A Scripting Language for Security Patching in Open RAN[*]

Dong Hyeok Kim, Jinseo Lee, and Min Suk Kang[†]

KAIST, Daejeon, South Korea
{kimdh98,jinseo.vik.lee,minsukk}@kaist.ac.kr

Abstract

Securing the radio access network (RAN) requires timely deployment of patches, yet patch cycles for virtualized RAN (vRAN) components remain slow and tied to vendor-controlled processes, leaving many vulnerabilities unaddressed for extended periods. We present a lightweight scripting language that leverages the programmability of Open RAN (O-RAN) systems to let network operators express and apply vulnerability patches in a simple, declarative, and vendor-agnostic manner. The language introduces human-readable constructs for defining rules that alter protocol message handling and system behavior, enabling patches to be authored and deployed without modifying underlying RAN source code. This approach lowers the barrier to rapid patch creation, facilitates the sharing and reuse of patch specifications, and ensures compatibility across heterogeneous O-RAN deployments. We demonstrate the practicality of our design by implementing representative vulnerability patches and show that they can be deployed with minimal runtime overhead. Our work highlights the value of domain-specific scripting in making cellular patching more agile, transparent, and operator-driven.

1 Introduction

Cellular networks are undergoing a major transformation with the adoption of virtualization and the disaggregation of radio access networks (RANs). Open RAN (O-RAN) [17] in particular introduces standardized interfaces and programmability that promise faster innovation and greater flexibility for mobile operators. Yet, despite this architectural shift, one of the most persistent challenges in operational security remains unresolved: the slow and vendor-dependent process of patch deployment. Vulnerabilities uncovered in cellular protocols often remain unpatched for years, constrained by lengthy standardization cycles [1], vendor-controlled update mechanisms, and the complexity of multi-vendor integration. As a result, operators frequently face extended windows of exposure, even when effective mitigations are already known [16, 15].

Traditional patching practices exacerbate these delays. Modifying low-level RAN source code requires privileged vendor access and specialized expertise, while vendor-driven software updates typically align with infrequent release schedules. These limitations leave operators with little recourse for addressing vulnerabilities on their own. Although the programmability of O-RAN has enabled new classes of management and monitoring applications [19, 28], practical support for expressing and applying security patches remains minimal. Operators currently lack lightweight mechanisms to codify known mitigations into portable, vendor-agnostic artifacts that can be deployed directly in live networks.

In this work, we propose a domain-specific scripting language that fills this gap by enabling operators to describe vulnerability patches in a simple, declarative, and human-readable form. The language is designed to capture patch logic at the level of protocol message handling and

[*]Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. 36, December 16-18, 2025, Sapporo, Japan. © The copyright of this paper remains with the author(s).

[†]Corresponding author

system behavior, rather than requiring invasive modifications to source code. By abstracting patches as rules that operate on standardized message flows, the language supports vendor independence, ease of sharing, and applicability across heterogeneous O-RAN deployments.

Our approach brings the benefits of scripting into the security patching process for cellular networks. First, it lowers the barrier for operators to rapidly author and deploy patches without relying on vendor intervention. Second, it provides a reusable and auditable representation of security fixes that can be exchanged within the operator community. Finally, by aligning with standardized protocol behaviors, the language ensures that patches remain broadly compatible while introducing negligible runtime overhead. Through these properties, the proposed scripting language transforms patching from a vendor-bound software engineering task into a flexible, operator-driven process.

2 Background

2.1 Patch Cycles in Cellular Networks

The discovery of vulnerabilities in cellular protocols has accelerated in recent years, fueled by advances in open-source implementations [27, 23, 22], formal verification methods [3, 4, 10, 11], and dynamic testing frameworks [6, 12, 14, 24]. Despite this progress, the deployment of patches in commercial networks remains remarkably slow. Fixes are generally tied to the 3GPP [2] release cycle, which introduces new specifications every 18–24 months [1], and in practice may not be realized until major generational upgrades occurring roughly once a decade. This creates a persistent "time-to-patch gap," during which known vulnerabilities remain exploitable in deployed systems.

