

Securing 5G OpenRAN with a Scalable Authorization Framework for xApps

Tolga O. Atalay*, Sudip Maitra*, Dragoslav Stojadinovic[†], Angelos Stavrou*[†], Haining Wang*

*Department of Electrical and Computer Engineering, Virginia Tech, USA

[†]Kryptowire, LLC, McLean, VA, USA

Email: tolgaoo@vt.edu, smaitra@vt.edu, dstojadinovic@kryptowire.com, angelos@vt.edu, hnw@vt.edu

Abstract—The ongoing transformation of mobile networks from proprietary physical network boxes to virtualized functions and deployment models has led to more scalable and flexible network architectures capable of adapting to specific use cases. As an enabler of this movement, the OpenRAN initiative promotes standardization allowing for a vendor-neutral radio access network with open APIs. Moreover, the O-RAN Alliance has begun specification efforts conforming to OpenRAN’s definitions. This includes the near-real-time RAN Intelligent Controller (RIC) overseeing a group of extensible applications (xApps). The use of these potentially untrusted third-party applications introduces a new attack surface to the mobile network plane with fundamental security and system design requirements that are yet to be addressed. To secure the 5G O-RAN xApp model, we introduce the xApp Repository Function (XRF) framework, which implements scalable authentication, authorization, and discovery for xApps. We first present the framework’s system design and implementation details, followed by operational benchmarks in a production-grade containerized environment. The evaluation results, centered on active processing and operation times, show that our proposed framework can scale efficiently in a multi-threaded Kubernetes microservice environment and support a large number of clients with minimal overhead.

Index Terms—5G, OpenRAN, system security, xApps

I. INTRODUCTION

The deployment of the next generation of mobile networks is underway. Mobile network functions which existed as physical network functions (PNFs) on proprietary hardware, are now implemented as virtual network functions (VNFs) on commercial off-the-shelf (COTS) servers. This transition enables the scalable and flexible deployment of the 5G radio access networks (RAN) and 5G core (5GC) in cloud-supported virtualized environments. To that end, the OpenRAN initiative, with the O-RAN Alliance [1], [2] as a leader, proposes the disaggregation and softwarization of the RAN by employing a functionality split where virtualized logical entities host different layers of the radio stack [3]. Additionally, by steering the 5G deployment effort towards the adoption of open RAN interfaces with standardized APIs, it promotes a vendor-neutral integration of the 5G RAN.

This material is based on research sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement number HR001120C0155. The views, opinions, and/or findings contained in this article are those of the author(s) and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

Furthermore, the O-RAN architecture introduces a new software-defined networking (SDN) controller called the near real-time RAN Intelligent Controller (near-RT RIC), which hosts various RAN-related utility VNFs in the form of extensible applications (xApps). Through xApps, mobile virtual network operators (MVNOs) are offered a high degree of freedom in accessing and manipulating information in the radio stack in a software-defined manner. Every gNB within the O-RAN framework is fitted with an agent [4], allowing it to interact with the near-RT RIC using a custom access protocol [5]. Fig. 1 shows the high-level integration of O-RAN into the 5G architecture. The scenario depicts a RAN functionality split where the lower layers are implemented in the distributed unit (DU) while the upper layers are inside the central unit (CU). The O-RAN near-real time RIC is deployed at the same hierarchy as the DU, where xApps will subscribe to information from the radio stack. The management back-end of O-RAN is deployed in the central cloud along with the 5G core components.

While O-RAN adoption supports a flexible RAN deployment, it inadvertently introduces a new attack surface due to xApps being supplied by third-party vendors with various levels of trust. Moreover, given the decentralized and modular nature of 5G networks which will be developed as microservices, it is impractical to implement bulky, monolithic functionalities in a single VNF.

This creates a demand for inter-xApp communications at a large scale, which requires secure and efficient execution of cross-component authorization and discovery in addition to permissions to access infrastructure functionality. Currently, both the 3rd Generation Partnership Project (3GPP) [6] and the community implementation of OpenRAN lack the necessary security fundamentals to enable a scalable deployment.

Collaboration efforts [7], [8] have started pursuing experimental work within O-RAN [9]–[12] with focus on real-life measurements. Additionally, smaller scale testbeds have been constructed [12]–[16] using open-source solutions [17], [18] with the aim of incorporating O-RAN compatible RAN nodes. However, none of these efforts addressed the inherent lack of security mechanisms inside the O-RAN framework.

To address this security gap, we propose the **xApp repository function (XRF)** framework, a microservice-based client-to-server augmentation to the O-RAN platform, which will

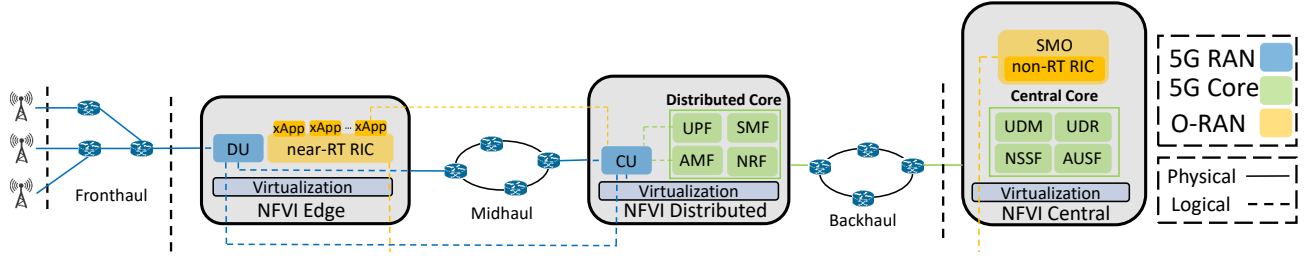


Fig. 1: High level overview of the 5G ecosystem with O-RAN augmentation

provide authentication, authorization, and discovery mechanism for xApps in the O-RAN architecture analogous to the network functions repository function (NRF) [19] in the 5GC. XRF provides a secure service discovery mechanism, facilitating the efficient interaction of xApps in a microservice architecture. In addition to maintaining xApp metadata, the XRF serves as an OpenAuthorization (OAuth) 2.0 server and distributes access tokens to xApps, enabling them to provide and consume services through secure API transactions.

We complete the system design, implementation, and testing tasks of the XRF framework, which includes the modular server-side as a centralized entity and the lightweight client-side applications adjacent to xApps. These entities are deployed on a highly-available (HA) Kubernetes cluster wrapped in a service mesh, hosted on an Openstack infrastructure, where various performance benchmarks demonstrate that the XRF framework scales efficiently in a multi-threaded environment under the microservice model of deployment.

The rest of the paper is structured as follows: Section II discusses related work and specifications regarding OpenRAN and O-RAN security. In Section III, we detail our threat model and framework requirements. Section IV describes the complete XRF system design and proposed O-RAN integration. The experimental setup followed by the performance benchmarks is presented in Section V. Finally, we conclude the paper with Section VI.

