An OpenRAN Security Framework for Scalable Authentication, Authorization, and Discovery of xApps with Isolated Critical Services

Tolga O. Atalay*, Sudip Maitra*, Dragoslav Stojadinovic†, Angelos Stavrou*†, Haining Wang*
*Department of Electrical and Computer Engineering, Virginia Tech, USA
†Kryptowire, LLC, Arlington, VA, USA
Email: tolgaoa@vt.edu, smaitra@vt.edu, dstojadinovic@kryptowire.com, angelos@vt.edu, hnw@vt.edu

Abstract—The OpenRAN initiative promotes an open Radio Access Network (RAN) and offers operators fine-grained control over the radio stack. To that end, O-RAN introduces new components to the 5G ecosystem, such as the near real-time RAN Intelligent Controller (near-RT RIC) and the accompanying extensible Applications (xApps). The introduction of these entities expands the 5G threat surface. Furthermore, with the movement from proprietary hardware to virtual environments enabled by Network Functions Virtualization (NFV), attack vectors that exploit the existing NFV attack surface pose additional threats. To deal with these threats, we propose the xApp repository function (XRF) framework for scalable authentication, authorization, and discovery of xApps. To harden the XRF microservices, we isolate them using Intel Software Guard Extensions (SGX). We benchmark the XRF modules individually and compare how different microservices behave in terms of computational overhead when deployed in virtual and hardware-based isolation sandboxes. Our evaluation shows that the XRF framework scales efficiently in a multi-threaded Kubernetes environment. The isolation of the XRF microservices introduces different amounts of processing overhead depending on the sandboxing strategy. Finally, a security analysis is conducted to show how the XRF framework addresses chosen key issues from the O-RAN and 5G standardization efforts.

Index Terms—O-RAN Security, System Design, NFV co-tenancy

1 Introduction

Next-generation cellular networks are designed to accommodate a wide range of use cases with versatile Quality of Service (QoS) requirements. To sustain such a system, service offerings are carried out over a logically isolated set of radio access and compute resources denoted as network slices [1]. Each network slice is tailored to support different use cases, such as enhanced Mobile Broadband (eMBB), Ultra Reliable Low Latency Communication (URLLC), massive Internet of Things (mIoT), and a larger selection of customized Slice Service Types (SSTs) [1]created by mobile virtual network operators (MVNOs). With the continuous advancements in 5G design by the 3rd Generation Partnership Project (3GPP), research efforts increasingly focus on introducing innovations that align with established standards. One prominent area of interest is the OpenRAN initiative, led by the O-RAN Alliance.as [2].

O-RAN proposes the disaggregation of the Radio Access Network (RAN) to promote collaboration among MVNOs and accelerate the delivery of a versatile infrastructure. A key enabling factor of this movement is the adoption of Network Function Virtualization (NFV) and Software-Defined Networking (SDN) to deploy and manage 5G entities as Virtual Network Functions (VNFs). Using NFV, mobile network functions that were bound to proprietary black boxes are now deployed in cloud environments as VNFs. Leveraging NFV and SDN, OpenRAN offers various functionality splits in the RAN [3] to further enhance the flexibility of operators in building mobile networks. While the integration of O-RAN promises to deliver a more flexible and configurable architecture, the introduction of new entities along with further commitment to NFV, results in the drastic expansion of the 5G attack surface. To that end, this paper introduces the xApp Repository Function (XRF) design, a security framework to enable scalable authentication, authorization, and discovery of select O-RAN entities while defending against NFV attack vectors.

The objective of O-RAN is to create an ecosystem in which different layers of the radio stack can be hosted in separate physical locations using virtualized entities. For a vendor-neutral RAN, these disaggregated components will communicate through standardized and open APIs. To realize this goal, the O-RAN Alliance [2] standardization efforts introduce the Near-Real-Time RAN Intelligent controller (near-RT RIC), a custom SDN controller that manages a set of extensible Applications (xApps). xApps are VNFs that interact with the RAN protocol stack in a software-defined manner to offer MVNOs fine-grained near real-time capabilities in monitoring and controlling RAN flows. In the context of the gNodeB (gNB), compliance with executing the O-RAN service chain is accomplished through a specific set of augmentations that adhere to the established standards [4]. Enhanced with service models [5]–[7] and equipped with a custom agent to communicate with the near-RT RIC [8], a closed-loop control mechanism is created.

