ORANClaw: Shredding E2 Nodes in O-RAN via Structure-aware MiTM Fuzzing

Geovani Benita
Singapore University of Technology and Design
Singapore
geovani_benita@mymail.sutd.edu.sg

Matheus E. Garbelini
Nanyang Technological University
Singapore
matheus.garbelini@ntu.edu.sg

Sudipta Chattopadhyay
University of Missouri-Kansas City
Kansas City, Missouri, USA
schattopadhyay@umkc.edu

Jianying Zhou
Singapore University of Technology and Design
Singapore
jianying_zhou@sutd.edu.sg

Abstract

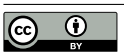
The open radio access network (O-RAN) standard provides a foundational move towards disaggregated RAN architecture, allowing flexibility and multi-vendor integration. For example, the Radio Intelligent Controller (RIC) may involve third-party applications (xApps) to dynamically control and monitor network behavior, facilitating significant opportunities for multi-party involvement, but allowing potentially untrusted integration with the RAN. In this paper, we propose, design and evaluate *ORANClaw* — a structure aware, man-in-the-middle fuzzing framework that takes full control over the E2 interface between the xApps and the RIC, and systematically mutates or duplicates packets communicated via this interface to disrupt the behavior of the base station (gNB). *ORANClaw* takes into account the structural and semantic constraints while systematically mutating the packets. Furthermore, it optimizes the mutation strategy based on the coverage of explored state transitions. We have implemented *ORANClaw* and evaluated it with FlexRIC, O-RAN SC RIC, OpenAirInterface, ns-3 simulator and commercial VIAVI TeraVM AI RAN Scenario Generator gNB/RIC. In total, *ORANClaw* has discovered 71 unique bugs (eight CVEs already assigned): 28 in FlexRIC, one in O-RAN SC RIC, 37 in the gNB implementations of OpenAirInterface and ns-3. Additionally, *ORANClaw* uncovered five distinct vulnerabilities in commercial VIAVI TeraVM AI RSG gNB/RIC. Our evaluation also reveals that structure and semantic-aware mutations within *ORANClaw* are key factors in revealing these bugs. Overall, *ORANClaw* provides an open platform to automatically validate both the RIC and gNB implementations via xApp manipulations.

CCS Concepts

• **Security and privacy → Denial-of-service attacks**.

Keywords

O-RAN, Structure-Aware Fuzzing, MiTM, E2AP, ASN.1



WiSec '26, Saarbrücken, Germany

ACM ISBN 979-8-4007-2201-1/2026/06
https://doi.org/10.1145/3765613.3797451

ACM Reference Format:
Geovani Benita, Matheus E. Garbelini, Sudipta Chattopadhyay, and Jianying Zhou. 2026. *ORANClaw:* Shredding E2 Nodes in O-RAN via Structure-aware MiTM Fuzzing. In *Proceedings of the 19th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '26), June 30-July 03, 2026, Saarbrücken, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3765613.3797451

1 Introduction

Traditional Radio Access Networks (RANs) were typically built as monolithic, vendor-specific systems, limiting flexibility, scalability, and innovation. To address these limitations, the 3rd Generation Partnership Project (3GPP) introduced a disaggregated architecture for 5G RAN in Release 15, separating the base station into three distinct components: the Central Unit (CU), Distributed Unit (DU), and Radio Unit (RU).

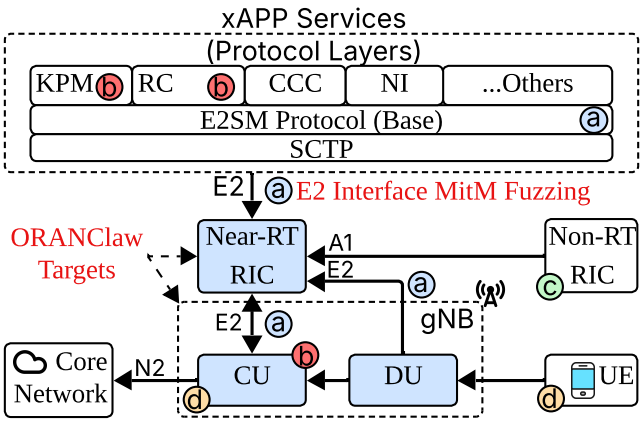


Figure 1: Position of ORANClaw w.r.t related work: a) *ORANClaw* (E2 interface fuzzing) targeting Near-RT RIC and gNB. b) Malicious gNB targeting Near-RT RIC [43], c) Malicious Non-RT RIC targeting Near-RT RIC [40], d) Malicious gNB targeting Core Network [5].

The Open RAN (O-RAN) standard fundamentally shifted this paradigm by disaggregating RAN functions across standardized interfaces, enabling multi-vendor interoperability and introducing programmable control through the Radio Intelligent Controller

(RIC). In particular, the RIC architecture allows third-party applications (xApps) to dynamically optimize network behavior via real-time control and monitoring capabilities through the E2 Application Protocol (E2AP). While this shift from closed, monolithic systems to open, disaggregated architectures enhances flexibility and innovation, it also expands the attack surface [13, 43] and introduces new security challenges that remain insufficiently explored in current research.

In this paper, we propose *ORANClaw* to systematically investigate a novel attack surface that targets the RAN via third-party and potentially untrusted xApps (see Figure 1ⓐ). *ORANClaw* is a structure-aware and man-in-the-middle (MITM) fuzzing framework, which is placed between the xApps and the RIC (i.e., the E2 interface), as illustrated in Figure 1❶. This allows *ORANClaw* to have full control over the packets through E2 interface and all of its E2SM services, hence acting like a potentially malicious/vulnerable xApp. An appealing feature in the design of *ORANClaw* is that it does not require any access to the RAN source code, thus, facilitating opportunities for testing commercial ORAN stacks. Earlier work on structure-aware fuzzing was targeted to Core Network (see Figure 1ⓓ) and is not directly applicable to our ORAN threat model. In addition, its implementation is unavailable [5] and therefore, extending such implementation remains infeasible in practice. In the O-RAN space, prior works have targeted testing the near real-time RIC (Near-RT RIC) via malicious gNB (see Figure 1ⓑ), or malicious Non real-time RIC (see Figure 1ⓒ). None of these prior works target the gNB or generalize to fuzz all E2SM services (see Figure 1ⓑ). Moreover, unlike *ORANClaw*, prior work on testing Near-RT RIC [43] requires access to the source code. Finally, the works illustrated in Figures 1ⓒ-ⓓ are only based on specific, handcrafted test cases and do not represent a fuzzing framework. *To the best of our knowledge, ORANClaw is the first fuzzing methodology that targets the gNB via systematic manipulation of E2 interface packets that generalizes to all E2SM services.*

ORANClaw leverages a benign packet trace to capture the reference state machine of E2AP protocol and xApp Service Models. Then it employs type-aware and constraints-aware mutation strategies for different ASN.1 constructs (integers, bitstrings, sequences) to preserve distinct message structures and semantic constraints. For instance, the Key Performance Measurement (KPM) service model defines specific measurement report formats that must adhere to precise encoding rules. Once the mutated packet is forwarded to the RIC (see Figure 1(e)), *ORANClaw* monitors the communication behaviour in the RAN and reconstructs the state machine on-the-fly to identify new behaviour. This is then used as a feedback to guide the mutation strategies in subsequent fuzzing session. In a nutshell, *ORANClaw* not only implements systematic mutations to carefully modify packets with certain structural and semantic features, but it also enables capabilities to find deeply rooted bugs in specific components of the RAN through stateful coverage of the E2AP protocol and xApp Service Models behaviour.

ORANClaw is not tied to a particular implementation of RAN since it focuses on the E2AP interface between xApp and RIC. Moreover, *ORANClaw* can also be easily extended by changing the xApp and collecting its normal communication behaviour. After providing some background information (Section 2), we make the following contributions:

(1) We present *ORANClaw*, a structure-aware and MITM fuzzing framework that systematically explores the stateful behavior of xApp E2SM service models and E2AP protocol (Section 3).
(2) We implement *ORANClaw* and evaluate it with FlexRIC [35], O-RAN SC RIC [21], OpenAirInterface (OAI) [20], ns-3 [18] and commercial VIAVI TeraVM AI RAN Scenario Generator gNB/RIC [41]. Our implementation is open source and available for community research (Section 4).
(3) We discover a total of 71 unique bugs (crashes): 19 bugs in the O-DU and O-CU-UP components of OAI in 5G SA configuration, 28 bugs in FlexRIC and its internal libraries (i.e., iApp/SDK), one bug in O-RAN SC RIC I-Release, 18 bugs in the ns-3 5G NSA network simulator and five bugs in VIAVI TeraVM AI RSG gNB/RIC (Section 5). All bugs are reported to the developers and **eight (8)** CVEs are already assigned.
(4) We compare *ORANClaw* with alternative strategies and we show that *ORANClaw* finds 13 more unique bugs in comparable fuzzing sessions.

After discussing the limitations of our framework and the closely related works (Section 6 and Section 7), we conclude in Section 8.

2 Background

O-RAN Alliance proposed an open radio access network (RAN) architecture (illustrated in Figure 2) that enables interoperability across multi-vendor environments (i.e., Split 7.2x [23]). This design follows a service-based architecture, in which network functions are implemented as independent microservices. These services, in turn, communicate over standardized interfaces, allowing them to be developed, deployed, and scaled independently on general purpose software and hardware. Beyond this disaggregation, O-RAN introduces two intelligent control entities i.e., the *Near-Real-Time RIC* (Near-RT RIC) and the *Non-Real-Time RIC* (Non-RT RIC) which enable automated, data-driven optimization and management of RAN behavior in response to changing network conditions.

Radio Intelligent Controllers (RIC): The *Near-Real-Time RIC (Near-RT RIC)* has latency requirements typically between 10 ms-1 s [19] and manages dynamic RAN functions e.g., load balancing, interference mitigation, and handover decisions. It interacts with the Central Unit (CU) and Distributed Unit (DU) for precise control. The Near-RT RIC hosts modular *xApps* that implement control and optimization algorithms; these xApps can be developed, deployed, and updated independently, allowing operators to adapt RAN behavior to real-time conditions.

The *Non-Real-Time RIC (Non-RT RIC)*, hosted in the SMO framework, operates at a higher abstraction level, handling policy configuration, service orchestration, and long-term optimization. It runs *rApps* that analyze aggregated data and set policies guiding the Near-RT RIC and other RAN elements, enabling AI-driven closed-loop automation via the A1, O1, and O2 interfaces.

