formally:

Definition 1 (Critical Invariant)**.** $i^{\rho} : \{v_1^{\rho}, \ldots, v_n^{\rho}\}$ is a **critical invariant** only if $\forall v_m^{\rho}$ $(1 \le m \le n)$ it holds: v_m^{ρ} is either **validated and impactful (Definition 2)** or **unchecked but used in semantic logic (Definition 3)**. Otherwise, i^{ρ} is non-critical.

Let v_{inv} be one such variable in i^{ρ}, and $\mathcal{V}_{\text{inv}}$ the set of variables that v_{inv} is data-dependent on (i.e., variables from which v_{inv} is derived). Let $\mathcal{V}_{\text{input}} \subseteq \mathcal{V}_{\text{inv}}$ denote the subset of $\mathcal{V}_{\text{inv}}$ derived from input messages. Dataflow analysis [69, 72] augmented with field-sensitive taint analysis [33, 52] allows the approximation of may- and must-dependencies on input fields. Let S_{ρ} denote the forward slice and $\widehat{S}_{\rho}$ the backward slice from ρ. Let $\#_{\text{comp}}^{\mathcal{V}_{\text{input}}}(S_{\rho})$ denote the number of computation instructions in S_{ρ} on $\mathcal{V}_{\text{input}}$(§4.1.2).

Definition 2 (Validated (§4.1.1) and Impactful (§4.1.2))**.** A variable v_{inv} is considered validated and impactful if:

- **Validated:** $\exists$ branch $b \in \widehat{S}_{\rho}$ whose condition depends on some $v \in \mathcal{V}_{\text{input}}$ s.t. ρ is control-dependent on b, and
- **Impactful:** $\#_{\text{comp}}^{\mathcal{V}_{\text{input}}}(S_{\rho}) > \theta$

If no such validation check exists, InvaRAN performs an extended forward analysis to search for downstream branches on variables in $\mathcal{V}_{\text{input}}$. If such a branch b' is found where one path leads to rejection ($\#_{\text{comp}}^{\mathcal{V}_{\text{input}}}(S_{b'}) \le \theta$) and the other to meaningful processing ($\#_{\text{comp}}^{\mathcal{V}_{\text{input}}}(S_{\neg b'}) > \theta$), the invariant is deemed non-critical, as it lies before the inferred validation check. If no downstream validation branch is found, InvaRAN examines the backward slice $\widehat{S}_{\rho}$ to check for evidence of processing-related behavior. Let $\#_{\text{proc}}^{\mathcal{V}_{\text{input}}}(\widehat{S}_{\rho})$ denote the number of instructions in $\widehat{S}_{\rho}$ that demonstrate processing cues defined in §4.1.3 (e.g., I/O operations). If $\#_{\text{proc}}^{\mathcal{V}_{\text{input}}}(\widehat{S}_{\rho}) > 0$, InvaRAN classifies the invariant as involved in processing and thus critical.

Definition 3 (Unchecked but Used in Semantic Logic (§4.1.3))**.** v_{inv} is considered as no explicit validation but used in semantically meaningful logic if all the following hold:

- **No preceding validation:** v_{inv} does not satisfy the validated condition from Definition 2, and
- **No post validation:** $\nexists$ branch $b' \in S_{\rho}$ with condition on $v \in \mathcal{V}_{\text{input}}$ s.t. $\#_{\text{comp}}^{\mathcal{V}_{\text{input}}}(S_{b'}) > \theta$, $\#_{\text{comp}}^{\mathcal{V}_{\text{input}}}(S_{\neg b'}) \le \theta$, and
- **Used in semantic logic and impactful:** Both of the following hold: $\#_{\text{proc}}^{\mathcal{V}_{\text{input}}}(\widehat{S}_{\rho}) > 0$ and $\#_{\text{comp}}^{\mathcal{V}_{\text{input}}}(S_{\rho}) > \theta$

For implementation details, we provide the algorithm for validation check detection in Appendix A, offering further insight into how InvaRAN identifies relevant branches.

4.2 Consistent Correlation Detection

To address Challenge C2 (§3.2), InvaRAN aims to expand invariant relationship detection beyond the existing rigid,

template-based scalar invariants. For example, an E2 node associated with a list of cell IDs, a policy containing multiple UE identifiers, or a measurement item carrying multiple bytes of reported values. These relationships are hard to capture because they encode rich semantic meanings, cannot be summarized by simple templates, and often reflect one-to-many or many-to-many associations rather than one-to-one mappings. To detect such relationships, we leverage a fundamental intuition: if two variables are semantically linked, they should exhibit *correlated changes*, i.e., changes in one variable should be consistently accompanied by changes in the other. Conceptually, if a variable a is a function of b (e.g., $a = f(b)$), then changes in b should induce changes in a. For example, a change in the E2 node ID typically leads to a different set of associated cell IDs, or a shift in service type may correspond to changes in the QoS profile. These co-occurrence patterns recur consistently across traces and form the foundation of InvaRAN's correlation inference.

Trace representation and cluster mappings. InvaRAN uses a trace-based correlation approach grounded in a simple principle: if two variables are consistently related, their values should change together across traces. That is, values for one variable should consistently co-occur with specific groups of values for another variable. This notion aligns with the concept of functional dependencies [38] to capture relational abstraction [35], extended here to support many-to-many relationships observed dynamically. While such associations may not follow a strict one-to-one mapping, they form stable, disjoint partitions, e.g., different E2 nodes associated with distinct sets of cell IDs. When this pattern recurs across many traces, it reveals an underlying semantic link. Conversely, if over time every value of one variable appears with every value of the other, i.e., only one partition exists, the variables are deemed unrelated. This observation allows InvaRAN to infer many-to-many correlations without knowledge of data schema, message format, or domain-specific semantics.

Formally, let $T = \{\tau_1, \ldots, \tau_m\}$ be the set of execution traces, where each trace τ_k records the values of all trace variables observed during a segment of execution. Same as prior inference, trace variables are lexically scoped to their program location (i.e., basic block), so the same code variable at different program locations is treated as distinct. For ease of exposition, we assume each τ_k corresponds to the processing of one input message. In Appendix B, we describe more trace segmentation strategies. We define $\texttt{val}_{\tau_k}(v)$ as the set of values observed for v within τ_k (possibly across multiple dynamic occurrences, e.g., loop iterations). For any variable pair (v_i, v_j) and trace τ_k, we define a *mapping cluster* as:

$$\mathcal{C}_k^{(i,j)} = \left(\texttt{val}_{\tau_k}(v_i),\ \texttt{val}_{\tau_k}(v_j)\right)$$

It captures the co-occurring sets of values for v_i and v_j during trace τ_k, forming the atomic unit for correlation inference.