A central factor behind this gap is the operator's reliance on vendor-controlled patching processes. In traditional RAN deployments, baseband and control-plane software is proprietary, and operators lack access to the source code needed to apply independent fixes. Even with the adoption of virtualized RAN (vRAN), where cellular network components run as software modules, patch deployment continues to depend on vendor releases. Targeted or incremental fixes are rarely possible, leaving operators unable to respond directly to security threats. Consequently, while vulnerability knowledge accumulates quickly, the ability to mitigate these vulnerabilities in practice remains tightly constrained by vendor timelines and closed development models.

2.2 O-RAN and Operator Programmability

The adoption of O-RAN introduces a new level of openness and programmability into the radio access network. In the O-RAN architecture as shown in fig. 1, the RAN is disaggregated into modular components: the Radio Unit (RU), which handles radio transmission and reception; the Distributed Unit (DU), which manages real-time baseband and scheduling functions; and the Centralized Unit (CU), which provides higher-layer control and user-plane processing. These components interact over standardized interfaces [17] such as the fronthaul (between RU and DU) and F1 (between DU and CU), ensuring interoperability across vendors.

To coordinate and control this disaggregated environment, O-RAN introduces the RAN Intelligent Controller (RIC). The RIC serves as a logically centralized platform where network functions can be extended or customized through modular applications. In particular, near-real-time control functions are implemented via xApps [20], lightweight applications that operators

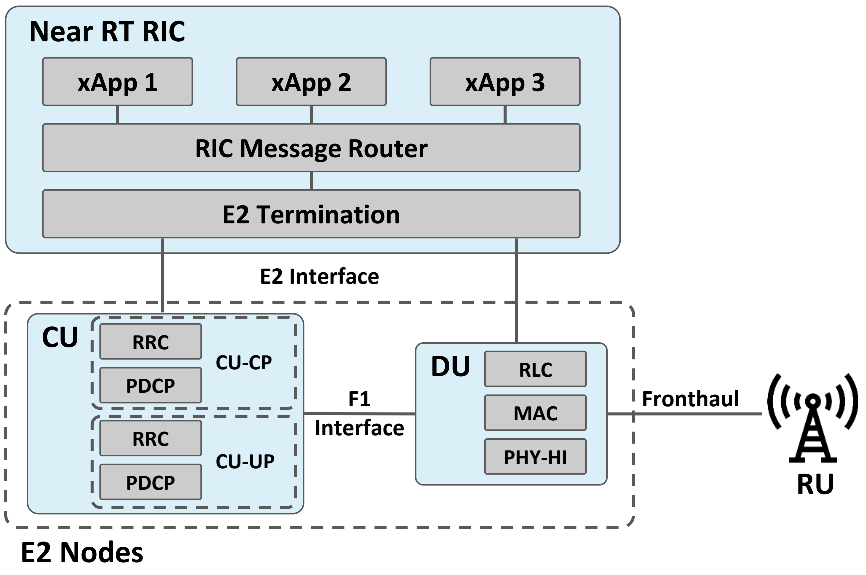

Figure 1: Overview of the O-RAN Architecture. [21]

can deploy to enforce monitoring, optimization, and policy decisions without modifying underlying RAN source code. xApps interact with the RAN through standardized APIs defined by their E2 service model (E2SM) [18], providing a vendor-neutral mechanism for operators to introduce new logic into live systems.

This architecture has already demonstrated its versatility in domains such as traffic engineering [13], performance monitoring [19], and anomaly detection [28]. However, despite the programmability of the RIC and the flexibility of xApps, current O-RAN practice has not yet extended these capabilities to systematic security patching. State-of-the-art research on enhancing O-RAN programmability, such as Janus [8], enables control over vRAN components, making it technically possible to deploy security patches via the RIC. Nonetheless, implementing these patches requires extensive knowledge and proficiency on the underlying vRAN source code, which is often inaccessible to operators. Consequently, operators still lack a structured mechanism to translate vulnerability knowledge into deployable xApp logic, leaving patching largely dependent on vendor releases rather than operator-driven intervention.

