OZTrust: An O-RAN Zero-Trust Security System

Hao Jiang*, Hyunseok Chang†, Sarit Mukherjee† and Jacobus Van der Merwe*
* School of Computing, University of Utah, Salt Lake City, United States
† Network Systems and Security Research Department, Nokia Bell Labs, Murray Hill, United States

***Abstract*—The Open Radio Access Network (O-RAN) has gained significant attention as a future RAN framework. However, its architectural characteristics introduce unprecedented security challenges from expanded attack surface and increased risk for proprietary data theft and RAN control manipulation. Despite extensive security analysis from industry, concrete security solutions for the evolving O-RAN framework are still lacking in the literature. In this paper, we propose OZTrust, a Zero-Trust security system tailored for the O-RAN environment. OZTrust comprises two components: access control module and policy management module. The former performs per-packet tagging and verification for each xApp as dictated by its access control policy, while the latter automatically derives necessary access control policies by discovering xApp's communication patterns through distributed tracing. Our prototype-based evaluation demonstrates that OZTrust provides more fine-grained access control for xApps than existing Role-Based Access Control (RBAC) and Container Network Interfaces (CNIs) and outperforms its predecessor.**

I. Introduction

The Open Radio Access Network (O-RAN) has emerged as a prominent framework for future RAN systems [1], [2]. Its disaggregated architecture allows flexible deployment and management of virtualized components in cloud-native environments, while its open and standardized interfaces enable interoperability among diverse vendors. However, alongside these benefits, O-RAN also introduces a host of unprecedented security challenges, as highlighted in recent studies [3]–[5]. The main reason is that, compared to the traditional monolithic RAN systems, its disaggregated architecture exposes additional entry points into the system, expanding its attack surface. Moreover, the ORAN's Near-Real-Time RAN Intelligent Controller (Near-RT RIC), which manages critical RAN functions while storing sensitive data like RAN configuration, subscriber identities, Machine Learning (ML) models and performance metrics, can become an ideal target to attackers due to devastating consequences from any security breaches.

In a recent study, the O-RAN Alliance Work Group 11 (WG11), which is dedicated to O-RAN security, identifies unauthorized use of APIs as one of the primary threats in the Near-RT RIC [6]. This attack could potentially grant attackers the ability to manipulate RAN controls or access sensitive data. WG11 recommends authentication and RBAC as mitigation solutions. However, these measures alone are insufficient, as authentication and authorization credentials and tokens can be stolen. Notably, credential or token leaks are not uncommon, even among prominent organizations like Facebook and GitHub [7], [8].

Additionally, the increasingly prevalent threat of lateral movement, described as one of the major tactics in the threat matrix summarized by Microsoft [9], poses a significant risk to the Near-RT RIC as well. Lateral movement involves attackers navigating deeper into a network after gaining initial access, seeking high-value assets and network controls. It enables attackers to evade detection and maintain access even if the initial entry point is discovered, making detection highly challenging. Consequently, existing proposals to mitigate lateral movement in cloud-native environments often lead to noticeable overhead due to heavy computation of complex algorithms or ML-based approaches [10], [11], rendering them inapplicable for time-critical Near-RT RIC operations.

Unfortunately, up to this point, no O-RAN-specific solution has been put forward to effectively address the security threats targeting the O-RAN Near-RT RIC. In this paper, we propose OZTrust, a Zero-Trust security system for O-RAN. OZTrust effectively addresses the issues pertaining to unauthorized API access and lateral movement in the O-RAN Near-RT RIC, ensuring the protection of sensitive and proprietary data from theft and preventing malicious manipulation of RAN controls. Moreover, OZTrust achieves these security objectives while incurring minimal overhead on the system.

OZTrust comprises two components: access control module and policy management module. The access control module builds upon eZTrust [12], an eBPF-based Zero-Trust access-control mechanism built for containerized environments. Its core concept involves tagging each outgoing packet and verifying each incoming packet based on the tag. We adapt and optimize eZTrust-based approach for the O-RAN environment. In O-RAN context, a unique tag represents the identity of an xApp, derived from its authentic contexts. Therefore, we extend eZTrust's generic context discovery mechanism to cover xApp-specific contexts. In addition, unlike eZTrust, where packet verification involves expensive slow path to populate local context maps on demand, OZTrust pre-populates the context maps by leveraging predictable xApp communication patterns, thereby avoiding expensive slow path processing.