A tentative O-RAN deployment co-located with the 5G RAN and core is illustrated in Figure 1. The scenario depicts

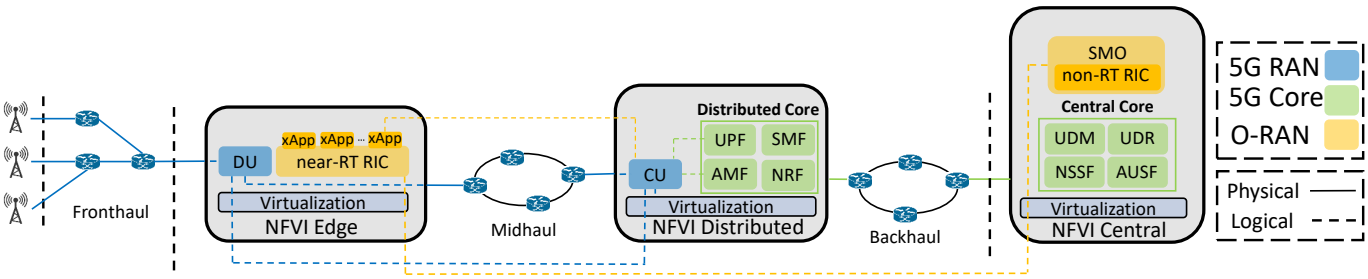


Fig. 1: High-level overview of the 5G ecosystem with O-RAN augmentation

a functionality split [9] in the 5G RAN protocol stack. As a result, the lower layers of the stack are deployed closer to the fronthaul inside the Distributed Unit (DU), while the upper layers are closer to the backhaul inside the Central Unit (CU). The O-RAN near-RT RIC is deployed next to the DU while the management back-end is deployed in the central cloud. Certain 5G core components have been deployed in a more distributed fashion and others have been centralized depending on operational requirements.

Although this approach provides a more adaptable and customizable RAN, it also leads to an expansion in the 5G attack surface due to the introduction of new entities. Especially the utilization of the xApps provided by third-party vendors with varying levels of trust requires a framework to orchestrate the secure interactions between xApps at a large scale. Meanwhile, independent groups are pursuing research [10]–[18] with a focus on machine learning-based optimization of RAN flows by leveraging the O-RAN closed-loop control. Smaller and larger scale 5G testbeds have been constructed [19]–[24] relying on open-source solutions [25]–[27] to create a high-fidelity testing environment to integrate with O-RAN compatible RAN nodes. While these efforts highlight the capabilities achievable with the O-RAN closed-loop control, they do not address the system's lack of fundamental security mechanisms.

To address this security gap, we introduce the XRF framework. The XRF framework is a client-to-server augmentation for the O-RAN platform which enables scalable authentication, authorization, and discovery of xApps at a large scale. The goal of XRF is to lay the foundation of a microservice-based security framework for the O-RAN ecosystem. This is achieved by attaching *XRF clients* as functional security Side-car Proxies (SCPs) to individual xApps upon instantiation. This design approach enables the seamless integration of the XRF security primitives into the O-RAN ecosystem. Furthermore, to improve the integration of the proposed solution with cloud-native deployments, the automated injection of this XRF client SCP is demonstrated using a Kubernetes-based service mesh.

The XRF framework uses robust security primitives to address fundamental O-RAN attack vectors. While it does provide functional security, it remains vulnerable against the NFV attack surface [28]–[31] as summarized in Figure 2 and highlighted by 3GPP [32] in a lengthy set of key issues. An attack that originates from a malicious application inside a co-resident container hosted on the same physical host as the XRF server and/or client services can use the illustrated attacks to compromise the critical security infrastructure of xApps. These attacks aim to break the virtualization boundaries by penetrating the Docker daemon [33] ❶, gaining access to the Virtual Machine (VM) guest host ❷, and escalating privileges into the hypervisor [34] ❸. This allows the attacker to target a co-resident container in the same VM or other VMs ❹ to attack their applications. To harden the XRF client and server functions against such NFV attack vectors, we refactor the building blocks into independent microservice programs and isolate them using Intel Software Guard Extensions (SGX) [35] enclaves.

Our contributions are summarized below.