Cluster merging for correlation inference. Let $\mathcal{C}_m^{(i,j)} = (\texttt{val}_{\tau_m}(v_i), \texttt{val}_{\tau_m}(v_j))$ and $\mathcal{C}_n^{(i,j)} = (\texttt{val}_{\tau_n}(v_i), \texttt{val}_{\tau_n}(v_j))$ be mapping clusters observed in traces τ_m and τ_n for variable pair (v_i, v_j). InvaRAN merges the clusters if:

$$\texttt{val}_{\tau_m}(v_i) \cap \texttt{val}_{\tau_n}(v_i) \neq \emptyset \text{ or } \texttt{val}_{\tau_m}(v_j) \cap \texttt{val}_{\tau_n}(v_j) \neq \emptyset$$

In that case, the merged cluster is:

$$\mathcal{C}_m^{(i,j)} \cup \mathcal{C}_n^{(i,j)} = \left(\texttt{val}_{\tau_m}(v_i) \cup \texttt{val}_{\tau_n}(v_i),\ \texttt{val}_{\tau_m}(v_j) \cup \texttt{val}_{\tau_n}(v_j)\right)$$

Otherwise, the clusters are kept separate. The result is a partitioning of clusters into a set of disjoint merged groups. This merging step accumulates consistent co-occurrence patterns observed across traces. If two clusters share any value for either variable, it suggests that the corresponding values are part of the same semantic group. InvaRAN consolidates partial observations by merging such clusters. In O-RAN implementations, correlated values may appear incrementally across messages. For example, a reporting message from a RAN node may include only the subset of cells with changed measurements, rather than all cells it manages. Merging thus enables InvaRAN to reconstruct the complete association over multiple messages. Each final merged cluster represents consistent co-occurring values for (v_i, v_j) across traces, suggesting a stable many-to-many correlation. Conversely, if merging collapses all observations into a single cluster containing all values of v_i and v_j, the variables are deemed *uncorrelated*. Overall, InvaRAN performs the following steps:

1. Hypothesize that every variable pair $(v_i, v_j) \in V \times V$ satisfying $v_i \neq v_j$ may be correlated.
2. For each $\tau_k \in T$, construct $\mathcal{C}_k^{(i,j)}$ for all such pairs.
3. Merge clusters according to the presented method.
4. After processing all traces, if a single cluster remains such that $\mathcal{C}^{(i,j)} = \left(\bigcup_k \texttt{val}_{\tau_k}(v_i),\ \bigcup_k \texttt{val}_{\tau_k}(v_j)\right)$, then (v_i, v_j) is considered to have no consistent correlation. Otherwise, multiple disjoint clusters indicate a *stable correlation*.
5. To reduce noise, a correlation between a variable pair is retained only if it is observed in at least a threshold number or ratio of traces. Within each correlation, individual value clusters are preserved only if the corresponding value pairs appear with sufficient frequency across traces.

The same criticality analysis described in §4.1 is then applied to each variable in the inferred correlations. A correlation is used as a logical oracle only if all participating variables are classified as critical. This extended analysis addresses two key limitations of existing template-based invariant inference. First, it removes the need for rigid templates and enables the discovery of semantically rich relationships purely through observed correlations. Second, due to its lightweight merging-based design, the analysis can be efficiently applied across all variable pairs in $V \times V$, rather than limited to variables observed at the same program point.

Example of Correlation Inference. To illustrate InvaRAN's correlation detection methodology, consider the simplified code in Listing 2. An incoming message is processed to extract the associated CGI (Cell Global Identifier, line 2), and each report item is separately processed to extract its UE ID value (line 10). Although the variables `cgi` and `ueid` are not used together in the same scope, each cell node

typically is connected with a stable set of UEs. Suppose the following trace observations τ_1, τ_2, and τ_3 are made.

```
1  func ProcessMessage(msg *indicationMessage) {
2      cgi := extractCGI(msg) // CGI extraction
3      for item := range msg.ReportItems {
4          processCellItem(item)
5      }
6      ... // further processing
7  }
8
9  func processCellItem(item *reportItem) {
10     ueid := getUEID(item) // UE ID extraction
11     ... // further processing
12 }
```

Listing 2: Illustrative processing logic of a component

First, in trace τ_1, we observe:

$$\texttt{val}_{\tau_1}(\texttt{cgi}) = \{1\}, \quad \texttt{val}_{\tau_1}(\texttt{ueid}) = \{a, b\}$$

Resulting an initial cluster: $(\{1\}, \{a, b\})$. Then in trace τ_2:

$$\texttt{val}_{\tau_2}(\texttt{cgi}) = \{3\}, \quad \texttt{val}_{\tau_2}(\texttt{ueid}) = \{c, d\}$$

A second disjoint cluster forms: $(\{3\}, \{c, d\})$. InvaRAN infers a potential correlation: CGI 1 maps to UE IDs $\{a, b\}$, and cell 3 maps to $\{c, d\}$. This partitioning suggests a stable correlation relationship. These isolated observations alone are insufficient to support any semantic inference. Instead, when such disjoint mappings are observed repeatedly and consistently across thousands of traces by InvaRAN, they provide strong statistical evidence of an underlying semantic association. Now consider a third trace τ_3:

$$\texttt{val}_{\tau_3}(\texttt{cgi}) = \{1\}, \quad \texttt{val}_{\tau_3}(\texttt{ueid}) = \{a, b, c\}$$

This observation may arise, for example, when UE c is handed over from cell 3 to cell 1. Because the UE ID c now overlaps with a previously disjoint cluster, InvaRAN merges the clusters according to its merging rule:

$$\mathcal{C}_{1,2,3}^{(\texttt{cgi},\texttt{ueid})} : (\{1, 3\}, \{a, b, c, d\})$$

The expanded cluster now includes all previously observed values, collapsing the partition. As such overlaps occur, the mapping may converge to an all-to-all relationship. Once a single cluster spans all values of both variables, InvaRAN concludes that no consistent correlation exists between `cgi` and `ueid`, as there is no longer any stable partitioning structure. Conversely, if clusters remain consistently disjoint over time (e.g., when each operator's E2 nodes manage a stable and non-overlapping set of UEs), InvaRAN infers a strong correlation between the variables.

5 Implementation

We implement context-aware thread tracing and message attribution (Appendix B) by manipulating the Go abstract syntax tree (AST). The same AST-based approach is used to instrument target programs for collecting runtime variable traces, which serve as input to invariant inference. InvaRAN supports two types of invariant inference: (1) scalar invariants using Daikon [29], and (2) correlation-based invariants (§4.2). The correlation analysis is implemented in Python in ∼500 lines of code, while inferred invariants are re-instrumented into components under test as Go AST assertions (∼1,400 lines of code) to provide runtime feedback during fuzzing. The fuzzing engine is implemented in Go by extending ORANalyst [76]. InvaRAN generates grammar-aware, semantically valid inputs, emulating messages from potentially malicious or misbehaving RAN nodes or xApps, and probabilistically applies havoc mutations to produce slightly malformed inputs [37, 76]. At runtime, the fuzzer retains inputs that break invariants and reports violations of critical invariants as potential issues for manual triaging.

6 Evaluation

We perform all evaluations on a machine with an Intel Xeon Gold 6448H @ 4.1GHz CPUs and 1007GB RAM. The invariant classification (§4.1) and consistent correlation detection (§4.2) components were allowed to utilize the full available memory. Each fuzzing instance was executed within a virtual machine running the corresponding O-RAN system, and allocated a maximum of 4GB RAM and 4 CPU cores. We evaluate InvaRAN with the following research questions:

- Can InvaRAN find logical issues in O-RAN? (§6.1)
- How effective is each component of InvaRAN? (§6.2)
- Is InvaRAN extensible beyond O-RAN RIC? (§6.3)

6.1 InvaRAN Findings

We evaluated InvaRAN on two widely adopted O-RAN implementations, SD-RAN [22] and O-RAN-SC [17], covering both platform components and xApps. Table 2 summarizes the results. In total, InvaRAN uncovered 9 previously unknown issues, consisting of 7 logical errors and 2 crashes. Importantly, although InvaRAN targeted the same components and used the same fuzzing setup as prior work [76], it discovered crashing bugs that earlier approaches failed to find. Manual inspection confirmed that these bugs were already present in the versions tested by prior work but went undetected, likely because InvaRAN leverages invariant violations as an additional feedback signal, enabling guided exploration and deeper coverage of the input space.