2.3 Scripting Languages for System Management

Domain-specific scripting languages have long been used to simplify configuration and control in complex systems. Declarative approaches, such as YAML [5] or JSON [7]-based configuration languages, enable human-readable specification of rules, policies, and automation workflows [25, 9]. This paradigm reduces the need for specialized programming expertise and lowers the barrier to system customization. Applying these principles to cellular patching offers a promising opportunity: operators could express vulnerability mitigations at the level of protocol message handling, without modifying source code or depending on vendor-specific implementations. A scripting-based approach therefore holds potential to shorten the time-to-patch gap and make patch deployment more transparent, auditable, and reusable.

3 Scripting Language Design

3.1 Design Goals

The primary goal of the scripting language is to enable operators to express vulnerability patches in a form that is simple, declarative, and vendor-agnostic, while still being precise enough to generate executable patch logic. To achieve this, the language is designed with the following principles:

- **Operator Accessibility.** The language should adopt lightweight, configuration-like constructs that lower the barrier for operators to author, understand, and share patches quickly, without requiring specialized programming knowledge.

- **Protocol Expressiveness.** The language must be expressive enough to capture the full semantics of 5G RAN protocols, allowing operators to describe patches in terms of protocol message flows and behaviors rather than low-level implementation details.

- **Action Modularity.** Patches should consist of independent rules that extract fields and apply actions, enabling reuse and composability.

- **Automated Code Generation.** Scripts must be directly translatable into executable message handlers, avoiding the need for manual coding or vendor intervention.

3.2 Language Structure

The scripting language adopts a declarative YAML-based format that is both human-readable and machine-parsable. fig. 2 presents the EBNF grammar representation of the scripting language. Each script contains three components: a channel specification that indicates where in the protocol stack the handler operates, a set of data structure declarations for maintaining state, and a collection of handlers that describe how particular messages are processed. Data structures are defined with attributes such as type, key and value representation, and maximum capacity, ensuring consistent initialization across patches. Handlers are message-centric: each one targets a specific protocol message type and is composed of two stages, field extraction and action execution. Field extraction rules allow operators to describe which protocol elements are relevant, while the actions determine how the system responds once those fields are obtained.

3.3 Handler Semantics and Actions

Handlers provide the core mechanism by which patches are expressed. Field extraction rules capture protocol semantics by identifying elements in RRC or NAS payloads through parameters such as type, identifier, and offset. Actions then encode the logic to be applied. The language supports a concise set of actions that cover common patching behaviors. Map operations enable values to be inserted, updated, or retrieved from declared data structures, making it possible to track state across messages. Conditional statements introduce branching based on equality, inequality, or null checks, while packet control primitives such as drop and release allow operators to suppress or terminate problematic flows. This action set strikes a balance between expressiveness and simplicity, ensuring that patches remain easy to author yet sufficiently powerful to enforce practical mitigations.

```
⟨syntax⟩ ::= channel: ⟨channel⟩
    datastructs: ⟨datastruct⟩
    handlers: ⟨handler⟩

⟨channel⟩ ::= dldcch | uldcch | dlccch | ulccch | . . .

⟨datastruct⟩ ::= - name: ⟨string⟩
    type: ⟨string⟩
    map_type: ⟨string⟩
    key_type: ⟨string⟩
    value_type: ⟨string⟩
    max_entries: ⟨integer⟩


⟨handler⟩ ::= - msg_type: ⟨msg_type⟩
    extract_fields: ⟨field⟩
    actions: ⟨action⟩

⟨msg_type⟩ ::= rrcConnectionReconfiguration | rrcConnectionSetup | . . .

⟨field⟩ ::= - nas: ⟨boolean⟩
    message_type: ⟨integer⟩
    name: ⟨string⟩
    eid: ⟨integer⟩
    type: ⟨string⟩
    offset: ⟨integer⟩

⟨action⟩ ::= ⟨update_map⟩ | ⟨lookup⟩ | ⟨if_action⟩ | ⟨drop⟩ | ⟨release⟩

⟨update_map⟩ ::= - action_type: update_map
    target: ⟨string⟩
    key: ⟨string⟩
    value: ⟨string⟩

⟨lookup⟩ ::= - action_type: lookup
    target: ⟨string⟩
    key: ⟨string⟩
    name: ⟨string⟩

⟨drop⟩ ::= - action_type: drop

⟨release⟩ ::= - action_type: release
    id: ⟨string⟩

⟨if_action⟩ ::= condition: ⟨condition⟩
    then: ⟨action_elems⟩
    else: ⟨action_elems⟩

⟨condition⟩ ::= cond_type: (isnull | gt | lt | eq)
    op1: ⟨string⟩
    [ op2: ⟨string⟩ ]
```