- We design and implement the XRF framework for the authentication, authorization, and discovery of xApps within the O-RAN ecosystem. Using Intel SGX, the security-critical components of the XRF system are hardened against the NFV attack vectors. We compare the computational overhead introduced by SGX enclaves with other isolation methods such as VM and container sandboxes, as well as full physical isolation.
- For scalability evaluation, we use a production-grade OpenStack infrastructure hosting a Highly Available (HA) Kubernetes cluster to deploy XRF clients at a large scale. We show that the XRF client is a lightweight entity that can serve as a reverse proxy for individual xApps. On the other hand, the XRF server can efficiently trade off compute resources for scalability in a multi-threaded environment.
- For isolation evaluation, we benchmark the XRF modules in different deployment topologies to understand the overhead of virtual and hardware sandboxes. The results show that, for certain modules, the SGX enclave introduces higher overhead in exchange for hardware-based isolation.

The organization of the paper is given below. In Section 2, we provide a comprehensive background on the O-RAN and NFV attack surface as well as the key design

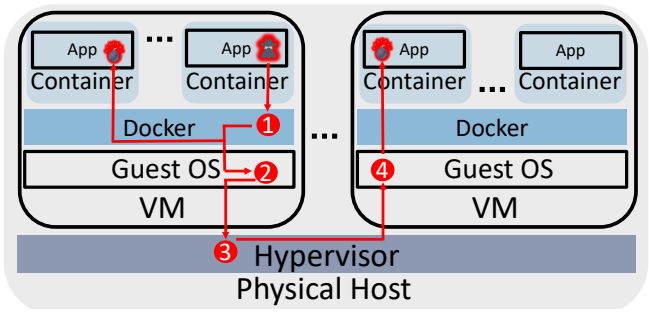


Fig. 2: NFV attack surface overview

principles used throughout the paper. Section 3 presents the threat model with different types of attacker capabilities and attack vectors. The XRF framework is explained in Section 4, starting with the functional design, the server-to-client message flow, and finally the implementation details. Next, the SGX methodology followed by the refactoring and isolation approach to the XRF microservices is presented in Section 5. The experimental setup for the scalability and isolation testing is described in Section 6 followed by the performance evaluations in Section 7. A security analysis is conducted in Section 8 to demonstrate how the XRF framework addresses several key issues from the O-RAN standardization [36] as well as NFV-related key issues from 3GPP [32]. The related work is presented in Section 9, after which the paper is concluded with Section 10.

2 Background

In this section, we provide the necessary background information on the O-RAN and NFV attack surface as well as the core design principles that are used throughout this paper. Last but not least, we summarize the isolation strategies. The acronyms in the paper are listed in Table 1.

2.1 O-RAN Attack Surface

The O-RAN architecture is vulnerable to several attack vectors, due to its open, disaggregated, and software-defined nature. The key components of O-RAN are shown in Figure 1, co-located with a 5G deployment. These key components include the Service Management and Orchestration (SMO), Near-Real-Time RAN Intelligent Controller (Near-RT RIC) hosting xApps, and disaggregated Radio Unit, Distributed Unit, and Central Unit, interact through multiple interfaces, including A1 [37], E2 [8], O1 [38] and many others. Each of these interfaces can be a potential entry point of attacks if not properly secured. The O-RAN system's reliance on virtualization and containerization adds another layer of complexity, as vulnerabilities in virtual machines or containers can lead to broader network compromises.

TABLE 1: List of Acronyms

Acronym	Description
3GPP	3rd Generation Partnership Project
5G	Fifth Generation
API	Application Programming Interface
CU	Central Unit
DU	Distributed Unit
eMBB	Enhanced Mobile Broadband
gNB	Next Generation Node B
HTTP	HyperText Transfer Protocol
JWT	JSON Web Token
KPI	Key Performance Indicator
MVNO	Mobile Virtual Network Operator
NFV	Network Functions Virtualization
near-RT RIC	near-real-time RIC
non-RT RIC	non-real-time RIC
OS	Operating System
O-RAN	Open Radio Access Network
RIC	RAN Intelligent Controller
RAN	Radio Access Network
RKE	Ranchers Kubernetes Engine
RU	Radio Unit
SCP	Sidecar Proxy
SGX	Software Guard Extensions
SMO	Service Management and Orchestration
SST	Slice Service Types
URLLC	Ultra Reliable Low Latency Communication
VM	Virtual Machine
VNF	Virtual Network Function
xApp	Extensible Application
XRF	xApp Repository Function

A primary concern within the O-RAN ecosystem is the possibility of unauthorized access to critical components, such as xApps, RU, DU, and CU. Attackers can exploit misconfigurations or weak authentication/authorization mechanisms to gain control over these entities, leading to service interruption and data leakage. These misconfigurations might include improperly set policy controls, default or weak credentials, or insufficiently secured APIs, all of which can allow attackers to bypass standard security checks. Once these vulnerabilities are exploited, attackers can gain access to sensitive resources to either take over critical services or extract sensitive data [39] The open interfaces within Open-RAN, designed for programmability and flexibility, can be targeted by such attacks if they lack robust authentication, authorization, and encryption mechanisms.