II. RELATED WORK

In this section, we summarize previous work with prototype solutions for certain issues in the O-RAN architecture and the O-RAN Security Focus Group (SFG) efforts.

O-RAN security surveys and prototypes: Several studies have been dedicated to mapping out the existing O-RAN attack surface and prototyping various security solutions independently from the specification efforts.

Authors of [20] have put together a primer, explaining the various workflows and communication interactions involved, as well as summarizing the work conducted by the O-RAN SFG [21]–[24]. A systematic breakdown of the attack surface has been conducted for O-RAN in [25] by identifying the threat vectors of known physical and virtualization layer attacks towards entities in the architecture. While it does provide an overview, specific system design issues related to certain sub-components such as xApps have not yet been analyzed. Another analysis is provided in [26] where certain privacy and mitigation strategies for cloud environments are discussed

in addition to discussing potential threats and actors. Authors conclude that the integration of O-RAN with specifically 5G will lead to an expanded attack surface.

A technical study on open fronthaul interface security is conducted in [27], where the mandatory security requirements for the management, control, user, and synchronization planes are presented. The authors then propose MACsec [28] as a candidate for securing the control user planes between the RAN entities where open interfaces are used.

O-RAN security specifications: The O-RAN development has been documented in a series of specifications by SFG.

The O-RAN specific components extend the attack surface of the 5G system defined by 3GPP. This introduces additional security challenges and risks to consider while implementing these augmentations to the existing 3GPP architecture. In [21], the O-RAN SFG outlines the implementation requirements for security protocols used in O-RAN compliant interfaces.

Furthermore, the group is working on specifying high-level security requirements and control mechanisms for O-RAN defined interfaces and network functions in [22]. However, security requirements for entities of interest maintained by O-RAN are yet to be addressed. It is stated that interfaces must provide basic information security primitives such as confidentiality, integrity, authentication and access control for services and applications.

Specifically for xApp and near-RT RIC security, key issues such as the deployment of malicious xApps or the compromise of existing xApps are identified in [29]. To address fundamental xApp security, network layer authentication solutions such as IPsec and TLS are proposed, followed by a high-level description of API authorization for xApps using the near-RT RIC as an authorization server. The proposed solution lacks specificity in how it will handle xApp authentication and authorization at a large scale. Furthermore, it creates a reliance on the near-RT RIC as a trusted entity which is required to serve as an authorization server. Such a reliance will result in the implementation of critical security functionality next to the generic management modules, which will expose the former to internal attack vectors from the latter.

In [23], roles and responsibilities are identified for maintaining and operating O-RAN systems which also includes prerequisites and assumptions to securely implement and run O-RAN defined components, interfaces, and protocols. A threat model is constructed by identifying the threat actors, describing the attack surface, identifying the vulnerabilities

and assets, and listing affected components for each threat.

A recent study summarizes the work done by the O-RAN SFG [30] while considering certain 5G related concerns mentioned in [31]. Currently, it provides a concise overview of the perspective on security from the side of the O-RAN SFG. While the O-RAN SFG has taken the preliminary steps towards formulating the attack surface and identifying certain requirements, the majority of the surface remains unmapped. Additionally, no concrete frameworks have been implemented or proposed to address the lack of security primitives in inter-xApp communication. To the best of our knowledge, this paper is the first to design, implement and analyze a scalable authentication, authorization and discovery framework for xApps that addresses the fundamental system security requirements.

III. THREAT MODEL

The O-RAN architecture builds on top of the concepts and standards laid out by 3GPP [6] for added functionality. However, new features such as virtualized functions, open interfaces, use of disaggregated COTS hardware, virtualization, and use of open-source code affect the 5G attack surface. Therefore, carefully defining the risks and scope considered for introducing a security element (i.e., the XRF) in the O-RAN architecture is crucial. Here we detail any relevant assumptions regarding the system with trusted and untrusted entities, along with the attacker model, objectives, and capabilities.

A. Assumptions

Operational and security requirements: The deployment environment of the entities is assumed to be hardened against virtualization attacks aiming to break through sandbox environments (i.e., containers, virtual machines) [32]–[35]. Our approach does not protect against side-channel attacks from neighboring sandboxes, malicious code injections during instantiation, and privilege escalation into infrastructure managing entities such as container engines and hypervisors.

Trusted entities:

- The RAN entities such as the RU, DU, and CU are equipped with the O-RAN E2 agent [5] for communicating with the near-RT RIC.
- *Management and orchestration (MANO)* entity handling resource orchestration, VNF instantiation, and infrastructure management.
- O-RAN central deployment, which includes the *service management and orchestration (SMO)* and the *non-real-time RIC (non-RT RIC)* and interfaces to the near-RT RIC

Untrusted entities:

- *MVNOs* that wish to instantiate an xApp.
- *near-RT RIC* of the O-RAN architecture which handles xApp registration and interfaces with the RAN nodes.

B. Attacker Model

Actor: Malicious xApps deployed through the MANO are seeking to subscribe to a RAN node for eavesdropping on or disrupting radio stack operations.

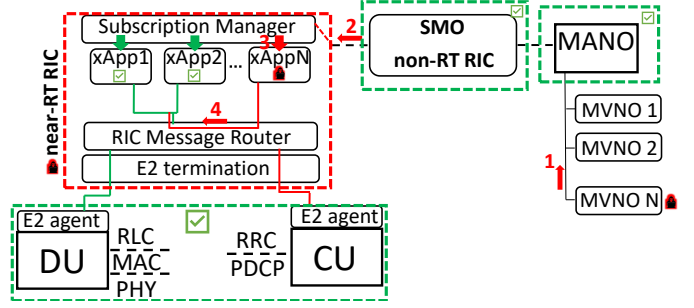


Fig. 2: Attack vector propagation in the O-RAN system with a malicious xAppN and trusted/untrusted entities

Objectives: Take advantage of the lack of proper authentication and authorization mechanisms to compromise the security of other xApps and the near-RT RIC internal communications. A malicious xApp can passively target the RAN to extract or alter sensitive information (e.g., track user location, infer identification). Moreover, a malicious xAPP can affect a RAN node’s performance or QoS requirements by exploiting resources and thereby affecting the availability of other services. As an active attacker, the malicious xApp can compromise the integrity of the data handled by legitimate xApps and perform unauthorized manipulation of the RAN through rogue signaling.

Capabilities: The xApp has access to the information available in the near-RT RIC message router (RMR) before subscribing to other endpoints in the RMR.

C. Attack Vector

Fig. 2 illustrates the attack vector used to reach the objective while delineating the trusted and untrusted entities. In the first step, a malicious MVNO will instantiate an xApp by submitting a generic cloud template to the MANO. The xApp is marked for deployment at the near-RT RIC in step 2. Next, the xApp will register with the near-RT RIC, where the messages between the near-RT RIC entities are handled through the RMR. Once the xApp is ready, it will subscribe to the desired RAN stacks through the E2 termination in the near-RT RICs and the E2 agents on the RAN nodes [4].