Figure 2: Grammar of the YAML-based scripting language for defining message handlers.

3.4 Code Generation Workflow

To transform high-level specifications into deployable logic, scripts are compiled through a two-stage workflow. First, a Python parser validates the YAML input against the formal grammar and constructs an intermediate representation of its data structures, handlers, and actions. This representation is then passed to a Jinja2 [26] template engine, which emits executable code. The templates ensure that data structures are properly instantiated, message fields are extracted according to the specified offsets, and actions are expanded into concrete code blocks. This design allows operators to focus entirely on the declarative specification of patches, while the generation pipeline handles low-level implementation details.

4 Implementation

Implementation overview. We implemented a prototype of the scripting language to demonstrate its practicality and effectiveness in applying vulnerability patches. The prototype consists of three main components: a YAML parser, a code generator, and a deployment mechanism. Together, these components allow an operator to author a patch specification, compile it into message handler code, and deploy it directly into an operational O-RAN environment. The entire workflow is automated, requiring no modifications to vendor source code or manual integration into the RAN software stack.

Parsing and code generation. Patch scripts are parsed by a Python front end that validates them against the formal grammar. The parser constructs an intermediate representation of the declared data structures, message handlers, and action sequences. This representation is passed to a Jinja2-based template engine, which generates executable handler code. The templates embed boilerplate for data structure initialization and message decoding, and expand action rules into control logic that performs updates, lookups, conditionals, or packet suppression. By decoupling scripts from implementation, this approach allows the same high-level specification to be reused across multiple deployments.

Deployment as an xApp. To integrate the generated handlers into a live O-RAN system, we implemented an xApp that runs within the near-real-time RIC. The xApp provides an interface for operators to upload YAML scripts. Upon receiving a script, the xApp invokes the parser and code generator, then deploys the resulting handler into the specified channel (e.g., dldcch, ulccch). This design leverages the O-RAN control plane for patch distribution, ensuring that patches can be applied dynamically and without vendor intervention. Because the deployment occurs through the RIC, the same xApp can be reused across heterogeneous RAN environments, reinforcing the vendor-agnostic design of the scripting language.

5 Evaluation

Evaluation against blind DoS attack. The evaluation examines whether the proposed scripting language can express and enforce a practical patch against a representative cellular vulnerability, the blind denial-of-service (DoS) [11, 14] attack. We focus on three aspects: the ability of the language to concisely specify the mitigation, the correctness of the generated handler in preventing service disruption, and the runtime performance overhead introduced by the patch. The blind DoS attack targets the control plane by exploiting temporary subscriber identifiers. An adversary repeatedly transmits forged `RRC Connection Request` that reuse the victim's TMSI. In the absence of additional checks, the RAN may erroneously accept the attacker's request and release the victim's existing RRC context, thereby terminating the victim's ongoing session. The attack is termed "blind" because the adversary does not need to observe the victim's traffic; it simply guesses and reuses identifiers. Prior work [14] has demonstrated that this attack reliably causes service interruption in unpatched systems.