Another significant threat arises from the use of open-source software within the O-RAN components. While open-source solutions offer flexibility and innovation, they can also introduce vulnerabilities if not properly vetted and maintained. Attackers can exploit known vulnerabilities in open-source components, leading to potential breaches in the O-RAN system. Additionally, the use of AI/ML models in the Near-RT RIC and other components introduces risks related to data integrity and model manipulation.

2.2 NFV Attack Surface

The NFV attack surface is illustrated in Figure 2, highlighting the potential vulnerabilities within a typical virtual environment where a physical host running a hypervisor, which in turn hosts multiple virtual machines (VMs). Each VM contains a guest Operating System (OS) and runs containers to host various applications. There are four primary attack vectors, each marked with a red arrow and numbered accordingly, under the different levels at which an attacker could potentially exploit.

The first attack vector ❶ occurs within the containerized application environment. The vulnerabilities within an application or container are exploited, allowing an attacker to gain unauthorized access or disrupt the container's functionality. Such vulnerabilities may include outdated software libraries, misconfigured container permissions, or exposed sensitive environment variables, all of which can be exploited for mounting attacks like privilege escalation, injection attacks, or attacks based on known container runtime flaws. These exploits allow attackers to gain unauthorized access to the host system or disrupt the services running inside a container, leading to possible service outages or compromised data [30], [31], [40]. This type of attack is often caused by insecure application code, insufficiently protected container configurations, or lack of proper access controls. Inside a container, an attacker might attempt to escalate privileges within the Docker environment or exploit vulnerabilities in other containers hosted on the same VM.

The second and third attack vectors (❷ and ❸) highlight vulnerabilities within the guest operating system and the

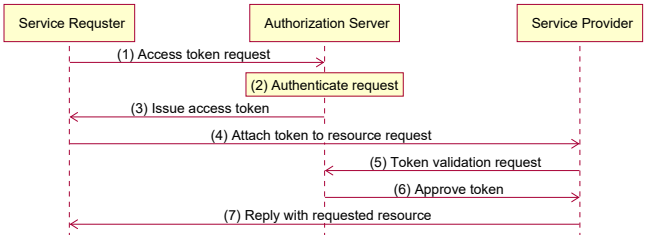


Fig. 3: Overview of OAuth 2.0 Framework

hypervisor. The second attack vector ❷ represents potential exploits within the guest OS itself, where attackers can take advantage of system vulnerabilities to gain control over the entire VM. The third attack vector ❸ represents the risks associated with the hypervisor, which manages the virtualized environment. If attackers successfully breach the hypervisor, they can potentially gain control over all VMs hosted on the physical host, leading to widespread compromise. Lastly, the fourth attack vector ❹ represents the possibility of lateral movement between VMs on the same host, where attackers, having compromised one VM, can attempt to exploit vulnerabilities to access and control other VMs, thereby expanding their reach and impact within the NFV infrastructure.

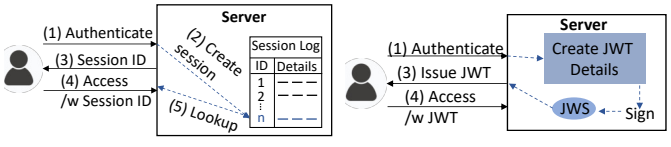
2.3 OpenAuthorization 2.0

OAuth 2.0 is a framework used by applications to delegate authorization distribution rights to a trusted independent party [41]. There are four main entities involved in the OAuth 2.0 framework, which are the service provider, service requester, service server, and authorization server. The service server and service provider are considered as a single entity throughout the remainder of this paper. The framework is illustrated in Figure 3.

The information flow is invoked when one service seeks to access the resources of another. In that case, this service requester will send an access token request ❶ to the authorization server. The authorization server will authenticate this request ❷ and issue the access token ❸. The service requester will attach this access token to any future API call made to the service provider ❹ in an attempt to access a given service endpoint resource. The provider will validate this token with the authorization server ❺ and upon approval of the token ❻ reply to the service requester with the desired information ❼.