The policy management module is responsible for populating xApp's access-control policies to be enforced by the access control module. For this purpose, instead of solely relying on vendor-supplied information, we independently discover and verify xApps' communication patterns via existing distributed tracing libraries and tools [13], [14]. Once their communication patterns are discovered, the policy management module combines the information with xApp contexts acquired through the access-control module to automatically derive

access-control policies for xApps. This approach reduces the risk of using accidental misinformation supplied by third-party vendors or open-source developers, and minimizes the possibility of human-induced errors during network policy creation process. Furthermore, the communication pattern tracing process enables us to optimize the context population mechanism of the access control module as described earlier.

Our contributions can be summarized as follows. First, we present the design and implementation of OZTrust's access-control mechanism, which we adapt from eZTrust for the O-RAN environment. Second, we design and implement a mechanism that automates the access-control policy generation process, which can mitigate the risk associated with relying on untrustworthy vendor-supplied information and minimize human errors that may arise during the manual policy configuration process. Finally, to the best of our knowledge, we are the first to propose and implement a practical security system tailored for the O-RAN Near-RT RIC. Our evaluation demonstrates that OZTrust delivers more fine-grained protection than widely-used RBAC and CNIs and outperforms eZTrust.

II. RELATED WORK

Several surveys [2]–[5] offer comprehensive analyses and evaluations of O-RAN security, but they do not present concrete solutions. WG11, a dedicated working group for O-RAN security, released several specifications and studies that outline potential attacks and provide suggestions for mitigating risks [6]. They recommend authentication and RBAC to mitigate unauthorized use of APIs in the Near-RT RIC, which are responsible for data accesses and RAN controls. However, authentication and RBAC measures alone are insufficient due to potential risk for credential and token leaks [7], [8].

O-RAN Near-RT RIC is commonly deployed in containerized environments, and therefore it is susceptible to tactics and techniques outlined in the threat matrix provided by Microsoft [9]. Among these tactics, lateral movement stands out as a serious threat for the Near-RT RIC. It empowers attackers to elude detection and retain access even if the initial entry point is detected, making detection highly challenging. Several strategies to mitigate lateral movement in cloud-native environments are proposed [10], [11]. However, these approaches often incur noticeable overhead due to their reliance on complex algorithm computations or ML-based techniques.

There are Zero-Trust security solutions for Kubernetes [15]–[17]. However, these solutions provide access control only at the pod/container-level, and hence are inadequate when pods or containers are compromised with malware.

OZTrust is designed to prevent unauthorized use of APIs and lateral movement with sufficient granularity, as will be described in the rest of the paper.

III. OZTRUST ARCHITECTURE DESIGN

A. Threat Model

1) Trusted vs. Untrusted: We assume that the infrastructure provider is reliable and free from vulnerabilities. Furthermore, we assume that the OZTrust framework is securely integrated with the provider's infrastructure, ensuring the secure collection of contexts and non-compromised packet tagging and policy enforcement. This integration also guarantees the secure storage and distribution of Zero-Trust access control policies and contexts. On the other hand, we do not trust xApps, containers housing them, and container images themselves.

2) Threat Surface: We outline possible threats for the O-RAN Near-RT RIC. Attack targets can be broadly classified into two categories: RAN controls and data. Examples of control-related attacks include: (i) Unauthorized access to disaggregated RAN components with the intent of deteriorating network performance. (ii) Manipulation of RAN controls such as mobility management of mobile users, resource allocation or scheduling, spectrum or infrastructure sharing, where the goal is to induce network malfunctions. (iii) Gaining unrestricted control over one or more O-RAN nodes to produce synthetic data, which can mislead ML-driven RIC operations to cause erroneous control decisions, performance degradation, or even service outages [3], [4]. Examples of data-related attacks encompass: (i) Obtaining sensitive information transmitted over O-RAN interfaces for RAN controls, management, and configuration purposes. (ii) Stealing proprietary data utilized for ML model training and testing. (iii) Exfiltrating subscriber's private data such as identities. (iv) Acquiring cryptographic keys deployed across network elements [2], [5].