The two crash vulnerabilities (#4, #7) result in component-level denial-of-service, requiring the affected module to restart and re-establish connections with the RIC and RAN to restore service, introducing temporary or prolonged network degradation depending on the timing and frequency of the fault. The remaining logical issues result in network degradation, optimization failure, or weakened isolation guarantees. Per-UE errors (#1) can push a UE into an inconsistent state that prevents correct mobility or measurement handling.

Table 2: Summary of discovered vulnerabilities

**Spec A: violates explicit requirement in specification; B: no security properties but violates generic functional requirements; C: no relevant specification*

Impl	#	Vulnerability Description	Type	Component	Spec*	Vulnerability Location	Oracle Type	Trace Level	Status
SD-RAN	1	UE enters inconsistent state without recovery	Inconsistent State	`rimedo-ts`	C	mho/mho.go#L115, L160, L199	Exact Value	Overall	Reported
	2	Lack of validation of cells controlled by E2 node	Input validation	`rimedo-ts`	B	mho/mho.go#L99, L146, L187	Correlation	Message	Confirmed
	3	Missing validation for negative RSRP values	Input validation	`rimedo-ts`	B	mho/mho.go#L134, L166	Range	Message	Patched
	4	Crash	Crash	`onos-lib-go`	C	pkg/asn1/aper/aper.go#L58	Crash	-	Patched [5]
	5	Missing validation for cell-node association	Input validation	`onos-kpimon`	B	monitoring/-monitor.go#L177	Correlation	Message	Reported
	6	No validation of delete stream operation	Access control	`onos-e2t`	A	northbound/e2/v1beta1/-subscription.go#L454	Correlation	Session	Reported
O-RAN-SC	7	Crash	Crash	`kpimon-go`	C	e2sm/wrapper.c#L992	Crash	-	Confirmed
	8	Missing validation check in KPI monitoring logic	Input validation	`kpimon-go`	B	control/control.go#L827	Range	Message	Confirmed
	9	No validation of delete subscription operation	Access Control	`plt-submgr`	A	control/control.go#L795	Correlation	Session	Reported

Correlation- and range-based validation failures (#2, #3, #5, #8) permit the acceptance of falsified or malformed reports. Depending on the affected component, these can propagate incorrect load, KPI, or measurement signals, undermining network-wide optimization logic and potentially contributing to congestion. Finally, session-level state-management flaws (#6, #9) break subscription invariants and allow unauthorized deletion of streams created by other applications, enabling cross-component denial-of-service and network degradation.

Each vulnerability is detected by different invariant types (*Oracle Type* in Table 2), aligned with the invariant types in Table 1. We also examined the relevant technical specifications to determine whether each issue constitutes a direct violation. As shown in the *Spec* column, issues fall into three categories: (A) clear violations of explicit requirements, (B) violations of functional expectations not explicitly formalized (an example is provided in §3.1), and (C) implementation-level flaws without a corresponding specification rule. Although type-B and type-C issues do not violate explicit specification clauses, they nevertheless produce behaviors that violate the semantic requirements of the standard. These problems often arise from implementation-specific logic, including missing checks, sink states, or crashes, that are not explicitly specified in natural-language requirements. Prior work [48, 49, 68], which relies on manual extraction of specification-derived oracles, is generally restricted to identifying only type-A violations. Detecting type-B and type-C issues requires deep implementation knowledge and is not feasible through specification matching alone. In contrast, InvaRAN uncovers these broader categories automatically by learning behavioral invariants from execution, enabling it to detect low-level semantic deviations that remain invisible to specification-guided testing. Below, we detail representative issues uncovered by InvaRAN.

Targeted UE DoS via falsified reports (#1 & 2). The two vulnerabilities highlighted in the motivating example (§3.1) can be chained to launch a targeted denial-of-service (DoS) attack against any UE managed by the xApp, assuming the attacker learns the UE's identifier and the CGI of the serving cell. Prior work has demonstrated that both identifiers are obtainable through exposed O-RAN control-plane and fronthaul channels [47, 74], where unprotected control and measurement messages leak UE identifiers and cell metadata.

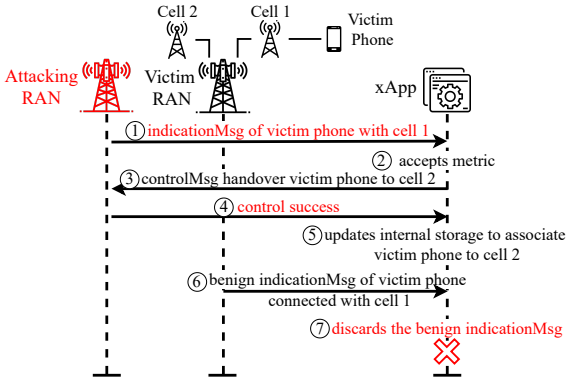

Figure 5: Attack flow for the targeted DoS attack

Figure 5 illustrates the attack flow. Suppose a benign RAN manages two cells, cell 1 and cell 2, and a victim UE is connected to cell 1. A malicious RAN, after learning the UE's ID and the CGI of its serving cell, falsely claims to control both cells and submits a manipulated measurement report. For example, it reports that the victim UE observes a stronger signal from cell 2 than from cell 1 (①). Due to the lack of cell-ownership validation (Issue #1), the xApp accepts this falsified report (②) and, attempting to optimize connectivity, issues a handover request to the malicious RAN, instructing it to move the UE to cell 2 (③). The malicious RAN acknowledges the request (④) despite having no actual control over the UE or either cell. Deceived by the acknowledgment, the xApp updates its internal state to believe that the UE is now served by cell 2 (⑤). In reality, the UE remains connected to cell 1 of the benign RAN. From this point onward, the xApp

silently drops all subsequent benign reports whose content is inconsistent with its internal state (Issue #2), causing legitimate measurement updates about the UE to be ignored (⑥, ⑦). This leads to a persistent DoS, where the UE remains active in the network, but the xApp no longer processes its measurement reports or issues valid control actions. Mitigating this attack requires addressing both vulnerabilities. First, the xApp must validate cell ownership to prevent adversaries from injecting falsified reports. Second, it must also resolve inconsistencies between internal state and actual network conditions, preventing DoS from malicious activity or benign anomalies such as delayed or lost acknowledgments.

Missing validation of stream deletion (#6). In SD-RAN's E2T component, stream deletion requests are not validated against the identity of the stream creator. As a result, any xApp with access to the E2T endpoint can delete streams created by other xApps if it knows the corresponding channelID. This enables a DoS attack in which legitimate xApps are prevented from receiving RAN metric reports. The channelID is also easy to infer, as it follows a predictable format based on the xApp name and E2 node ID. This issue was uncovered through per-session correlation of created channelIDs. The most relevant security guideline, from the O-RAN Security Requirements and Controls [54], states: *"REQ-SEC-API-11: APIs used in O-RAN shall default-deny properties that should not be accessed by clients."* While this vulnerability directly violates an explicit specification requirement, it is non-trivial to detect through specification-guided analysis. Because vendors retain flexibility in defining internal APIs and endpoints, the specification provides no concrete enforcement details. As a result, manual or specification-driven analysis would still need to examine all API implementations to determine whether such requirements are properly enforced.

6.2 Per-Component Ablation Evaluation

6.2.1 Evaluation of invariant classification

We evaluated the accuracy of InvaRAN's invariant classification (§4.1) by manually analyzing results from two representative xApps, `rimedo-ts` and `onos-kpimon`. These xApps were chosen for their semantically rich processing logic and moderate codebase size, which make them feasible for detailed inspection. In contrast, platforming components such as SD-RAN's E2T generate more than 4,000 invariants, rendering exhaustive manual analysis impractical.