As a passive attacker, without any authentication or authorization method to control the access of xApps to the APIs of the critical entities, the malicious xAppN in Fig. 2 can gain access to the information intended for the other xApps through the RMR. As an active attacker, xAppN can send rogue signals to the radio stack pretending to be xApp1 or xApp2, resulting in unwanted behavior from the RAN entities.

IV. XRF FRAMEWORK

Our ultimate goal with this framework is to build a mechanism which will oversee the authentication, authorization, and discovery of xApps at a large scale. In the current microservice dominant implementation ecosystem, a given xApp will be required to consume services of other xApps for service-chaining their individual functionalities into a complete application. The XRF framework will facilitate this

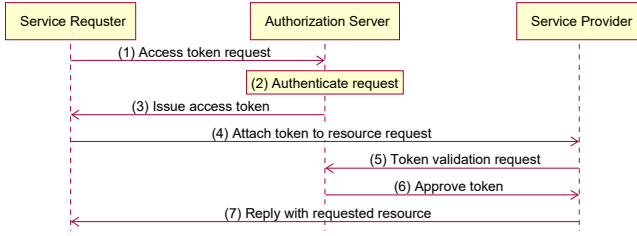


Fig. 3: OAuth 2.0 Framework

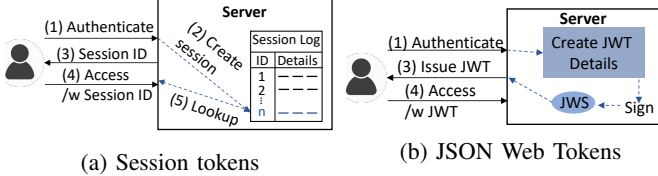


Fig. 4: Access token mechanisms

by servicing local xApps as a local metadata database and an OAuth2.0 server. It will distribute access tokens to service consuming xApps for them to securely access the APIs of service providing xApps.

In this section we first provide the reader with a brief overview of the relevant background on some of the fundamental design components that were used in the creation of the XRF framework. Next, details of the functional system design are provided followed by the message flow between server and client entities that were designed. We then dive into the non-functional details of the implementation in a production-grade environment and finally show a proposed integration of the framework into the O-RAN architecture to address the threat model described in Section III.

A. XRF Fundamentals

Here we give a brief overview of the relevant background on some of the fundamental design components that were used in the creation of the XRF framework.

1) *Open-Authorization 2.0*: OAuth 2.0 is a well-established standard where a service provider delegates authorization distribution rights to a trusted middle-man entity [36]. The main actors in the OAuth 2.0 framework are the service provider, service requester (client), service server and the authorization server. For the remainder of this paper, we will consider the service provider and the service server to be the same entity. The overall flow of the framework is given in Fig. 3. A service will start by requesting an access token from the authorization server for a target service. The server will authenticate this request and issue an access token. The service requester will use this token in accessing the API of the service provider. The provider will validate this request through the authorization server and, if the token is valid, will respond to the original request with the desired information.

2) *Access Tokens*: The OAuth 2.0 framework [36] uses access tokens [37] to authorize service requests. These tokens are typically either session tokens or JSON web tokens (JWTs) [38]. The two frameworks are depicted in Fig. 4.

For session tokens shown in Fig. 4a, the server will maintain a database of sessions with all their details and associate each session with a unique ID. It will then issue this ID to the user and they can use it on subsequent access requests. This mechanism requires the server to maintain a log of sessions and perform look-ups to validate access requests. While it is a very popular method of authorization for web servers, in a microservice deployment, due to the number of services and volume of required lookups, it is not scalable.

JWTs, which are depicted in Fig. 4b, are more suitable for this purpose. In this case, all the access details for a particular service that would be required in any future request are parameterized in JSON format and signed by the server, forming a JSON web signature (JWS). This token is issued to the user and attached to any API call that will require it.

A signed JWT contains three elements, namely – header, payload, and signature. The elements are Base64 encoded JSON key value pairs. The header contains general information such as signature algorithm (e.g., alg: RS256), unique key id (kid) associated with the token and media type (e.g., typ: JWT). Payload element contains the relevant user information such as unique identifier of the intended recipient or the service consumer (aud), expiration time of the JWT (exp), issuer (iss) of the JWT (e.g., authorization server), access type denoted as scope (e.g., read, write) and unique identifier of the token subject (sub). The issuer signs the header and the payload for the recipient to verify the authenticity of the token.

3) *Sidecar Proxies and Service Mesh*: In the design of microservices for containerized environments, the concept of sidecar proxies (SCPs) [39] has become a popular choice for abstracting non-functional requirements from the main application. This is depicted in Fig. 5.

In a Kubernetes environment, where the smallest unit of deployment is a pod, a single pod can be comprised of multiple containers. One of these containers is the microservice application which is not exposed to the outside of the pod but supported by SCP containers. These SCPs provide non-functional security as well as other reverse proxy services to the primary application such as monitoring, load balancing and various other platform abstractions. The concept of SCPs has also been adopted in 3GPP standardization for 5GC VNFs in Rel.16 as a viable method of enabling inter-VNF communication [6].

These SCPs are connected by a service mesh [40] and

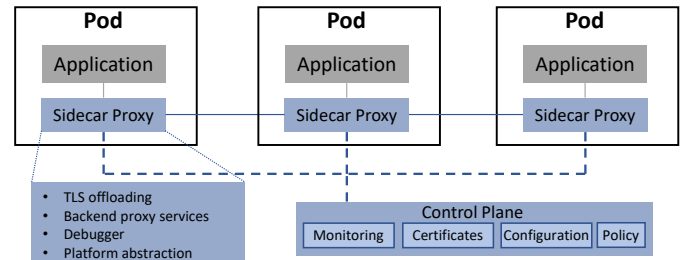


Fig. 5: Sidecar proxy utilization in containerized environments over a service mesh

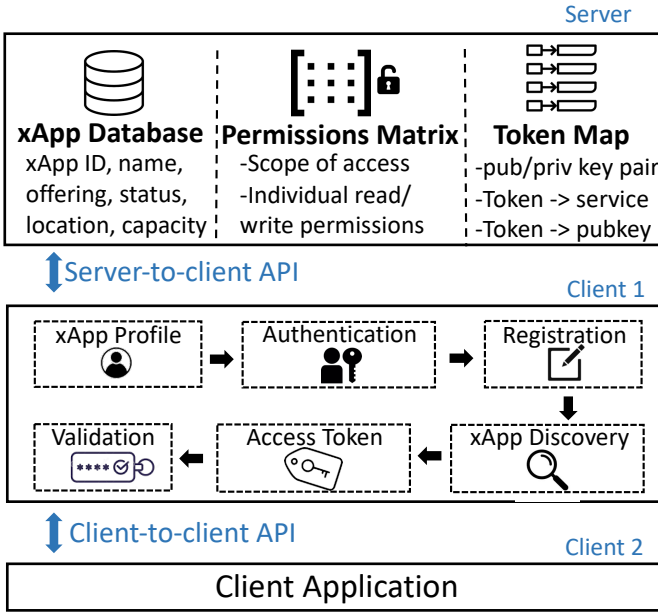


Fig. 6: XRF functional system design overview with server and client internals

share a common control plane in a Kubernetes deployment. The control plane can provide transport layer security (TLS) certificate management along with long-term monitoring, non-functional configurations and policies.

