ORANalyst: Systematic Testing Framework for Open RAN Implementations

Tianchang Yang, Syed Md Mukit Rashid, Ali Ranjbar, Gang Tan, Syed Rafiul Hussain
The Pennsylvania State University
{tzy5088, szr5848, aranjbar, gtan, hussain1}@psu.edu

Abstract

We develop ORANalyst, the first systematic testing framework tailored for analyzing the robustness and operational integrity of Open RAN (O-RAN) implementations. O-RAN systems are composed of numerous microservice-based components. ORANalyst initially gains insights into these complex component dependencies by combining efficient static analysis with dynamic tracing. Applying these insights, ORANalyst crafts test inputs that effectively navigate these dependencies and thoroughly test each target component. We evaluate ORANalyst on two O-RAN implementations, O-RAN-SC and SD-RAN, and identify 19 previously undiscovered vulnerabilities. If exploited, these vulnerabilities could lead to various denial-of-service attacks, resulting from component crashes and disruptions in communication channels.

1 Introduction

The standard body for mobile telecommunications [1] has designed a new disaggregated architecture for 5G radio access network (RAN) to enable software-centric deployment on general-purpose hardware. This new architecture splits the RAN's stack into multiple nodes, each capable of operating independently on geographically separated machines to form the RAN network. As 5G RAN evolves towards more diverse and adaptable deployments, the O-RAN Alliance [21], a worldwide community of mobile operators, vendors, and research institutions, proposes *Open RAN (O-RAN)*. It introduces *RAN intelligence controllers* to reshape RANs into smarter, open, virtualized, and interoperable multi-supplier solutions. These advanced controllers interact with different RAN nodes through standardized interfaces, enabling them to manage and optimize RANs in near real-time based on network metrics. O-RAN's software-centric intelligence controller also supports third-party applications to enhance the network's adaptive capabilities, thereby improving RAN performance and user experience. This new and complex RAN design, with its focus on flexibility, openness, and a software-centric architecture, unfortunately also greatly expands the attack surface of O-RAN. For instance, with nodes like radio units of base stations sourced from multiple vendors interfacing collectively with O-RAN, each interaction and dependency introduces a potential security risk. This heterogeneity and complex RAN architecture demand O-RAN's resilience against unexpected inputs, which could arise from malformed or potentially malicious data sent from its connected nodes.

Prior research has demonstrated that mobile RANs are susceptible to misconfigurations [45], implementation and dependency vulnerabilities [42], and can also be compromised by malicious user devices [5]. Consequently, adversaries exploiting a vulnerable RAN node may deliver unpredictable inputs to O-RAN components through its public-facing interfaces, leading to crashes that violate O-RAN's robustness and operational integrity. Such security lapses may lead to large-scale network collapses, substantial financial losses, and could even threaten national security. The active adoption of O-RAN technologies further amplifies these risks. More than 31 major operators from over 45 countries, including Deutsche Telekom and AT&T, are actively deploying or trailing commercial O-RAN [22, 53], with reports of these systems incorporating or are derived from open-source implementations [15, 24]. The urgency of these concerns underscores an in-depth security analysis of O-RAN implementations to certify the security and reliability of this next-generation network system. In this paper, we, therefore, aim to address the following research question: *can we develop an automated reasoning framework to analyze the robustness and operational integrity of O-RAN implementations, providing high-security assurances prior to their commercial deployments?*

Challenges. O-RAN's unique threat model and its service-based architecture (SBA) pose distinct challenges that existing software testing techniques [2, 4] struglle to address. Components in O-RAN are divided into separate microservice programs, such as the database, the subscription manager, and other applications. These components are deployed across different logical containers and communicate via network traffic. Functioning as independent, long-running programs, these microservices are continuously ready to process incoming re-

quests. They engage in complex network communications to fulfill O-RAN's functionalities. Each input originating from base stations or cellular devices traverses a complex path in the O-RAN, navigating through multiple components, undergoing numerous layers of sanitization checks and parsing processes, and engaging in a sequence of network request interactions before reaching components laying in deeper layers of O-RAN. The typical software testing approach of analyzing only a single component in isolation, without considering its dependencies on other components [18,50], would lead to a high number of false positive results. These false-positive vulnerabilities are not exploitable in a complete, i.e., end-to-end deployment of O-RAN, due to the sanity checks and preprocessings conducted by upstream components. Furthermore, unlike the well-defined protocols and messaging formats of O-RAN's public interfaces for communication with the RAN, the internal messaging systems between O-RAN's internal components lack standardization in technical specifications [68] and vary significantly across different implementations. Consequently, isolated testing further requires manually crafting messages that conform to the protocol and format each component expects to yield meaningful results.

Approach. To address these unique challenges, we introduce `ORANalyst`, a dynamic feedback-guided end-to-end testing methodology that intelligently identifies and navigates dependencies between O-RAN components to find threat-model-compliant problematic inputs. In our end-to-end testing method, testing inputs are sent to a component only through O-RAN's public-facing interfaces, e.g., the interface with RAN that is well-specified. By focusing on generating and testing the standardized messages that are interchanged between the RAN and the O-RAN, `ORANalyst` bypasses the need for detailed knowledge of each component's internal messaging implementations. Yet, developing such an end-to-end testing framework introduces its own set of challenges, most notably when testing and sending inputs to a deeply rooted component within O-RAN. Inputs targeting these components must navigate through several layers of decoding and validation conducted by upstream components. Test inputs may get discarded by these checks and never reach their intended targets unless the knowledge of the sequence and order of those upstream components (i.e., dependency among components) is strategically incorporated while crafting inputs.

To craft inputs that can effectively explore each component, `ORANalyst` first computes the dependencies among components involved in different O-RAN operations. It dynamically records the network and program execution traces of inputs during regular O-RAN operations, and combines both traces to identify each input's entry and exit points in each component. By leveraging this component-level reachability analysis, `ORANalyst` designs a dependency-aware incremental testing strategy. With this approach, `ORANalyst` starts testing with the components directly connected to the RAN (i.e., the root in the component-level dependency tree), and then progressively moves to those located deeper within the O-RAN's communication chain (i.e., subsequent successors). This gradual approach helps us resolve the dependencies and interactions of components while thoroughly analyzing each layer of the system. To ensure that the inputs generated by `ORANalyst` can effectively reach the component under test (CUT) and still provide enough variations to test the component, we design an inter-procedural static analysis to extract path constraints along each component's entry point to the exit point, while mitigating the well-known state explosion problem. Path constraints collected from components upstream of the CUT are combined to guide the generated inputs past those previous components to reach the CUT. At the same time, path constraints of the CUT enable `ORANalyst` to create inputs that probe into CUT's deeper logic.

Results. We have evaluated `ORANalyst` on the only two widely-used, commercially-adopted O-RAN open-source implementations. `ORANalyst` identified 19 issues in both the platforming components and the application programs running on both implementations. These issues can lead to component crashes and messaging channel disruptions.

Responsible disclosure and open-sourcing. We have reported all 19 issues to developers and received confirmation on 9 issues. 15 CVEs are assigned for all 19 reported issues. `ORANalyst` is publicly available on GitHub [29].

Contributions. In summary, our contributions include:

- We develop the first systematic analysis technique tailored for O-RAN systems dubbed `ORANalyst` that can compute the complex dependencies among components in O-RAN's service-based architecture.
- We design an effective test input generation mechanism that combines dynamic analysis with efficient static path constraint extraction, enabling the generated inputs to effectively probe and test deep O-RAN components.
- We evaluate `ORANalyst` on two O-RAN implementations and uncovered 19 critical issues that lead to component crashes and messaging channel disruptions.

2 Background

In Release 15, the mobile telecommunication standard body, the Third Generation Partnership Project (3GPP) [1], defined a flexible, disaggregated architecture for the 5G-RAN, splitting the base station into the Central Unit (CU), the Distributed Unit (DU), and the Radio Unit (RU), each designed to host different layers/functions of the 5G radio stack. To accommodate 5G RAN's transition towards diverse and software-centric deployment, the O-RAN Alliance [21] designs O-RAN (depicted in Figure 1) to create a multi-supplier solution where equipment from various vendors can be easily plugged and played to form the RAN. It incorporates a service-based architecture (SBA), where different functions and services are implemented as separate microservice programs, communicating over network traffic. Each microservice can be

independently developed, deployed, and scaled on general-purpose hardware. In addition to the disaggregation of the base station, O-RAN introduces two RAN Intelligent Controllers (RICs), the *Near-Real-Time RIC (Near-RT RIC)* and the *Non-Real-Time RIC (Non-RT RIC)*, for automated optimization and control of network operations and resources in response to real-time network conditions.

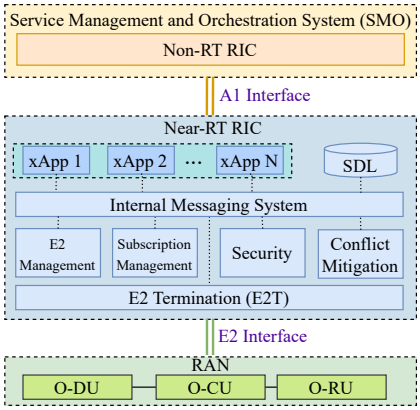

Figure 1: O-RAN RIC Architecture

❑ **Near-RT RIC and xApps.** Near-RT RIC's role is to dynamically manage and optimize the RAN by responding to network conditions in near real-time (10*ms* to 1*s* scale) [68]. Near-RT RIC's functionality is driven by modular applications known as *xApps*. These xApps can be sourced from third-party developers and may also be open-sourced, each designed to perform specific tasks, including load balancing, interference management, and Quality of Service (QoS) control. To support xApps' functionalities, Near-RT RIC provides various *platforming components*, such as a Shared Data Layer (SDL), Security, Conflict Mitigation, E2 Management, etc.

❑ **Non-RT RIC.** The Non-RT RIC operates on a timescale above 1 second and is part of the Service Management and Orchestration (SMO) framework. Its functions are typically strategic, e.g., policy control, network planning, and broader long-term RAN management tasks.

❑ **E2 open interface.** An O-RAN system can simultaneously govern multiple RAN nodes (i.e., CUs and DUs) from different operators. Each RAN node interfaces with the Near-RT RIC through the *E2 interface*. *E2 Termination (E2T)* is the end point of this interface in the Near-RT RIC, and *E2 Management* handles the setup, maintenance, and teardown of E2 sessions. The E2 interface runs over SCTP protocol [78] and is logically divided into two protocols: E2 Application Protocol (E2AP) [66] and E2 Service Model (E2SM) [67]. Both protocols are defined using ASN.1 formats [3] in the technical specifications. E2AP provides signaling procedures for the setup and modification of the E2 connection, error indication, and the reporting of general RAN status. E2SM supports service-level interactions between xApps and the RAN, e.g., reporting of various RAN-/cell-level metrics, control of RAN operations, etc.