3) Attackers and Their Capabilities: We consider that an attacker possesses malicious intent and seeks to execute the aforementioned control- or data-related attacks. Potential attackers may include: (i) Open-source xApp developers or those affiliated with xApp vendors who deliberately introduce backdoors or malware into the code. (ii) Developers responsible for upstream libraries that are beyond the control of xApp vendors or open-source developers. (iii) Individuals outside open-source organizations or vendors, who have knowledge of intentionally or unintentionally embedded backdoors or vulnerabilities in xApps [2]–[5].

An attacker can compromise an xApp container by exploiting vulnerabilities, backdoors or pre-existing malware in the container. The attacker can then exploit the compromised state to acquire authentication and authorization credentials and tokens, thereby enabling unauthorized access to APIs in the O-RAN Near-RT RIC. Moreover, upon successfully gaining initial access to the system, the attacker may employ lateral movement techniques, as summarized in [9], in an attempt to obtain unauthorized access to other deployed xApps.

B. OZTrust Architecture

In the following we describe the OZTrust architecture highlighted in Figure 1.

1) Access-Control Module: The access-control module in OZTrust is adapted from eZTrust, an eBPF-based access-control solution originally designed for containerized environments [12]. We first briefly summarize the original eZTrust design, and then describe how we adapt its design for the O-RAN Near-RT RIC environment.

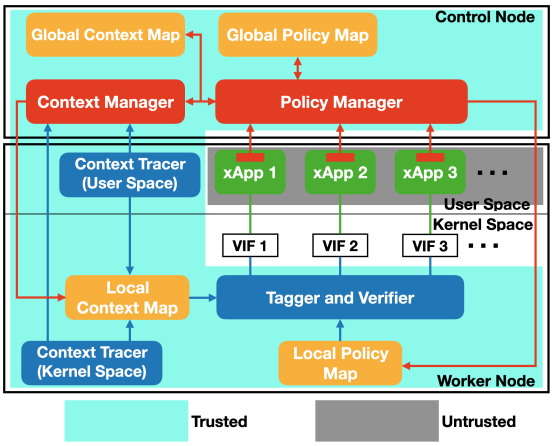

Fig. 1: OZTrust architecture.

eZTrust design. eZTrust is characterized by three design features: context tracing, packet tagging, and packet verification. Context tracing in eZTrust is carried out by dedicated tracing components which discover, upon the launch of each microservice, a set of contexts associated with it and its runtime environment (e.g., app name, version, SSL version, geographic location, filesystem image, OS version). Discovered contexts are stored in the global/local context maps as <tag, set of contexts>, where the tag serves as a globally unique key in the context maps. The tag is added to each outgoing network packet on sender-side and is used on receiver-side to retrieve a set of contexts associated with the microservice that has generated the packet. To verify (i.e., access control) each tagged packet on receiver-side, eZTrust looks up the sender's contexts in the local context map by using the tag, derives the intended receiver's contexts from the packet, and performs verification based on these identified contexts and pre-configured access control policies.

OZTrust adaptation. As a Zero-Trust security solution for generic microservices, eZTrust is not designed to trace xApps in the O-RAN environment. To address this limitation, we extend its context tracing mechanism to support *xApp-specific contexts*. OZtrust's extended context tracing component (called "Context Tracer") runs in both kernel space and user space. With the OSC's Kubernetes-based O-RAN Near-RT RIC implementation [18], in-kernel Context Tracer can trace contexts such as an xApp's name, version, inode number, pod's name and its UUID, and container's name and its UUID. Context Tracer in user space can further collect an xApp's subscribed message types in the RIC Message Router (RMR), subscribed service types in the E2 termination, and control capabilities.