For `rimedo-ts`, Daikon inferred 632 invariants, of which InvaRAN classified 85 as critical. Manual inspection showed that 6 of the 85 were false positives, primarily due to limited diversity in the benign traces used for inference. For instance, one such invariant was `ueData.FiveQi one of {1, 2}`. While the FiveQI (5G QoS Identifier) can take various values corresponding to different QoS levels, our benign trace collection only involved UEs performing basic tasks, hence only values 1 and 2 were observed. We also noted that some critical invariants appear unintuitive but are nonetheless meaningful. For example, the invariant `ueData.CGIString < ueData.E2NodeID` compares two strings lexicographically. Though this may seem odd, it holds because the CGI is represented as a numeric string and the E2NodeID as an alphanumeric string. We consider this a true positive, as a violation could indicate formatting inconsistencies between these fields, even if the comparison itself is a byproduct of Daikon's inference mechanism. During dynamic testing, across 5 fuzzing runs, a total of 35 invariant violations were observed, of which only 3 were critical invariants. We manually triaged the 3 critical invariant violations. Among these, 2 were true positives (issues #2 and #3 in Table 2), and 1 was a false positive related to the aforementioned FiveQI invariant. The remaining 32 non-critical violations did not correspond to logical errors. These false positives fall into two main categories: (1) invariants that summarize benign input patterns but are not used in semantically meaningful processing (22 violations), and (2) invariants violated prior to validation checks, where subsequent checks correctly rejected the malformed input before it influenced program behavior (10 violations). For `onos-kpimon`, Daikon inferred 349 invariants, of which 23 were critical. Manual triaging found 3 false positives. During fuzzing, 26 violations were observed, and only 1 critical invariant was violated, which was a false positive. None of the remaining non-critical invariant violations corresponded to true logical issues. Due to the large number of non-critical invariants (more than 500 per component), we did not attempt to exhaustively evaluate potential false negatives (i.e., non-critical ones potentially indicative of logical error).

6.2.2 Evaluation of correlation inference

To evaluate the effectiveness of correlation inference (§4.2), we manually analyzed the inferred results of the same xApps.

Accuracy & critical classification. For `rimedo-ts`, InvaRAN identified 798 consistent variable correlations (both intra- and inter-program-point) out of 127,270 possible pairwise mappings. Of these, 97 were classified as critical. Manual inspection confirmed that all 798 correlations were supported by the traces and revealed 12 false positives among the critical set. Violations of these false positives did not correspond to true logical errors. For example, one inferred correlation linked the `UE ID` with reported `RSRP`. While valid, since different UEs naturally experience distinct signal strengths, violating it (e.g., two UEs reporting identical RSRP values) is not semantically problematic. During fuzzing, 8 correlations were violated, including 2 critical ones: 1 true positive (#1 in Table 2) and 1 false positive (the UE ID–RSRP correlation). Violations of non-critical correlations did not indicate logical vulnerabilities. For `onos-kpimon`, InvaRAN identified 642 correlations out of 21,042 possible mappings, with 67 classified as critical. Manual validation found 3 false posi-

tives. During fuzzing, 8 violations were observed: 3 critical (2 true positives, both corresponding to #5, and 1 false positive), while the non-critical violations did not lead to vulnerabilities.

Comparative analysis. We compared these results with Daikon [29], a widely used template-based invariant inference tool [39, 61]. Out of the 798 correlations detected in `rimedo-ts`, only 181 are expressible by Daikon's predefined templates (e.g., equality or simple arithmetic relations). Furthermore, just 27 of the 798 correlations occur within the same program point, which is the scope supported by Daikon. Of these, 8 are detectable by Daikon's template-based inference, but only 7 were identified using Daikon's default templates. The remaining one (a substring relation) would require adding custom templates. For `onos-kpimon`, of the 642 correlations detected by InvaRAN, only 51 match Daikon's templates, and 39 involve variables within the same program point. Among these, 23 are detectable by Daikon's approach, but only 18 were identified with Daikon's default templates.

6.2.3 Coverage comparison & vulnerability discovery

To assess the dynamic testing capabilities of InvaRAN, we compare it against ORANalyst [76], which applies grammar-aware fuzzing to find crashing bugs in O-RAN components. InvaRAN augments this baseline with invariant-violation feedback and a critical-invariant oracle for detecting logical errors. We also evaluate against AFL++ [40], a state-of-the-art general-purpose fuzzer. In addition to the baseline AFL++, we include a variant augmented with invariant feedback and the critical-invariant oracle, allowing us to isolate the impact of invariant-guided exploration on coverage and bug discovery. Experiments were conducted on the same xApps, with each tool run five times for 24 hours each run. Results (Table 3) show that invariant feedback and the critical-invariant oracle substantially improve bug-finding effectiveness. Additionally, invariant feedback yields only slight coverage improvements, consistent with prior observations on using invariants as additional fuzzing feedback [39]. Finally, we verified that InvaRAN also detects all four issues previously reported by ORANalyst [76] on the unpatched versions of these two components, in addition to the newly discovered issues.

Comppnent	Tool	bbCov	edgeCov	inv	critical inv	bugs
`rimedo-ts`	InvaRAN	10388	5344	30	4	3
	ORANalyst	10194	5128	24	0	0
	AFL	10093	5014	15	0	0
	AFL w/ inv	10204	5159	22	4	3
`onos-kpimon`	InvaRAN	5794	2730	24	1	1
	ORANalyst	5764	2721	20	1	0
	AFL	5472	2697	15	0	0
	AFL w/ inv	5613	2718	20	1	1

Table 3: Average coverage/invariant violation comparison
bbCov: basic block coverage; **edgeCov**: edge coverage; **inv**: # violated invariants (critical + non-critical); **critical inv**: # violated critical invariants

Table 4: InvaRAN capability comparison with existing tools
* Can be used as oracles, but lead to significant false positives (6.2.1)

Capability	AFL++ [1]	ORANalyst [76]	Daikon [29]	InvaRAN
logical bug	✗	✗	✗	✓
crashing bug	✓	✓	✗	✓
semantic oracles	✗	✗	✓*	✓
scalar invariants	✗	✗	✓	✓
global invariants	✗	✗	✗	✓
domain awareness	✗	✓	✗	✓

6.3 Generality and Comparison of InvaRAN

To assess the generality of InvaRAN beyond O-RAN components, we evaluated it against 6 known logical bugs in widely used Go programs. We deliberately focused on silent bugs that do not cause crashes or hangs, but instead lead to semantically incorrect behavior while the program continues running. The selected vulnerabilities span different categories, including resource exhaustion, access control violations, and privilege escalation. To align with InvaRAN's setup, we chose input-dependent vulnerabilities, where adversarial inputs trigger the flaw while benign inputs do not (i.e., omitting configuration-dependent vulnerabilities). InvaRAN detected 5 of the 6 vulnerabilities out of the box, covering all bug categories, and was able to detect the remaining one with additional instrumentation monitoring the number of active handlers (which was not tracked by any existing program variable), to detect its abnormal growth during attacks. Moreover, during testing, InvaRAN uncovered a previously unknown DoS issue, showing its effectiveness beyond O-RAN. Details are provided in Appendix C and vulnerabilities are summarized in Table 5.

Feature comparison with existing tools. We provide a capability comparison of InvaRAN against representative baseline tools (Table 4), including AFL++, ORANalyst, and Daikon. These baselines reflect primary approaches used in prior work relevant to components in InvaRAN, and the comparison highlights their respective strengths and limitations.