E2 Interface: The E2 interface (between Near-RT RIC and RAN) operates through the *E2 Application Protocol* (E2AP), which exposes essential RAN internals to enable intelligent control and monitoring [22]. Through the E2 interface, external applications (xApps) running on the Near-RT RIC can continuously monitor network conditions and execute real-time operations. This is facilitated by *E2*

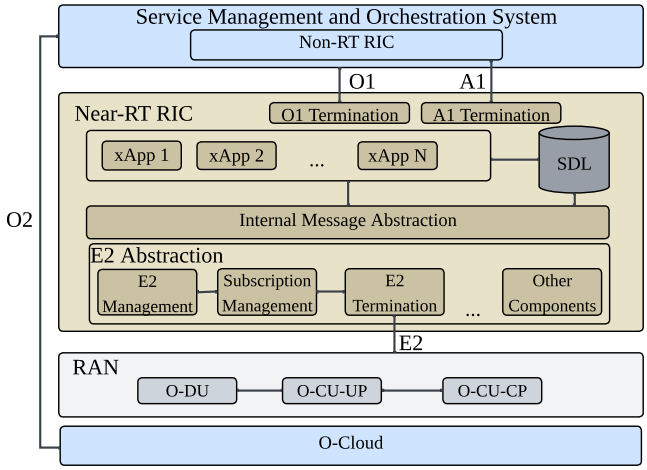


Figure 2: 5G O-RAN Architecture

Service Models (E2SM), which define standardized interaction patterns for specific operations such as reporting performance metrics and issuing control commands (e.g., *KPM, RC, etc.*).

Open Source Implementations: The O-RAN Software Community (O-RAN-SC) [21], in collaboration with the Linux Foundation, provides reference implementations for both Near-RT and Non-RT RIC platforms, containerized with Docker and orchestrated via Kubernetes. In particular, the OSC RIC Platform offers full xApp lifecycle management and E2 interface support; FlexRIC [35] delivers a lightweight, easily deployable alternative enabling multi-vendor E2 integration; and ColO-RAN [32] extends the OSC Near-RT RIC, while μONOS RIC from the SD-RAN project [34] emphasizes software-defined RAN capabilities. Non-RT RIC implementations typically enable policy-based control and AI/ML model inference, supported by orchestration frameworks such as ONAP [16], which integrate policy management and RIC control workflows.

Motivation & Threat Model: The O-RAN architecture (Figure 2) introduces new attack surfaces that require careful security analysis. Existing research has primarily examined threats from malicious RAN nodes (e.g., gNB) targeting the RIC [43] or malicious UEs compromising the core network [5]. However, the open nature of O-RAN's xApp ecosystem presents a different and underexplored threat vector: malicious xApps may directly target RAN components (e.g., O-DU, O-CU) and disrupt the network (Figure 1). This risk is recognized by O-RAN Alliance security documents. WG11 identifies threats such as T-NEAR-RT-01/02 (unauthorized access and malicious behavior in the Near-RT RIC) and T-xAPP-01 (malicious xApps) [24, 25]. These reports stress two principles: the Near-RT RIC must not trust data received from the RAN, and strong access control and xApp isolation are required to prevent one xApp from affecting others. Threats T-E2-01 and T-E2-03 further highlight that malformed or injected E2 messages can degrade or disrupt RAN services. To assess this threat in practice, we design an automated testing, in which adversarial xApps systematically inject malformed E2 messages to evaluate RAN robustness. Intuitively, xApps resemble mobile app–store applications and are often produced by third-party vendors. While easy to deploy, such applications may be untrusted or insufficiently tested. *ORANClaw* investigates whether such untrusted xApps can disrupt gNB availability via the E2 interface when deployed within the Near-RT RIC.

3 Design of *ORANClaw*

We discuss key components of *ORANClaw* (Figure 3) below:

3.1 MiTM Proxy

We employ a man-in-the-middle (MiTM) approach to intercept and forward packets exchanged between an xApp and the near-RT RIC. *ORANClaw* implements the component `SCTP Proxy` (as shown in Figure 3) using an SCTP socket in a client-server architecture. Specifically, the `Server` component listens for incoming connections from the xApp, typically on the default E2AP port (i.e., 36422), while the `Client` component initiates a new connection to the actual near-RT RIC. Since different RIC implementations may use different ports (e.g., srsRAN + FlexRIC uses port 34621), *ORANClaw* allows configuring the outbound connection to match the RIC's expected port. This setup enables *ORANClaw* to transparently relay SCTP messages between the xApp and RIC without requiring changes to either party ***(step 1)*** in Figure 3). Furthermore, it enables recovery from any connection failures by either the xApp or the RIC. Using an SCTP socket closely mirrors the real-world communication setup between the near-RT RIC and E2 nodes in production environments, where E2AP messages are transported over SCTP as mandated by the O-RAN specifications [22]. This makes our MiTM compatible with implementations like FlexRIC, which rely on SCTP for E2 message exchange and proprietary E42 messages that adhere to E2AP. From the perspective of an xApp, this communication is entirely transparent. Since xApps typically interact with the RIC through abstracted messaging interfaces and are unaware of the underlying transport details (e.g., SCTP over E2AP), they operate under the assumption that they are directly exchanging information with the RIC. In reality, even if a Man-in-the-Middle (MiTM) entity is inserted between the RIC and the E2 nodes, the xApp continues to function normally sending messages and receiving responses as if the original communication path remained intact. This transparency is crucial for our MiTM-based analysis and fuzzing setup, as it allows us to manipulate, and re-inject E2AP messages without disrupting the logical flow of information perceived by the xApp. Furthermore, if mutations are not desired, the proxy can seamlessly switch to a passive mode, simply forwarding packets between the RIC and the E2 node to preserve baseline communication behavior with a minimal overhead (detailed in Section 5.2).

3.2 Specification Agnostic

ORANClaw is designed to be specification-agnostic, supporting a wide range of E2 Application Protocol (E2AP) versions and E2 Service Models (E2SMs) defined by the O-RAN Alliance [22]. This flexibility is crucial, as different RAN and RIC implementations adopt varying versions of these specifications depending on vendor requirements or research goals. For instance, a setup involving FlexRIC and OpenAirInterface (OAI) gNB may utilize E2AP v2.03 and E2SM-KPM v2.03, whereas a simulation environment like the ns-O-RAN module in ns-3 may adopt older or divergent specifications such as E2AP v1.01, E2SM-KPM v3.00, or E2SM-RC v1.03.

ORANClaw does not rely on hardcoded and/or static message formats; instead, it supports a dynamic code generation workflow in which the desired versions of E2AP and E2SM ASN.1 specification files are discharged to a code generation tool (e.g., `asn1c` [26]). In particular, the `Encoder` and `Decoder` blocks for messages are generated via such a tool. Thus, even though the ASN.1 specification may evolve in future generations, our design can generate the required encoder and decoder for syntactically correct specification files. As the O-RAN ecosystem continues to evolve with newer versions of its protocol stack, *ORANClaw*'s specification-agnostic design ensures it remains both compatible and adaptable over time. Concretely, whenever a new version of the E2AP or an E2SM (e.g., KPM, RC, etc.) is released, *ORANClaw* can generate the required encoders and decoders using the updated ASN.1 files. However, *ORANClaw*'s internal components including its MiTM proxy and `Fuzzing Engine` (discussed next) remain unaffected. This is because they operate over abstracted message representations (e.g., JSON-formatted JER) that are automatically derived from the active specification. Moreover, the process of generating constraints for valid mutations of messages also remain unchanged, as these constraints are retrieved from the same specification files (see Section 3.3.1).

In summary, *ORANClaw* targets comprehensive fuzzing of both the binary-encoded Service Model payload and the higher-level ProtocolIEs. By targeting both structural and semantic (control-level) layers of the message, *ORANClaw*'s `Fuzzing Engine` broadens the protocol's attack surface and increases the likelihood of uncovering implementation-level vulnerabilities in Service Models as well as in high-level semantics of the E2AP protocol.

Service Model (E2SM) generalizability. Unlike prior tools [43] that rely on manually referencing service models data structures (e.g., KPM, RC), *ORANClaw* operates directly on the E2SM base schema (see label *(a)* in Figure 1), without hardcoding references to protocol fields or headers. This zero-coding, schema-driven design allows *ORANClaw* to support any E2 protocol while preserving the semantics of each E2SM service model. Thus, the design and implementation of *ORANClaw* fuzzing engine is agnostic to vendor-specific extensions or future versions of the E2AP, thus effectively generalizing *ORANClaw* MiTM E2 Proxy.

3.3 *ORANClaw* Fuzzing Engine

E2AP messages are encoded using ASN.1 Packed Encoding Rules (PER), which represent data in a compact, bit-level format optimized for transmission efficiency. Directly manipulating such PER-encoded messages is challenging due to the strict bitwise constraints and complex encoding rules. To facilitate precise and flexible fuzzing without violating the protocol specifications, `Fuzzing Engine` internally translates the PER-encoded messages into a structured JSON representation using JSON Encoding Rules (JER). Upon receiving a raw E2AP message, `Fuzzing Engine` first leverages external ASN.1 parsing tools (i.e., *asn1c*) to decode the PER-encoded data into its JSON (JER) equivalent. This transformation maintains the integrity of all bitwise specifications defined by the E2AP standard, ensuring that any modifications applied at the JSON level remain consistent with the protocol's strict constraints.

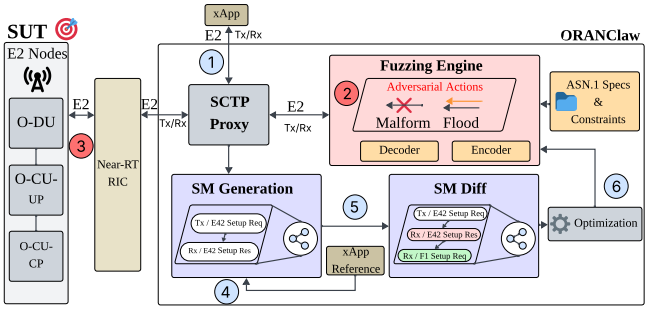


Figure 3: Software Components of *ORANClaw*

After mutations are applied on this JSON representation, the modified message is re-encoded back into PER format before transmission ***(step ❷)***. This decode–mutate–re-encode process allows *ORANClaw* to conduct targeted fuzzing of E2AP fields without breaking protocol compliance or corrupting bit-level details. Furthermore, *ORANClaw* additionally can fuzz and inject invalid mutated values aiming to find issues within the Encoder/Decoder of the RIC's implementation. This is discussed in detail in Section 4.

3.3.1 Type-Aware Fuzzing. To maximize the impact and precision of mutations, *ORANClaw* incorporates a type-aware fuzzing mechanism that leverages the type information embedded in ASN.1 specifications. Rather than relying on blind or random mutations, type-aware fuzzing systematically understands the expected structure, data types, open types and constraints of each field in an E2AP message. This allows the fuzzer to generate inputs that are not only syntactically valid but also semantically plausible, thereby increasing the probability of traversing deeper into the protocol logic and triggering non-trivial behaviors. With this approach, *ORANClaw* respects field-specific properties such as permissible ranges for integers, valid enumeration sets, and string length bounds (as shown in Table 2), ensuring that mutated messages remain within the expected domain of each type. This minimizes early rejections and enables effective exploration of the protocol state space.