Unlike eZTrust, OZTrust *pre-populates* local context maps with relevant xApp contexts retrieved from the global context map, instead of relying on on-demand population. Such pre-population is possible because OZTrust performs distributed tracing prior to system launch to learn each xApp's communication patterns (see Section III-B2). From distributed tracing, OZTrust can predict what other xApps will communicate with a given xApp, and hence pre-populate contexts of those xApps in advance. As a result, when an xApp receives a tagged

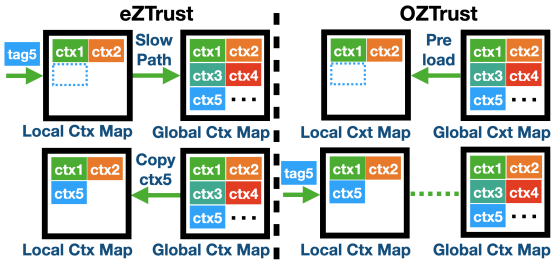

Fig. 2: Context populating mechanism comparison.

packet, the contexts associated with the tag can be immediately retrieved locally without querying the global context map, which significantly speeds up the packet verification process. This approach solves the trade-off dilemma between excessive memory usage (i.e., pre-copying all contexts from the global context map to local maps on all nodes) and degraded performance (i.e., on-demand retrieval of required contexts from the global context map upon context lookup miss). Since eZTrust is a general Zero-Trust security solution without any prior knowledge of microservices' communication patterns, it opts for the latter approach, sacrificing performance. In contrast, OZTrust eliminates the need for "slow path" by exploiting know communication patterns of xApps. Figure 2 illustrates the difference between the two.

In eZTrust, packet verification occurs at the physical NIC interface (PIF) of a worker node. This approach has two drawbacks. First, it incurs additional overhead for discovering a target xApp for each incoming packet. More importantly, it ends up setting protection boundaries *at the node level*, and hence cannot enforce access control policies among the xApps co-located on the same node. To address these drawbacks, OZTrust moves packet verification to the virtual interface (VIF) of each xApp. This enhances flexibility in xApp deployment and improves verification performance. One downside is that OZTrust must deploy as many instances of Verifier as the number of deployed xApps, resulting in more memory usage. However, since the eBPF implementation of Verifier is lightweight, the additional memory usage is negligible. Thus, the benefits of this approach easily outweigh the cost.

2) The Policy Management Module: The policy management module, a new component in OZTrust, handles two key functions: auto-discovery of xApps' communication patterns and auto-generation of access control policies. Figure 3 shows its internal design.

Auto-generation of access control policies is motivated by three concerns. First, as advocated by Zero-Trust principles, it is imperative to not bestow any trust upon xApps and xApp-related information supplied by their vendors or developers since these xApps are commonly sourced from various third-party entities or open-source repositories [2], [5]. Second, when a large number of xApps are deployed with complex access control policies in place, manual configuration processes often result in errors [19]. Finally, the on-demand context population mechanism of eZTrust, triggered by context lookup misses, leads to degraded performance if many xApp communication flows are short-lived. Driven by these factors,

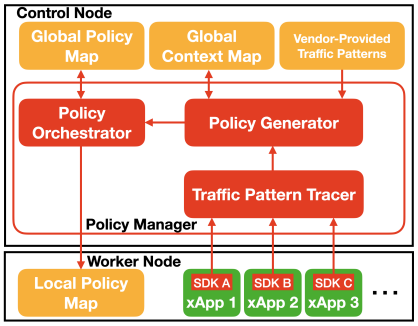

Fig. 3: OZTrust policy management module.

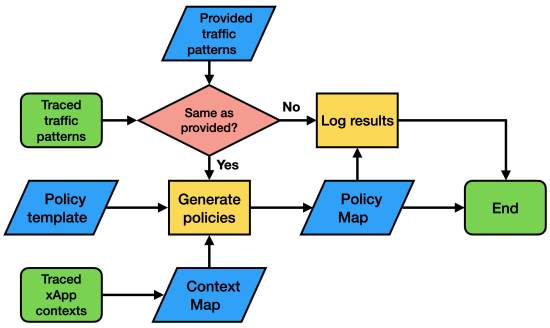

Fig. 4: Policy generation procedures.

the policy management module carries out two key functions: traffic pattern tracing and access-control policy generation.