7 Discussions

Manual effort and triage workflow. Identifying logical issues in complex systems traditionally involves manual effort in: (a) understanding functional requirements [48, 49, 68], (b) constructing test oracles [48, 49, 68], (c) generating test cases [27, 44, 48, 49, 68, 70], and (d) triaging issues [27, 44, 48, 49, 68, 70]. InvaRAN substantially reduces this burden by automating (a), (b), and (c). It leverages invariants inferred from benign traces as oracles and systematically mutates inputs to explore violations, enabling broad and deep exploration with minimal manual intervention. Manual effort remains necessary for (d), to triage invariant violations and determine whether they indicate true vulnerabilities, as well as to analyze the root causes and impacts. During triage, InvaRAN

produces three artifacts that guide manual inspection: (i) the violated invariant, (ii) the concrete runtime values triggering the violation, and (iii) the minimal reproducing input sequence. Together, these artifacts enable precise localization of the affected code region and allow developers to replay the execution path to expose the deviation deterministically. In practice, this substantially reduces analysis effort. While automatically inferred invariants may yield more false positives than manually crafted oracles, InvaRAN mitigates this through program analysis that prioritizes semantically meaningful invariants (§6.2.1). Combined with a streamlined triage workflow, the remaining manual effort is significantly reduced compared to existing approaches.

Additional manual effort involves defining rules for identifying network and I/O operations (§4.1.2). InvaRAN includes base rules for tracking low-level standard libraries (e.g., `io`, `net`), and optionally allows users to specify higher-level, implementation-specific wrappers (e.g., `grpc` handlers in SD-RAN). Capturing these wrapper functions improves static analysis precision and avoids unnecessarily deep traversals into standard libraries. However, custom I/O implementations that bypass these base libraries may be missed unless explicitly specified, reducing classification completeness. Extending InvaRAN with syscall-level tracing to generalize I/O identification remains future work.

Quality of invariant-based logical error detection. InvaRAN's invariant-based analysis reduces the need to manually interpret specifications or reason about expected behaviors, making it more scalable for large, complex systems like O-RAN. It can also detect subtle logical errors that traditional specification-guided oracles may miss, such as violations of implicit assumptions (Table 2). However, the effectiveness of this approach depends on the quality and expressiveness of the inferred invariants. Complex semantic violations or multi-step logical errors beyond the scope of traditional scalar invariants may go undetected. InvaRAN mitigates this through correlation analysis, which helps uncover structured relationships among variables, but it may still miss higher-order or non-obvious patterns. Exploring richer representations, such as machine learning or neural invariant inference, is a direction for future work. Despite the limitation, InvaRAN provides actionable insights by pinpointing exactly which invariants are violated, enabling developers to focus their debugging efforts without manually tracing through code or control flows.

Language support. Implementation of InvaRAN (§5) targets Go because most available RIC platforming components (all SD-RAN modules and most O-RAN-SC) are written in Go. However, the underlying methodology is language agnostic. Invariant inference (both Daikon-based and correlation-based) already supports all languages compatible with Daikon's instrumentation front ends (e.g., C/C++, Java, Python). The invariant classification analysis (§4.1) relies on AST- and CFG-based traversal, and can be extended to other languages utilizing language-specific static analysis.

Extension to binary targets. InvaRAN assumes source code access to support trace instrumentation for invariant inference and static analyses for invariant classification (§4.1). Both steps can be adapted to gray-box targets. Trace collection can be performed using binary-level instrumentation (e.g., Kvasir [7] for C/C++), though this may require binaries compiled with debugging symbols, and the resulting traces may lack stable variable names or consistent source mappings due to compiler optimizations, complicating triaging. Invariant classification relies on control- and dataflow analysis and can be applied through binary analysis [2], but such analyses may miss validation logic, over-approximate unreachable code, or introduce other imprecision, leading to higher false-positive or false-negative rates compared to source-level analysis.

8 Related Work

Logical error detection. Other than approaches discussed in §3.1, prior work has mined attribute-based access control (ABAC) rules from audit logs to detect access control issues [31, 34, 45, 46, 75]. These approaches require manual specification of entities (users, resources, attributes), which is impractical in O-RAN's heterogeneous setting with multiple users (UEs, CUs, DUs, xApps) and resources (xApp services, runtime state, user data). This effort must also be repeated for each component, limiting scalability. In contrast, InvaRAN infers semantically related entities and behaviors through correlation analysis of trace variables, enabling detection of implicit access control violations even without explicit policies (Table 2). Additionally, IPPO [50] detects missing validation by comparing execution paths, but it cannot detect when a validation is absent from all paths. InvaRAN instead learns invariants from benign traces and flags violations during testing, allowing it to uncover malformed or unauthorized inputs even when checks are missing in all execution paths.

Fuzzing. Fuzzing has proven effective in detecting memory corruption issues [3, 11, 14, 40, 63]. Recently, it has also shown success in uncovering logical bugs [48, 49, 68] across various domains, including Android applications [68], database management systems [49], and extended Berkeley Packet Filter [48]. However, these approaches target a narrow class of systems and rely on manually crafted, domain-specific oracles that are not easily transferable to O-RAN. In contrast, InvaRAN leverages invariants as proxies for logical expectations, treating them as oracles in a general and automated manner. Prior efforts have also explored using inferred invariants as feedback to guide fuzzing [39, 43], but these approaches are limited to detecting traditional memory corruption bugs.

O-RAN security. Prior O-RAN testing efforts mostly focus on detecting memory or crashing bugs via fuzzing [67, 76] or over-the-air testing [30, 71]. Other efforts conduct conformance testing through LLM-devised [41] or manually created [42, 53] tests, but they require extensive harnessing and manual intervention, and can only find issues the test cases

are designed to uncover. In parallel, several studies discuss theoretical security challenges and research directions for O-RAN [36, 55–58, 65, 66]. InvaRAN is the first to systematically test O-RAN systems for logical errors, moving beyond memory corruption and compliance checks.

9 Conclusion

We introduced InvaRAN, the first systematic framework for detecting logical errors in O-RAN using dynamically inferred invariants and correlation analysis. Applied to two production-grade platforms, it uncovered 9 previously unknown issues that traditional testing missed. InvaRAN offers a foundation for more resilient O-RAN deployments and advances automated security analysis of next-generation mobile networks.

Ethical Considerations

We present a new framework for uncovering logical vulnerabilities in O-RAN components and other general programs. While our work highlights flaws that could be exploited for denial-of-service, access control bypass, and other logic-driven attacks, the purpose of this research is to advance the security and robustness of mobile networks and related software ecosystems. Identifying such vulnerabilities is critical to mitigating risks in next-generation RAN deployments, where logical flaws can lead to degraded performance, service disruptions, or weakened security guarantees. By automating the discovery of these issues, InvaRAN provides developers and operators with actionable insights to harden their systems.

Stakeholders. This work mainly affects the following parties. For developers, InvaRAN provides support for identifying semantic flaws and improving code quality. Network vendors may need to update components or harden validation logic to address reported issues. Mobile operators benefit from more thorough testing of RIC components prior to deployment. Mobile users could gain from increased network reliability, though unpatched systems may expose risks if the vulnerabilities discussed were exploited. Standardization bodies may also benefit, as several findings highlight underspecified behaviors where clearer guidance or testing requirements could reduce implementation oversights. Finally, the research community could extend our research to improve testing methodologies towards automated logical vulnerability detection.