B. XRF Functional System Design

The functional entities of the XRF system are presented in Fig. 6. The framework is composed of two standalone entities, which are the XRF client module and XRF server. The high-level client flow and server internals are depicted in Fig. 6.

To enable server-to-client and client-to-client communication, representational state transfer (REST) APIs have been constructed on both the server and the client. These are described in Table I. The XRF server runs a multi-threaded HTTP server with seven distinct endpoint handlers to offer a concise collection of request and response loops necessary to handle the lifecycle operations of a client.

When it is first instantiated, the XRF client will create a metadata profile for the xApp. The profile parameters are given in Table II. They are the instance identifier, an operator-chosen descriptive name, a standardized service offering code, current

TABLE II: xApp profile metadata

Data	Description
xAppInstanceID	a universally unique identifier (UUID)
xAppInstanceName	human-readable name for the instance
xAppOffering	<i>ServiceRequestAPI</i> endpoints offered
xAppStatus	availability status of the xApp
xAppLocation	physical deployment location
xAppLoad	number of serviced xApps

status, the physical location, and finally service load indicating how many clients a given xApp is currently serving.

After the local creation of the xApp profile, the XRF client will start a sequence of lifecycle operations until it becomes service ready. First, it will create an authentication challenge for the *InitialAuthentication* endpoint of the server and receive a counter-challenge for mutual authentication. It will then transform the xApp profile metadata into JSON format and send it to the *RegistrationHandler* endpoint of the XRF server where it will be stored in the xApp database.

The XRF server will maintain the metadata of clients using an in-memory key-value storage. It will keep track of the service consumers of any registered client and in the event that the profile of a provider changes, it will notify the relevant XRF clients of xApps through their *ProfileUpdate* endpoint.

In the steps towards joining a microservice chain, the client will craft a discovery request to the XRF server *xAppDiscoveryHandler* endpoint for a target client that is advertising the desired *xAppOffering* code in the profile. To respond to a discovery request, the server will query the internal xApp database and respond with a list of eligible profiles.

The client will then formulate a request for the *AccessTokenRequest* endpoint of the server to retrieve an access token for consuming services from a target provider. This token can be attached in the header of API calls whenever required.

With token distribution, the XRF server performs access enforcement on target clients where the scope of a token is determined by an internal permissions matrix populated using the value of the *xAppOffering* parameter from Table II.

The *xAppOffering* advertises a standardized and globally comprehensible xApp functionality while also indicating the predetermined access rights allowed at the service endpoints of a client. If an incoming access token request conflicts with the recorded entry (e.g. write request to a read-only endpoint), the server will reject this request.

Furthermore, the server will use the permissions matrix to resolve any conflicting access rights so that no more than one consumer is granted critical access to specific services of the same provider at the same time. For instance, two consumers might concurrently try to write to the same data in a provider, which can result in service conflicts.

The distribution of access tokens makes the XRF server an OAuth 2.0 authorization server where the XRF client modules have a service consumer and provider back-and-forth flow. The tokens in our framework are JWTs which are bound to a unique RSA key pair where the private key is used to sign

TABLE I: XRF server and client APIs

System	API
Server	<i>POST/InitialAuthentication(Challenge)</i> → Counter
	<i>PUT/RegistrationHandler(xAppProfile)</i> → HTTP::OK
	<i>PUT/ProfileUpdateHandler(xAppData)</i> → HTTP::OK
	<i>GET/xAppDiscoveryHandler(targetProfile)</i> → List
	<i>POST/AccessTokenRequest(Func, Loc)</i> → JWT
	<i>POST/TokenIntrospection(JWT)</i> → HTTP::OK
	<i>GET/JWKSRequestHandler(keyID)</i> → token pubkey
Client	<i>PUT/ProfileUpdate(xAppData)</i> → HTTP::OK
	<i>X/ServiceRequestAPI -H Bearer "JWT"</i> → desired

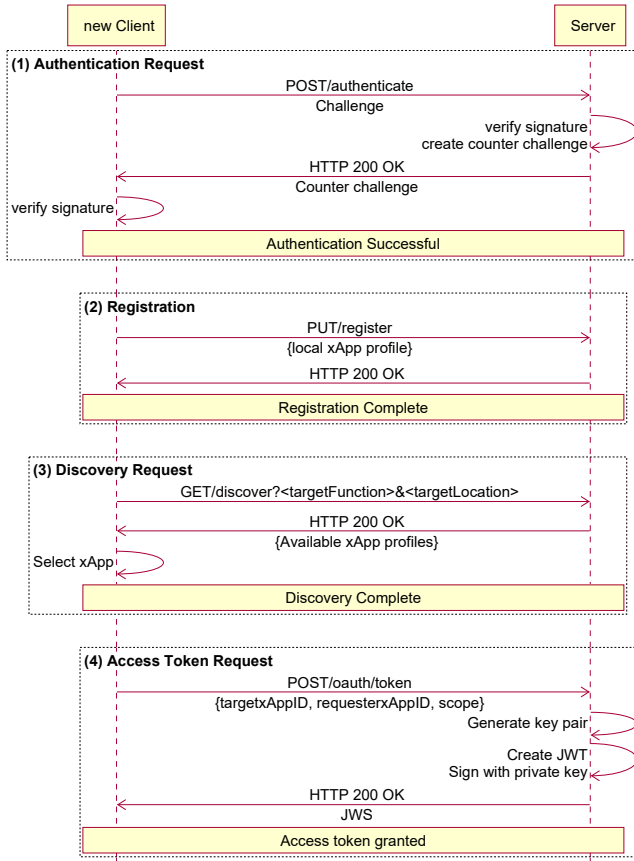


Fig. 7: Preliminary operation flow between XRF client and server showing (1) initial authentication, (2) registration, (3) discovery request, (4) access token request

the token and the public key is used to verify them.

Any client can perform token validation on a service request through either a self-contained validation using the *JWTKeyRequestHandler*, where the token is validated internally through the server, or assisted validation where the token is forwarded to the *TokenIntrospection* of the server.

C. XRF Server-Client Information Flow

The first set of messages exchanged between the XRF client and the server are shown in Fig. 7 starting with the instantiation of the client until it becomes service ready after obtaining an access token for consuming services from other client modules.

The first step the client performs after creating a profile is to authenticate (1) with the server through public key authentication.

Once the authentication is successful, the client will register (2) by sending the required metadata in a PUT request to the server. This includes the parameters described in Table II.