Additionally, *ORANClaw*'s `Fuzzing Engine` respects and enforces field-specific constraints derived from ASN.1 specifications during mutation. These constraints include value ranges for integers, size limits for sequences or strings, and required presence conditions for optional fields. By holding such semantic boundaries, constraint-aware fuzzing ensures that generated mutations do not violate basic protocol invariants, allowing the mutated messages to be accepted and parsed by downstream components (see Table 2). This process is agnostic to the version of the Service Model (e.g., KPM v1, KPM v3), and *ORANClaw* automatically extracts these constraints during preprocessing, which starts from the E2SM base layer. After mutation, the payload is re-encoded into its PER (Packed Encoding Rules) format and converted into a hexadecimal string, which replaces the original message content and is subsequently forwarded to the RIC by the `Client` component ***(step ❸)*** in Figure 3).

`Fuzzing Engine` also applies mutations directly to the top-level Protocol Information Elements (*ProtocolIEs*), which encode the high-level control semantics of the E2AP protocol. These are structurally distinct from the embedded Service Model (E2SM) payloads carried

within specific IEs, although both are defined in ASN.1 and encoded using the same encoding rules. These mutations typically target simpler, non-nested fields that define control parameters, message types, and identifiers. While this "top-level" JSON does not include the Service Model payload, it encapsulates essential structural elements such as the *procedureCode* (e.g., *RICSubscriptionRequest*) and its associated *ProtocolIEs*, which are critical for correct message interpretation and protocol flow control. *ORANClaw* employs mutations to these fields, while respecting schema-derived constraints to ensure syntactic validity.

3.3.2 Mutation Strategy and Field Selection. We heuristically guide the selection and mutation of message fields as follows:

- **Constraint-Driven Mutation:** When detailed constraints are available from the loaded JSON schema files, they guide the generation of random yet valid values for fields such as enumerated types, bounded integers, and patterned strings. This constraint-aware approach helps keep mutations within semantically meaningful boundaries, thereby increasing the chances of uncovering subtle protocol-handling bugs while avoiding immediate rejections due to format violations.
- **Field Importance Weighting:** Initially, the fuzzing process assigns random weights for mutating different message types and fields. However, these weights are iteratively refined based on the impact of mutations (e.g., finding additional state transitions). We discuss details in Algorithm 1.
- **Payload Complexity Awareness:** The fuzzing engine also targets nested E2SM payloads, which often encapsulate semantic structures such as measurement definitions, triggers, and subscription filters. For example, *ricActionDefinition* is a specific command or instruction that the RIC sends to a Radio Access Network (RAN) node when subscribing to a specific RAN function, and/or to modify its behavior or retrieve information (i.e., E2SM-KPM, that defines the measurement granularity or measurement object IDs). These payloads tend to be more complex and varied.

3.3.3 State Machine Generation and Diff. To capture the runtime behavior of the RAN control interface under fuzzing conditions, we employ an automated state machine generation pipeline. Each session between the xApp and the RIC is transparently intercepted by our MITM proxy, which records the entire interaction into a `.pcapng` file via `TShark` [42]. Once a session concludes, the proxy automatically invokes the `SM Generation` tool on the captured traffic to generate the protocol state machine ***(step 4)*** in Figure 3). In particular, to generate such a state machine, we provide a few rules to identify packet types and protocols along the interfaces within the gNB (i.e., E1 and F1); and between the gNB and the RIC (i.e., E2) from the traffic. Using these rules, we then derive the relevant message sequences and their abstract states, reflecting a behavioral model for a given session. While these rules involve manual input, it is fairly flexible and easy to derive for someone working on the protocol, as shown in prior works [10, 11]. Indeed, within *ORANClaw*, we only needed six rules for generating the state machine (see Appendix B for details).

Based on the intuition mentioned in the previous paragraph, we track protocol deviations or behavioral anomalies introduced by fuzzing. To this end, we generate a differential model by comparing the current session's state machine against a pre-computed baseline model derived from a benign, non-fuzzed execution of the same xApp (i.e., `xApp Reference` in Figure 3). This is performed by invoking `SM DIFF`, which generates a differential model (in the form of a JSON-formatted diff file) that highlights *Added* and *Removed* states (represented by green and red boxes, respectively, in Figure 3), as well as the transitions between them ***(step 5)***. Such a differential model allows us to determine whether a mutated input had an effect, such as triggering new message exchanges and/or activating previously unobserved protocol behaviour. Finally, irrespective of whether the `Optimization` block is enabled or disabled during fuzzing sessions, *ORANClaw* workflow continues iteratively ***(step 6)***. In the subsequent section, we discuss the utility of this differential model within the fuzzing workflow.

3.3.4 Overall Fuzzer workflow. Algorithm 1 presents the overall workflow for *ORANClaw* fuzzing that orchestrates the interaction between packet interception, mutation, and optimization. The algorithm initializes the fuzzing environment by loading ASN.1 constraints C, setting up the fuzzing engine $\mathcal{F}$, and creating the genetic optimizer population $\mathcal{G}$ with initial population (lines 2-7).

During operation, the `SCTP Proxy` intercepts packets $\mathcal{P}$ from xApps and applies probabilistic fuzzing (exemplified by random() $< P_{\text{fuzz}}$ in Algorithm 1) for a supported packet (i.e., packets at E2 interface). When fuzzing is triggered, the current weight vector $\mathbf{w}i$ guides the location of mutation. In particular, $\mathbf{w}i$ is a vector of weights $[w_{\text{SM}}, w_{\text{json}}, w_{\text{both}}]$ where w_{SM} and w_{json} capture the probabilities of mutating either the service model or the top-level JSON representation (e.g., *RIC-SubscriptionRequest*), respectively, whereas w_{both} denotes the probability to mutate both the service model and top-level JSON. Intuitively, $\mathbf{w}i$ keeps track of the *importance* of the different parts of messages (i.e., the service model, the top-level JSON representation or both) from the viewpoint of fuzzing. In other words, if mutating service models predominantly leads to functional anomalies (e.g., crashes) as compared to mutating other parts of the message, then the genetic algorithm embodied within *ORANClaw* naturally assigns more weights to w_{SM}. While this idea could be extended to any number of message fields trivially, the implementation of *ORANClaw* sticks to the current $\mathbf{w}i$ for the sake of simplicity. Once the location of mutation (i.e., service model, top-level JSON or both) is chosen in line with $\mathbf{w}i$, the fuzzing engine selects $\mathcal{F}_{\text{fields}}$ for up to $M \leq M_{\text{max}}$ mutations. These fields are then leveraged to apply type-aware mutations $\mathcal{F}.\texttt{mutate}(\mathcal{P}, \mathcal{F}_{\text{fields}}, C)$ to produce the mutated packet $\mathcal{P}'$.

The algorithm supports optional packet duplication when enabled and random() $< P_{\text{dup}}$, sending $D \leq D_{\text{max}}$ copies of $\mathcal{P}'$ to stress-test the target RIC's handling of repeated messages. To enable adaptive mutations, we record mutation traces and update fuzzing statistics (e.g., found crashes or additional state transitions) for fitness evaluation when $\mathcal{P}' \neq \mathcal{P}$. In particular, session boundaries are detected through timeouts or natural termination points, at which the system extracts protocol states $\mathcal{S}$ from logs and computes the cost function $c(\mathbf{w}_i)$ based on observed protocol transitions and applied mutations (see lines 34-36 in Algorithm 1). Finally, the function $\mathcal{G}.\texttt{update}(c(\mathbf{w}_i))$ evolves the population and a new fuzzing session starts (line 13 in Algorithm 1). This continuous feedback

loop enables adaptive fuzzing strategy refinement across sessions while maintaining comprehensive logging through session recording and differential model generation for subsequent analysis.

Algorithm 1 Main steps of *ORANClaw*

procedure ORANCLAW_FUZZING
  $\mathcal{C} \leftarrow$ Load `Constraints`
  $\mathcal{F} \leftarrow$ Initialize `Fuzzing Engine`
  $\mathcal{G} \leftarrow$ Initialize `GeneticFuzzerOptimizer`
  Start MiTM proxy and capture interface
  ▷ Initialize population with uniform Dirichlet
  $\mathbb{W} \leftarrow \{\mathbf{w}_i \sim \text{Dirichlet}(\mathbf{1}) : i = 1, \ldots, N\}$
  while proxy is running **do**
    ▷ Run Fuzzing session for each individual
    for each $\mathbf{w}_i \in \mathbb{W}$ **do**
      $c(\mathbf{w}_i) \leftarrow$ `ORANClaw_Session`$(\mathbf{w}_i)$
    ▷ Update population
    $\mathbb{W} \leftarrow \mathcal{G}$.`update`$([\mathbf{w}_i, c(\mathbf{w}_i)]_{1\ldots N})$

procedure ORANCLAW_SESSION($\mathbf{w}_i$)
  while true **do**
    Receive: $\mathcal{P} \leftarrow$ packet from xApp
    if random() $< P_{\text{fuzz}}$ **and** $\mathcal{P}$ is supported **then**
      $\mathbf{w}_i \leftarrow$ current weights from $\mathcal{G}$
      $M \leftarrow$ #mutations sampled s.t. $M \leq M_{\max}$
      $\mathcal{F}_{\text{fields}} \leftarrow$ select fields to mutate using $\mathbf{w}_i$
      $\mathcal{P}' \leftarrow \mathcal{F}$.`mutate`$(\mathcal{P}, \mathcal{F}_{\text{fields}}, \mathcal{C})$
      if $\mathcal{P}' \neq \mathcal{P}$ **then**
        Record mutation trace for fitness
        Update fuzzing stats
      if duplication enabled **and** random() $< P_{\text{dup}}$ **then**
        $D \leftarrow$ random count $\leq D_{\max}$
        Send D copies of $\mathcal{P}'$ to target ▷ Duplicate
      else
        Send $\mathcal{P}'$ to target ▷ Send fuzzed packet once
    else
      Send original $\mathcal{P}$ to target ▷ No fuzzing applied
    if session ends or timeout **then**
      Extract protocol states $\mathcal{S}$ from logs
      Save session, generate state machine diff
      Compute cost $c(\mathbf{w}_i)$ ▷ Cost Function
      return $c(\mathbf{w}_i)$

3.3.5 Optimization Objective (Cost Function). The state machine embodied in *ORANClaw* (see "SM Generation" in Figure 3) captures states (message types and direction) that directly correspond to attacker-controlled payloads over E2 interface. Intuitively, *ORANClaw* aims to diversify mutations on such attacker controlled payloads. This is to discover unexpected or erroneous behaviour. As a result, the optimizer aims to maximize the transitions in the state machine. Concretely, we design the cost function as follows:

$$c = \beta \cdot \frac{T}{\tau} + \mu \cdot M \tag{1}$$

where:

- T: Number of observed transitions during a fuzzing session.
- τ: Normalization constant representing the number of transitions in the reference state machine model (i.e., the behavioral model in the absence of fuzzing).
- M: Total number of mutations applied to achieve the observed state changes and transitions.
- β, μ: Hyperparameters weighting the relative importance of transition exploration and penalty for mutation of fields, respectively.