Traffic pattern tracing. Automatic traffic pattern tracing is facilitated by leveraging distributed tracing libraries and tools [13], [14]. Distributed tracing is originally designed to trace distributed microservice workflows as part of performance troubleshooting. We re-purpose this technique to obtain a comprehensive view of actual traffic patterns among deployed xApps. To enable distributed tracing in an xApp, we need to instrument it, which involves inserting tracing APIs and their dependencies into the xApp's source code. OpenTelemetry [14] already provides tracing support for a wide range of libraries used for application communication, which are written in popular programming languages such as Python, C++, Go, and Java. This includes support for libraries handling HTTP, gRPC, database operations (e.g., Mongo DB, Redis, Memcached). If an xApp is developed with one of those supported libraries, no manual instrumentation is necessary.

The traffic pattern tracing is conducted as a pre-deployment procedure taking place before the commercial launch of xApps. It is reasonable for carriers to require that third-party xApp vendors collaborate in the instrumentation and automatic traffic pattern tracing process. When xApps are obtained from open-source repositories, the process becomes even simpler, as carriers have the option to perform automatic or manual instrumentation themselves.

Once distributed tracing is enabled in xApps, it automatically generates telemetry data, referred to as *Spans* [14], whenever the instrumented libraries are invoked by xApp communication. Traffic Pattern Tracer gathers these Spans from the xApps and stores them in the shared database in the O-RAN Near-RT RIC. Subsequently, these Spans are utilized to generate access-control policies as described next.

Policy generation. Once xApps' traffic patterns are traced, the policy generation process proceeds as follows.

Step 1. The collected Spans stored in the shared database are pre-processed to trim non-essential metadata to produce concise traffic pattern information.

Step 2. The resulting traffic pattern information is cross-checked with xApp's vendors-supplied information. In case of a mismatch, the results are logged, and the process is terminated. Otherwise, we proceed to the next step.

Step 3. xApps' access-control policies are generated based on traffic patterns, policy template, and xApp contexts. The policy template, prepared by carriers, contains <sender xApp's name, receiver xApp's name, sender's context keys, receiver's context keys> tuples. Essentially this template indicates which set of xApp contexts are used by carriers to define their access control policies. The policy generator iterates through each entry in the policy template. It looks up global context map for the sender and receiver using their tags, fetches their corresponding contexts, and retrieves the context values based on the context keys specified in the template. It then creates a policy entry by populating it with the retrieved context values. Next, the policy generator examines the traced traffic patterns to determine if communication is expected between the sender and the receiver. If it is, the policy action of the entry is set to `ACCEPT`. Otherwise it is set to `DROP`. Generated policy entries are stored in global policy map.

Step 4. The newly created policies are distributed to the respective local policy maps on worker nodes. It selectively populates the local policy maps with relevant policies, considering xApps deployed on each node and their communication requirements. Carriers have the flexibility to define multiple policies for each xApp based on various subsets of xApp contexts. This enhances the versatility of policy creation. Section IV provides a concrete example of policy generation.

IV. Implementation

We implement an OZTrust prototype and integrate it with the OSC's Kubernetes-based Near-RT RIC implementation [18]. We highlight key aspects of the prototype.

We implement packet tagging and verification with eBPF, based on eZTrust implementation. Unlike eZTrust, we attach corresponding eBPF programs to ingress and egress of the same VIF of an xApp by leveraging `clsact` qdisc.

To adapt xApp contexts in the O-RAN Near-RT RIC environment, we extend the OZTrust's context-tracing mechanism as follows. Context Tracer in kernel space captures essential events such as xApp deployment and TCP/UDP sockets-related activities using relevant eBPF hooks. The context-collecting routines are implemented in Python, utilizing `client.CoreV1Api()` from the Kubernetes Python client library. Upon xApp-related events detected by eBPF hooks, these routines collect xApp-specific contexts such as the xApp's name, version, pod's name and UUID, and container's name and UUID. Context Tracer in user space is a Python