Experiment setup. We conducted our evaluation on an O-RAN compliant testbed composed of open source implementations of a near-real-time RIC, a virtualized RAN implementation, and EPC. The RAN stack was deployed on a commodity server equipped with a 16-core Intel i9-12900K processor and 128 GB RAM running Ubuntu 22.04. A USRP B210 software-defined radio provided the radio front-end for the RAN. The victim device was a commercial smartphone (Samsung A20), while the adversary was instantiated on a separate machine running an open-source UE implementation (srsUE) with its own USRP B210. The RIC hosted the custom xApp described in section 4, which accepts YAML scripts, generates the corresponding handler code, and deploys the handler into the specified logical channel.

Experiment procedure. We compare two scenarios. In the baseline, the system operates without the patch. The attacker repeatedly issues spoofed connection requests using the victim's TMSI while the victim maintains its video stream. In the patched scenario, we generate and deploy the YAML script as shown in listing 1 encoding the mitigation to the RIC xApp.

```
# Dldcch script for recording new TMSIs
channel: dldcch
datastructs:
- name: tmsi_rnti_map
  type: bpf_map
  map_type: hash
  key_type: uint32_t
  value_type: uint32_t
  max_entries: 1024

handlers:
- msg_type: rrcConnectionReconfiguration
  extract_fields:
  - nas: true
    message_type: 66
    name: tmsi
    eid: 80
    type: uint32_t
    offset: 8
  actions:
  - action_type: update_map
    target: tmsi_rnti_map
    key: tmsi
    value: rnti

# Ulccch script for blocking requests with invalid TMSI-RNTI pair
channel: ulccch
datastructs:
- name: tmsi_rnti_map
  type: bpf_map
  map_type: hash
  key_type: uint32_t
  value_type: uint32_t
  max_entries: 1024
```

```
handlers:
- msg_type: rrcConnectionRequest
  extract_fields:
    - nas: false
      name: tmsi
      field: criticalExtensions.choice.rrcConnectionRequest_r8.ue_Identity.
          choice.s_TMSI.m_TMSI
      type: uint32_t
      offset: 0
  actions:
  - action_type: lookup
    target: tmsi_rnti_map
    name: rnti
    key: tmsi
  - action_type: if
    condition:
      cond_type: isnull
      op: rnti
    then:
      - action_type: drop
    else:
      - action_type: update_map
        target: tmsi_rnti_map
        key: tmsi
        value: rnti
```

Listing 1: Example script for blind DoS

The script declares a hash map binding TMSIs to RNTIs and defined a handler for `RRC Connection Reconfiguration` messages that updates the map and a handler for `RRC Connection Request` messages that drops requests with invalid TMSI-RNTI pairings. The generated code is deployed automatically into the downlink DCCH and uplink CCCH channels through the xApp. The attack is then repeated under identical conditions. To drive continuous downlink traffic, the victim streams a live video throughout each experiment. Tcpdump is used to capture downlink packets at the UE, providing a direct measure of service continuity.

Results. The patch was concisely expressed in fewer than 40 lines of YAML, consisting of a single data structure declaration and one message handler. This demonstrates that the scripting language can capture meaningful vulnerability mitigations without requiring low-level programming effort. fig. 3 shows the cellular data traffic in the victim UE in both the baseline and patch deployed cases while under the blind DoS attack. In the baseline case, the victim's downlink traffic is terminated when the attacker launches the attack, consistent with the expected effect of the blind DoS attack. With the scripted patch deployed, the victim's traffic remained continuous throughout the attack window. Packet traces collected at the UE showed uninterrupted bursts of downlink packets even as the adversary attempted to disrupt the connection. In other words, the generated handler correctly enforced the TMSI–RNTI binding and neutralized the attack without affecting legitimate signaling.