Intuitively, the term $\beta \cdot \frac{T}{\tau}$ elevates the value of the cost function when crashes or functional anomalies occur, as such events typically increase the number of new messages exchanged between the RIC and the gNB, leading to several new transitions in the state machines. μ is typically a small value (set to 0.1 in *ORANClaw*), which ensures that the increased number of mutations does not have a significant impact on the value of the cost function. This ensures that the optimizer does not disproportionately favor populations with higher mutations.

4 Implementation and Experimental Setup

The *ORANClaw* implementation is shown in Figure 3. Our implementation is built around a custom MitM `SCTP Proxy` that intercepts SCTP traffic between xApps and the RIC, which then forwards communication to E2 Nodes. This architecture enables manipulation and analysis of E2AP protocol messages during runtime.

ORANClaw is developed using Python 3.12 (1261 LoC) and runs on a machine with an AMD Ryzen 7 5800H CPU @ 4.46GHz, with 64GB of RAM for a single session. Additionally, concurrent fuzzing sessions used in the experiments for **RQ1** (discussed in Section 5.1), were run on a machine with an AMD Ryzen Threadripper PRO 7985WX CPU @5.37GHz and 512GB of RAM.

MitM `SCTP Proxy`: It implements a client-server architecture using Python and integrates the following components:

- *Core Communication Layer:* `pysctp` [30] is used for low-level SCTP socket communication, establishing bi-directional connections between xApps and the RIC platform.
- *Traffic Analysis:* We leverage `TShark` to capture and log intercepted packets, generating *pcapng* files for post-fuzzing session analysis.
- *Message Processing: ORANClaw* processes E2AP messages using two complementary approaches: the Objective Systems ASN1C [26] compiler generates C++ runtime libraries from ASN.1 specification files, while *asn1tools* provides further Python-based encoding/decoding capabilities on the JER representation.

Bidirectional Message Conversion: As discussed in Section 3.3, *ORANClaw* implements a bi-directional conversion enabling PER encoding/decoding allowing proper communication between the xApp and the RIC:

(1) `Decoder:` This component (127 LOC) receives hexadecimal streams representing PER-encoded E2AP messages and converts them to JER (JSON Encoding Rules) format. The decoder extracts protocol data from the binary PER representation and generates human-readable JSON structures (JER encoding) for mutation.
(2) `Encoder:` This component (156 LOC) processes the JER-compliant E2AP message structures and converts them back to PER format (possibly after mutation). This enables modified messages to be transmitted properly.

Both the `Encoder` and `Decoder` components are custom-built wrappers around C++ functions, which are automatically generated by the *asn1c* compiler. Moreover, such a *decode-modify-encode* workflow introduces minimal overhead: it is within the typical RIC–xApp communication interval ($1ms$ to $1s$) [19].

A challenge arises when processing certain E2AP message types (i.e., *RICSubscriptionRequest*). These messages contain Service Model definitions (e.g., E2SM KPM and E2SM RC) within fields such as *ricActionDefinition*. The standard *asn1c* compiler cannot decode these complex PER payloads that define Service Model semantics. To overcome this limitation, we leverage the *asn1tools* Python library specifically for Service Model encoding and decoding. This ensures consistency by using the same ASN.1 specification files.

Radio Intelligent Controller: Due to the complexity and overhead of deploying the full O-RAN Software Community (SC) Near-RT RIC, which involves orchestrating multiple containers (e.g., E2 Termination, Submgr, Rtmgr, and more), we adopt FlexRIC for most of our experiments. FlexRIC is a modular alternative that implements the essential E2 interface components with minimal dependencies, making it significantly easier to deploy, and integrate for testing. This simplicity is particularly advantageous in our context, where deep packet inspection, fuzzing instrumentation, and packet mutations are required. By using FlexRIC, we adhere to the E2AP protocol while avoiding the deployment complexity of O-RAN-SC RIC. To validate the generalizability of *ORANClaw* beyond FlexRIC, we also evaluate it with the full O-RAN SC RIC deployment, as discussed in Section 5.5.

System under test (SUT): *ORANClaw* primarily evaluated two open-source RAN implementations:

The first deployment leverages the OpenAirInterface (OAI) gNB stack [28] *(comnit: ff58b5e1)*, integrated with FlexRIC *(comnit: beabdd07)* as the near-Real-Time RIC platform to support xApp deployment. The setup is completed with Open5GS [27] as the core network (5GC), forming a full end-to-end 5G system. The OAI stack used in this setup use the disaggregated gNB architecture aligned with 3GPP and O-RAN specifications. Specifically, it includes the O-DU, O-CU-UP, and O-CU-CP functional components, instantiated through the `nr-softmodem` application. These components communicate via standardized interfaces F1 (between O-DU and O-CU) and E1 (between O-CU-CP and O-CU-UP) supporting a realistic split architecture. For testing, two simulated UEs are deployed within the environment (i.e., *nr-uesoftmodem*), interacting with the gNB and core network. The entire setup is deployed within a containerized environment to facilitate the testing and reproducibility.

We also deploy a 5G NSA setup via the *ns3-mmwave-oran* module (`scenario-zero.cc`) [37] *(commit: 7936f62f)*. The experimental setup has three simulated gNBs (`MmWave Nodes`), one eNB (`Lte Nodes`), and five UEs with random-walk mobility. FlexRIC runs the xApps, and uses the Evolved Packet Core (EPC) as its core network. The gNB, instantiated via *MmWaveEnbNetDevice*, implements a monolithic stack without real CU/DU or control/user-plane separation. While ns-3 is an academic tool rather than production software, it enables systematic vulnerability analysis of O-RAN's attack surface, informing real-world security considerations.. Although not a fully disaggregated deployment, we argue that this abstraction is sufficient for evaluating most of the xApp-driven control logic.

Testing commercial solutions: To demonstrate *ORANClaw*'s applicability beyond academic and open source implementations, we evaluated TeraVM VIAVI RAN Simulator Generator (v2.4-40470-43580c20) [41], a commercial O-RAN testing solution in production environments, uncovering distinct vulnerabilities in the proprietary gNB and RIC (Table 3). Most critically, **VulnVIAVI-02** triggered complete system failure, simultaneously crashing the GUI, gNB, and RIC while rendering the Docker container unhealthy, a full denial-of-service in production. These findings validate that structure-aware mutations can identify real-world vulnerabilities in commercial O-RAN systems.

5 Results

We answer the following research questions (RQs) to evaluate the effectiveness and efficiency of our *ORANClaw*:

5.1 RQ1: How effective is *ORANClaw* in finding bugs?

In Table 3, we showcase the effectiveness of *ORANClaw* in finding bugs in open source implementation (Implementation column) of ORAN components such as near-RT RIC (FlexRIC), O-DU, O-CU (OAI) and monolithic gNB and eNB (ns-3). To this end, each target implementation is evaluated against *ORANClaw* five times for a period of 24 hours with the same optimization parameters (i.e., β, μ) to keep consistency during the evaluation. The results are classified by their *Vulnerability* type, *General Cause*, affected ORAN *Component* and *Threat*.

Furthermore, to highlight *ORANClaw* effectiveness over time, we evaluate different configurations of *ORANClaw* (i.e., Optimization, Mutation only, Duplication only, etc.) by running five fuzzing sessions for each configuration and we show the average of the cumulative unique bugs in Figures 4 and 5. Additionally, in the OAI setup we performed a sensitivity analysis of the cost function in Eq. 1 by increasing the population size (i.e., $N = 100$), while keeping the rest of the experiments at $N = 6$. We also reduced the mutation hyperparameter P_{mut} to 5%, compared to the 30% used in the other experiments. Finally, we evaluated an additional xApp test case in FlexRIC that combines the KPM and RC service models to demonstrate the generalizability of E2SM.

Overall, *ORANClaw* and its variations were able to identify a total of **71** vulnerabilities: **19** issues in the O-DU, O-CU-UP components of OAI in 5G SA configuration, **28** bugs in FlexRIC and its internal libraries (i.e., *iApp/SDK*), 1 bug in O-RAN SC I release and **18** bugs in the ns-3 5G NSA network simulator. Additionally, uncovering five vulnerabilities in commercial VIAVI RAN. Interestingly, *ORANClaw* also uncovered a DoS issue in OAI related to the E1 interface (*VulnOAI-11*), which is not directly connected to xApps nor the near-RT RIC. Such vulnerability is not triggered by a malformed message, but rather from the O-CU-UP's inability to recover properly after a previous fuzzing session that led to a crash. Such results demonstrate *ORANClaw*'s effectiveness of finding bugs in several ORAN components that are indirectly connected to xApps (gNB, CU, DU), highlighting the importance of securing the RAN against malicious xApps. We note that 68 out of 71 cases (Table 3)

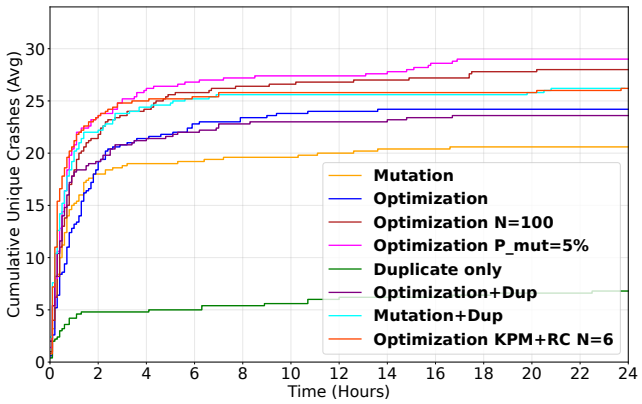


Figure 4: Average of Cumulative Bugs in OAI RAN.

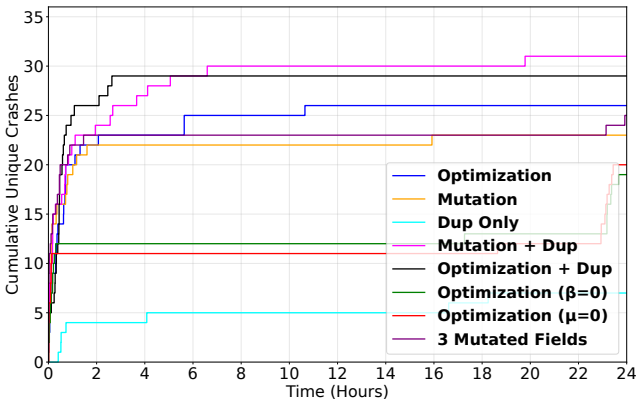


Figure 6: Unique Crashes in OAI w.r.t Time with different fuzzing strategies

do not require continuous adversarial presence; a single malformed message triggers the fault. In these cases, the system fails to recover without manual restart (persistent DoS). All issues were reported to the developers of the target implementation, with many bugs already having their corresponding CVEs assigned.