Policy Template Entry	Sender's Contexts	Receiver's Contexts	Traffic	Generated Policy Entry
<AD, TS, <UUID>, <UUID>>	{**Name:** AD, **UUID:** abc, ...}	{**Name:** TS, **UUID:** xyz, ...}	AD → TS	<abc, xyz, Accept>
<AD, TS, <UUID, Version>, Inode>	{**Name:** AD, **UUID:** abc, **Version:** 0.2, ...}	{**Name:** TS, **Inode:** 123, ...}	AD → TS	<<abc, 0.2>, 123, Accept>
<QP, AD, A1_RMR_SUB, TS_RMR_SUB>	{**Name:** TS, **A1_RMR_SUB:** 10, ...}	{**Name:** QP, **TS_RMR_SUB:** 20, ...}	QP → TS	<10, 20, Drop>

TABLE I: An example of policy generation.

program that gathers xApp-specific contexts from the runtime information in xApp containers, such as xApp's subscribed message types in the RMR, subscribed service types in the E2 termination, and control capabilities.

We integrate the OZTrust prototype with O-RAN Near-RT RIC OSC implementation to demonstrate real-world use cases. In Section V, we will show how OZTrust can protect communication among three real-world xApps: Anomaly Detection (AD), Traffic Steering (TS), and QoE Predictor (QP). AD retrieves user equipment (UE) data from the shared database (InfluxDB) and applies ML-based algorithms to detect anomalous UEs based on the data. AD then sends to TS the information about anomalous UEs along with their corresponding degradation type. Upon receiving messages from AD, TS sends a prediction request to QP, which includes the list of detected anomalous UEs. QP retrieves the most up-to-date UE and cell metrics for the identified anomalous UEs. Using ML-based algorithms, it predicts the expected throughput in the neighboring cells for these UEs. The prediction results are then sent back to TS. Based on the received prediction results, TS makes a final decision on whether to perform a handover for the identified anomalous UEs.

We instrument these xApps with OpenTelemetry [14] for automatic traffic pattern tracing which is then used by the policy management module. AD and QP are developed in Python, while TS is written in C++. We manually instrument them with OpenTelemetry SDK since the RMR library used by these xApps for communication is not supported for automatic instrumentation. To collect the traced Spans, we utilize the Jaeger collector [13].

The Policy Manager application, implemented in Python, serves as the central module responsible for policy generation. Table I provides a concrete example illustrating the generation of policy entries for the anomaly detection use case.

Our extensions, new developments and manual xApp instrumentation amount to approximately 600 SLOC of Python, 300 SLOC of C, and 50 SLOC of C++.

V. EVALUATION

Our evaluation aims to address the following key questions: (i) Does OZTrust effectively prevent unauthorized use of APIs? (ii) Does it successfully prevent lateral movement in the O-RAN Near-RT RIC? (iii) What is its performance compared to other schemes? To answer these questions, we deploy the prototype and conduct experiments on AMD EPYC Rome servers, each equipped with 16 CPU cores and 128GB memory, provisioned in the POWDER testbed infrastructure [20].

A. Functional Evaluation

To evaluate OZTrust's effectiveness for preventing unauthorized API access and lateral movement, we conduct experiments with the real-world anomaly detection use case shown in Section IV and compare OZTrust with Kubernetes RBAC [15] and two popular CNIs: Cilium [16] and Calico [17].

1) Unauthorized use of APIs: We configure the following access control policies for OZTrust, along with equivalent network policies for Cilium and Calico: AD and QP have legitimate access to InfluxDB, whereas TS does not. For Kubernetes RBAC, we specify "Pods" as the resource permitted to be accessed by all the xApps. As shown in Figure 9, at time 0–10s, all three xApps perform normally. Starting from time 10s, we launch a malware process in AD, TS, and QP containers one by one in this sequence with 10-second intervals. This malware sends illegitimate API accesses to InfluxDB. We assume that the malware is using a stolen database access credential. Purely for visual aid (i.e., to show whether traffic is blocked or allowed), we set baseline normal traffic rate for OZTrust, Cilium, and Calico differently, and let each malware process send traffic at 10Mbps fixed rate.