Balancing benefits and risks. Publishing this work entails both benefits and potential risks. The proposed methodologies enable more automated detection of logical issues, benefiting both the research community and developers by streamlining the identification of subtle logical flaws and strengthening software systems in the long term. At the same time, disclosure of identified vulnerabilities could be misused if affected systems remain unpatched. However, withholding such findings would leave developers, operators, and users unaware of latent flaws that can silently undermine network correctness and availability. We believe the benefits of improving network resilience outweigh these risks, particularly given our coordinated disclosure process, the absence of exploit code, and our focus on design-level insights rather than exploitation.

Responsible disclosure. We adhered to standard coordinated disclosure practices. All discovered issues were reported to the maintainers of O-RAN-SC, SD-RAN, and other evaluated components, along with technical reports to facilitate remediation. The disclosure status of each issue is summarized in Table 2. So far, five issues have been confirmed by developers. Two have been patched, with a CVE assigned [5], and the remaining confirmed issues are undergoing remediation and awaiting CVE assignment.

Mitigating potential harm. All experiments were performed on isolated, self-hosted testbeds. No experiments were conducted on production systems or operational carrier networks. Testing of general-purpose programs was similarly confined to controlled environments.

Open Science

To facilitate future research, we will make all artifacts for InvaRAN publicly available through https://doi.org/10.5281/zenodo.17969521. They cover the full pipeline of invariant-guided logical testing for O-RAN, including trace collection, invariant inference, classification (§4.1), correlation analysis (§4.2), and runtime assertion checking. Examples are also provided to demonstrate each component.

Acknowledgment

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, Public Wireless Supply Chain Innovation Fund (PWSCIF) under Federal Award ID Number 51-60-IF007, 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] American fuzzy lop. https://github.com/google/AFL.

[2] angr: open-source binary analysis platform. https://angr.io/.

[3] boofuzz. https://boofuzz.readthedocs.io/en/stable/.

[4] *Clang MemorySanitizer*. https://clang.llvm.org/docs/MemorySanitizer.html.

[5] CVE-2025-30077. https://nvd.nist.gov/vuln/detail/CVE-2025-30077.

[6] Daikon's Chicory. https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Chicory.

[7] Daikon's Kvasir. https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Kvasir.

[8] DeepSig and SRS Chosen by DOD Future G Office to Lead OCUDU: The Open-Source 5G/6G RAN Initiative. https://www.deepsig.ai/deepsig-and-srs-chosen-by-dod-futureg-office-to-lead-ocudu-the-open-source-5g-6g-ran-initiative/.

[9] Docker. https://www.docker.com/.

[10] Gitlab's protocol fuzzing framework. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[11] go-fuzz. https://github.com/dvyukov/go-fuzz.

[12] Intelligent Private 5G Solution Based on Near-RT RIC. https://stage-o-ran-v2.azurewebsites.net/classic/generation/2023/category/intelligent-ran-control-demonstrations/sub/intelligent-control/251.

[13] Kubernetes. https://kubernetes.io/.

[14] libFuzzer. llvm.org/docs/LibFuzzer.html.

[15] Linux Foundation. https://www.linuxfoundation.org/.

[16] *O-RAN ALLIANCE*. www.o-ran.org.

[17] O-RAN Software Community. https://o-ran-sc.org/.

[18] ONF and Deutsche Telekom demonstrate fully disaggregated Open RAN. https://opennetworking.org/news-and-events/press-releases/onf-and-deutsche-telekom-demonstrate-fully-disaggregated-open-ran-with-open-ric-platform/.

[19] ONOS Project: Rimedo Lab Traffic Steering xApp. https://github.com/onosproject/rimedo-ts/tree/master.

[20] Open Networking Foundation. https://opennetworking.org/.

[21] Open RAN: Attack of the xApps. https://www.trendmicro.com/vinfo/us/security/news/vulnerabilities-and-exploits/open-ran-attack-of-the-xapps.

[22] SD-RAN. https://opennetworking.org/open-ran/.

[23] A. Aijaz, S. Gufran, T. Farnham, S. Chintalapati, A. Sánchez-Mompó, and P. Li. Open RAN for 5G Supply Chain Diversification: The BEACON-5G Approach and Key Achievements. In *2023 IEEE Conference on Standards for Communications and Networking (CSCN)*.

[24] 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*, CCS '23. Association for Computing Machinery.

[25] C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*. Proceeding of the Second International Workshop on Numerical and Symbolic Abstract Domains: NSAD 2010.

[26] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

[27] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1690–1701. Association for Computing Machinery.

[28] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*.

[29] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering*, 41(4):408–428, Apr. 2015.

[30] L. Bonati, M. Polese, S. D'Oro, P. B. del Prever, and T. Melodia. 5G-CT: Automated Deployment and Over-the-Air Testing of End-to-End Open Radio Access Networks. *IEEE Communications Magazine*.

[31] T. Bui, S. D. Stoller, and J. Li. Mining relationship-based access control policies. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, SACMAT '17 Abstracts, page 239–246. Association for Computing Machinery.

[32] S. Chakraborty, S. K. Lahiri, S. Fakhoury, M. Musuvathi, A. Lal, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy. Ranking llm-generated loop invariants for program verification, 2024.

[33] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[34] C. Cotrini, T. Weghorn, and D. Basin. Mining abac rules from sparse logs. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 31–46, 2018.

[35] P. Cousot. *Abstract Interpretation Based Formal Methods and Future Challenges*, pages 138–156. Springer Berlin Heidelberg, 2001.

[36] D. Dik and M. S. Berger. Open-ran fronthaul transport security architecture and implementation. *IEEE Access*, 2023.

[37] Y. Dong, T. Yang, A. Al Ishtiaq, S. M. M. Rashid, A. Ranjbar, K. Tu, T. Wu, M. S. Mahmud, and S. R. Hussain. CORECRISIS: threat-guided and context-aware iterative learning and fuzzing of 5G core networks. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

[38] R. Fagin. Functional dependencies in a relational database and propositional logic. *IBM Journal of Research and Development*.

[39] A. Fioraldi, D. C. D'Elia, and D. Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, Aug. 2021.

[40] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies*, Aug. 2020.

[41] A. Ganiyu, P. Gajjar, and V. K. Shah. AI5GTest: AI-Driven Specification-Aware Automated Testing and Validation of 5G O-RAN Components. In *18th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec 2025, 2025.

[42] G. Gemmi, M. Polese, P. Johari, S. Maxenti, M. Seltser, and T. Melodia. Open6G OTIC: A Blueprint for Programmable O-RAN and 3GPP Testing Infrastructure. In *2024 IEEE 100th Vehicular Technology Conference (VTC2024-Fall)*, pages 1–5, 2024.

[43] H. Huang, A. Zhou, M. Payer, and C. Zhang. Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference. In *2024 IEEE Symposium on Security and Privacy*.

[44] 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*.

[45] J. Hwang, T. Xie, V. Hu, and M. Altunay. Mining likely properties of access control policies via association rule mining.

[46] P. Iyer and A. Masoumzadeh. Mining positive and negative attribute-based access control policy rules. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*.

[47] 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)*.

[48] Y. Li, W. Niu, Y. Zhu, J. Gong, B. Li, and X. Zhang. Fuzzing logical bugs in ebpf verifier with bound-violation indicator. In *ICC 2023 - IEEE International Conference on Communications*.

[49] Y. Liang, S. Liu, and H. Hu. Detecting logical bugs of DBMS with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Aug. 2022.

[50] D. Liu, Q. Wu, S. Ji, K. Lu, Z. Liu, J. Chen, and Q. He. Detecting missed security operations through differential checking of object-based similar paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21.