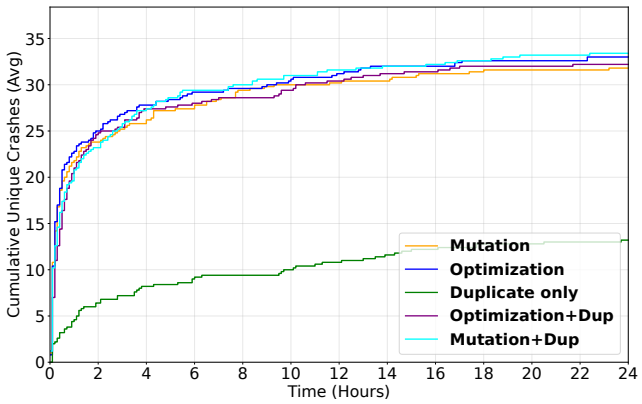


Figure 5: Average of Cumulative Bugs in ns-3 RAN.

5.2 RQ2: How do components of *ORANClaw* contribute to its overall effectiveness?

To investigate the contribution of our fuzzing strategies, we evaluate variants of *ORANClaw* solely employing mutation, duplication, optimization and a combination of such. Furthermore, for the optimization, parameters of the cost function such as β and μ are disabled to study their impact throughout the fuzzing process. The results of our evaluation for different 5G stacks (OAI and ns-3) are highlighted in Figures 6 and 7.

We further study the effect of state-machine completeness on *ORANClaw*'s performance (Figure 8). Incomplete state-machine traces (green, olive, dark red) sometimes match or outperform the fully reconstructed trace (blue) in terms of unique crash discovery (Figure 7). This suggests that partial traces can encourage exploration of alternative or out-of-order transitions that the complete trace constraints, enabling the fuzzer to reach different states. Thus, completeness is not always strictly beneficial in some cases, a less prescriptive trace can increase coverage of unexpected behaviours.

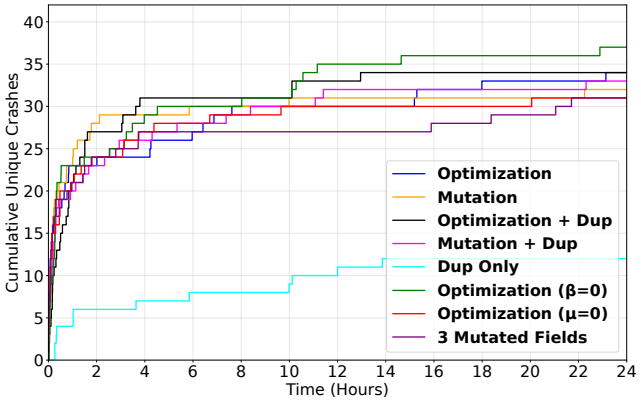


Figure 7: Unique Crashes in ns-3 w.r.t Time with different fuzzing strategies

In summary, variants *optimization + duplication* and *mutation + duplication* consistently find most bugs across the evaluated RAN implementations. Moreover, *mutation + duplication* often matches or even outperforms *optimization + duplication*. This highlights that semantic and constraint-aware mutations are the critical factors at exploiting protocol fields to reveal bugs. However, the variant with the number of transitions disabled ($\beta = 0$), performs better with ns-3 since majority of ns-3's vulnerabilities are located in the early states of the communication (xApp registration). This is because the full optimization variant incentivizes exploration of bugs in further protocol transitions, making it less effective to find bugs early in the state machine. Nonetheless, we argue that leveraging protocol transitions as cost function is appropriate since it targets a broader use case for RAN service and can find most of the bugs in the initial hours of the fuzzing session for both evaluated RAN implementations (see Figures 6 and 7).

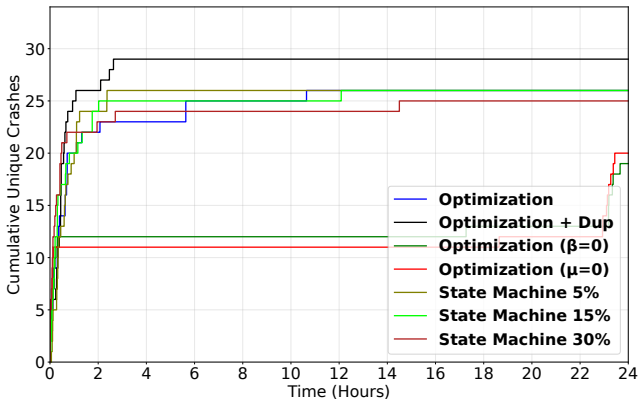


Figure 8: Unique Crashes in OAI w.r.t Time with state machine completeness

5.3 RQ3: How efficient is *ORANClaw*?

In order to fuzz the RAN, *ORANClaw* intercepts E2AP messages via its internal bridge and performs packet processing which includes encoding and decoding of ASN.1 messages. Therefore, to measure the overhead of *ORANClaw* when intercepting through the E2 interface, we measure the time it takes for every packet to enter and leave the E2 bridge over a period of 24 hours. We perform this experiment by enabling and disabling packet processing and showcase the overhead results in the boxpot of Figure 9.

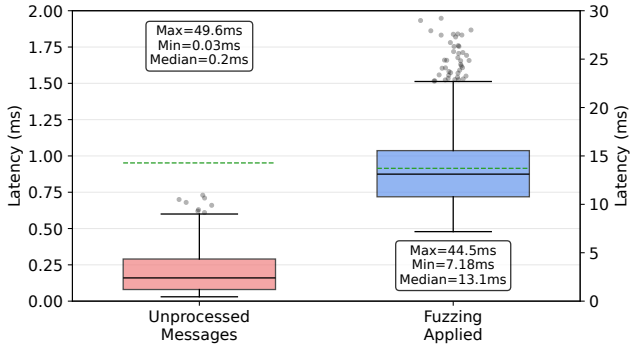


Figure 9: Latency of *ORANClaw*

The results show that *ORANClaw* adds around 13*ms* of overhead to the E2 interface when processing packets. This delay is low enough to satisfy the timing requirements (10*ms* to 1*s*) of the E2 interface [19]. This demonstrates the feasibility in employing *ORANClaw* to time-critical RAN interfaces.

5.4 RQ4: How *ORANClaw* compares w.r.t state of the art testing tools?

Due to the limited tooling that can mutate ASN.1 fields in the context of ORAN, we evaluate the competitiveness of *ORANClaw* by modifying it to match the fuzzing behaviour of other tools e.g., Boofuzz [29]. This is done by conducting a fuzzing session in the OAI setup with basic random JSON mutations without considering ASN.1 constraints (structure-unaware). Our results show that 87.9% of the messages (904/1029) leads to basic encoding/decoding issues due to mutations, often breaking ASN.1 parsing constraints. Moreover, due to such ASN.1 parsing errors, the session only lasts for six hours as no differences are observed beyond such a timeframe.

Overall, only eight crashes are triggered within the internal message parser of the RIC. Moreover, two other crashes are triggered within the gNB due to failures in the RIC. In contrast, *ORANClaw* finds ∼23 bugs within the same six hours time window (see Figure 5). This shows the effectiveness of *ORANClaw* and its type-aware, constraint-based mutations.

Additionally, we evaluated the 5G fuzzer available in U-Fuzz [36]. However, U-Fuzz is unable to fuzz the E2AP ASN.1 SCTP payloads sent from xApp to the RIC. This is because U-Fuzz relies on Wireshark to decode ASN.1 messages, which, in turn does not support decoding the same version of the ASN.1 E2AP protocol employed in FlexRIC. Therefore, we were unable to evaluate U-Fuzz further due to its limitation in parsing different versions of ASN.1 protocols. We note that *ORANClaw* does not face such an issue and it is robust to different versions of ASN.1. This is because *ORANClaw* leverages compiled code from arbitrary ASN.1 specifications for message encoding and decoding.

5.5 *ORANClaw* Generalizability

ORANClaw fuzzes E2AP messages over E2AP interface (between Near-RT RIC and E2 RAN nodes), testing both RIC implementations (via E2 Termination) and RAN nodes (via gNB) by simply changing the direction of the fuzzer. While internal RIC communications between xApps and other RIC components use implementation-specific protocols (e.g., gRPC and RMR in O-RAN SC), these are outside the scope of E2AP fuzzing. *ORANClaw*'s MiTM flexibility allows positioning it, thus intercepting and fuzzing standardized E2AP traffic regardless of the internal RIC architecture, making it agnostic to whether the RIC is monolithic or microservice-based.

To demonstrate that *ORANClaw* does not require fundamental changes to its core fuzzing engine or overall architecture, we evaluated it against the O-RAN SC RIC platform [21] together with the srsRAN gNB [39], demonstrating its deployability across multiple RIC and base station implementations. For this evaluation, we used the O-RAN SC I-Release and the single-binary gNB, following the official tutorial [38]. By simply placing *ORANClaw* in between the E2 Termination and the gNB (see Figure 2), we deployed two virtual machines (VMs): one running the O-RAN SC RIC and another running both the gNB and *ORANClaw*, which we controlled via SSH. We redirected *ORANClaw*'s `Fuzzing Engine` to target the O-RAN SC RIC instead of the original target (i.e., gNB). After 11 hours of fuzzing, we discovered one vulnerability in the E2 Termination component (**VulnOSCRIC-01**, as shown in Table 3). This demonstrates the generalizability of the MiTM fuzzing engine embodied in *ORANClaw* across multiple ORAN deployment scenarios.