2.4 Access Tokens

Access tokens [42] are a crucial part of the OAuth 2.0 framework. They serve as the information element in carrying authorization credentials in various forms across service



(a) Session tokens (b) JSON Web Tokens

Fig. 4: Access token mechanisms

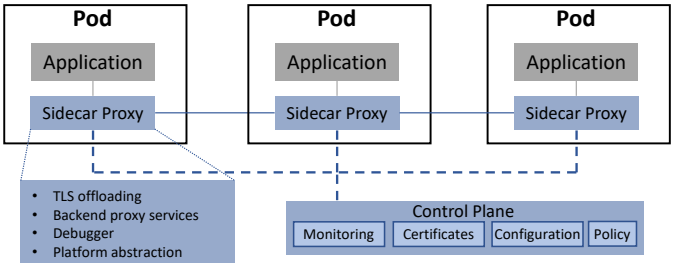


Fig. 5: Sidecar proxy utilization in containerized environments over a service mesh

providers. The two most popular access token methods are the session tokens and the JSON Web Tokens (JWTs) [43]. These are depicted in Figure 4.

In the case of session tokens shown in Figure 4a, the server maintains a database of all the sessions and performs frequent look-ups which can be inefficient for a high-volume environment. JWTs shown in Figure 4b are more appropriate for this use case. Using a JWT, the access credentials of a user are all contained inside a single information element. The contents of a JWT are parameterized in JSON format and signed by the authorization server, creating a JSON Web Signature (JWS). The service requester will present the JWS to the service provider in an HTTP request and the latter will verify it through the authorization server. A signed JWT contains three parts, elements of which are Base64 encoded JSON key-value pairs. These are the header, payload, and the signature.

2.5 Sidecar Proxies and Service Mesh

In modern-day containerized environments, applications are packaged as microservices instead of monolithic functions. To enhance the deployment of these microservices, SCPs [44] offload non-functional services from the main application as illustrated in Figure 5.

In a Kubernetes environment, the smallest unit of deployment is defined as a pod, where each pod is comprised of multiple containers, some of which carry out functional services while others serve as SCPs to support them. The SCP containers provide the application services with various platform abstractions such as security offloading, policy management, and load balancing. 3GPP has standardized the use of SCPs among VNFs as a viable method of indirect communication to enable 5G core network interactions [1]. A service mesh [45] is used to connect SCPs to a control plane in Kubernetes to transparently orchestrate the insertion of non-functional services.

2.6 Isolation Strategies

Different technologies are used to isolate applications into various virtual sandboxes as illustrated in Figure 6.

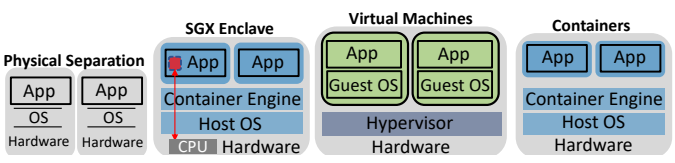


Fig. 6: Isolation strategies

The most flexible method of deployment in cloud environments is **containerization**, where applications share the same OS. This lightweight approach to isolation allows higher scalability and faster service instantiation. However, due to the shared OS, the applications are subject to a wider attack surface through the same kernel.

A stronger sandbox option is the VM, where the OS is no longer shared. This way, an attack targeting the kernel is contained within the VM environment and cannot escape to neighbor applications. Compared to containers, VM has higher resource isolation but longer instantiation times, which leads to less flexibility and efficiency in deployment.

2.7 Intel SGX

To protect against virtualization layer attacks that can penetrate VMs and containers, Intel has introduced SGX [35], [46], a hardware-based trusted execution environment that uses enclaves. An enclave is a portion of memory encrypted by the Central Processing Unit (CPU) using a hardware key. It requires the preparation of source code in a predetermined way to allow SGX-related commands to be executed. An application deployed inside an enclave is executed in a protected and limited (e.g., 92 MB) region of memory called Enclave Page Cache (EPC). Only the CPU package and by extension, the manufacturer are trusted in the SGX threat model while other privileged pieces of software (e.g., OS, BIOS, kernel, hypervisor) are untrusted.

SGX prevents virtualization layer attacks, but it has higher implementation complexity and requires the application to be adapted for deployment. Running applications inside the enclave incurs high-performance overhead, which can be narrowed down to three sources: (1) data encryption/decryption to and from the enclave, (2) accessing OS services, and (3) applications requiring more memory than the EPC size.

Finally, in case of hardware attacks that can break the SGX-enclave [47], **physical separation** can be used by placing the target service in a different physical host. This will completely stop any adversary seeking sandbox escape or side channel exploitation at the cost of efficiency, flexibility, scalability, and added capital and operational expenditure.