❑ **Workflow between O-RAN RIC and RAN.** In a typical O-RAN RIC communication, after the SCTP connection between the Near-RT RIC and a RAN node is established, the RAN node first transmits an E2AP `E2SetupRequest` listing the service models it supports and the performance metrics it can report. The near-RT RIC processes this information and replies with an `E2SetupResponse` message. If the E2 setup is successful, xApps can send `subscription` messages to the connected RAN node specifying the E2SM used and the metrics they want to subscribe to. During this subscription negotiation, an xApp can specify trigger events or the periodicity with which the RAN node should send report metrics. Following a successful subscription, xApps continuously receive and process `indication` messages sent by the RAN, which contain the RAN's performance metrics, and optionally sending `control` messages to optimize RAN operations.

❑ **Open-source implementations.** Two commercially-adopted, widely-used O-RAN implementations are available as open-source projects: the O-RAN Software Community (O-RAN-SC) [23], which is developed through a collaboration between the O-RAN Alliance [21] and Linux Foundation [20]; and SD-RAN [32], developed by Open Networking Foundation (ONF) [27]. Both open-source projects are being utilized for commercial deployments. For example, O-RAN-SC's Near-RT RIC is being used by China Mobile in its private 5G network solution [15], whereas Deutsche Telekom is conducting a commercial field trial in Berlin using SD-RAN's RIC [24]. Both implementations are cloud-native where each component is shipped as a Docker image [6] and deployed in Kubernetes [17]. However, the two implementations differ in their approaches to handling logical components. For instance, the SD-RAN implementation consolidates the logical components of E2T, E2 management, and subscription management into a unified E2T microservice. In contrast, the O-RAN-SC implementation treats each of these components as distinct, individual microservices running in separate containers. Components in both implementations are developed using a variety of programming languages, with C/C++ and Go being the most predominant.

3 Threat Model & Motivation

In this section, we discuss our threat model and summarize the limitations of existing approaches when testing O-RAN RIC implementations, which motivates the design of `ORANalyst`.

3.1 Threat Model

O-RAN's interoperability [71], while offering significant benefits to network operators, also introduces substantial security challenges for the RIC. The RIC must manage interactions with RAN components from different vendors and network operators, each with distinct system implementations and differing security protocols. These variations not only exposes each individual RAN to potential security issues but

Category	Fuzzer	RT	NG	GA	NR	Notes on Requirements, Effectiveness and Efficiency
General Fuzzers	AFL [50]/LibFuzzer [18]	✗	✓	✗	✓	Requires complete control over the target program to fork, execute, and terminate for each testing run. Does not support long-running servers and can not explore the interactions between components.
Protocol Fuzzers	AFLNET [70]	✗	✓	✗	✗*	Similar to general fuzzers, requires complete control over the target server to fork, execute, and terminate for each testing run. *The stateful mode requires a response for each request.
Protocol Fuzzers	BooFuzz [4]/Peach [11]	✓	✗	✓	✓	Manual specification of message grammar is required. As generation fuzzers, no feedback is accepted, hence input generation is not guided by dynamic code exploration. Due to no feedback, a timeout mechanism is employed, which delays sending the next input, resulting in inefficiencies.
Protocol Fuzzers	Frizzer [9]	✓	✓	✗	✓	Dynamically instrument the code using Frida which is extremely slow. More importantly, Frida instrumentation is unstable and can lead to crashes of the target program, generating numerous false positives.
API Fuzzers	Restler [39]/EvoMaster [7]	✓	✓	✗*	✗	Requires a response with each request as feedback. *Can automatically generate Rest API-compliant messages, but does not work with the ASN.1 used in O-RAN.
Microservice Fuzzer	Evomaster RPC [83]	✓	✗	✓	✗	Requires manually developing a driver to send input and collect coverage for each testing component.
O-RAN Tester	`ORANalyst`	✓	✓	✓	✓	Can generate highly structured inputs automatically from the ASN.1 message definitions, while utilizing insights on the interactions of O-RAN components to test each target component efficiently.

Table 1: Comparison of Different Fuzzing Techniques. RT: support (R)emote (T)arget, NG: (N)o manual definition of (G)rammar is required, GA: (G)rammar-(A)ware input generation/mutation, NR: (N)o (R)esponse message required for each test case

also cumulatively heightens the overall network's susceptibility to breaches. Key security concerns include misconfigurations [45], vulnerabilities in implementation [45], dependency weaknesses [81], and potential compromises by malicious user devices, also known as User Equipments (UEs) [5]. Consequently, the RIC must be prepared for a variety of unexpected, malformed, or malicious inputs. Our threat model considers one misbehaving RAN transmitting unexpected or malicious inputs to the RIC. We do not consider vulnerabilities that require multiple misbehaving RANs collaborating to exploit in this work. We assume that RIC components are benign (but vulnerable) and do not attack each other as they are owned and deployed by the same party. This threat model aligns with the threat IDs T-O-RAN-05, T-E2-01, and T-E2-03 considered in O-RAN's official risk assessment [63]. O-RAN's study on security for Near-RT RIC and xApps [64] also identifies this threat as a critical issue and concludes that the Near-RT RIC should not assume that the data it received is valid or trusted. We align our work with the security assessments report by O-RAN. Under this threat model, our primary objective is to ensure the robustness and operational integrity of RIC against unpredictable and malicious inputs from RAN. This involves ensuring the RIC handles any unexpected data without crashing or hanging.

3.2 Limitations of Existing Testing Methods

Directly applying existing testing techniques for analyzing the O-RAN RIC encounters significant limitations, as summarized in Table 1. ($\mathbf{L_1}$): General program fuzzers, such as AFL [50], LibFuzzer [18], and Driller [77], are optimized for testing single monolithic command-line applications and struggle to navigate the intricate component-to-component interactions and diverse communication protocols inherent to O-RAN's distributed and service-based architecture. ($\mathbf{L_2}$): Protocol fuzzers like AFLNET [70] and BooFuzz [4] support sending testing inputs over network traffic, allowing them to target remote systems. However, they are generally tailored towards testing individual servers rather than an interconnected microservices environment. ($\mathbf{L_3}$): These fuzzers [4, 11] also often require the labor-intensive and error-prone task of manually constructing message grammars. ($\mathbf{L_4}$): Existing microservice fuzzers [83], though designed to address some aspects of service-based architectures, require the manual creation of driver code to feed inputs into test components and gather coverage feedback. ($\mathbf{L_5}$): API fuzzers like RESTler [39] and EvoMaster [7] primarily depend on analyzing response messages to guide their mutation strategies, whereas in O-RAN RIC, requests from the RAN often result in no observable responses. Besides, existing API fuzzers typically only support Rest APIs [28], instead of the ASN.1 messages on SCTP connections used by O-RAN's public interfaces.

3.3 Motivation for End-to-End Testing

`ORANalyst` employs an end-to-end design where all test inputs are sent through O-RAN's public-facing E2 interface. An alternative approach would be to adapt existing fuzzers [18, 50, 70, 77] to test individual components of the RIC in isolation. With this approach, the fuzzer abstracts away the interactions between the component under test (CUT) and other RIC components by using stubs, i.e., simplified components that perform no actual functionality. Upon finding a crash-inducing input, the fuzzer also needs to validate that the input can cause the crash in an end-to-end setup. However, O-RAN operations often involve complex inter-component communications, such as querying a database for cell configuration data of a RAN. Testing in isolation not only demands extensive effort to manually stub these interactions to produce realistic data [59] but also increases the risk of analysis errors from over-simplified stubbing. Furthermore, this isolated testing approach tends to generate many false positives, as many crash-inducing inputs detected under this approach would likely be filtered out by intermediate components in an end-to-end deployment, thus never reaching the CUT.

To illustrate this scenario, consider the high-level E2T processing logic presented in Listing 1. When it receives a new message from the RAN, it first decodes the message (line 2), validates the message content (line 6), and then processes the message or forwards it to other components based on the

```
1  func processE2AP(e2apByte []byte) error {
2    e2ap, err := decode(e2apByte)
3    if err != nil {
4      return err
5    }
6    if err = validateE2AP(e2ap); err != nil {
7      return err
8    }
9    switch e2ap.type{
10   case indication:
11     processIndication(e2ap)
12     // ... other cases
13   }
14 }
15
16 func processIndication(e2ap E2AP) {
17   indMsg := e2ap.IndicationMsg
18   if err := validateIndication(indMsg); err != nil {
19     return err
20   }
21   if stream, ok := getStream(indMsg); ok {
22     sendIndMsg(indMsg)
23   } else {
24     return errors.NewNotFound("stream not found")
25   }
26 }
```

Listing (1) Illustrative E2T Implementation

```
1 func GetUeID(ueID *e2sm_v2_ies.Ueid) (int64, error) {
2   switch ue := ueID.Ueid.(type) {
3   case *e2sm_v2_ies.Ueid_GNbUeid:
4     return ue.GNbUeid.GetAmfUeNgapId().GetValue(), nil
5   // ... other UE ID type cases
6 }
```

Listing (2) False Positive example, taken from SD-RAN's rimedo-ts xApp

```
1  struct encode_result encode_action_definition(const
       char *action_bytes){
2    // ... setup variables to be used for decoding
3    // decode action_bytes to e2smKpmRanFunctDescrip
4    asn_dec_rval_t rval =  asn_decode((void**)&
       e2smKpmRanFunctDescrip, action_bytes);
5    char **actionArr;
6    if(rval.code == RC_OK) {
7      // ... extract the action items from
       e2smKpmRanFunctDescrip to actionArr
8    }
9    // ... encode variables setup, setup buffer buf for
       encoding result
10   int encodeRes = encode_action_arr(&buf[0], &buf_size
       , actionArr);
11   int encoded[encodeRes]; // * vulnerable line *
12   // ... further processing using encoded
13 }
14
15 size_t encode_action_arr(unsigned char *buf, size_t *
       buf_size, char **actionArr) {
16   // allocate memory for actionDef, the structure to
       be encoded
17   for (int index = 0; index < actionDefCount; i++) {
18     // ... constructing the action definition list to
       actionDef for each index in actionArr
19   }
20   // ... constructing other fields of actionDef
21   asn_enc_rval_t encode_result = asn_encode_to_buffer(
       actionDef, buf, *buf_size);
22   if (encode_result.encoded == -1) {
23     return -1; // returns -1 when encounters an
       encoding error
24   }
25   // ... normal return of positive encoded size
26 }
```

Listing (3) Example of an unchecked return value issue, taken from O-RAN-SC's kpimon-go xApp, simplified for presentation

message type (lines 9 - 13). Therefore, all inputs must pass the initial checks and validations enforced by the E2T before reaching any other components. Now consider the simplified code in Listing 3, which demonstrates a crashing vulnerability found by `ORANalyst` from O-RAN-SC's kpimon-go xApp [16]. The vulnerability's root cause lies in lines 10 and 11, where the return value `encodeRes` can be negative in the event of an encoding error. This value is then used as the length for initializing an array without checking for negativity, leading to unexpected behavior when using the array and a program crash. At first glance, the issue might seem straightforward and detectable by certain existing static or dynamic analysis techniques [40, 41, 43, 46, 72, 74]. However, such detection results in numerous false positives. To illustrate, consider Listing 2 showing an SD-RAN's xApp [25]. Here, a pointer's field access in line 2 is not validated for nullity. A tool analyzing only the xApp would likely flag this as a crash-inducing bug. Yet, in an end-to-end setup where the input is sent from the RAN, we found that the E2T component enforces a non-null check on this field when validating the input (in a function represented by line 18 in Listing 1), leading to early rejection of the message. Consequently, such inputs can never reach the vulnerable xApp to trigger a crash.