[51] A. Mehrban, Z. A. El Houda, H. Moudoud, B. Brik, and L. Khoukhi. Securing O-RAN Equipment Using Blockchain-Based Supply Chain Verification. In *2025 International Wireless Communications and Mobile Computing (IWCMC)*, pages 1570–1575, 2025.

[52] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4, 2005.

[53] T. V. Ngo, M. V. Ngo, B. Chen, G. Gemmi, E. Baena, M. Polese, T. Melodia, W. Chien, and T. Quek. Consistent and repeatable testing of o-ran distributed unit (o-du) across continents. In *2024 IEEE 100th Vehicular Technology Conference (VTC2024-Fall)*, pages 1–5, 2024.

[54] O-RAN Working Group 11. *O-RAN Security Requirements and Controls Specifications 11.0*. www.o-ran.org/specifications.

[55] O-RAN Working Group 11. *O-RAN Security Test Specifications 5.0*. www.o-ran.org/specifications.

[56] O-RAN Working Group 11. *O-RAN Security Threat Modeling and Risk Assessment v5.0*. www.o-ran.org/specifications.

[57] O-RAN Working Group 11. *O-RAN Study on Security for Near Real Time RIC and xApps v5.0*. www.o-ran.org/specifications.

[58] O-RAN Working Group 11. *O-RAN Study on Security for O-Cloud 4.0*. www.o-ran.org/specifications.

[59] O-RAN Working Group 3. *O-RAN Near-RT RIC Architecture 5.0 Technical Specification*. www.o-ran.org/specifications.

[60] O-RAN Working Group 3. *O-RAN Use Cases and Requirements 8.0*. www.o-ran.org/specifications.

[61] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*. PMLR, 2023.

[62] A. Peltonen, R. Sasse, and D. Basin. A comprehensive formal analysis of 5G handover. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*.

[63] V.-T. Pham, M. Böhme, and A. Roychoudhury. AFLNET: A Greybox Fuzzer for Network Protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, 2020.

[64] M. A. A. Pirzada, G. Reger, A. Bhayat, and L. C. Cordeiro. LLM-Generated Invariants for Bounded Model Checking Without Loop Unrolling. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24.

[65] M. Polese, L. Bonati, S. D'oro, S. Basagni, and T. Melodia. Understanding o-ran: Architecture, interfaces, algorithms, security, and research challenges. *IEEE Communications Surveys & Tutorials*, 2023.

[66] C. Shen, Y. Xiao, Y. Ma, J. Chen, C.-M. Chiang, S. Chen, and Y. Pan. Security threat analysis and treatment strategy for ORAN. In *2022 24th International Conference on Advanced Communication Technology*.

[67] I. Siroš, D. Singelée, and B. Preneel. CovFUZZ: Coverage-based fuzzer for 4G&5G protocols, 2024.

[68] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su. Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.*, (OOPSLA), 2021.

[69] F. Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[70] 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)*. USENIX Association, Aug.

[71] C. M. Ueltschey, J. Moore, A. S. Abdalla, and V. Marojevic. RAN Tester UE: An Automated Declarative UE Centric Security Testing Platform [Dataset/Tool Paper]. In *Proceedings of the 30th ACM Symposium on Access Control Models and Technologies*.

[72] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, 1981.

[73] Y. Xia, A. Pingle, D. Sur, J. Deshmukh, M. Raghothaman, and S. Ravi. LLM-guided Predicate Discovery and Data Augmentation for Learning Likely Program Invariants. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*, SAC '25.

[74] J. Xing, S. Yoo, X. Foukas, D. Kim, and M. K. Reiter. On the Criticality of Integrity Protection in 5G Fronthaul Networks, booktitle = 33rd USENIX Security Symposium (USENIX Security 24).

[75] Z. Xu and S. D. Stoller. Mining attribute-based access control policies from logs. In V. Atluri and G. Pernul, editors, *Data and Applications Security and Privacy XXVIII*. Springer Berlin Heidelberg.

[76] 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)*.

[77] F. Zhu and J. Wei. Static analysis based invariant detection for commodity operating systems. *Computers & Security*.

A Dataflow Analysis Implementations

Dataflow analysis for validation checks identification. InvaRAN performs both backward and forward dataflow analyses to identify validation checks, as described in Algorithm 1. These analyses determine whether an invariant is preceded by validation logic that governs whether the program proceeds to the point where the invariant is observed. All analyses are performed over the program's SSA (Static Single Assignment) form to ensure precise data tracking. The process begins with the variable associated with the invariant, denoted as v_{inv}. The backward dataflow analysis (`BackwardDataflow`, lines 14–34) traces all variables that directly or indirectly influence v_{inv} via field access operations. The objective is to identify the set of input-derived variables that can affect the computation or control flow leading to the invariant. InvaRAN limits its analysis to variables originating from input message fields, as these are the only values an attacker can control. Variables derived solely from static or constant values are excluded, as they do not contribute to potential vulnerabilities. Once all relevant variables assigned via field accesses are identified, the algorithm filters out any that are not directly derived from the input message (line 3). The resulting set, denoted V, represents the fields that are both attacker-controlled and relevant to the invariant. These are then subjected to further analysis to detect associated validation checks.

Next, the forward dataflow analysis (`ForwardDataflow`, lines 36–53) traces how each variable in V propagates through the program to influence downstream conditional branches. This enables InvaRAN to capture both direct and indirect dependencies between input fields and control-flow decisions. Finally, InvaRAN identifies potential validation checks by analyzing the program's control dependence graph (lines 6–12). Branch conditions that influence the reachability of v_{inv}

Algorithm 1 Dataflow Analysis for Validation Checks

Require:
v_{inv} : The variable in the invariant relation, assumed in SSA form
Output: List of Boolean Expressions Representing the Validation Checks Performed Before the Identified Invariant Relation

procedure INVDATAFLOW(v_{inv})
  assigned_variables ← BACKWARDDATAFLOW(v_{inv})
  $V \leftarrow \{x \in \textit{assigned\_variables} \mid x \text{ is from input message field access}\}$
  ▷ Filter to keep only variables assigned from field access of the input message
  branch_conditions ← FORWARDDATAFLOW(V)
  relevant_branches ← {}
  for each branch condition in *branch_conditions* **do**
    if reachability of instruction containing v_{inv} is control-dependent on the branch (using PDG) **then**
      relevant_branches ← *relevant_branches* ∪ {branch}
    end if
  end for
  return *relevant_branches*
end procedure

procedure BACKWARDDATAFLOW(v)
  ▷ Given a variable v, perform backward dataflow analysis to find all variables v assigns from and is assigned from field access statements
  Initialize an empty set *FieldAccessVars*
  Initialize a worklist *Worklist* with v
  while *Worklist* is not empty **do**
    $v \leftarrow$ *Worklist*.pop()
    for each usage u of v in the form $v =$ expression **do**
      if expression is of the form $x.field$ **then** ▷ A field access statement
        FieldAccessVars.add(x)
      end if
      if expression involves other variables w **then**
        for each w **do**
          if w has not been processed **then**
            Worklist.add(w)
          end if
        end for
      end if
    end for
  end while
  return *FieldAccessVars*
end procedure

procedure FORWARDDATAFLOW(V)
  ▷ Perform forward dataflow analysis to find all branch conditions influenced by variables in V
  Initialize an empty set $AffectedBranches$
  Initialize a worklist *Worklist* with variables from V
  while *Worklist* is not empty **do**
    $x \leftarrow$ *Worklist*.pop()
    for each statement s where x influences or is part of an expression **do**
      Extract resulting variable y if s is of the form $y = f(x,\ldots)$
      if s is a branch condition **then**
        $AffectedBranches$.add(s)
      end if
      if y is defined and y not visited **then**
        Worklist.add(y)
      end if
    end for
  end while
  return $AffectedBranches$
end procedure