The client has now concluded the preliminary operations and is ready to become part of the microservice-chain. To proceed, it will query the server for a discovery (3) request for a target xApp. This request will include the desired *xAppOffering* and *xAppLocation* from Table II that the client is

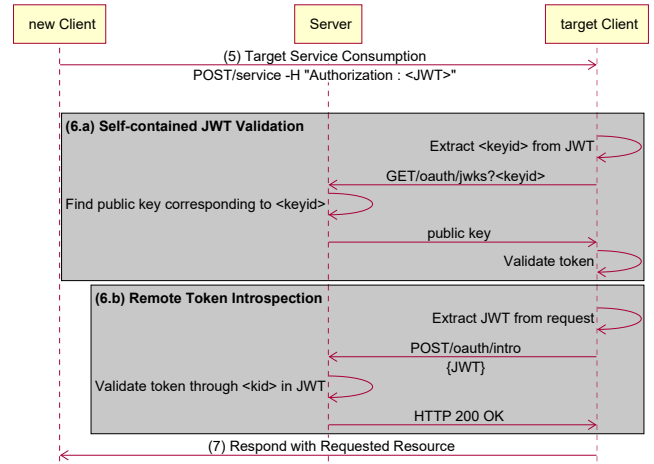


Fig. 8: (5) Service consumption from a target client of another xApp using a token bearer with (6.a) self-contained JWT validation and (6.b) remote token introspection

expecting from a *ServiceRequestAPI*. The server will respond with the available list of profiles and the client will select the candidate with the lowest current *xAppLoad* among the received profiles.

In order to consume the API of a target provider, the client will create an access token request (4) with the UUID of this target, its own UUID and the scope of the access that it wants from the chosen client. If the requested scope is allowed in the permissions matrix of the server as described in Section IV-B, a JWT will be created using a new public-private RSA key pair. The server will sign the JWT with the private key to create a JWS and deliver it to the client.

Having received the access token, the client can now request service consumption from the target XRF client of another xApp, which is depicted in Fig. 8.

The token obtained in step (4) is attached to the header of the API calls (5) destined for the XRF client of the chosen xApp. Upon receiving the API request, the client can utilize two methods of validating the token (6). These are depicted in Fig. 9.

(6.a) Self-contained JWT validation is shown in Fig. 9a. After receiving the service request (i) the target client will

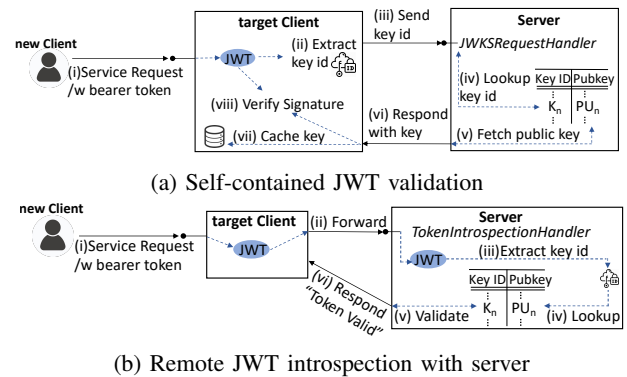


Fig. 9: Implemented access token validation mechanisms

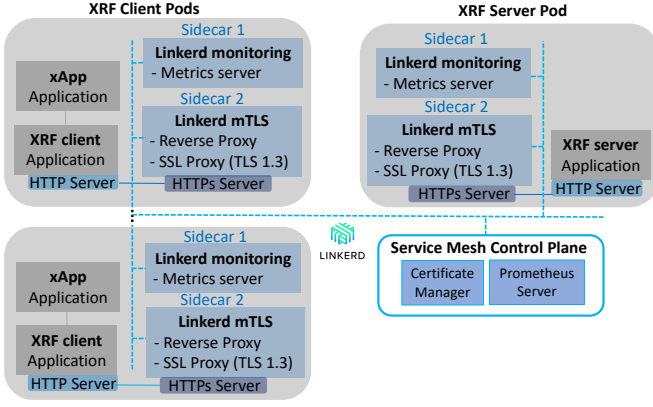


Fig. 10: Full Kubernetes deployment model with application containers and auxiliary functionality providing SCPs connected with a Linkerd service mesh

extract the key id (ii) from the header of the JWT and query the JSON web key set (JWKS) endpoint *JWKSRequestHandler* in Table I of the server with this value (iii). This key id will be internally mapped to the public key of the private key used in signing the JWT (iv-v). The XRF server will respond with this public key (vi) and the target client will cache it (vii) for future API calls with the same key id and then proceed to validate the token (viii). The advantage of this approach is that, after the first API call, the target client can perform validation on the same key id independently from the server. However, the approach expects token validation functionality to be implemented on the XRF client.

(6.b) Remote token introspection is shown in Fig. 9b. The target client will extract the JWT from the header of the HTTP request and forward it entirely to the *TokenIntrospection* endpoint of the server (ii) where the latter will validate it in the same way (iii-v) as the client in (6.a) and respond with the "token valid" message (vi).

Once the token has been validated, the target client can now respond (7) to the new client with the requested resource.

D. XRF Implementation

The implementation of the XRF server and client applications is done in C++17 in Ubuntu 20.04. Both applications are running an HTTP server with REST API endpoints crafted using the Pistache library [41]. Pistache is a mature C++ API compatible with OpenAPI [42]. To make the API accessible, we have first prepared the human-readable YAML files with the OpenAPI 3.0.0 specification and later used the OpenAPI generator to create template APIs that were populated with our custom endpoint handlers.

For handling access tokens, we used a JWT library [43] which uses OpenSSL as a primary dependency. It provides encoding and decoding of access tokens which can be used for creation, signing and validation.

The Kubernetes deployment model is given in Fig. 10. It shows the deployment of the XRF client and XRF server pods. The former is made up of four containers, where the xApp application conceptually sits in the pod back-end and only

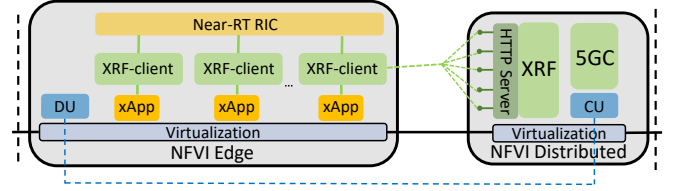


Fig. 11: Proposed XRF integration into the O-RAN ecosystem

has communication with the XRF client container. Depending on the design, this can be inter-process communication (IPC) or direct integration of the client libraries into the xApp application. The client container is running an HTTP server which is not exposed to the network but is connected to the Linkerd mutual TLS (mTLS) [40] SCP, which is serving as a back-end reverse proxy. The main use of this SCP is to serve as a secure socket layer (SSL) proxy, which encrypts the HTTP traffic from the client module with TLS 1.3 and exposes an HTTPs server to the network using self-signed certificates.

The SCPs are connected using Linkerd [40], a lightweight service mesh solution which provides a centralized monitoring server and certificate manager that rotates TLS certificates over time. To take advantage of the monitoring tool, we also deploy the Linkerd monitoring SCP which connects to the central Prometheus server to enable long-term metrics gathering.