As shown in Figure 5, Cilium and Calico fail to prevent unauthorized database access from malware in AD and QP at time 10–20s and 30–40s. This is because, by default, both of them enforce network policies at the pod/container level. In contrast, OZTrust enforces access control policies via process-level contexts, thereby effectively preventing unauthorized database access by the unknown malware. At time 20-30s, OZTrust, Cilium, and Calico successfully prevent unauthorized database access from TS. Cilium and Calico succeed in this scenario because their pod-level network policies prevent any traffic from the TS pod to InfluxDB. However, Kubernetes RBAC fails to prevent the attacks because the service account's role grants access to all pods. If Kubernetes RBAC is redefined at pod-level by leveraging Kubernetes Custom Resource Definitions (CRDs), it could at best achieve the same results as Cilium and Calico. To block unauthorized access by malware in all scenarios like OZTrust, Kubernetes RBAC, Cilium and Calico have to resort to additional entities specifically responsible for process-level security [21], [22].

2) Lateral Movement: We conduct another series of experiments depicted in Figure 10 with the same access-control and network policies among AD, QP and TS. The experiments are structured as follows: At time 0–10s, an attacker compromises TS. At time 10–20s, the attacker transfers a piece of malware to AD by exploiting its vulnerability. At time 20–30s, the attacker installs the malware on AD and successfully initiates lateral movement attack from its initial access point. Finally, at time 30–40s, the malware in AD begins sending illegitimate traffic to InfluxDB. Again, for visual convenience, we set baseline normal data rates for OZTrust, RBAC, Cilium, and Calico differently. We measure the aggregated data rates (including malware traffic) on AD and InfluxDB. The results, as shown in Figure 6, demonstrate that OZTrust effectively prevents the lateral movement attack. In contrast, in case of

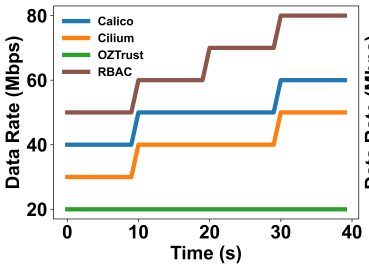

Fig. 5: Use of APIs.

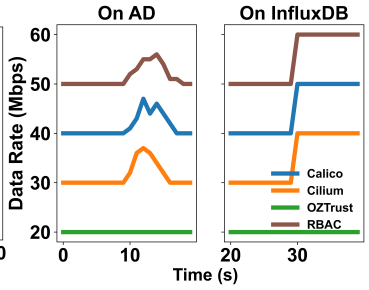

Fig. 6: Lateral movement.

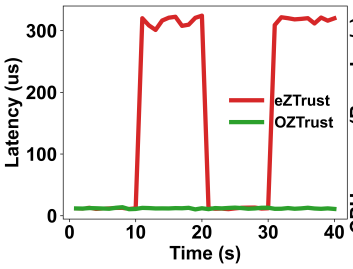

Fig. 7: OZ vs. eZ latency.

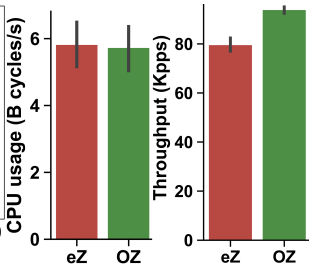

Fig. 8: Tput and CPU.

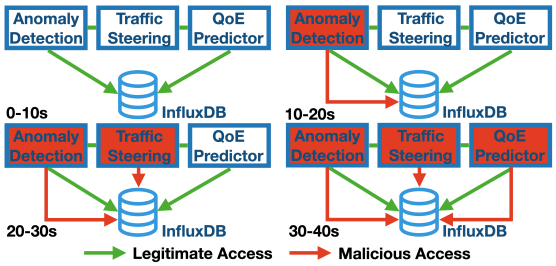

Fig. 9: Scenario 1: Unauthorized use of APIs.

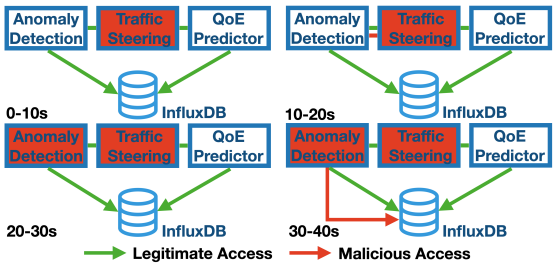

Fig. 10: Scenario 2: Lateral movement.

RBAC, Cilium, and Calico, additional traffic is shown at time 10s (caused by the transfer of malware code) and at time 30s (caused by illegitimate access to InfluxDB).