5.6 Example Vulnerabilities

5.6.1 Mutating E2SM-KPM. When deploying *ORANClaw* alongside the OAI implementation, we evaluated the default FlexRIC xApp using the KPM Service Model (i.e., xapp_kmp_moni) that retrieves downlink and uplink metrics from UEs at specified intervals

(granularityPeriod). As demonstrated in **VulnOAI-06** (see Table 3), *ORANClaw* successfully decodes the KPM payload's Protocol Information Elements (IEs) and exploits the measurement reporting structure. Specifically, *ORANClaw* mutates the KPM payload by modifying the *measName* field with a malformed structure, while maintaining a valid *measID* format (see Listing 1). This targeted manipulation of E2SM-KPM IEs triggers a parsing error in the O-DU that leads to an assertion failure and crash.

```
        "measInfoList": [
          {
- "measType": ("measName", "DRB.Pdcp...DL"),
+ "measType": ("measID", 5354),
            "labelInfoList": [{"measLabel": {"noLabel":
                "true"}}]
          },
          {
- "measType": ("measName", "DRB.Pdcp...UL"),
+ "measType": ("measID", 5354),
            "labelInfoList": [{"measLabel": {"noLabel":
                "true"}}] }, ...
```

Listing 1: Structure Mutation by ORANClaw in E2SM-KPM

5.6.2 Single-Field Mutation & LTE eNB. *ORANClaw* also mutates single fields (e.g., single byte) within ASN.1 structures. For instance, when mutating the *Criticality* field from *id-RICsubscriptionDetails* ProtocolIE (i.e., "id:30"), a vulnerability is triggered in the eNB from ns-3 Simulator. It processes the mutated RICSubscriptionRequest but its *e2sim* module fails during ASN.1 APER encoding, triggering an assertion error in the channel validation logic (`m_channel`), ultimately crashing the entire eNB with a SIGIOT signal. This implementation vulnerability is noted as **VulnNS-06** in Table 3. In general, this security flaw shows that malformed messages via the E2 interface may propagate through and even disrupt the eNB in a 5G NSA configuration.

```
        {
          "id": 30,
- "criticality": "reject",
+ "criticality": "notify",
          "value": {
            "ricEventTriggerDefinition": "0063",
            "ricAction-ToBeSetup-List": [
```

Listing 2: Criticality Field Mutation in RIC Subscription Request

6 Discussion & Limitations

O-RAN Testing: Our field selection strategy is primarily based on its location (for example, at the service-model level or among top-level JSON fields) rather than more nuanced and fine-grained selection strategy for each individual field. Nonetheless, the open platform and architecture of *ORANClaw* allow design and implementation of more sophisticated selection and optimization strategies for message mutations.

Because O-RAN does not mandate IPsec [22], our threat model includes deployments without IPsec (e.g., OpenAirInterface, srsRAN, O-RAN SC) e.g., insider threats from compromised or buggy xApps with valid credentials, and misconfigurations. Importantly, ORANClaw's E2AP vulnerabilities remain exploitable by authenticated endpoints regardless of IPsec, since malicious or buggy xApps can send malformed messages over encrypted channels, which secure transport but not message semantics. Furthermore, despite operator vetting, our evaluation of a commercial VIAVI gNB (see Table 3) shows that deployed xApps/RICs can still introduce vulnerabilities in commercial O-RAN systems, confirming xApps as practical attack vectors via malicious code or bugs.

Code/branch coverage: *ORANClaw* is designed as a black-box testing tool. While this approach enables testing of proprietary implementations, it may result in lower code coverage. However, *ORANClaw* can be easily extended via standard code instrumentation tools used in greybox fuzzer family [33, 44].

Testing Completeness: *ORANClaw* only focuses on mutating the messages captured over the E2 interface. Hence, it does not find vulnerabilities that are not caused by these messages. Nonetheless, the systematic mutation strategies implemented over ASN.1 messages and the optimization process based on the differential model are generic. We aim to adapt *ORANClaw* for fuzzing other ORAN interfaces in the future.

Compiler-generated code: *ORANClaw* generates the necessary *C++* components (`Encoder, Decoder`) by discharging ASN.1 specification files to *asn1c* compiler. While such an architecture makes *ORANClaw* robust to future changes in ASN.1 specification, it also creates a dependency on the correctness and completeness of the compiler. In our work, we preferred a compiled version of the Message Encoder and Decoder, as such is more reliable and extensible over manual and hard-coded rules to generate messages.

Manual efforts: *ORANClaw* requires a benign xApp trace file (e.g., *.pcapng*) to establish a baseline for the `SM DIFF module` (see Figure 3) when constructing state machines. Additionally, it also requires a few rules to identify the packet types and protocols. Nonetheless, these rules are easy to be specified with minimal knowledge of the target protocol (E2AP) and wireshark [11].

7 Related Work

Table 1 illustrates the comparison between *ORANClaw* and other related works. In the following, we discuss and position *ORANClaw* with respect to prior works.

Greybox Fuzzing: Automated testing within the RIC environment is challenging due to the nature of the messages defined in ASN.1 structures and the dynamic nature of E2AP protocol. Specifically, existing general fuzzing tools [33, 44] involve source-code instrumentation and they do not focus on stateful protocols. While stateful and generic greybox fuzzing have made some progress in the last few years [4, 31], they are not directly applicable within the RIC environment (see columns *Type* and *Attack Model Applicability* in Table 1). For example, they are based on mutating static seed input (packet sequences), whereas AFLNET specifically requires a response for every request. Mutating static seeds are not appropriate for fuzzing dynamic protocols, as the mutated packets are often invalid and rejected by the target node. In contrast, the mutations embodied in *ORANClaw* are aware of the ASN.1 structure and it

also involves man-in-the-middle fuzzing (at E2) to deal with the complexity of dynamic protocols.

Work	Stateful Execution	Type	Grammar/Structure Awareness	ASN.1 Support	Attack Model Applicability
AFL [44]	○	Greybox	○	○	NA
AFLNET [31]	◗	Greybox	○	○	NA
LibFuzzer [33]	○	Greybox	○	○	NA
Frizzer [17]	○	Blackbox	◗	○	NA
BooFuzz [29]	○	Blackbox	◗	○	Protocol-specific
Peach [7]	○	Generation-based	◗	○	Protocol-specific
U-Fuzz [36]	●	Blackbox	●	○	Protocol-specific
5Ghoul [11]	●	Blackbox	●	○	UE focused
ORANALYST [43]	○	Whitebox	●	◗	RIC focused
Berserker [2]	○	Whitebox	●	●	UE/RAN focused
RANsacked [5]	○	Whitebox	●	●	Core Network focused
ORANClaw	●	Blackbox	●	●	RAN focused

Table 1: *ORANClaw* vs other testing tools. ●: Full consideration, ○: Exclusion, ◗: Partial consideration.

Protocol Fuzzing: In contrast to *ORANClaw*, BooFuzz [29] and Peach [7] require manual specification of message grammars (see column *Grammar* in Table 1). Frizzer [17] dynamically instruments the target using the Frida toolkit. However, due to the dynamic instrumentation, it is inefficient and involves reverse engineering to indicate target function address. On the contrary, *ORANClaw* does not involve any reverse engineering or instrumentation. Moreover, it automatically constructs the state machine from the communication captured in E2 interface. Hence, it is suitable to effectively and efficiently fuzz both open- and closed-sourced targets.

Limited ASN.1 support: Efforts have been made to develop tools that aim to generate inputs based on ASN.1 structures and grammar. For instance, prior works targeting the RIC [43] lacks statefulness and it also involves manual and direct implementation of mutation rules on certain ASN.1 structures. As a result, such an implementation is not easily extensible for future evolutions of ASN.1 specification (see column *ASN.1 Support* in Table 1). Furthermore, libprotobuf-mutator [12] enables the generation of ASN.1 messages using protobuf definitions, but it relies on a manual translation from ASN.1 to protobuf. This manual process is particularly challenging when testing O-RAN protocols such as E2AP and E2SM [22], which involve hundreds of complex structures across various types.

RAN-focused fuzzers: Although Berserker [2] can generate valid ASN.1 messages, its RRC focus and stateless execution limit testing scope (see column *Stateful Execution* in Table 1). Other approaches such as RANsacked [5] and prior core/RAN work [8] attack the network only via a malicious UE, while U-Fuzz [36] targets the UE from a rogue base station. Likewise, some efforts target only the near-RT RIC via a malicious base station [43] or the A1 interface [40], narrowing their applicability. By contrast, *ORANClaw* exercises multiple RAN components over the E2 interface without deploying malicious devices, producing practical findings since arbitrary malicious xApps can inject E2 attacks. Prior xApp-triggered testcases [13] lack flexible, automated discovery as O-RAN and ASN.1 evolve; *ORANClaw* addresses this by using existing xApps and an ASN.1-agnostic, automated testing approach.

O-RAN Attacks and Defense: Several works uncover vulnerabilities specific to ORAN and its implementations [9, 15], while other works focus on defense [1, 3, 6, 14]. Instead of competing, *ORANClaw* complements these works by providing a framework that allows packet analysis regardless of the ASN.1 specification. Consequently, future research can build upon *ORANClaw*.

8 Conclusion

In this paper, we comprehensively investigate the attack surface that targets the RAN via third-party and potentially untrusted xApps. To this end, we propose and design *ORANClaw*, which acts as a man-in-the-middle (MITM) fuzzer at the E2 interface between the xApp and the RIC. The MITM fuzzing capabilities embodied in *ORANClaw* allows us to impersonate the behaviors of arbitrary vulnerable and malicious xApps via systematic packet mutations and duplications. An appealing feature of *ORANClaw* is its capability to perform type-aware, constrained mutations that stress test the behavior of the RAN (RIC and the base station) without being rejected early by the packet decoder at the RAN. Besides, *ORANClaw* employs optimizations of mutation strategies via a differential state model to find deeply rooted bugs in the E2AP protocol. The effectiveness of *ORANClaw* is demonstrated by finding 71 unique bugs across widely used RIC (FlexRIC, O-RAN SC RIC), gNB implementations (OAI and ns3-simulator) and commercial solutions such as VIAVI TeraVM AI RSG. While *ORANClaw* is currently applied for fuzzing the RAN, we believe its core fuzzing engine can be extended for fuzzing different segments of the ORAN architecture. This is because *ORANClaw*'s fuzzing engine is based on ASN.1 specification. We hope that the design of *ORANClaw* opens the possibility of security testing for the evolving ORAN architecture at scale. For reproducing our results and advancing research in the area, we have made our code and data available at: https://anonymous.4open.science/r/ORANClaw-E2-MitM-Fuzzing-AFFC/

9 Research Ethics

All experiments were conducted in isolated environments using O-RAN software with no impact on live networks or user data. Discovered vulnerabilities were reported through coordinated disclosure to strengthen O-RAN security before widespread deployment.

Acknowledgement: We thank the anonymous reviewers for their insightful comments on our paper. This research is supported by National Research Foundation, Singapore, under its National Satellite of Excellence Programme "Design Science and Technology for Secure Critical Infrastructure: Phase II" (Award No: NRF-NCR25-NSOE05-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the respective funding agency.

References

[1] Zakaria Abou El Houda, Hajar Moudoud, and Bouziane Brik. 2024. Federated deep reinforcement learning for efficient jamming attack mitigation in O-RAN. *IEEE Transactions on Vehicular Technology* 73, 7 (2024), 9334–9343.
[2] Researchers at KTH. 2021. Berserker: ASN.1-Aware Telecom Protocol Fuzzing. In *Telecom Security Conference.* focus on NGAP, F1AP, XnAP.
[3] Tolga O. Atalay et al. 2023. Securing 5G openran with a scalable authorization framework for xapps. In *IEEE INFOCOM*. IEEE, 1–10.
[4] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *USENIX Security Symposium*. 3255–3272.
[5] Nathaniel Bennett et al. 2024. Ransacked: A domain-informed approach for fuzzing lte and 5g ran-core interfaces. In *CCS*. 2027–2041.
[6] Shin-Ming Cheng, Bing-Kai Hong, and Cheng-Feng Hung. 2022. Attack Detection and Mitigation in MEC-Enabled 5G Networks for AIoT. *IEEE Internet of Things Magazine* 5, 3 (2022), 76–81.