The same implementation is mirrored on the XRF server pod where the server application is separate from the Linkerd SCPs. This model of deployment ensures that the microservice application is not burdened with implementing any non-functional security features.

A proposed integration of the XRF framework into the O-RAN and 5G architecture is presented in Fig. 11. The XRF server functionality is exposed to the XRF clients through REST API endpoints. The client module is a microservice application independent from the primary xApp. It plays the role of a middle-man entity between the xApp and the near-RT RIC, performing the operations described in Section IV-B. The information routing through the RMR, mentioned in Section III, is bounced through the XRF client pod (as shown in Fig. 10) before reaching the xApps.

This procedure requires the xApp to supply the XRF client with the metadata in Table II when it is first instantiated. From this point forward, in any xApp-to-xApp or xApp-to-near-RT RIC communication, the XRF client will behave as a proxy to provide functional security with the flow illustrated in Fig. 6.

As an entity that oversees multiple edge domains at the same time, the XRF server can be deployed in the distributed cloud. Depending on the operational requirements it can be placed in the edge network for lower latency or further centralized for a broader management scope.

V. PERFORMANCE EVALUATION

In this section, we present our experimental setup for testing with the XRF framework, followed by our findings from various performance benchmarks on the XRF client and server. First, a set of operational throughput and latency experiments

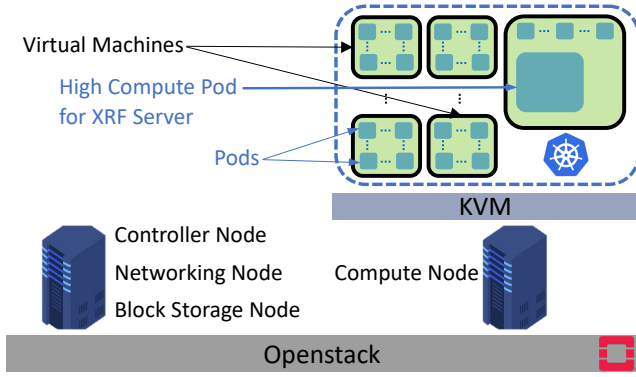


Fig. 12: Testbed with Openstack virtual machines hosting a Kubernetes cluster

are carried out in a multi-threaded environment with varying wall and CPU times. Next, micro-benchmarks are performed over the individual modules in the XRF framework.

A. Experimental Setup

Our testing environment, depicted in Fig. 12, was used to deploy XRF client and server connections at a large scale where each client is instantiated inside a separate container.

We have two servers which are running a small-scale Openstack infrastructure. One of the nodes is serving as a controller, networking and block storage node while the other serves as a compute node with the KVM hypervisor. The nodes used are two Precision 7920 Tower servers with 2 x Intel Xeon Gold 5218R 2.1GHz CPUs, 512GB RAM, 1TB disk space, and running Ubuntu 20.04.

The compute node is running a total of 11 virtual machines (VMs) for the HA-Kubernetes cluster constructed through the Ranchers Kubernetes Engine (RKE) [44]. Three VMs are control nodes and the remaining eight are workers, one of which is large (L-worker) and dedicated to the XRF server enabling all the vCPUs to prioritize server threads. The others are small workers (S-worker). The flavors are given in Table III.

TABLE III: HA Kubernetes cluster VM flavors

Node	Instances	vCPUs	RAM (GB)	Disk (GB)
Control	3	2	8	40
S-worker	7	6	16	80
L-worker	1	20	200	80

B. Results and Discussion

We use end-to-end operational throughput on the server side and end-to-end latency on the client side for performance measurements. Our raw metrics are the system wall and CPU times. The former corresponds to the total operation time and the latter is the active processing time spent across the CPUs for all the threads. The results are given in Fig. 13.

In each experiment, we instantiate the given number of concurrent clients from separate containerized environments for end-to-end benchmarking of the preliminary flows given

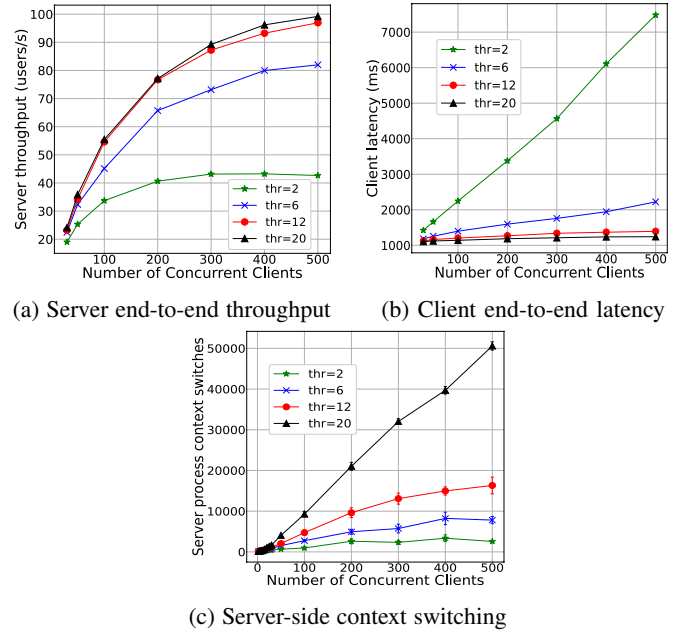


Fig. 13: Operational benchmarking of the XRF server and client in a multi-threaded environment

in Fig. 7. For measuring the server throughput, an internal wall clock timer is started inside the application with the arrival of the first user and stopped when the final user is fully processed.

We can see in Fig. 13a that increasing the number of threads running on the XRF server significantly impacts the number of users processed per second, indicating concurrency among the modules. The throughput saturates after a while regardless of the number of threads, which is a result of increased processor scheduling explained in the next paragraph. The saturation can be seen by comparing the marginal increase from 12 to 20 threads as opposed to the increase in 2 to 6 threads.

A lower-level explanation for this is provided by analyzing the context switching measurements in Fig. 13c. To obtain these numbers, we lookup the process ID (PID) of the XRF server process on the L-worker in Section V-A in the Linux file system. Later, we isolate the individual thread IDs (TIDs) belonging to this PID and record the total context switches performed across all the threads. This shows that there is a considerable increase going from 12 to 20 threads, which creates increased scheduling overhead in the processors.

The client end-to-end latency measurements given in Fig. 13b complement the throughput results in Fig. 13a. The latency is measured individually on each XRF client and averaged. With fewer threads on the XRF server, clients are suspended in scheduling, leading to higher wait times.

In a multithreaded system, the combined CPU time will be higher than the overall wall time with multi-processing taking place over a set of CPUs. In Fig. 14, we show the total client and server CPU times with a varying number of threads.