Furthermore, testing any RIC component in isolation by directly interfacing the fuzzer with the CUT poses several additional challenges due to unspecified message formats for internal components and difficulties in triaging the results. O-RAN specifications do not define messaging protocols and formats used for communications between two *internal* RIC components. As a result, different RIC implementations employ different internal messaging infrastructures using custom APIs or formats, even for components performing similar functionalities. For instance, O-RAN-SC utilizes a customized routing system to relay indication messages from the E2T to the targeted xApps. However, SD-RAN's implementation uses gRPC's streams to complete the same task. Identifying these diverse messaging formats and crafting appropriate messages for each component demands manual and error-prone code scanning. On the other hand, verifying that the identified vulnerabilities in each component are exploitable from RAN inputs is also challenging. It requires complex and manual mapping of the tested internal message to the messages sent from RAN. In contrast, in an end-to-end testing setup, the protocols and messages accepted by the RIC from the RAN are well-defined in the E2AP and E2SM protocols by O-RAN specifications. In addition, any issues found by test inputs sent over the E2 interface are directly exploitable by a misbehaving RAN node. These reasons justify an end-to-end testing approach that aligns with our proposed threat model.

3.4 Scope of Testing

Our threat model specifically targets the E2 interface, the Near-RT RIC that is directly connected with E2, and the xApps that operate within the Near-RT RIC, due to their critical role in responding directly to RAN conditions. For the sake of exposition, we refer to the O-RAN RIC broadly as O-RAN occasionally in this paper. In this work, our main focus is on effectively reasoning about component dependencies and

interactions in O-RAN. We limit our scope to finding memory corruption issues that can cause O-RAN components to crash or become unresponsive, and do not consider logical or stateful bugs. Each of our testing runs is conducted in a stateless manner, which also reflects the RAN's limited control over the RIC's operational states. From the RAN's perspective, interactions with the RIC can be broadly categorized into two phases: *pre-connection* and *post-connection* with the RIC. The transition from the pre-connection to the post-connection is through RAN sending a successful `E2SetupRequest` message to the E2T. We test both scenarios by configuring the RIC to the corresponding state before starting testing. We only test individual inputs instead of sequences of inputs. While testing in the pre-connection state, if an `E2SetupReueqst` message sent from the RAN is accepted by the E2T, the state is transitioned to post-connection. `ORANalyst` terminates the established connection and initiates a new SCTP connection to the E2T, effectively resetting the state to pre-connection. O-RAN's technical specifications mandate SBA for implementing its RIC [61], so monolithic implementations are not specification-compliant. In this work, we focus on testing service-based implementations.

4 Challenges & Methodology

We first summarize the challenges in designing an end-to-end testing framework for O-RAN, then present `ORANalyst`'s design which aims to address these challenges.

4.1 Challenges of End-to-End O-RAN Testing

(C1) Generating targeted and meaningful test inputs. For end-to-end testing of an O-RAN RIC, inputs from the RAN are initially processed by the E2T (entry point in the RIC from RAN), where they undergo decoding and validation before being either routed to their target components or discarded. For testing components downstream of the E2T, e.g., xApps (as shown in Figure 1), it is essential to generate inputs that can successfully reach these targets to ensure effective testing. Moreover, without inputs reliably reaching the intended target, a timeout mechanism has to be employed to infer the inputs' failure to reach the target [51], resulting in more delays.

Motivating Example. To better explain this challenge, we illustrate it with the running example in Listing 3. Test inputs sent to the RIC from the fuzzer emulating as a RAN connecting to the RIC first pass through intermediate components (e.g., E2T) before reaching their target (i.e., the xApp in this example). To exploit the vulnerability in lines 10 and 11, the `encode_action_arr` function must return a negative value. We observe that the encoding process in line 21 may result in an error if the action definition list, constructed in the while loop from lines 17 to 19, is empty. The length and contents of this list are determined by the parameter `actionArr`, which is extracted in the `encode_action_definition` function from lines 6 to 8. Therefore, to uncover the vulnerability, the `action_bytes` variable, which contains the encoded information about supported RAN functions for RIC control, must either be decoded successfully but contain no action items or fail to decode. It is important to note that `action_bytes` are embedded within an `E2SetupRequest`, sent by the RAN during registration to the RIC. Should the entire `E2SetupRequest` message fail to decode or contain malformed information, it will be rejected by the E2T during the checks in lines 2 or 11 in Listing 1, preventing the message from reaching the affected xApp.

In summary, for a fuzzer to reveal the vulnerability, the test input it generates must satisfy the following criteria: **(i)** It must first navigate past decoding checks and validations of earlier components, such as the E2T, to reach the targeted component. **(ii)** Upon reaching the test component, the input must successfully pass initial decoding (syntactic correctness check) to prevent an early exit and ensure that it reaches the problematic code segment. **(iii)** Finally, the input must also contain some anomaly to trigger the vulnerability (e.g., an encoding error at line 22 in Listing 3 by providing an empty action definition list within `actionDef`.

(C2) Constantly evolving message formats. Existing general-purpose grammar-aware input generation tools like Peach [11] or BooFuzz [4] require manual extraction of message formats. No automated ASN.1 message generation tool currently exists. Libprotobuf-mutator-asn1 [19] can generate ASN.1 messages from protobuf [30] definitions but requires a manual ASN.1-to-protobuf translation. These manual efforts pose challenges for testing O-RAN protocols, i.e., E2AP [66] and E2SM [67], where hundreds of structures are used to define numerous message types. To make matters worse, both protocols are frequently updated by the standard body, rendering the manual construction and maintenance of grammar from the technical documents for each version impractical. Since O-RAN's initial release in 2020, both E2AP and E2SM protocols have undergone four major revision versions alongside numerous minor updates. Additionally, O-RAN implementations often utilize different versions of these protocols [23, 26, 32].

(C3) Identifying O-RAN's end-of-processing of an input. For efficient testing, a new test input should be dispatched as soon as the previous one finishes processing in the CUT. On the other hand, for accurate coverage collection and fault attribution, inputs must not overlap with previous processing ones. Precisely identifying the end-of-processing for input is often challenging in O-RAN RIC, as the RIC does not respond to certain types of inputs from the RAN (e.g., indication messages). This differs from the expectations of API fuzzers [7, 39], which depend on response messages to determine the end of a request. Moreover, unlike traditional fuzzing conventions, remote microservice targets in O-RAN do not signal input processing completion through function returns or control transfers back to the fuzzer.

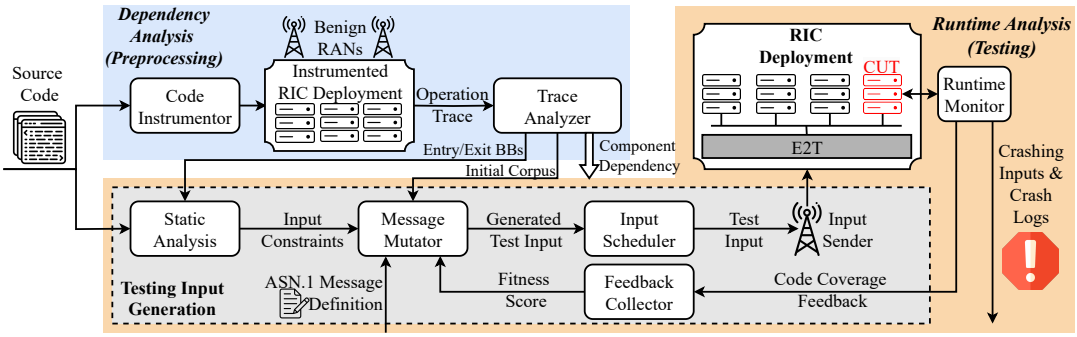

Figure 3: Architecture of ORANalyst. Details are discussed in §4.2

4.2 ORANalyst's Methodology

To address the challenges detailed in §4.1, we design ORANalyst, an end-to-end, grammar-guided, feedback-driven fuzzing framework for O-RAN. In this design, the fuzzer emulates a RAN node connected with the target O-RAN RIC system. To sidestep challenge (C1), ORANalyst takes an incremental testing approach, where it focuses on testing a singular O-RAN component at any time, while also considering the complex interactions among other components in the RIC without stubbing any components. Initially, ORANalyst focuses on the "shallow" components that are directly connected to the fuzzer. In this workflow, the first component tested is E2T, as it directly interfaces with the RAN, and is responsible for managing all message decoding and routing to other components of the RIC. During the testing of each component, ORANalyst also collects the path conditions required for the inputs to propagate to the next component through an efficient static analysis. We discuss in detail how we circumvent challenges in static analysis, such as the path explosion problem in §5. As the testing progresses, we move to components deeper in the system, aggregating path conditions collected from the previous components to constrain input mutation so that the generated message can reach the component under test (CUT) consistently. The architecture of ORANalyst is detailed in Figure 3. To achieve the proposed methodology, ORANalyst operates in two stages: a *preprocessing* stage, where it extracts component dependency information, and a *runtime testing* stage.

4.2.1 Dependency Analysis Through Dynamic Tracing

When an input is sent to the RIC, it sequentially passes through several RIC components for processing. While static analysis allows ORANalyst to find execution paths of an input within a component, it cannot directly identify the subsequent component an input flows to. This inter-component information flow occurs via network traffic, such as API [28] or gRPC [13] calls, and the destination endpoint cannot be easily extracted through static methods. This is because endpoints from different components are connected through RIC's internal routing and messaging systems, which are managed dynamically during O-RAN's runtime. Additionally, extracting this information from O-RAN specifications is also not possible since implementations vary in their designs of the internal components. For example, SD-RAN merges the logical E2T, E2 management, and subscription management into a single E2T component, while O-RAN-SC separates each function into individual components. Manually scanning the code and compiling this information is an arduous and error-prone task. To address this, ORANalyst collects dynamic execution and network traces during the normal operation of O-RAN and constructs a *component dependency tree*.