[7] Peach Tech Community. 2021. Peach Fuzzer Community Edition. https://peachtech.gitlab.io/peach-fuzzer-community/. supports both generation- and mutation-based fuzzing via Peach Pit grammar files.
[8] Dimitri Dessources et al. 2024. In *MILCOM*. 129–134.
[9] Yared Abera Ergu et al. 2024. Efficient adversarial attacks against DRL-based resource allocation in intelligent O-RAN for V2X. *IEEE Transactions on Vehicular Technology* (2024).
[10] Matheus E. Garbelini et al. 2022. BrakTooth: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing. In *USENIX Security Symposium*. USENIX Association, 1025–1042.
[11] Matheus E. Garbelini et al. 2025. 5GHOUL: Unleashing Chaos on 5G Edge Devices via Stateful Multi-layer Fuzzing. *IEEE TDSC* (2025).
[12] Google. 2023. libprotobuf-mutator. https://github.com/google/libprotobuf-mutator. Accessed: 2025-08-02.
[13] Cheng-Feng Hung et al. 2024. Security threats to xApps access control and E2 interface in O-RAN. *IEEE Open Journal of the Communications Society* 5 (2024), 1197–1203.
[14] Cheng-Feng Hung, Chi-Heng Tseng, and Shin-Ming Cheng. 2025. Anomaly Detection for Mitigating xApp and E2 Interface Threats in O-RAN Near-RT RIC. *IEEE Open Journal of the Communications Society* 6 (2025), 1682–1694.
[15] Felix Klement et al. [n. d.]. Endless Subscriptions: Open RAN is Open to RIC E2 Subscription Denial of Service Attacks. ([n. d.]).
[16] Hoejoo Lee et al. 2020. Hosting ai/ml workflows on o-ran ric platform. In *IEEE Globecom Workshops*. IEEE, 1–6.
[17] Dennis Mantz and the Frizzer community. 2025. Frizzer: A Coverage-Guided Black-Box Fuzzer Based on Frida. https://github.com/demantz/frizzer.
[18] Marco Mezzavilla et al. 2015. 5G mmWave module for the ns-3 network simulator. In *Proceedings of the 18th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. 283–290.
[19] Esteban Municio et al. 2023. O-ran: Analysis of latency-critical interfaces and overview of time sensitive networking solutions. *IEEE Communications Standards Magazine* 7, 3 (2023), 82–89.
[20] Navid Nikaein et al. 2014. OpenAirInterface: A flexible platform for 5G research. *ACM SIGCOMM Computer Communication Review* 44, 5 (2014), 33–38.
[21] O-RAN. 2025. O-RAN Software Community. https://o-ran-sc.org/.
[22] O-RAN. 2025. O-RAN Specifications. https://orandownloadsweb.azurewebsites.net/specifications.
[23] O-RAN Alliance. 2023. O-RAN Fronthaul Working Group 4: Control, User and Synchronization Plane Specification. https://www.o-ran.org/specifications. O-RAN.WG4.CUS.0-R003-v11.00, Release 003, Version 11.00.
[24] O-RAN Alliance. 2023. *O-RAN Security Threat Modeling and Remediation Analysis*. Technical Report. O-RAN Alliance WG11. O-RAN.WG11.Threat-Model.O-R003-v06.00.
[25] O-RAN Alliance. 2023. *Security for Near-RT RIC and xApps*. Technical Report. O-RAN Alliance WG11. O-RAN.WG11.Security-Near-RT-RIC-xApps-TR.0-R003-v03.
[26] Objective Systems, Inc. 2024. ASN1C ASN.1 Compiler. https://www.obj-sys.com/products/asn1c/. ASN.1 C/C++ Compiler and Runtime Libraries.
[27] Open5GS Project. 2024. Open5GS: Open Source Implementation of 5G Core and EPC. https://open5gs.org/. Accessed: 2025-08-05.
[28] OpenAirInterface Software Alliance. 2024. OpenAirInterface 5G Software Alliance. https://openairinterface.org/. Accessed: 2025-08-05.
[29] Josh Pereyda and the BooFuzz community. 2023. BooFuzz: Network Protocol Fuzzing for Humans. https://github.com/jtpereyda/boofuzz.
[30] Elvis Pfutzenreuter. 2022. pysctp: Python bindings for the SCTP transport protocol. https://pypi.org/project/pysctp/. Maintained by Benoit Michau; accessed August 5, 2025.
[31] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. (2020), 460–465.
[32] Michele Polese et al. 2022. ColO-RAN: Developing machine learning-based xApps for open RAN closed-loop control on programmable experimental platforms. *IEEE Transactions on Mobile Computing* 22, 10 (2022), 5787–5800.
[33] LLVM Project. 2021. libFuzzer: In-process, coverage-guided, evolutionary fuzzing engine for LLVM. https://llvm.org/docs/LibFuzzer.html.
[34] Onos Project. 2025. SDRAN-in-a-Box for SD-RAN project. https://github.com/onosproject/sdran-in-a-box.
[35] Robert Schmidt, Mikel Irazabal, and Navid Nikaein. 2021. FlexRIC: An SDK for next-generation SD-RANs. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 411–425.
[36] Zewen Shang, Matheus E Garbelini, and Sudipta Chattopadhyay. 2024. U-Fuzz: Stateful Fuzzing of IoT Protocols on COTS Devices. In *ICST*.
[37] Orange Open Source. 2023. ns-O-RAN-flexric: O-RAN and FlexRIC Integration for ns-3. https://github.com/Orange-OpenSource/ns-O-RAN-flexric. Accessed: 2025-08-05.
[38] Software Radio Systems. 2025. O-RAN NearRT-RIC and xApp tutorial. srsRAN Project documentation. https://docs.srsran.com/projects/project/en/latest/tutorials/source/near-rt-ric/source/index.html Accessed: 2025-01-19.
[39] Software Radio Systems. 2025. srsRAN Project: Open-Source 5G CU/DU. GitHub repository. https://github.com/srsran/srsRAN_Project.
[40] Kashyap Thimmaraju et al. 2024. Security testing the o-ran near-real time ric & a1 interface. In *WiSec*. 277–287.
[41] VIAVI Solutions Inc. 2024. TeraVM RIC Test: RAN Simulator Generator. https://www.viavisolutions.com/en-us/products/teravm-ai-rsg. Version 2.4-40470-43580c20.
[42] Wireshark Foundation. 2024. Wireshark. https://www.wireshark.org. Network Protocol Analyzer.
[43] Tianchang Yang et al. 2024. {ORANalyst}: Systematic Testing Framework for Open {RAN} Implementations. In *USENIX Security Symposium*. 1921–1938.
[44] Michał Zalewski. 2020. American Fuzzy Lop (AFL): Security-oriented Grey-box Fuzzer. https://github.com/google/AFL. Latest official release 2.57b; uses lightweight instrumentation and genetic mutation for code coverage.

A Appendix

Field Type	Constraints	Mutation Method
Boolean	True, False	Flip the boolean value.
Integer	Range, bounds	Random integer within boundaries.
Enumerated	Valid enums (ASN.1)	Replace with an adjacent enumeration option.
Bit String	Size constraint	Change bit-string length, insert malformed sequences invalid bit patterns.
Octet String	Length constraint	Randomly mutate bytes or insert invalid octet values within the allowed length.
Sequence / Sequence Of	Length, element types	Mutate sequence length or element values.
Choice	Element choice	Switch to another valid choice variant.
Optional	–	Insert or mutate present optional fields.

Table 2: Generation and mutation field types used in *ORAN-Claw*.

A.1 Genetic Algorithm Optimization

Algorithm 2 Updating population (Genetic Optimizer)

1: **procedure** UPDATE($[\mathbf{w}_i, c(\mathbf{w}_i)]_{1\ldots N}$)
2: $P_t \leftarrow$ sorted $[\mathbf{w}_i]_{1\ldots N}$ based on decreasing $[c(\mathbf{w}_i)]_{1\ldots N}$
3: $\mathbf{w}_A, \mathbf{w}_B \leftarrow P_t[0], P_t[1]$
4: $P_{t+1} \leftarrow \{\mathbf{w}_A, \mathbf{w}_B\}$ ▷ Select top two
5: **while** $|P_{t+1}| < N$ **do**
6: $\mathbf{w}_c \leftarrow \frac{\mathbf{w}_A + \mathbf{w}_B}{2}$
7: $\mathbf{w}_c \leftarrow \frac{\mathbf{w}_c}{\|\mathbf{w}_c\|_1}$ ▷ Crossover + normalize
8: **if** random() $< p_{\text{mut}}$ **then** ▷ Mutation
9: $j \sim \text{Uniform}(\{0, 1, 2\})$
10: $\mathbf{w}_{c,j} \leftarrow \max(\mathbf{w}_{c,j} + U(-0.2, 0.2), 0.01)$
11: $\mathbf{w}_c \leftarrow \frac{\mathbf{w}_c}{\|\mathbf{w}_c\|_1}$ ▷ Re-normalize
12: **return** P_{t+1} ▷ Advance generation

Algorithm 2 outlines the creation of a new population (i.e., $\mathcal{G}$.update in line 13 of Algorithm 1). In particular, we preserve the top two individuals as parents. To create a new individual, we perform arithmetic crossover $\mathbf{w}_c \leftarrow \frac{\mathbf{w}_A + \mathbf{w}_B}{2}$ with L1-normalization $\mathbf{w}_c \leftarrow \frac{\mathbf{w}_c}{|\mathbf{w}_c|_1}$ and preserve convex combination properties of the weights vector $w = [w_{\text{SM}}, w_{\text{json}}, w_{\text{both}}]$. This child is then subject to probabilistic mutation (with probability $p_{\text{mut}} = 0.3$) with uniform perturbation $U(-0.2, 0.2)$ to random components of $\mathbf{w}_c$. This is followed by constraining from below ($\max(\cdot, 0.01)$) and re-normalization to maintain feasible probability space.