The CPU time on the client side in Fig. 14a is inversely proportional to the number of threads on the XRF server. Fewer threads cause slower scheduling for the client, resulting

in operational HTTP connections being kept open longer than necessary, leading to a slight overhead on the XRF clients. On the other hand, increasing the number of threads on the server allows for more processors to be engaged simultaneously which means the XRF server can handle tasks more efficiently, provided there are available CPUs.

To understand the difference between wall and CPU time, we compare the two for the XRF client and server in Fig. 15 for a server with 20 threads. On the client side shown in Fig. 15a, CPU time is much lower than wall time because the majority of the latency is caused by processing on the server. This shows that the XRF client is a lightweight entity with minimal overhead, enabling xApps to efficiently scale in a production-grade framework. On the server side, however, as shown in Fig. 15b, CPU time is much higher than wall time because multiple processors are engaged simultaneously to execute different handlers within the XRF server.

Furthermore, we carried out operational benchmarks for the individual modules of the XRF framework in Fig. 16 with 50 concurrent clients. The analysis given in Fig. 16a illustrates the results for the initial authentication, registration, discovery, and access token modules. We can see that initial authentication, due to the cryptography operations, consumes the most resources both on the client and server sides. For the remainder of the operations, client side CPU time is negligible because the XRF client will only supply the XRF server with the desired HTTP request. On the server side, we can see that registration uses very little time, while discovery and access token requests are more considerable. The discovery operation needs to traverse a key-value map while the access token creation module performs the creation of a JWT payload as well as permission validation on the request.

In Fig. 16b, we carry out a service request scenario with one XRF server and two XRF clients deployed where one of the clients is a service requester and the other a provider. In this experiment, the same requester sends 100 consecutive service requests to the provider with the same access token. We show the total CPU and wall time for the two token verification flows shown in Fig. 8 for the 100 requests.

Due to the overhead of contacting the XRF server each time, the wall time on the requester side is approximately twice as long with remote token introspection compared to JWKS

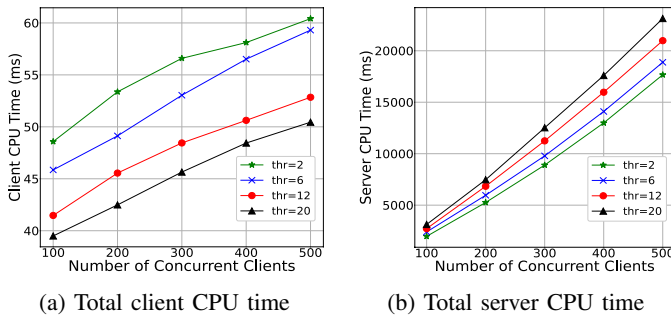


Fig. 14: XRF server and client active CPU time measurements with multiple threads

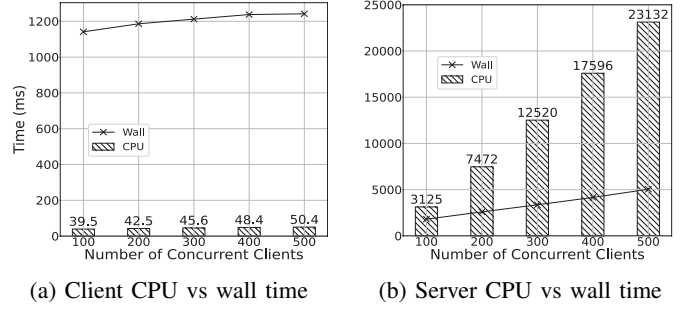


Fig. 15: Comparing CPU and wall time on the XRF client and server for an HTTP server with 20 threads

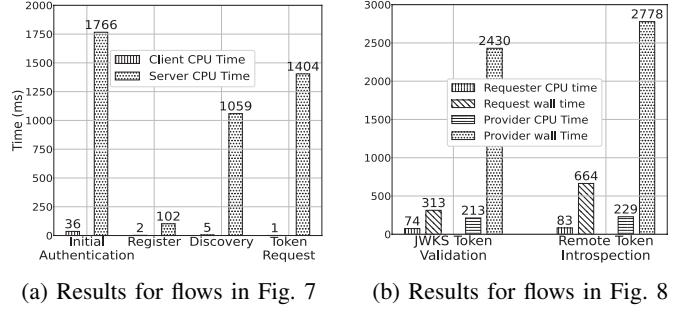


Fig. 16: Benchmarks for individual modules of XRF server and client operations

token validation. On the provider side, we can see comparable wall times since both cases use the same verification method, regardless of whether the client or the server carries it out.

VI. CONCLUSION

The OpenRAN framework is a next-generation mobile network platform aiming to achieve softwarization of the RAN and standardization of the interfaces across vendors. As with all new architectures, it expands the attack surface, requiring new security frameworks to address emerging threats. In this paper, we presented the system design, implementation, and evaluation of the XRF framework, which ensures the scalable authentication, authorization, and discovery of xApps within O-RAN, a software community realization of the OpenRAN initiative. XRF is built using proven and robust concepts tested in mobile app ecosystems, compatible with the containerized microservice model, offering seamless operational security for xApps. To fulfill the non-functional security requirements, we wrapped the deployment with the Linkerd service mesh to provide mTLS between entities. We conducted large-scale deployments in a HA Kubernetes cluster to evaluate XRF using operational benchmarks. Our results show that the XRF client has lightweight processing requirements, making it ideal for xApp adjacent deployment in a decentralized microservice environment. At the same time, the server-side handles the majority of the workload. Additionally, we demonstrate that the XRF server can scale in a multi-threaded environment while supporting concurrent clients with minimal overhead, trading off time for computing resources.