The goal of dynamic tracing is to capture the flow of all message types in the O-RAN implementation to understand the implementation's components dependency. Specifically, for each message type, dynamic tracing provides three types of information: **(i)** A sequence of components the input passes through to guide ORANalyst's testing order from shallow- to deep-lying components during runtime analysis; **(ii)** Detailed information on the entry and exit points (basic blocks) of the input in each component, which the static analysis uses to generate path constraints for ORANalyst to create targeted inputs for testing the CUT (detailed in §5); and **(iii)** Raw packets collected over the E2 interface, used as initial corpus during runtime analysis. To extract these three pieces of information, we set up the target O-RAN implementation with all available xApps configured to communicate with unmodified benign RANs, recording naturally occurring traffic from O-RAN's regular operations for 24 hours. We collect two types of traces: **(i)** we use tcpdump [34] to record all raw packets sent and received by each RIC component, forming a network trace. **(ii)** We instrument each RIC component to assign unique identifiers to each edge and basic block in the source code, tracing code execution paths by collecting all executed basic blocks and edges during runtime to form an execution trace.

ORANalyst combines the collected network and execution traces using timing, order, and message type information to form a comprehensive operation trace. By aligning the order and timestamps of the network packets with the corresponding execution paths, it maps the sequence of components that each message type traverses. From the execution trace of each component, it extracts the entry and exit basic blocks

for the message traversed in a component. Although the execution path information is also available, we do not use it as the traces may not cover all execution paths from entry to exit, potentially over-restricting the input generation. Instead, we use static analysis to identify all potential execution paths and associated path constraints required for the input to flow from entry to exit, i.e., to the next component. We group the extracted information by message types and cross-validate to ensure consistency across the same message types. If messages of the same message type result in different extracted component dependencies or entry and exit points, it may indicate that the message type is not fine-grained enough. For example, indication messages of the same service model may have different reporting formats, resulting in different extracted dependencies. In case of any inconsistencies, we refine the message type distinctions to be more fine-grained if necessary, by sub-categorizing the message types.

As dynamic tracing only records messages that naturally occur during a 24-hour operation of O-RAN with benign RANs, it is possible that some rarely occurring messages are not captured. In that case, we do not test those messages. This is acceptable as this work aims for correctness, i.e., ensuring that all detected issues are true vulnerabilities exploitable by a misbehaving RAN node, rather than completeness, i.e., we do not claim to find all vulnerabilities. We provide an evaluation of our dynamic tracing in §7.2. Note that we did not encounter any circular dependencies, i.e., no single input traverses the same component more than once.

4.2.2 Runtime Analysis

The component dependency extracted during preprocessing guides `ORANalyst`'s runtime analysis. `ORANalyst` focuses on testing one component at a time, beginning with the first component the input encounters in the RIC (E2T), and progressively moves to deeper components. To ensure that test inputs can navigate past previous components to reach the deeper ones, input generation is refined iteratively during the testing of each CUT. `ORANalyst` extracts the necessary path constraints that ensure the input navigates past the CUT by designing an efficient *static analysis*, as detailed in §5. This allows `ORANalyst` to systematically explore and test each component in a dependency sequence. Below, we introduce other key components in the runtime analysis process.

❑ **Grammar-aware input generation.** Inputs sent to the RIC from RAN through the public-facing E2 interface follow the highly structured ASN.1 format, as specified by E2AP [66] and E2SM [67] protocols and detailed in the corresponding technical specifications. Traditional byte-level input generation and mutation methods [18, 50] view the entire input as bytes, ignoring its underlying structure and context. This approach leads to a significant proportion of test inputs being discarded by the RIC's decoding functions checking the syntactic correctness of the inputs. As a result, those inputs cannot reach the CUT's input processing logic, where the main functionalities of the component reside. Consequently, byte-level input generation and mutation methods miss the opportunity to uncover potentially more impactful vulnerabilities within the processing logic.

To address this, `ORANalyst` employs a grammar-aware mutation strategy that generates inputs conforming to the expected input structure but incorporating unexpected values. The grammar-aware generation strategy also allows us to control specific field values in the generated inputs to satisfy input constraints collected from static analysis. However, as mentioned in (C2), manually translating message formats from ASN.1 to fuzzer-specific formats [4, 11, 19] for various message types and versions in O-RAN protocols is not feasible. To overcome this, we leverage the limited number of datatypes in ASN.1 to directly implement mutation rules on ASN.1 formats without translating them into internal representations. By operating directly on ASN.1 formats, we can also support ASN.1-specific features like field constraints and encoding/decoding operations using existing ASN.1 compilers [35] to auto-generate information from ASN.1 grammars.

We implement a universal set of generation and mutation rules based on the *field datatypes* and the *constraints* each datatype may enforce, rather than on specific fields in a message. `ORANalyst` applies these universal rules across different message types and protocol versions during testing, eliminating the need for manual updates with each new revision. Raw packets collected over the E2 interface during dependency analysis are used as the initial corpus for the *message mutator*. The mutator decodes and mutates messages in the corpus, and can also generate missing message fields. Table 2 details the generation and mutation methods applied to each ASN.1 datatype. All inputs generated by the grammar-aware message mutator are in valid ASN.1 syntax. However, `ORANalyst` also selectively incorporates byte-level mutations into the generated and encoded test inputs to craft slightly malformed packets for testing the decoding logic of CUT. The motivating example in Listing 3 demonstrates that some encoding errors are required to trigger certain vulnerabilities, emphasizing the need to also test slightly malformed packets.

❑ **Identifying the end of input processing.** Test inputs generated by the message mutator are scheduled by the *input scheduler* and sent to the CUT by the *input sender*. However, as discussed in challenge (C3), it is difficult to determine when a test input has finished being processed by the CUT. To address this, we observe that like typical web server implementations, each component in the RIC operates in a continuously running loop that accepts new requests indefinitely. The return to the beginning of a new loop iteration, where a new request can be accepted, might indicate the completion of processing the previous input. However, in O-RAN, new threads are often spawned for parallel input processing. This means that even if the main loop is ready to accept new inputs, the processing of the previous input may still be ongoing in

Field Type	Constraints	Generation Method	Mutation Method
Boolean	-	Generate both True and False.	Change the boolean value to the alternative.
Integer	Value Constraint	Generate "interesting" values (e.g., 0, positive, negative, boundary values) within constraints.	Randomly select an integer value or assign boundary values within the given constraints.
Enumerated	Valid Enums	Generate all possible enumeration values.	Randomly change to a different enumeration option.
Bit String	Bit Size Constraint	Create bit strings of varying lengths, including boundary cases, within constraints.	Within size constraints, perform byte-level mutations, or generate a new random bit string.
Octet String	Octet Size Constraint	Generate random octet strings of diverse lengths, including edge cases.	Modify to a random octet sequence within size limits, or opt for boundary lengths.
Sequence / Sequence Of	Sequence Length	Produce sequences of varied lengths, including boundary lengths, randomly populating each entry according to its type.	Randomize length in constraints or set to lower/upper constraints, and randomly delete/add/mutate each element based on type.
Choice	-	Create instances of all available choice options using methods described in this table depending on the field type.	Mutate to an alternative choice option or modify the existing one while introducing random mutations in the subfields.
Optional	-	Populate the optional field although it is allowed to be empty.	Apply probabilistic mutations to the values of optional fields, randomly remove existing fields, or introduce previously absent ones.

Table 2: Generation and Mutation Methods for Each Field Type

different threads. To accurately determine the end of request processing, we establish two criteria: **(i)** the readiness of the main message-receiving loop to accept the next input, which `ORANalyst` monitors by instrumenting the start of the loop; and **(ii)** the termination of all threads initiated for processing the prior input, which `ORANalyst` tracks by instrumenting all thread creations and terminations. Existing O-RAN implementations we tested do not use thread pools to manage their threads. This is because, in Go, threads (goroutines) are lightweight and inexpensive to create and terminate, and there are simpler ways to limit concurrency than using thread pools. Consequently, `ORANalyst` only monitors thread creation and termination. However, support for thread pools can be added to `ORANalyst` by instrumenting the tasks submitted to the thread pool and monitoring the callbacks when tasks finish.

❑ **Feedback loop.** Leveraging components introduced above, we design `ORANalyst` to generate inputs that effectively reach any specific CUT in the O-RAN RIC. The *runtime monitor* oversees the processing of each input in the CUT and evaluates the outcome. Once input processing is complete, the runtime monitor first checks for and reports any crashes triggered by the input, and then restarts the crashed component. If no crash occurs, it collects code coverage data from the execution, which is then sent back to the *feedback collector* for fitness score calculation and subsequent test input scheduling. To handle scenarios where an input fails to reach the CUT, `ORANalyst` employs a timeout mechanism. If a series of consecutive inputs or a significant portion of inputs fail to reach the CUT, the runtime monitor raises an alert indicating a potential stall in the CUT or an issue with the input constraints. We detail this identification mechanism in §7.2.

5 Input Constraints Generation

To generate inputs that effectively reach and explore each component in the RIC, `ORANalyst` combines static analysis with dynamic tracing to generate path constraints that guide inputs from the E2 interface to the target component. Using dynamic tracing, as detailed in §4.2.1, `ORANalyst` extracts information about the specific components an input reaches, the sequence/order of traversal, and the entry and exit points (basic blocks) in each component. `ORANalyst` begins its testing of each component with an intra-component static dataflow analysis to collect path constraints required to reach the exit point (e.g., API or gRPC calls to another component) of the CUT from its entry point. Satisfying these path constraints ensures test inputs can reach the next component. These path conditions serve two critical purposes: First, they provide semantic guidance for the fuzzer when exploring the CUT, enabling it to produce inputs that are not only syntactically valid but also semantically interesting for exploring different branching paths of the CUT. In this setup, values satisfying and violating branch conditions are both tested. Second, in the context of testing subsequent components in the communication chain (based on the component dependency tree), these conditions are used to generate inputs that can navigate through the current component and reach the next one. In this scenario, `ORANalyst` specifically generates inputs that meet the branch conditions necessary to reach the exit point of the current component. Below we discuss details about the static analysis to ensure efficiency and accuracy.

5.1 Critical Path Conditions

When performing the static analysis on a CUT, we observe that enumerating all paths and branch conditions the input traverses from the entry to the exit quickly leads to path explosion issues. The problem arises from the need to exhaustively explore possible paths at each branch point, demanding significant memory and computing resources. This challenge is further exacerbated by the numerous checks and complex processing logic dependent on various message fields that are common in RIC components. We, however, observe that while many branch conditions evaluate different field processing routines, they usually do not alter the control flow leading to the exit. Moreover, we notice that certain errors in input processing or validation logic, despite triggering logging activities, do not result in input rejection, indicating these errors do not affect the message's progression to the exit.