6 Discussion and Future Work

Our results demonstrate that declarative scripting provides a viable foundation for operator-driven patching in O-RAN systems. By expressing vulnerability mitigations as high-level rules,

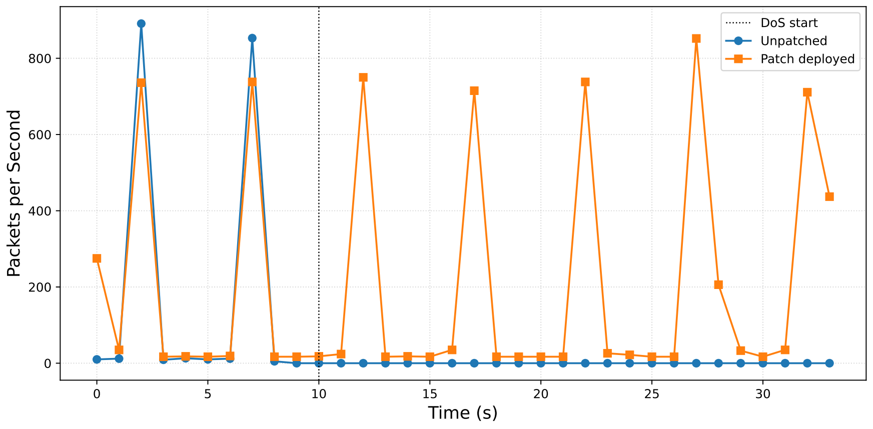

Figure 3: Data traffic observed by the victim UE (in terms of the number of packets received) under the Blind DoS attack.

operators can translate security insights into deployable defenses with minimal effort and runtime cost. The successful defense against the blind DoS attack illustrates how this approach enables rapid protection of users while avoiding disruption to legitimate traffic. Beyond this initial case study, several promising avenues exist to extend and strengthen the framework.

One opportunity lies in expanding the expressiveness of the language. While the current design focuses on message parsing, stateful mappings, and conditional actions, many future vulnerabilities may require richer constructs such as temporal properties across message sequences, multi-channel coordination, or protocol-level transformations. Extending the scripting language in these directions could make it suitable for a wider variety of security and performance use cases.

Another direction is the incorporation of automated validation and assurance. Declarative scripts could be paired with static analyzers or formal verification tools that check for semantic consistency, rule conflicts, or unintended side effects. Such mechanisms would not only improve reliability but also increase operator confidence in deploying patches quickly, even in critical production environments.

7 Conclusion

This work presents a lightweight scripting language for operator-driven patching in O-RAN systems. The language introduces a declarative interface for defining message handlers and actions, enabling operators to encode vulnerability mitigations without modifying vendor source code or waiting for lengthy release cycles. By lowering the barrier to creating and deploying patches, this work highlights the potential of domain-specific scripting to make cellular security more agile, transparent, and responsive. Our results demonstrate that empowering operators with practical tools for patch expression and deployment can significantly shorten the time-to-patch gap and strengthen the resilience of next-generation RAN infrastructures. We envision scripting-based patching becoming a cornerstone of agile defense in O-RAN, enabling networks to adapt quickly and securely to an evolving threat landscape.

8 Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00444170, Research and international collaboration on trust model-based intelligent incident response technologies in 6G open network environment)

References

[1] 3GPP. 3GPP Releases. https://portal.3gpp.org/#/55934-releases. [Online; accessed May 2025].

[2] 3GPP. Introducing 3GPP. https://www.3gpp.org/about-us/introducing-3gpp. [Online; accessed May 2025].

[3] Mujtahid Akon, Tianchang Yang, Yilu Dong, and Syed Rafiul Hussain. Formal Analysis of Access Control Mechanism of 5G Core Network. In *ACM CCS*, 2023.

[4] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A Formal Analysis of 5G Authentication. In *ACM CCS*, 2018.