REFERENCES

- [1] "O-RAN ALLIANCE e.V," <https://www.o-ran.org/>, (Accessed on 05/31/2022).
- [2] A. Garcia-Saavedra and X. Costa-Pérez, "O-RAN: Disrupting the Virtualized RAN Ecosystem," *IEEE Communications Standards Magazine*, vol. 5, no. 4, pp. 96–103, 2021.
- [3] 3GPP, "Study on New Radio Access Technology: Radio Access Architecture and Interfaces," 3rd Generation Partnership Project (3GPP), TR 38.801 V14.0.0, Mar. 2017.
- [4] O-RAN WG3, "O-RAN Near-Real-time RAN Intelligent Controller Architecture & E2 General Aspects and Principles," O-RAN Alliance e.V., Technical Specification v02.01, Feb. 2022.
- [5] —, "Near-Real-time RAN Intelligent Controller, E2 Application Protocol (E2AP)," O-RAN Alliance e.V., Technical Specification v02.01, Feb. 2022.
- [6] 3GPP, "System architecture for the 5G System (5GS); Stage 2," 3rd Generation Partnership Project (3GPP), TR 23.501 V17.4.0, Mar. 2022.
- [7] "Commonwealth Cyber Initiative — Commonwealth Cyber Initiative — Virginia Tech," <https://cyberinitiative.org/>, (Accessed on 06/02/2022).
- [8] "Public Repository for POWDER-RENEW project / srslte-ric gitlab," <https://gitlab.flux.utah.edu/powderrenewpublic/srslte-ric>, (Accessed on 06/01/2022).
- [9] D. Johnson, D. Maas, and J. 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*, ser. WiNTECH'21. New York, NY, USA: Association for Computing Machinery, 2022, pp. 17–23. [Online]. Available: <https://doi.org/10.1145/3477086.3480842>
- [10] —, "Open Source RAN Slicing on POWDER: A Top-to-Bottom O-RAN Use Case," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 507–508. [Online]. Available: <https://doi.org/10.1145/3458864.3466912>
- [11] Y. Wang, A. Gorski, and A. P. da Silva, "Development of a Data-Driven Mobile 5G Testbed: Platform for Experimental Research," in *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, 2021, pp. 324–329.
- [12] P. S. Upadhyaya, A. S. Abdalla, V. Marojevic, J. H. Reed, and V. K. Shah, "Prototyping Next-Generation O-RAN Research Testbeds with SDRs," 2022.
- [13] L. Bonati, M. Polese, S. D'Oro, S. Basagni, and T. Melodia, "OpenRAN Gym: An Open Toolbox for Data Collection and Experimentation with AI in O-RAN," in *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, 2022, pp. 518–523.
- [14] T.-H. Wang, Y.-C. Chen, S.-J. Huang, K.-S. Hsu, and C.-H. Hu, "Design of a Network Management System for 5G Open RAN," in *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2021, pp. 138–141.
- [15] T. O. Atalay, D. Stojadinovic, A. Stavrou, and H. Wang, "Scaling Network Slices with a 5G Testbed: A Resource Consumption Study," in *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, 2022, pp. 2649–2654.
- [16] S. Niknam, A. Roy, H. S. Dhillon, S. Singh, R. Banerji, J. H. Reed, N. Saxena, and S. Yoon, "Intelligent O-RAN for Beyond 5G and 6G Wireless Networks," 2020.
- [17] L. Bonati, M. Polese, S. D'Oro, S. Basagni, and T. Melodia, "Open, Programmable, and Virtualized 5G Networks: State-of-the-art and the Road Ahead," *Computer Networks*, vol. 182, p. 107516, 2020.
- [18] A. Esmaily and K. Kralevska, "Small-Scale 5G Testbeds for Network Slicing Deployment: A Systematic Review," *Wireless Communications and Mobile Computing*, vol. 2021, 2021.
- [19] 3GPP, "Network Function Repository Services; Stage 3," 3rd Generation Partnership Project (3GPP), TS 29.510 V17.6.0, Jun. 2022.
- [20] M. Polese, L. Bonati, S. D'Oro, S. Basagni, and T. Melodia, "Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges," 2022.
- [21] O-RAN SFG, "O-RAN Security Protocols Specifications," O-RAN Alliance e.V., Technical Specification v03.00, Jul. 2022.
- [22] —, "O-RAN Security Requirements Specifications," O-RAN Alliance e.V., Technical Specification v03.00, Jul. 2022.
- [23] —, "O-RAN Security Threat Modelling and Remediation Analysis," O-RAN Alliance e.V., Technical Specification v03.00, Jul. 2022.
- [24] —, "O-RAN Security Test Specifications," O-RAN Alliance e.V., Technical Specification v02.00, Jul. 2022.
- [25] D. Mimran, R. Bitton, Y. Kfir, E. Klevansky, O. Brodt, H. Lehmann, Y. Elovici, and A. Shabtai, "Evaluating the Security of Open Radio Access Networks," 2022.
- [26] F. Klement, S. Katzenbeisser, V. Ulitzsch, J. Krämer, S. Stanczak, Z. Utkovski, I. Bjelakovic, and G. Wunder, "Open or Not Open: Are Conventional Radio Access Networks More Secure and Trustworthy than Open-RAN?" 2022.
- [27] J. Y. Cho and A. Sergeev, "Secure Open Fronthaul Interface for 5G Networks," in *The 16th International Conference on Availability, Reliability and Security*, ser. ARES 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3465481.3470080>
- [28] "IEEE Standard for Local and Metropolitan area networks-Media Access Control (MAC) Security," *IEEE Std 802.1AE-2018 (Revision of IEEE Std 802.1AE-2006)*, pp. 1–239, 2018.
- [29] O-RAN SFG, "Study on Security for Near Real Time RIC and xApps," O-RAN Alliance e.V., Technical Specification v01.00, Jul. 2022.
- [30] C. Shen, Y. Xiao, Y. Ma, J. Chen, C.-M. Chiang, S. Chen, and Y. Pan, "Security Threat Analysis and Treatment Strategy for ORAN," in *2022 24th International Conference on Advanced Communication Technology (ICACT)*, 2022, pp. 417–422.
- [31] J. Cao, M. Ma, H. Li, R. Ma, Y. Sun, P. Yu, and L. Xiong, "A Survey on Security Aspects for 3GPP 5G Networks," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 170–195, 2020.
- [32] R. Khan, P. Kumar, D. N. K. Jayakody, and M. Liyanage, "A Survey on Security and Privacy of 5G Technologies: Potential Solutions, Recent Advancements, and Future Directions," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 196–248, 2020.
- [33] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. Marvel, "Malicious co-residency on the cloud: Attacks and defense," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.
- [34] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "A Study on the Security Implications of Information Leakages in Container Clouds," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 1, pp. 174–191, 2021.
- [35] Z. Wang, R. Yang, X. Fu, X. Du, and B. Luo, "A Shared Memory based Cross-VM Side Channel Attacks in IaaS Cloud," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2016, pp. 181–186.
- [36] E. Hardt, D., "The OAuth 2.0 Authorization Framework," RFC Editor, RFC 6749, October 2012. [Online]. Available: <https://www.rfc-editor.org/info/rfc6749>
- [37] M. Jones and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage," RFC Editor, RFC 6750, October 2012. [Online]. Available: <https://www.rfc-editor.org/info/rfc6749>
- [38] N. S. M. Jones, J. Bradley, "JSON Web Token (JWT)," RFC Editor, RFC 7519, May 2015. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc7519>
- [39] Microsoft, "Sidecar Pattern - Azure Architecture Center," Jun. 2022, [Online; accessed 26. Jun. 2022]. [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar>
- [40] "What is a service mesh? — Linkerd," <https://linkerd.io/what-is-a-service-mesh/>, (Accessed on 07/15/2022).
- [41] "Github - pistacheio/pistache: A high-performance REST toolkit written in C++," <https://github.com/pistacheio/pistache>, (Accessed on 06/27/2022).
- [42] "OpenAPI Generator," <https://openapi-generator.tech/>, (Accessed on 06/05/2022).
- [43] "Github - arun11299/cpp-jwt: JSON Web Token library for c++," <https://github.com/arun11299/cpp-jwt>, (Accessed on 06/27/2022).
- [44] "Rancher Kubernetes Engine (RKE)," Jul. 2022, [Online; accessed 20. Jul. 2022]. [Online]. Available: <https://rancher.com/products/rke>