For instance, consider the motivating example in Listing 3, where `action_bytes` undergo decoding in line 4. As these bytes are extracted from the input and are therefore tainted during the static analysis, an inter-procedural dataflow analysis would examine the `asn_decode` function's complex logic

and branches, resulting in path explosion. However, note that the return value `rval`, while used to guide input processing in line 6, does not lead to the early termination of the function even if the return value indicates an error, thereby not affecting the overall control flow. This observation allows us to bypass a detailed analysis of the decode function since the decoding result does not affect the control flow.

```
1 func AssociateRanToE2THandlerImpl(data models.
      RanE2tMap) error {
2   err := validateE2TAddressRANListData(data)
3   if err != nil {
4     xapp.Logger.Warn(" Association of RAN to E2T
       Instance data validation failed: " + err.Error()
      )
5     return err
6   }
7   // further processing
8   return sendRoutesToAll()
9 }
```

Listing 4: Critical Path Condition Example

Based on the above observation, we found that only a small portion of all path conditions checked along an execution path from the entry to the exit point need to be satisfied for the input to reach the exit. We denote such path conditions as *critical conditions*, and refer to the branches depending on these critical conditions as *critical branches*. Listing 4 illustrates an example of such critical path conditions, which depicts a simplified implementation for associating a RAN to an E2T instance in O-RAN-SC's routing manager. Here, if the validation check in line 2 fails, the input is discarded in line 5, preventing further processing or its acceptance. Thus, the condition in line 3 is deemed critical, and `ORANalyst` focuses on finding the input constraints that can fulfill this condition. This targeted focus on critical branches allows us to circumvent the common path explosion problem while computing path constraints using static analysis. This approach also ensures that identified critical conditions are enough to enable the input to traverse to the exit.

5.2 Extracting Path Conditions

To efficiently identify critical branches within a component whilst avoiding path explosion, we design an inter-procedural static analysis based on the Program Dependency Graph (PDG) [49] which comprises both the Control Dependency Graph (CDG) and the Data Dependency Graph (DDG). Using a function call graph, we first identify all sequences of function calls from an entry point to an exit point of the CUT, which we denote as function call paths. The task of identifying critical path conditions from entry to exit is then reduced to identifying the conditional branches that affect the execution of subsequent function calls along each function call path.

The algorithm `ORANalyst` uses to extract the critical path conditions is presented in Algorithm 1. It traces each function call in a function call path (lines 1-6) and identifies the basic blocks containing the call site that invokes the subsequent function in the call path (line 7). It then analyzes the callee

Algorithm 1 Critical Path Constraint Collection

Input:
$\mathcal{F}_{entry}$: Function Containing the Entry Point
$\mathcal{F}_{exit}$: Function Containing the Exit Point
Output: List of Boolean Expressions Representing the Critical Path Conditions Required to be Satisfied to Reach $\mathcal{F}_{exit}$ from $\mathcal{F}_{entry}$

1: **procedure** CollectPathConstraint
2: totalPathConstraint = { }
3: **for** each function call path P_i from $\mathcal{F}_{entry}$ to $\mathcal{F}_{exit}$ **do**
4: pathConstraint = { }
5: **for** each edge E_j in P_i **do**
// Get caller and callee of edge
6: f_{caller}, f_{callee} = getCallerAndCallee(E_j)
// Get target basic block in the caller containing call-site to callee
7: $b_{callsite}$ = getCallSiteBlock(f_{caller}, f_{callee})
// Get control blocks of $b_{callsite}$ in f_{caller}
8: $B_{control}$ = getControlDependentBlocks(CDG(f_{caller}), $b_{callsite}$)
// Append conditions in last instructions of control blocks
9: **for** each control block b_k in $B_{control}$ **do**
10: pathConstraint = pathConstraint ∪ getCondition(b_k)
11: **end for**
12: **end for**
13: totalPathConstraint = totalPathConstraint ∪ pathConstraint
14: **end for**
15: **return** totalPathConstraint
16: **end procedure**

function's CDG to determine the control dependency of these target basic blocks (line 8). Given a target basic block in a function, the CDG shows all basic blocks whose control decisions at the end of the block directly determine whether the target basic block is reachable. All these control decisions must be satisfied to reach the target basic block. These control decisions are collected as path conditions, and we extract all such conditions for all function calls in a function call path. By combining these conditions (lines 9-16), we obtain the path constraints required to follow all function call paths, i.e., the path constraints to reach the exit point from the entry point of a CUT. To handle indirect function calls, we use off-the-shelf points-to-analysis techniques [79] available for Go-SSA IR for Go programs and LLVM-IR for C/C++ programs. We observe that the use of pointers and function pointers (i.e., indirect calls) in existing O-RAN source code is very limited. Therefore, existing points-to-analysis tools are adequate for our analysis. This observation may not be true for all future O-RAN implementations, but it is a common trait of most protocol implementations.

We illustrate the above idea with the example presented in Listing 4. In this example, the target line to reach is line 8, and the critical branch is the `if` branch in line 3. If the branch condition `err` $!=$ `nil` is evaluated to be true, line 8 would not be reachable. Hence, we collect the condition `err` $!=$ `nil` as well as its required value `False` as the critical condition. Since we only focus on extracting critical path conditions utilizing CDG, our methodology can effectively disregard non-critical branches, such as those pertaining to error logging that do not result in early function termination. Furthermore, this approach also sidesteps the exponential increase in

enumerating paths that typically results in path explosion.

5.3 Selective Function Processing

Some conditions collected during the previous step may depend on the return values of function calls. For example, the collected condition `err!=nil` in Listing 4 depends on the result of the `validateE2TAddressRANListData` function. To determine what inputs can satisfy this condition, `ORANalyst` needs to further analyze the `validateE2TAddressRANListData` function. However, some functions, such as library functions, typically do not contain meaningful constraints on the input. To alleviate path explosion, we selectively analyze only functions that yield meaningful constraints on the input. To this end, we first discern functions that contain meaningful input constraints, which we call *validating functions*, as they typically apply validation checks that inputs must satisfy to reach the exit point. We call other functions *generic*.

The `validateE2TAddressRANListData` function is an example of a validating function, as it validates the input and contains critical conditions on inputs. Inputs containing values that fail this validating function get discarded by the component. Conversely, *generic* functions can generate errors in case of failure and can also affect control flows, but they do not yield meaningful constraints on the inputs. For example, in Listing 5, the database retrieval function SDL.Get is a generic function. We would skip analyzing the implementation details of this function since it would not result in meaningful constraints over the input. Instead, our static analysis would only record the function itself as a condition, i.e., that the data retrieval corresponding to the key is successful.

```
data, err := SDL.Get(key)
if err != nil {
  return err
}
// further processing using data
```

Listing 5: Generic Function Example

Based on this insight and the following observation, we design a heuristic approach to distinguish between generic and validating functions. We observe that generic functions are typically called multiple times in different contexts due to their broad use cases, while validating functions are used in more specific contexts. We use the following criteria to identify the functions we want to process: **(i)** The function should not be from a library or an imported open-source shared project, as these functions are often generic and involved in data management, logging, or networking (e.g., `SDL.Get` in Listing 5). **(ii)** The function should not be invoked by more than a threshold number of functions across more than a number of packages of the CUT. This criterion helps identify and exclude generic functions that are likely to be invoked in different contexts due to their broad applicability. In practice, we use empirical observations to set the threshold. During testing, if a function is invoked by more than three functions in the CUT or by functions from more than two different packages, we label it as generic and do not analyze it. We provide some assessments of the classification accuracy in §7.2.

Finally, for the functions we decide to analyze, we apply a similar methodology as described in §5.2 to collect conditions. We identify the basic blocks generating the required return value and utilize the function's CDG to determine the conditions for reaching these blocks. For example, in the `validateE2TAddressRANListData` function in Listing 4, we examine the basic blocks returning no errors and use the CDG to identify the conditions for reaching these blocks. This process is recursive, continuing until all conditions are resolved without dependencies on other validating functions.

5.4 Path Conditions To Input Constraints

After extracting the path conditions of each function call path, we map the collected conditions as constraints to specific fields in the generated test inputs. Each extracted path condition is represented as a boolean variable, along with its requisite value (True/False) for reaching the target block. For example, Listing 4 produces the boolean variable condition err != nil, and the required value False, as explained in §5.2. To generate constraints on inputs based on each condition, we employ a *backward dataflow analysis*. This analysis traces the variable involved in the condition back to a field access operation from the input through a backward taint analysis. `ORANalyst` then maps the condition on the boolean variable to constraints of specific fields in the input. The extracted constraints are used as additional rules for the input generator.

Loops. Performing backward analysis on loops can create scalability issues and potentially unsound results. However, due to the characteristics of protocol implementations, we observe that the majority of loops in O-RAN's RIC components iterate over a list of items in the message. In such scenarios, each iteration functions independently of the others, and the execution results from each iteration do not carry over to other iterations. We identify these scenarios by confirming that in the loop body, there are no cross-iteration dependencies (e.g., in SSA form, no variables defined inside the loop are used outside of their defined iteration, and no *phi* instructions are used to select a value from previous iterations). For such cases, we only analyze the loop once, treating the loop index as a symbolic value representing the list index. For other generic loops performing calculations where results are carried over iterations, we limit the number of loop iterations we process by capping the number of loop unrollings `ORANalyst` performs.

6 Implementation

We implement `ORANalyst` with five main components: (i) automated instrumentation for feedback collection from CUT, (ii) inter-component input tracing, (iii) static analysis for in-

put constraints, (iv) grammar-aware input generator, and (v) fuzzing engine to compute fitness score and schedule inputs.

Feedback instrumentation. For C/C++ components, we use AFL++ [50]'s afl-clang as our instrumentation tool. For Go, we extend go-fuzz [12]'s AST-based basic block coverage tool to include edge coverage by instrumenting branching statements for comprehensive path tracking. A universal customized runtime monitor is injected in each CUT to monitor the runtime information, manage shared memory for coverage data, and collect and relay coverage feedback to the fuzzer.

Dynamic tracing. We employ tcpdump [34] to capture network traces for each RIC component. Using our instrumentation tool, we assign unique identifiers to each edge and basic block to track code execution and collect execution traces.

Static analysis. We construct PDG using the LT algorithm [54] and the algorithm by Ferrante et al. [49]. For Go programs, we perform the static analysis on SSA intermediate representation (IR), while for C/C++ programs, we utilize SVF [79] to work with LLVM IR.

Grammar-aware input generation. The input generation utilizes datatype definitions, constraint specifications, and encoding/decoding routines from code generated by the ASN.1 C compiler [35] using E2AP and E2SM ASN.1 grammars [66, 67]. Additionally, we incorporate input constraints identified through static analysis to input generation by applying special rules on the specific messages and fields involved.

Fuzzing engine. We build our fuzzing engine by extending go-fuzz [12] to support remote coverage data reception, static analysis, and grammar-aware input generation.

7 Evaluation

❏ **Experiment setup.** For our experiments, we use a laptop with Intel i7-9750H CPU and 16GB DDR4 RAM. Only two matured and commercially-adopted open-source O-RAN RIC implementations exist: O-RAN-SC [21] and SD-RAN [32]. We evaluated both implementations. Specifically, we tested on SD-RAN's latest 1.4 release and O-RAN-SC's most recent I-release. Both O-RAN implementations offer a Kubernetes-based deployment for their RIC, wherein each microservice component is separately compiled into a Docker image and then deployed as a pod within the Kubernetes deployment, communicating with other components via network traffic.