[5] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008*, 5(11), 2009.

[6] Nathaniel Bennett, Weidong Zhu, Benjamin Simon, Ryon Kennedy, William Enck, Patrick Traynor, and Kevin RB Butler. RANsacked: A Domain-Informed Approach for Fuzzing LTE and 5G RAN-Core Interfaces. In *ACM CCS*, 2024.

[7] Douglas Crockford. Introducing json, 2001. Accessed: 2025-09-28.

[8] Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, and Zhihua Lai. Taking 5G RAN Analytics and Control to a New Level. In *ACM MobiCom*, 2023.

[9] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 618–635. IEEE, 2016.

[10] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *NDSS*, 2018.

[11] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5GReasoner: A Property-directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol. In *ACM CCS*, 2019.

[12] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. Noncompliance as Deviant Behavior: An Automated Black-box Noncompliance Checker for 4g LTE Cellular Devices. In *ACM CCS*, 2021.

[13] David Johnson, Dustin Maas, and Jacobus Van Der Merwe. Nexran: Closed-loop ran slicing in powder-a top-to-bottom open-source open-ran use case. In *Proceedings of the 15th ACM Workshop on Wireless Network Testbeds, Experimental evaluation & CHaracterization*, pages 17–23, 2022.

[14] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *IEEE S&P*, 2019.

[15] Oscar Lasierra, Gines Garcia-Aviles, Esteban Municio, Antonio Skarmeta, and Xavier Costa-Pérez. European 5G Security in the Wild: Reality versus Expectations. In *ACM WiSec*, 2023.

[16] Shiyue Nie, Yiming Zhang, Tao Wan, Haixin Duan, and Song Li. Measuring the Deployment of 5G Security Enhancement. In *ACM WiSec*, 2022.

[17] O-RAN Alliance. O-RAN Architecture Description 13.0 - O-RAN.WG1.TS.OAD-R004-v13.00, 2025.

[18] O-RAN Alliance. O-RAN E2 Service Model (E2SM) 7.0 - O-RAN.WG3.TS.E2SM-R004-v07.00, 2025.

[19] O-RAN Alliance. O-RAN E2 Service Model (E2SM) KPM 6.0 - O-RAN.WG3.TS.E2SM-KPM-R004-v06.00, 2025.

[20] O-RAN Alliance. O-RAN Near-RT RIC Architecture 7.0s - O-RAN.WG3.TS.RICARCH-R004-v07.00, 2025.

[21] O-RAN Project. O-RAN Architecture Overview. https://docs.o-ran-sc.org/en/latest/architecture/architecture.html, 2022. [Online; accessed May 2025].

[22] Open5GS. Open5GS. https://open5gs.org/. [Online; accessed May 2024].

[23] OpenAirInterface Software Alliance. OpenAirInterface5G: Open Source 5G Software. https://gitlab.eurecom.fr/oai/openairinterface5g. [Online; accessed Nov 2024].

[24] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. {DoLTEst}: In-depth Downlink Negative Testing Framework for {LTE} Devices. In *USENIX Security*, 2022.

[25] Saurabh Pujar, Luca Buratti, Xiaojie Guo, Nicolas Dupuis, Burn Lewis, Sahil Suneja, Atin Sood, Ganesh Nalawade, Matt Jones, Alessandro Morari, et al. Automated code generation for information technology tasks in yaml through large language models. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4. IEEE, 2023.

[26] Armin Ronacher. Jinja2 documentation. *Welcome to Jinja2—Jinja2 Documentation (2.8-dev)*, 2008.

[27] Software Radio Systems. srsRAN_4G. https://github.com/srsran/srsRAN_4G. [Online; accessed Mar 2024].

[28] Haohuang Wen, Phillip Porras, Vinod Yegneswaran, Ashish Gehani, and Zhiqiang Lin. 5G-spector: An O-RAN Compliant Layer-3 Cellular Attack Detection Service. In *NDSS*, 2024.