B. Performance Evaluation

Finally, we compare OZTrust and eZTrust in terms of performance. For this, we set up two simple Python-based xApps based on the OSC O-RAN Near-RT RIC implementation and deploy them across two back-to-back connected nodes. One xApp sends messages to the other at a customizable rate, and the other echoes the messages back to the sender xApp. In case of eZTrust, since it lacks the ability to trace their xApp-specific contexts, we deploy two simple containers for eZTrust, running simply Python-based client and server apps. Then we configure eZTrust to alternate between "fast path" and "slow path" at 10-second intervals to emulate multi-flow handling.

Figure 7 shows latency comparison between eZTrust and OZTrust, while Figure 8 presents throughput and CPU usage comparison. The results indicate that OZTrust outperforms eZTrust in terms of latency and throughput. This performance improvement is due to the OZTrust's context pre-population mechanism which eliminates inefficient slow path. There is no substantial difference in CPU usage, as both solutions utilize the eBPF technology for packet tagging and verification.

VI. CONCLUSION

In this paper, we present OZTrust, a Zero-Trust security system designed to address security challenges in the O-RAN Near-RT RIC. We show how OZTrust can effectively mitigate unauthorized API access and lateral movement, ensuring the protection of RAN controls and valuable data. Our evaluation demonstrates that OZTrust offers fine-grained protection with low overhead. As part of future work, we plan to develop tracing support for the RMR library of OSC xApps to easily integrate OZTrust in the existing O-RAN platform.

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grant Number 1827940.

REFERENCES

[1] "O-RAN Alliance," https://www.o-ran.org/.
[2] M. Polese *et al.*, "Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges," *IEEE Communications Surveys & Tutorials*, 2023.
[3] D. Mimran *et al.*, "Evaluating the Security of Open Radio Access Networks," *arXiv preprint arXiv:2201.06080*, 2022.
[4] M. Liyanage *et al.*, "Open RAN Security: Challenges and Opportunities," *Journal of Network and Computer Applications*, vol. 214, 2023.
[5] D. Mimran *et al.*, "Security of Open Radio Access Networks," *Computers & Security*, vol. 122, p. 102890, 2022.
[6] "O-RAN alliance specifications," https://orandownloadsweb.azurewebsites.net/specifications/.
[7] C. Guan *et al.*, "Dangerneighbor attack: Information leakage via postmessage mechanism in html5," *Computers & Security*, vol. 80, 2019.
[8] M. Jackson, "How hackers used stolen github tokens to access private source code," https://blog.gitguardian.com/how-hackers-used-stolen-github-oauth-tokens/, 2022.
[9] Y. Weizman, "Secure containerized environments with updated threat matrix for kubernetes," 2022, Microsoft Security Blog.
[10] B. Bowman *et al.*, "Detecting Lateral Movement in Enterprise Computer Networks with Unsupervised Graph AI," in *Proc. RAID '20*, 2020.
[11] A. Bohara *et al.*, "An unsupervised multi-detector approach for identifying malicious lateral movement," in *Proc. 2017 IEEE SRDS*, 2017.
[12] Z. Zaheer *et al.*, "eZTrust: Network-Independent Zero-Trust Perimeterization for Microservices," in *Proc. ACM SOSR '19*, 2019.
[13] "Jaeger," https://www.jaegertracing.io/.
[14] "OpenTelemetry," https://opentelemetry.io/.
[15] "Kubernetes RBAC," https://kubernetes.io/docs/reference/access-authn-authz/rbac/.
[16] "Cilium," https://cilium.io.
[17] "Calico," https://docs.tigera.io/calico.
[18] "Non-RealTime RIC," https://wiki.o-ran-sc.org/display/RICNR/.
[19] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "Xi Commandments of Kubernetes Security," *Proc. 2020 IEEE SecDev*, 2020.
[20] "Powder: Platform for open wireless data-driven experimental research," https://powderwireless.net.
[21] "Tetragon," https://github.com/cilium/tetragon.
[22] "Calico cloud – container threat detection," https://docs.tigera.io/calico-cloud/threat/container-threat-detection.