To prepare an O-RAN implementation for testing, we first instrument the source code of the CUT and recompile it following the original Dockerfile into an instrumented Docker image. We then replace the original CUT's Docker image with the instrumented image in the Kubernetes deployment. For each vulnerability identified by `ORANalyst`, we conduct manual validation to ensure that the vulnerability can be reproduced in the original deployment, thereby confirming that the instrumentation does not introduce any false-positive issues. We have not encountered any false positives caused by our instrumentation. When we discover a crash that is triggered continuously and hinders the deeper testing of the component, we manually devise and apply a patch to suppress the crash.

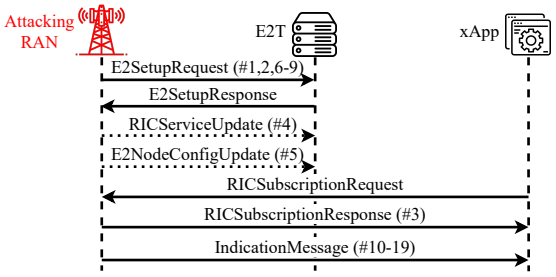

Figure 4: Flow of exploitable messages between the attacking RAN, E2T, and xApp. Other RIC components are omitted for conciseness. Dotted lines represent optional messages, while solid lines represent required messages in the procedure. # denotes the vulnerability IDs corresponding to the message.

7.1 Identified Issues & Potential Exploits

We deploy `ORANalyst` to test 4 xApps and 6 platform components across two O-RAN RIC implementations. These components vary in size, with each one's lines of code (LoC) typically ranging from several thousand to 100,000. The testing is conducted in both the pre-connection and post-connection state, each for a 24-hour period for each component. We identify 19 distinct issues across 7 components. Among these, 17 issues led to crashes, and 2 resulted in the blockage of communication channels, thereby obstructing all subsequent messages, as detailed in Table 3. Notably, only 7 of the 19 issues (#1, 6, 7, 8, 9, 13, 14) were triggered by malformed packets, which either caused segmentation faults or index out-of-bounds panics during decoding. Conversely, the remaining 12 issues were exploitable through well-formed inputs. The discovery of these issues underscores `ORANalyst`'s efficacy in reaching and exploring the processing logic, while findings through malformed inputs demonstrate that `ORANalyst` is also capable of discovering issues in the decoding logic. Figure 4 illustrates the flow of messages within the O-RAN procedures and highlights the messages that trigger vulnerabilities listed in Table 3. The figure shows the sequence in which these messages interact with the E2T and an xApp within the RIC. For simplicity, this diagram does not include other components involved in the RIC, such as E2 Management.

❏ **Memory issues.** The most common problems we identified are memory-related, such as crashes caused by index out-of-bounds panics. For example, the Key Performance Measurement (KPM) E2SM indication message type includes a field for an array of measurement items and an integer count of the number of items in the array. In SD-RAN's kpimon xApp, we observed that the count value is used to iterate over the array items without validating the array's boundaries. Consequently, when the actual number of items is less than the reported count, an index out-of-bounds panic is triggered, leading to the xApp's crash (issue #19). Attackers can cause

	Component	#	Cause	Vulnerability Description	Exploit Message	Impact of Vulnerability	Location
ORAN-SC	e2t	1	**IA**	Unexpected input causes 'std::invalid_argument' exception, leading to termination of E2T. This can be exploited to cause a DoS of all RIC components and RANs managed by the E2T.	E2SetupRequest	Crash & DoS	sctpThread-.cpp:1941
		2	**SF**	Unexpected input leads to segmentation fault. This can be exploited to crash the service, causing a DoS.	E2SetupRequest	Crash & DoS	sctpThread-.cpp:2180
		3	**SF**	Similar to above, leading to service crash and potential DoS.	RICSubscription-Response	Crash & DoS	sctpThread-.cpp:2451
	e2mgr	4	**OR**	Runtime panic due to array index out of range. This can be exploited to disrupt the service by causing unexpected crashes.	RICServiceUpdate	Crash & DoS	ric_service_update_-handler.go:107
		5	**OR**	Runtime panic due to index out of range, leading to service disruption.	E2NodeConfigur-ationUpdate	Crash & DoS	e2_node_config-uration_update_-ack.go:279
	kpimon-go	6	**UR**	Return value used as size to initialize array without checks. This may lead to a negative-sized array, which can be exploited to cause memory corruption or execute arbitrary code.	E2SetupRequest	Crash, DoS, & Memory Corruption	wrapper.c:L246
		7	**UR**	Similar to above, leading to crashes and memory corruptions.	E2SetupRequest	Crash, DoS, & Memory Corruption	wrapper.c:266
		8	**UR**	Similar to above, leading to crashes and memory corruptions..	E2SetupRequest	Crash, DoS, & Memory Corruption	wrapper.c:277
		9	**UR**	Similar to above, leading to crashes and memory corruptions.	E2SetupRequest	Crash, DoS, & Memory Corruption	wrapper.c:288
		10	**OR**	Index out of range error, which can be exploited to cause unexpected crashes and disrupt the service.	IndicationHeader	Crash & DoS	control.go:485
		11	**OR**	Index out of range error, similar to above, leading to DoS.	IndicationMessage	Crash & DoS	control.go:530
		12	**SF**	Exploit message cause unexpected signal during runtime execution. This can crash the service, causing a DoS.	IndicationMessage	Crash & DoS	Not yet identified
SD-RAN	onos-e2t	13	**OR**	Runtime panic due to index out of range. This can be exploited to cause crashes and disrupt the service.	IndicationMessage	Crash & DoS	marshal.go:57
		14	**OR**	Slice bounds out of range, which can be exploited to corrupt data and cause crashes.	IndicationMessage	Crash & DoS	aper.go:140
	ric-sdk-go	15	**EH**	Errors not handled properly, which can be exploited to crash the entire stream and block communication channels.	IndicationMessage	Communication Channel Blockage	node.go:223
	rimedo-ts	16	**OR**	Slice bounds out of range, which can lead to DoS.	IndicationMessage	Crash & DoS	reader.go:36
		17	**OR**	Slice bounds out of range, similar to above, DoS.	IndicationMessage	Crash & DoS	reader.go:18
	onos-kpimon	18	**EH**	Error channel blocking due to unhandled errors. This can be exploited to block subsequent messages and disrupt service.	IndicationMessage	Communication Channel Blockage	monitor.go:207
		19	**OR**	Index out of range error, which can lead to DoS.	IndicationMessage	Crash & DoS	monitor.go:133

Table 3: Summary of Discovered Vulnerabilities

All vulnerabilities are exploitable by malicious RAN nodes. Vulnerability Causes: **SF**: (S)egmentation (F)ault; **OR**: index/slice bounds (O)ut of (R)ange; **EH**: improper (E)rror (H)andling; **UR**: (U)nhandled (R)eturn value; **IA**: (I)nvalid (A)rgument error (due to unexpected input formats).

the affected components to crash by sending carefully crafted requests that exploit these memory issues in memory-safe languages like Go, leading to service disruptions. For components implemented in C/C++, which are not memory-safe, an attacker may exploit memory issues to trigger component crashes or even remote code execution.

❏ **Improper error handling.** Our evaluation uncovers two error-handling issues (#15, 18) that lead to the disruption of communication channels. Specifically, `ORANalyst` identified a flaw (#15) within the error handling of indication message streams of onos-sdk-go, an SDK used by xApps in SD-RAN. xApps employ this SDK to receive indication messages over gRPC streams. According to the usage of gRPC streams, any error in the stream requires the abortion of the current stream and the re-initialization of a new one. However, when `ORANalyst` generates inputs that trigger an error in the stream, the SDK tries to read repeatedly from the same stream. The read operation is enclosed in an infinite loop, and continuously trying to read from the aborted stream traps the process in an endless loop of reprocessing the same error, rendering the xApp incapable of processing any new messages. All SD-RAN xApps using onos-sdk-go are vulnerable to this issue.

❏ **Attacks and impacts.** As shown in Figure 4, an adversary exploiting the identified vulnerabilities can craft malicious inputs and send those to the target components from the malicious RAN to trigger crashes. Crashes within the O-RAN's RIC components can further lead to substantial DoS impacts on other RANs connected to the same RIC. Restarting the crashed component and re-establishing communication with other components and RANs could take several minutes. An adversary can also repeatedly crash the component after each restart. E2T is particularly vulnerable, as it serves as the primary conduit for message distribution from and to RANs, and its failure could deny all O-RAN services for the connected RANs. For example, a malicious RAN could send carefully crafted `E2Setup` messages to exploit vulnerability #2, triggering segmentation-fault to crash E2T, disrupting all communications between O-RAN-RIC and connected RANs. Moreover, the communication channel blockage issues (#15, 18) are undetectable from the RAN, as the affected RAN does not receive any error messages and can continue transmitting to the blocked channel. This effectively disables communication between the xApp and the RAN indefinitely.

7.2 Evaluation of Intermediate Components

Below, we present an evaluation of the correctness and completeness of `ORANalyst`'s intermediate components against manually constructed baselines to provide insights into the efficacy of these components. Additionally, we conducted ablation studies of components in `ORANalyst` in §7.3.

❏ **Dynamic tracing (§4.2.1).** We instrumented four xApps and six platform components across two O-RAN RIC implementations and collected traces of messages that occurred

during a 24-hour operation of O-RAN with benign RANs. We collected 14 GB of execution traces, containing sequences of executed basic block and edge identifiers, and 23 GB of network traces in pcap format. From the collected traces, we extracted 15 types of messages across the two RICs, including `E2SetupRequest`, `E2ConfigurationUpdate`, `E2ConnectionUpdate`, and `indication` messages of different service models. However, we did not collect any traces of `resetRequest` in SD-RAN as this message only occurs to reset the connection from an abnormal failure. Similarly, we did not collect any `errorIndication` message trace in O-RAN-SC as this message is only used to report errors not reportable by other appropriate response messages.

❏ **Generic/validating functions classification (§5.3).** Mislabeling generic functions as validating (i.e., false-positives) overloads static analysis to process unnecessary functions, wasting time and resources and risking path explosion if the false-positive rate is high. Conversely, false-negative cases might result in missing path conditions, preventing some generated test inputs from reaching the CUT. Calculating the false-negative rate is challenging due to the large code base size of O-RAN implementations. For example, SD-RAN's E2T alone contains over 6,000 functions and over 100,000 LoC, excluding imported libraries. However, the results of static analysis are used to construct input constraints that ensure the test inputs generated by `ORANalyst` can effectively reach and explore the CUT. Instead of directly finding the false-negative cases, we monitor the percentage of test inputs that successfully reach the CUT. If this success rate falls below a threshold (99% in our testing), we manually analyze rejected inputs to identify any missed validating functions. We have not experienced the percentage dropping below the threshold during testing. On the other hand, manually verifying false-positive cases is manageable because, for each entry/exit pair, only around 30 functions are labeled as validating in each CUT. During our testing, we identified only 4 false-positive cases. Including these false-positive functions would not cause path explosion during static analysis.