B StateMapper Configuration

```
{ "Filter": "e2ap", "StateNameField": "e2ap.procedureCode"},
{ "Filter": "x2ap", "StateNameField": "x2ap.procedureCode"},
{ "Filter": "f1ap", "StateNameField": "f1ap.procedureCode"},
{ "Filter": "e1ap", "StateNameField": "e1ap.procedureCode"},
{ "Filter": "ngap", "StateNameField": "ngap.procedureCode"},
{ "Filter": "sctp", "StateNameField": "sctp.chunk_type"   }
```

Listing 3: StateMapper Rules

Implementation	VulnID	CVE	Vulnerability	General Cause	Component	Threat	Location
O-RAN SC RIC	VulnOSCRIC-01	CVE-2025-67398	Unhandled Exception	Flooding E2SetupRequest	E2 Termination	DoS	Not Specified
VIAVI TeraVM RSG	VulnVIAVI-01	Pending	Heap/Stack Buffer Overflow	**Malformed RC SM Structure**	gNB	Mem. Corrupt	Not Specified
	VulnVIAVI-02	Pending	Assertion	Malformed KPM SM Field	RIC	DoS	nr-gnb-mac.cc:1136
	VulnVIAVI-03	Pending	Unhandled Exception	Malformed RICSubscriptionRequest Field	gNB	DoS	Not Specified
	VulnVIAVI-04	Pending	Buffer Over-read	Malformed KPM SM Field	RIC	Mem. Corrupt	Not Specified
	VulnVIAVI-05	Pending	Unhandled Exception	**Malformed KPM SM Structure**	GUI, gNB, RIC	DoS	Not Specified
OAI	VulnOAI-01	CVE-2024-48408	Assertion	Malformed MAC SM Field	O-DU	DoS	ran_func_mac.c:126
	VulnOAI-02	CVE-2025-52142	Assertion	Malformed KPM SM Field	O-DU/CU-UP	DoS	ran_func_kpm_subs.c:226
	VulnOAI-03	CVE-2025-52146	Assertion	Malformed KPM SM Field	O-DU/CU-UP	DoS	msg_handler_agent.c:136
	VulnOAI-04	CVE-2025-52150	Assertion	**Truncated MAC SM Structure**	O-DU	DoS	mac_dec_plain.c:190
	VulnOAI-05	Pending	Assertion	Malformed KPM SM Field	O-DU	DoS	ran_func_kpm.c:267/268
	VulnOAI-06	Pending	Assertion	**Malformed KPM SM Structure**	O-DU	DoS	ran_func_kpm.c:72
	VulnOAI-07	CVE-2025-52148	Assertion	Unexpected TC SM Field	O-CU-UP	DoS	tc_dec_plain.c:1418
	VulnOAI-08	CVE-2025-52151	Assertion	Malformed KPM SM Field	O-CU-UP	DoS	ran_func_kpm.c:230
	VulnOAI-9	CVE-2025-52147	Assertion	Zero-ed RC SM Field	O-CU-CP	DoS	rc_dec_asn.c:953
	VulnOAI-10	Pending	Assertion	Unable To Recover	O-CU-CP (E1)	DoS	cucp_cuup_e1ap.c:31
	VulnOAI-11	Pending	Assertion	**Malformed KPM SM Structure**	O-CU-UP	DoS	ran_func_kpm.c:177
	VulnOAI-12	Pending	Assertion	Malformed E42RICsubscriptionDeleteRequest Field	O-CU-CP	DoS	bimap.c:126
	VulnOAI-13	Pending	Assertion	Malformed KPM SM Field	O-CU-UP	DoS	ran_func_kpm.c:168
	VulnOAI-14	Pending	Assertion	Multiple Malformed KPM Fields	O-DU/CU-UP	DoS	ran_func_kpm.c:71
	VulnOAI-15	Pending	Assertion	Invalid KPM SM Field	O-CU-CP/CU-UP	DoS	ran_func_kpm.c:229
	VulnOAI-16	Pending	Assertion	Malformed KPM SM Field	O-CU-CP/UP	DoS	ran_func_kpm.c:176
	VulnOAI-17	Pending	Assertion	Malformed KPM SM Field	O-CU-CP/UP	DoS	ran_func_kpm.c:167
	VulnOAI-18	Pending	Assertion	Unable To Recover	O-CU-UP/CP	DoS	e2_agent.c:242
	VulnOAI-19	Pending	Assertion	Invalid KPM SM Field	O-CU-UP/CP	DoS	plugin_agent.c:286
NS-3	VulnNS-01	Pending	Buffer Overflow	Malformed RICSubscriptionRequest Field	gNB	Mem. Corrupt	Not Specified
	VulnNS-02	Pending	Heap-based Buffer Overflow	Malformed E42SetupRequest Field	gNB	Mem. Corrupt	Not Specified
	VulnNS-03	Pending	Heap-Based Buffer Overflow	E42SetupRequest Duplication	gNB	Mem. Corrupt	Not Specified
	VulnNS-04	Pending	Assertion	Malformed E42RICSubscriptionDeleteRequest	gNB	DoS	ipv4-l3-protocol.cc:972
	VulnNS-05	Pending	Assertion	Malformed RICSubscriptionRequest Field	gNB	DoS	ipv4-l3-protocol.cc:580
	VulnNS-06	Pending	Assertion	Malformed RICSubscriptionRequest Field	LTE eNB	DoS	lte-spectrum-phy.cc:486
	VulnNS-07	Pending	Null pointer dereference	Malformed RICSubscriptionRequest Field	gNB	Mem. Corrupt	ptr.h:638
	VulnNS-08	Pending	Out-of-bonds read	Malformed RICSubscriptionRequest Field	gNB	Info. Disclosure	net-device-queue-interface.cc:216
	VulnNS-09	Pending	Null pointer dereference	Malformed RICSubscriptionRequest Field	gNB	Mem. Corrupt	ptr.h:630
	VulnNS-10	Pending	Assertion	Malformed RICSubscriptionRequest Field	gNB	DoS	object.cc:349
	VulnNS-11	Pending	Assertion	Malformed RICSubscriptionRequest Field	gNB	DoS	traffic-control-layer.cc:337
	VulnNS-12	Pending	Improper Input Validation	Malformed E42RICSubscriptionDeleteRequest Field	gNB	DoS	point-to-point-net-device.cc:279
	VulnNS-13	Pending	Improper Check for Unusual or Exceptional Conditions	Malformed E42RICSubscriptionDeleteRequest Field	gNB	Improper File Handling	mmwave-phy-trace.cc:399
	VulnNS-14	Pending	Assertion	Invalid KPM RC Field	gNB	DoS	default-simulator--impl.cc:235
	VulnNS-15	Pending	Assertion	Malformed RICcontrolRequest Field	gNB	DoS	buffer.cc:183
	VulnNS-16	Pending	Improper Input Validation	Malformed E42RICSubscriptionDeleteRequest Field	gNB	DoS	mmwave-phy-trace.cc:220
	VulnNS-17	Pending	Improper Check for Unusual or Exceptional Conditions	Malformed E42RICSubscriptionDeleteRequest Field	gNB	Improper File Handling	mmwave-phy-trace.cc:71
	VulnNS-18	Pending	Assertion	Malformed RICSubscriptionRequest Field	LTE eNB	Dos	mmwave-enb-phy.cc:1141
FlexRIC	-	CVE-2024-34034 (Existing)	Assertion	Flooding E42SubscriptionRequest	RIC	DoS	Not Specified
	VulnFlex-01	Pending	Assertion	Malformed RICSubscriptionRequest Field	RIC	DoS	msg_handler_iapp.c:343
	VulnFlex-02	Pending	Assertion	Malformed E42RICsubscriptionDelete Fields	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:2534
	VulnFlex-03	Pending	Assertion	Malformed E42SubscriptionRequest Fields	RIC	DoS	msg_handler_ric.c:117
	VulnFlex-04	Pending	Assertion	Malformed E42SubscriptionRequest Fields	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:544
	VulnFlex-05	Pending	Assertion	Malformed E42SubscriptionRequest Field	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:487
	VulnFlex-06	Pending	Assertion	Malformed MAC SM Field	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:1107
	VulnFlex-07	Pending	Assertion	Malformed E42SubscriptionDelete Field	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:2523
	VulnFlex-08	Pending	Assertion	Malformed E42SubscriptionDelete Field	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:2540
	VulnFlex-09	Pending	Assertion	Flooding E42SetupRequest	RIC	DoS	e2ap_msg_enc_asn.c:3165
	VulnFlex-10	Pending	Assertion	Malformed E42SubscriptionRequest Field	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:536
	VulnFlex-11	Pending	Assertion	Malformed E42SubscriptionDeleteRequest Field	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:2531
	VulnFlex-12	Pending	Assertion	Malformed E42SubscriptionRequest Field	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:527
	VulnFlex-13	Pending	Assertion	E42SubscriptionDeleteRequest Field	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:2548
	VulnFlex-14	Pending	Assertion	Malformed E42SubscriptionRequest Field	RIC E2AP(v2.03)	DoS	e2ap_msg_dec_asn.c:477
	VulnFlex-15	Pending	Assertion	Unable to Recover	RIC	DoS	map_e2_node_sockaddr.c:154
	VulnFlex-16	Pending	Assertion	Malformed KPM SM Field	RIC	DoS	reg_e2_nodes.c:174
	VulnFlex-17	Pending	Assertion	Unable to Recover	RIC	DoS	map_ric_id.c:227
	VulnFlex-18	Pending	Assertion	Unable to Recover	RIC	DoS	assoc_rb_tree.c:457
	VulnFlex-19	Pending	Assertion	Flooding E42SubscriptionRequest	RIC	DoS	msg_handler_iapp.c:342
	VulnFlex-20	Pending	Assertion	Malformed E42SubscriptionRequest Field	RIC E2AP(v1.01)	DoS	e2ap_msg_dec_asn.c:418
	VulnFlex-21	Pending	Assertion	E42SubscriptionDeleteRequest Field	RIC E2AP(v1.01)	DoS	e2ap_msg_dec_asn.c:435
	VulnFlex-22	Pending	Assertion	Malformed E42SubscriptionRequest Field	RIC E2AP(v1.01)	DoS	e2ap_msg_dec_asn.c:378
	VulnFlex-23	Pending	Assertion	Malformed E42SubscriptionRequest Field	RIC E2AP(v1.01)	DoS	e2ap_msg_dec_asn.c:368
	VulnFlex-24	Pending	Assertion	Malformed E42SetupRequest Field	RIC E2AP(v1.01)	DoS	e2ap_msg_dec_asn.c:2113
	VulnFlex-25	Pending	Assertion	Unable To Recover	RIC E2AP(v1.01)	DoS	e2ap_msg_enc_asn.c:2731
	VulnFlex-26	Pending	Assertion	Malformed E42SubscriptionRequest Field	RIC E2AP(v1.01)	DoS	e2ap_msg_dec_asn.c:427
	VulnFlex-27	Pending	Assertion	Malformed E42SetupRequest Field	RIC E2AP(v1.01)	DoS	e2ap_msg_dec_asn.c:2101
	VulnFlex-28	Pending	Assertion	Unable To Recover	RIC	DoS	endpoint_ric.c:64

Table 3: Vulnerabilities Discovered by *ORANClaw*. Entries marked in bold under *General Cause* indicate vulnerabilities resulting from structural message mutations rather than single-field modifications. "*Unable To Recover*" denotes failures to restore normal operation of the Component from crashes in previous sessions.