Algorithm 2 Pointer Analysis for Validation Checks

Require:
v_{inv} : The variable in the invariant relation, assumed in SSA form
Output: List of conditions from Pointer Analysis that impact the reachability to v_{inv}

1: **procedure** POINTERANALYSIS(v_{inv})
2: $V \leftarrow$ BACKWARDDATAFLOW(v_{inv}) ▷ Find all variables assigned from field access of the input message
3: Initialize an empty set P
4: **for** each variable v in V **do**
5: **if** v can point **then**
6: Collect all pointers P_{all} along the call graph to v
7: Perform points-to analysis on P_{all} to find variables possibly pointing to the same location as v, call them P
8: **for** each p in P **do**
9: Perform FORWARDDATAFLOW(p)
10: **for** each condition c affecting reachability from p to v_{inv} **do**
11: Collect the condition
12: **end for**
13: **end for**
14: **end if**
15: **end for**
16: **return** Collected conditions as *ValidationChecks*
17: **end procedure**

are collected, while those that merely log or monitor values without affecting control flow are discarded. Only conditions that, if not satisfied, prevent the invariant's location from being reached are considered meaningful validations.

Pointer analysis for validation checks identification. In O-RAN implementations, direct assignments may not capture all control- or data-flow dependencies due to the prevalence of pointer-based operations and interprocedural message handling. For example, a component may validate an input field, then later dereference a separate pointer to re-access that field for further processing. Traditional dataflow analysis may fail to track these indirect relationships, leading to incomplete validation detection. To address this, INVARAN incorporates pointer analysis, as demonstrated in Algorithm 2, to complement its dataflow-based inference. The algorithm begins by reusing the `BackwardDataflow` procedure to collect all variables assigned via field accesses from the input message (line 2). It then identifies which of these variables are pointers (line 5), and collects all related pointers along the program's call graph (line 6), narrowing the analysis scope to relevant pointer interactions. A points-to analysis [26] is then performed on these variables (line 7) to determine all memory locations and aliases potentially referenced by the original input-derived pointers. For each resulting alias, the `ForwardDataflow` procedure is invoked (line 9) to analyze how control flow may be influenced by these memory references. This step reveals additional validation checks that rely on the state of aliased variables and determine access to the invariant location.

B Context-Aware Trace Attribution

To address Challenges C3 (§3.2), INVARAN introduces a context-aware trace attribution mechanism that precisely segments execution into logically coherent message-processing units. This enables accurate invariant inference even in highly concurrent environments like O-RAN, where interleaved executions from different inputs can otherwise contaminate trace semantics. Formally, let $T = \{\tau_1, \tau_2, \ldots, \tau_m\}$ denote the set of segmented traces, where each $\tau_k \in T$ corresponds to the processing of a single input message or session. Each trace τ_k captures a partial execution of the CUT, including a sequence of program point invocations and observed variable values $\texttt{val}_{\tau_k}(v)$ for trace variables $v \in V$ (as defined in §4.2).

INVARAN performs this segmentation by instrumenting thread creation and message dispatch logic in the CUT. When

Table 5: Demonstrating extensibility of INVARAN on real-world CVEs.* *Vulnerability detectable only with additional instrumentation.*

CVE	Affected Program	Detectable	Invariant Type	Bug Class
CVE-2025-46816	`patrickhener/goshs`	Yes	Range Constraints	Remote Code Execution
Discovered by INVARAN	`patrickhener/goshs`	Yes	Crash	Denial-of-Service
CVE-2025-3445	`mholt/archiver`	Yes	Correlation	Privilege Escalation
CVE-2022-21698	`prometheus/client_golang`	Yes	Range Constraints	Unbounded Resource Consumption
CVE-2020-26160	`dgrijalva/jwt-go`	Yes	Variable Comparison	Access Control Bypass
CVE-2025-46721	`justinas/nosurf`	Yes	Variable Comparison	Validation Bypass & CSRF
CVE-2023-39325	`net/http2`	No*	Variable Comparison	Excessive Resource Consumption

a new input message is received, INVARAN identifies the thread handling the message and recursively tracks all threads it spawns until the next message is accepted. This set of threads defines a *message-processing context*, and their collective execution is grouped into a single trace $\tau_k \in T$. This enables INVARAN to build per-message thread trees, attributing all trace events from related threads to the same message instance. To facilitate this, INVARAN propagates a lightweight identifier through the execution context, automatically instrumented at thread entry or message dispatch. This identifier is retrieved from the runtime thread-local metadata during trace collection and is used to tag each log entry and variable observation. Because logs from concurrent threads may be interleaved, INVARAN performs post-processing to reorder and reconstruct coherent traces τ_k, correcting for concurrency-induced mixing.

This context attribution supports trace segmentation and invariant inference at multiple semantic levels. (1) *Component-level*: All traces $\tau_k \in T$ are treated as a single unit. Invariants at this level reflect global properties and long-term consistency of the entire component. (2) *Message-type-level*: Traces are partitioned by input message type. Let $T^{(m)} \subset T$ be the set of traces handling message type m; invariants inferred over $T^{(m)}$ capture message-specific behavior, such as constraints unique to setup requests or indication messages. (3) *Session-level*: Traces are grouped by higher-level context identifiers (e.g., E2 node ID). Traces $\tau_i, \tau_j \in T$ belong to the same session if they operate on the same logical entity. This allows inference of cross-message invariants, such as identifier reuse or consistent resource mappings over time. These abstraction levels allow INVARAN to reason about behavior not just within individual messages but across sequences of operations and component lifecycles, enabling detection of logic bugs that span multiple inputs or sessions.

Instrumentation of Invariant Violations. To support runtime detection, INVARAN instruments all inferred invariants across all abstraction levels back into the CUT. These checks are used to provide feedback to the fuzzer. Whenever an invariant is violated by a generated input, the input is added to the fuzzing corpus to guide further exploration of the surrounding state space. However, some invariants are only valid under specific preconditions (e.g., for a particular message type or configuration). In these cases, a violation may not indicate a true logical error unless the associated preconditions are satisfied. Nonetheless, INVARAN still uses all violations as feedback signals during fuzzing. Since each violation contributes at most one input to the corpus, this design avoids uncontrolled corpus growth or redundant input generation.

To ensure precision in final reporting, INVARAN performs post-testing validation: it re-evaluates the preconditions of each violated critical invariant to confirm whether they held at runtime. If the precondition is satisfied, the violation is treated as a genuine logical error and flagged for manual triage. If the precondition is not satisfied, the violation is ignored in final analysis, though the input may still aid fuzzing by exposing new execution paths. This design maximizes the utility of inferred invariants for input generation, while maintains high precision in identifying true semantic violations.

C Extensibility of INVARAN

To further demonstrate the generality of INVARAN, we evaluated it against a diverse set of known real-world logical vulnerabilities in Go programs beyond O-RAN components. The selected programs cover a wide range of domains, including lightweight servers, utility libraries, and security middleware, each with distinct coding styles and attack surfaces. We focus on input-dependent logical flaws spanning categories such as resource exhaustion, access control violations, privilege escalation, and remote code execution. As summarized in Table 5, INVARAN successfully detected 5 out of 6 known CVEs out of the box, and was able to expose the sixth with additional instrumentation. Notably, it also uncovered a previously unknown denial-of-service vulnerability, highlighting its ability to reveal new flaws in widely used systems.