❏ **Generated input constraints (§5.4).** To provide insights into the accuracy of the extracted input constraints, we manually validated the constraints for the `E2SetupRequest` and `indication` messages in SD-RAN's E2T component. For an `E2SetupRequest` to pass through the E2T, `ORANalyst` extracts 13 constraints on input fields, which are enforced to confirm the message type and validate field types and contents. Although the static analysis encountered a loop iterating over a list of information elements in the message, this loop was correctly resolved using the approach discussed in §5.4, as there were no dependencies between loop iterations. We confirmed that all 13 extracted constraints were accurate. For an `indication` message to pass through the E2T, `ORANalyst` collected three input constraints, which we validated to be accurate. These constraints specify that the input is of the correct type (indication message) and that the specified stream for forwarding the indication message exists. To further complement this evaluation, we evaluate the overall impact of the input constraints on the efficacy of `ORANalyst` by conducting ablation studies testing `ORANalyst` without using the extracted input constraints in §7.3.

7.3 Benchmark with State-of-the-Art

❏ **Benchmark tools adaptions.** We test state-of-the-art tools to benchmark against `ORANalyst`. However, none of these tools can support O-RAN implementations off-the-shelf. We discuss the adaptations made to these tools below.

(a) AFLNET [70]. AFLNET, an adaptation of AFL [2] for network protocol fuzzing, uses response messages to extract state information that guides mutations. It requires complete control over the server binary for forking, execution, and termination. To use it with O-RAN, we developed an intermediate program to manage remote coverage feedback and forward it to AFLNET. Since inputs sent to the RIC often yield no response messages, AFLNET infers incorrect state transitions for these messages, which negatively impacts its performance. Hence, we disabled AFLNET's stateful testing.

(b) BooFuzz [4]. As a successor to Sulley [33], BooFuzz is a generation-based, extensible network protocol fuzzer used in several recent protocol fuzzing research [76, 82]. Unlike feedback-driven fuzzers, BooFuzz relies on predefined message grammars to generate test inputs. Constructing these grammars is manual and error-prone, especially for complex protocols like E2AP and E2SM. For our analysis, we simplify this by allowing BooFuzz to directly generate test inputs from the initial corpus by treating each message as a byte array. The corpus column for BooFuzz in Table 4 represents the number of total test inputs generated by BooFuzz. In addition, BooFuzz does not accept any feedback. To prevent input overlap, a brief delay between tests is introduced for BooFuzz, which highlights its inefficiency. Completing the testing of all BooFuzz-generated inputs required just under 22 hours out of the allocated 24 hours of testing budget.

(c) Frizzer [9]. Frizzer is a coverage-guided black-box fuzzer based on the Frida [8] instrumentation toolkit. It can remotely connect to the Frida instance running on the target program's host, making it suitable for testing remote targets like O-RAN components directly. However, it requires manual identification of the address of the main network protocol handler function in the target server. We manually identified this address by inspecting the CUT binary. During testing, we found that Frizzer's Frida injections frequently caused segmentation faults in the E2T, resulting in crashes unrelated to the test inputs and generating numerous false positives. These crashes also triggered E2T restarts, with each restart taking considerable time, significantly reducing testing efficiency. After reviewing all crashes discovered by Frizzer, we confirmed that they were all false positives. Consequently, we have not included Frizzer's results in Table 4.

O-RAN-SC Component	E2T				Kpimon					
Fuzzer	crashes	corpus	cover	% decoded	crashes	corpus	bb cover	edge cover	% reaching xApp	% decoded
`ORANalyst`	3	2149	4326	72.35	3	73	1838	910	100/100	55.64
`ORANalyst` w/o input constraints	3	2149	4326	72.35	1	47	1828	907	47.27/59.01	53.50
`ORANalyst` w/o grammar	0	1433	4647	3.9	1	59	1831	906	40.64/80.81	16.76
AFLNET	0	245	3663	21.78	0	41	1824	901	32.81/97.83	12.37
BooFuzz	1	427033*	3655	81.96	1	427033*	1824	899	10.71/11.65	33.40
Radamsa	0	1323	3916	3.76	0	66	1827	901	11.39/78.20	4.40
Radamsa-filter	0	137	3467	100	1	35	1820	896	62.54/62.54	86.13

Table 4: Comparative Testing Results

(d) Radamsa [31]. Radamsa is used by Frizzer for message mutation. To avoid the instability caused by Frizzer's Frida instrumentation, we replace Frida with our coverage collection mechanism and evaluate Radamsa's effectiveness.

(e) Radamsa with filter. Our evaluation shows that most inputs generated by Radamsa fail to pass the initial decoding checks, and Radamsa struggles to generate meaningful test inputs that can thoroughly explore the CUT. To address this, we implement a pre-filter to examine the format of generated messages. Only those messages that successfully passed the decoding checks were forwarded to the target in this mode.

❏ **Benchmark CUTs.** To evaluate `ORANalyst`'s performance under O-RAN's multi-language environment, we selected two components as benchmark targets: (i) the ORAN-SC's E2T component, developed in C++, and (ii) the kpimon-go xApp, implemented in Go. We perform the testing in the post-connection to the RIC state, i.e., after RIC's acceptance of a successful `E2SetupRequest` as part of the testing setup. In this testing state, `ORANalyst` identifies 3 crashes for kpimon-go (i.e., #10-12 in Table 3), as opposed to 7 discovered in both states. For fairness, we evaluated all benchmark tools, along with `ORANalyst` on the same hardware setup, using identical initial corpus sets for a 24-hour period on each target. We conducted the testing of two targets separately by accepting code coverage feedback from the corresponding target component. Inputs aimed at testing E2T may also flow to the xApp, and any inputs targeting the xApp must first pass through and be routed by E2T. Since BooFuzz [4] is a generation fuzzer and does not accept feedback to guide its input generation during test-time, testings of both E2T and the xApp use the same set of test inputs generated by BooFuzz from the provided initial corpus set.

❏ **Metrics.** We collect the following statistics for each test, summarized in Table 4: **(i)** The number of issues found by each tool. **(ii)** The number of interesting corpora generated. In mutation fuzzers, an input is saved as a corpus if it explores previously uncovered code regions, so this metric roughly represents the fuzzer's ability to generate diverse inputs exploring different code regions. For BooFuzz, the corpus column reflects the total test cases generated. **(iii)** The number of basic blocks and edges covered by test inputs. Given that the aflclang tool used for instrumenting C++ components is limited to edge coverage, for E2T, we only collect the edge coverage. **(iv)** The percentage of inputs successfully decoded by the CUT and thus reaching its processing logic is also computed. We manually identify the start point of the input processing logic and record the number of inputs reaching this point. In this testing setup, E2T decodes the E2AP layer, while the xApp decodes the E2SM layer of inputs. **(v)** For the kpimon xApp, which is not directly connected to the fuzzer, we record the percentage of all test cases that can reach the xApp. The first value represents the percentage of all inputs that reach the xApp, and the second value represents the percentage of inputs successfully decoded by E2T that reach the xApp.

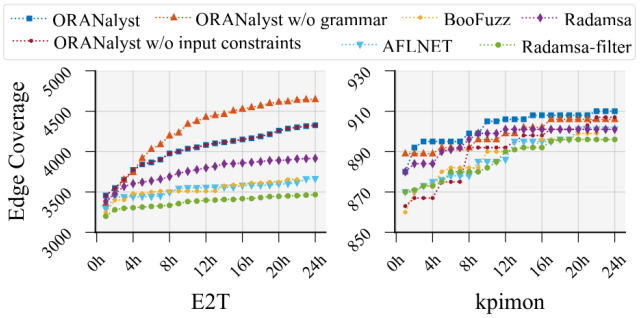

Figure 5: Discovered Edges Over Time by Evaluated Fuzzers

❏ **Ablation study.** We conduct an ablation study and present the results in Table 4. This study evaluates `ORANalyst` without extracted input constraints but still employing message grammar, as well as disabling the grammar-aware mutation entirely (also no input constraints). Since all inputs sent by `ORANalyst` through the E2 interface reach E2T, no input constraints are required to test E2T. Therefore, we directly use the evaluation results for `ORANalyst` to demonstrate the outcome of testing the E2T without input constraints. While `ORANalyst` without grammar-aware mutation achieves higher coverage for E2T, most of the test cases fail the decoding checks, indicating that the increased coverage likely comes from exploring the decoding logic. `ORANalyst` without grammar also cannot find any issues in E2T, and only finds one crash in the xApp. When testing the xApp, `ORANalyst` without input constraints performs only slightly better than `ORANalyst` with no grammar at all in terms of test inputs reaching the xApp or decoded by the xApp. This demonstrates that `ORANalyst` without input constraints struggles to generate inputs that efficiently reach and explore the xApp.

❏ **Benchmark results.** Comparative evaluation results of other tools is also presented in Table 4. All identified issues

are crashes. Figure 5 visualizes the number of covered unique edges by the benchmark tools. ORANalyst significantly outperforms all other tools, achieving both the highest coverage and the most discovered crashes. As discussed in §4.2.2, ORANalyst probabilistically applies byte-level mutations on well-formed inputs generated by the grammar-aware mutator to produce slightly malformed inputs, to test the decoding logic in addition to the processing logic. These malformed inputs that reach previously unexplored code regions are then added to the corpus for further mutations. As illustrated in Listing 3, carefully mutated malformed inputs can uncover vulnerabilities that strictly valid inputs cannot expose. During the evaluation, we configured the probability for a byte-level mutation at 10%, which explains why a subset of inputs generated by ORANalyst fails to be decoded. Setting this byte-level mutation probability to 0% would ensure all inputs generated by ORANalyst pass the decoding checks.

Although Radamsa with filter results in less coverage than Radamsa without filter in both components, it finds more issues due to its focus on input processing logic rather than decoding logic. Not all inputs generated by Radamsa-filter are decoded by the xApp because the E2SM layer contains some byte fields with encoded data that the filter does not validate, leading to possible decoding errors when the xApp tries to decode those bytes. BooFuzz identifies issues #1 in the E2T and #12 in the xApp. Both ORANalyst without grammar and Radamsa-filter identify issue #10 in the xApp. ORANalyst without input constraints is the same as ORANalyst when testing the E2T, and it finds issue #12 in the xApp.

8 Discussion and Limitations

Manual efforts. To prepare an O-RAN implementation for testing, we manually modify the original Dockerfile for each component to incorporate the instrumentation tool during the build process. We then build the instrumented Docker image and replace the original one in the O-RAN deployment. To collect network traces for dynamic tracing, communications between components must be unencrypted, or the encryption key must be saved to decode the collected traffic. We opt for saving the encryption key to decrypt the network traffic. While ORANalyst's components are mostly automated, some steps involve manual intervention. For example, transferring outputs from one component to another, such as inserting identified input constraints from static analysis into input generation, is currently manual. Automating these steps through scripting is primarily an engineering effort. Furthermore, we manually triage and diagnose the root cause of each issue.

Generalizability of ORANalyst. There are only two widely-used, commercially-adopted O-RAN implementations available, and we demonstrate ORANalyst's generalizability by testing both. While our input generation currently supports only ASN.1-specified messages, ORANalyst's testing methodology can be extended to other protocol implementations and microservice systems. Our observations about the limited use of complex loops and pointers discussed in §5 are based on common characteristics typical of protocol implementations in general, not just O-RAN.

ORANalyst assumes the availability of source code or intermediate representation (IR) for two reasons: **(i)** to instrument the components and collect execution traces to understand component dependencies, and **(ii)** to dynamically monitor test input execution and collect coverage data as testing feedback. For scenarios where only program binaries are available, additional methodologies and toolkits are required to perform these tasks. This can be achieved through reverse engineering [10, 14], binary instrumentation [55], or runtime analysis [36]. When even the binaries are unavailable, dependency analysis and feedback collection can be conducted through network traffic analysis [34, 37] or log analysis [38].

Monitoring multiple components simultaneously. Although our approach utilizes insights about component dependencies and the complex communication traversal of inputs, the testing focuses on a single component at a time, while also taking into account components' inter-dependencies and interactions. Extending our monitoring to multiple components simultaneously could uncover more complex vulnerabilities stemming from the interplay of components.

9 Related Work

Protocol testing. Several protocol fuzzers focus on performing state-aware testing from response codes [70], state tracking graphs [56] and dynamic queries to server [52, 60], but they cannot be applied to testing service-based systems. Also, many approaches use grammar-guided fuzzing to generate semantically valid inputs, either through manually constructed state and input message models [4, 11, 80] or through information extracted from source code [57, 75]. However, they either require manual work or assume specific source code formats, and cannot be applied to O-RAN.

Testing leveraging source code analysis. Several approaches leverage lightweight static analysis or concolic execution to guide fuzzers in generating interesting inputs [44, 48, 58, 69, 77]. Driller [77] leverages concolic execution to solve path constraints to reach deeper code branches and unexplored compartments. However, its concolic execution cannot avoid path explosions nor guarantee comprehensive component coverage. ORANalyst's dynamic and static analysis combination sidesteps path explosion and guarantees test input's reachability to all components. Peng et al. [69] adopt lightweight static analysis to extract an abstract state machine of the implementation and guide the fuzzer to generate inputs exploring new states and transitions. However, all of the previous works focus on monolithic programs and cannot handle inter-program communications between components in O-RAN RIC.

O-RAN security. The O-RAN Security Work Group has conducted several threat models, risk assessment, and security

studies of O-RAN components [62–65], but they only offer broad guidelines. Polese et al. [71] outline general research directions for O-RAN, including its security aspects. Shen et al. [73] explores the possibility of authentication and authorization attacks in the O1 interface of O-RAN, and Dik et al. [47] assess security threats at the transport layer of O-RAN's open fronthaul interface. Existing research on the emerging O-RAN technology has been concentrated on theoretical security issues or potential research avenues. This study is the first effort aiming at identifying actual security vulnerabilities within O-RAN implementations.

10 Conclusion

We propose the first systematic testing framework designed for O-RAN implementations, `ORANalyst`. By integrating efficient static analysis with dynamic trace analysis, `ORANalyst` resolves complex component dependencies in O-RAN RIC. It applies these insights to generate test inputs that can pass the initial decoding and validation checks to effectively explore deeply rooted components. Evaluation of `ORANalyst` on two major open-source O-RAN implementations yields the discovery of 19 previously uncovered vulnerabilities.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their feedback and suggestions. We also thank O-RAN RIC developers for cooperating with us during responsible disclosure. This work has been supported by the NSF under grants 2145631, 2215017, 2226447, 1801534, and 1900873, the Defense Advanced Research Projects Agency (DARPA) under contract number D22AP00148, the NSF and Office of the Under Secretary of Defense Research and Engineering under grant ITE 2326898, and National Telecommunications and Information Administration (NTIA)'s Public Wireless Supply Chain Innovation Fund.

References

[1] *3GPP - The Mobile Broadband Standard*. www.3gpp.org.

[2] American fuzzy lop. https://github.com/google/AFL.

[3] ASN.1 Project. https://www.itu.int/en/ITU-T/asn1/Pages/asn1_project.aspx.

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

[5] CVE-2021-45462. https://nvd.nist.gov/vuln/detail/CVE-2021-45462.

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

[7] EvoMaster. https://github.com/EMResearch/EvoMaster.

[8] Frida: A dynamic instrumentation toolkit. https://frida.re.

[9] Frizzer. https://github.com/demantz/frizzer.

[10] Ghidra. https://ghidra-sre.org/.

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

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

[13] gRPC: a high-performance RPC framework. https://grpc.io/.

[14] IDA-Pro. https://hex-rays.com/ida-pro/.

[15] 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.

[16] kpimon xApp of O-RAN-SC. https://gerrit.o-ran-sc.org/r/admin/repos/ric-app/kpimon-go,general.

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

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

[19] Libprotobuf-mutator-asn1. https://github.com/google/libprotobuf-mutator-asn1.

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

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

[22] O-RAN in the news. https://www.o-ran.org/in-the-news.

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

[24] 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/.

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

[26] Open AI Cellular. https://www.openaicellular.org/.

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

[28] OpenAPI. https://www.openapis.org/.

[29] ORANalyst. https://github.com/SyNSec-den/ORANalyst.

[30] Protocol Buffers. https://protobuf.dev/.

[31] Radamsa. https://gitlab.com/akihe/radamsa.

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

[33] Sulley. https://github.com/OpenRCE/sulley.

[34] tcpdump. https://www.tcpdump.org/.

[35] The ASN.1 Compiler. https://github.com/vlm/asn1c.

[36] Valgrind. https://valgrind.org/.

[37] Wireshark. https://www.wireshark.org/.

[38] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. Android SmartTVs vulnerability discovery via Log-Guided fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2759–2776. USENIX Association, August 2021.

[39] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, page 748–758. IEEE Press, 2019.

[40] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, apr 1975.

[41] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[42] Thomas Byrd, Vuk Marojevic, and Roger Piqueras Jover. Csai: Open-source cellular radio access network security analysis instrument. In *2020 IEEE 91st Vehicular Technology Conference*, pages 1–5, 2020.

[43] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of High-Coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, December 2008.

[44] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[45] Merlin Chlosta, David Rupprecht, Thorsten Holz, and Christina Pöpper. Lte security disabled: Misconfiguration in commercial networks. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, page 261–266, New York, NY, USA, 2019. Association for Computing Machinery.

[46] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, page 196–206, New York, NY, USA, 2007. Association for Computing Machinery.

[47] Daniel Dik and Michael Stübert Berger. Open-ran fronthaul transport security architecture and implementation. *IEEE Access*, 2023.

[48] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. Finding fixed vulnerabilities with off-the-shelf static analysis. In *2023 IEEE 8th European Symposium on Security and Privacy*. IEEE, 2023.

[49] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987.

[50] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies*, August 2020.

[51] Eduardo Gonzalez, Stan McClellan, and Wuxu Peng. Rtomin as a balancing parameter between fast retransmissions and timeouts within stream control transmission protocol (sctp). In *The 2014 2nd International Conference on Systems and Informatics*, pages 687–691, 2014.

[52] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. Dynsql: Stateful fuzzing for database management systems with complex and valid sql query generation. In *Proceedings of USENIX Security Symposium*, 2023.

[53] Hema Kadia. Current State of Open RAN. https://tecknexus.com/5g-network/5g-magazine-open-ran-june-2021/current-state-of-open-ran-countries-operators-deploying-trialing-open-ran/, June 2021.

[54] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1979.

[55] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 190–200. Association for Computing Machinery, 2005.

[56] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4481–4498.

[57] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. Polar: Function code aware fuzz testing of ics protocol. *ACM Transactions on Embedded Computing Systems*, 18:1–22, 2019.

[58] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*.

[59] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.*, volume 18, pages 1–11, 2018.

[60] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. MundoFuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1257–1274, Boston, MA, August 2022. USENIX Association.

[61] O-RAN Working Group 1. *O-RAN Architecture Description 11.0* . www.o-ran.org/specifications.

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

[63] O-RAN Working Group 11. *O-RAN Security Threat Modeling and Risk Assessment 1.0*. www.o-ran.org/specifications.

[64] O-RAN Working Group 11. *O-RAN Study on Security for Near Real Time RIC and xApps 4.0*. www.o-ran.org/specifications.

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

[66] O-RAN Working Group 3. *O-RAN E2 Application Protocol (E2AP) 4.0*. www.o-ran.org/specifications.

[67] O-RAN Working Group 3. *O-RAN E2 Service Model (E2SM) 4.0*. www.o-ran.org/specifications.

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

[69] Hui Peng and Mathias Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.

[70] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, 2020.

[71] Michele Polese, Leonardo Bonati, Salvatore D'oro, Stefano Basagni, and Tommaso Melodia. Understanding o-ran: Architecture, interfaces, algorithms, security, and research challenges. *IEEE Communications Surveys & Tutorials*, 2023.

[72] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, page 179–180. Association for Computing Machinery, 2010.

[73] CT Shen, YY Xiao, YW Ma, JL Chen, Cheng-Mou Chiang, SJ Chen, and YC Pan. Security threat analysis and treatment strategy for oran. In *2022 24th International Conference on Advanced Communication Technology (ICACT)*, pages 417–422. IEEE, 2022.

[74] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.

[75] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[76] Congxi Song, Bo Yu, Xu Zhou, and Qiang Yang. Spfuzz: A hierarchical scheduling framework for stateful network protocol fuzzing. *IEEE Access*, 7:18490–18499, 2019.

[77] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[78] Randall R. Stewart. Stream Control Transmission Protocol. RFC 4960, September 2007.

[79] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[80] Andreas Walz and Axel Sikora. Exploiting dissent: towards fuzzing-based differential black-box testing of tls implementations. *IEEE Transactions on Dependable and Secure Computing*, 17:278–291, 2017.

[81] Wentao Wang, Faryn Dumont, Nan Niu, and Glen Horton. Detecting software security vulnerabilities via requirements dependency analysis. *IEEE Transactions on Software Engineering*, 48(5):1665–1675, 2022.

[82] Yingchao Yu, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. Sgpfuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. *IEEE Access*, 8:198668–198678, 2020.

[83] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. White-box fuzzing rpc-based apis with evomaster: An industrial case study. *ACM Trans. Softw. Eng. Methodol.*, 32(5), jul 2023.