3 Threat Model

Our threat model considers the combined attack surface formed by both the O-RAN (§ 2.1) and NFV (§ 2.2) attack vectors. To highlight the key differences between the base O-RAN threat model and the complete O-RAN + NFV expanded threat model, we compare their environments, trusted/untrusted entities, and the actors in Table 2.

The complete threat model shown in Figure 7 includes attackers capable of exploiting the NFV attack surface in addition to the vulnerabilities within O-RAN. In the following subsections, we present our assumptions, the actors, and the attack vectors associated with this threat model.

3.1 Assumptions

In the base O-RAN threat model, all the XRF components are assumed to be deployed in a trusted **environment**. For the NFV expanded threat model, this assumption is

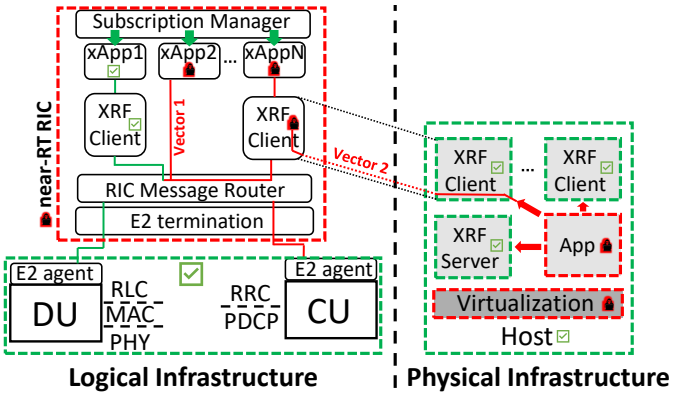


Fig. 7: Attack propagation in the O-RAN/XRF system with malicious xApps and co-residents

relaxed and the environment is one that is vulnerable to co-residency attacks [28]–[31]. These can be privilege escalation into infrastructure managing entities (e.g., hypervisor, container engine), and other *virtual* sandbox (e.g., VMs, containers) invasions.

Trusted entities are the *RAN entities* such as the RU, DU, and CU which use the E2 agent [8] to interact with the near-RT RIC. Additionally, in the physical infrastructure, we assume the *host CPU*, which encrypts SGX information elements with a hardware key, is also trusted. Last but not least, the *architectural enclave service manager daemon* is trusted.

Untrusted entities are *Malicious or inexperienced MVNOs* seeking to instantiate rogue, faulty, or misconfigured xApps on the O-RAN platform. Additionally, we consider the *near-RT RIC* of the O-RAN platform which orchestrates the communication between xApps to be untrusted. Finally, any *XRF co-resident applications* that share the virtual and/or physical infrastructure with the critical microservices of the XRF server/client are not trusted.

3.2 Attacker Model

A comparison of the attacker types is given in Table 3. Type 1 attacker exploits architectural vulnerabilities in the O-RAN platform while type 2 attacker exploits the NFV attack surface.

We consider two types of attackers in the expanded model. The *actor* of the first one corresponds to xApp2 in Figure 7 instantiated by an untrusted or inexperienced MVNO. The *objective* of this attacker is to exploit the lack of authentication and authorization within the O-RAN architecture to either passively eavesdrop on the information intended for other xApps or modify the sensitive information in the closed-loop control. Their *capability* is access to the information over the RIC message router (RMR), which can be seen by any xApp.

The *actor* of the second type of attack is the malicious application in Figure 7 sharing physical infrastructure with XRF entities. Their *objective* is to use co-residency attacks on the XRF server/client by exploiting the vulnerabilities in the virtualization infrastructure to compromise the integrity of the XRF operations. This can result in the malicious xAppN in Figure 7 being mistaken for a trusted xApp. The *capability*

TABLE 2: The comparison of threat model parameters between the O-RAN functional and NFV-expanded attack surface

	O-RAN Functional Model	NFV Expanded Model
Environment	XRF hardened against co-residency attacks	XRF vulnerable to co-residency attacks
Trusted	RAN entities, XRF infrastructure	RAN entities, Host CPU, SGX enclave manager
Untrusted	Malicious MVNOs, near-RT RIC	XRF infrastructure, Malicious MVNOs, near-RT RIC
Actors	Malicious xApps	Malicious xApps, XRF co-residents

TABLE 3: Comparison of the attacker types within the threat model illustrated in Figure 7

	Type 1 Attacker	**Type 2 Attacker**
Actor	Malicious xApp2	Malicious application on XRF infrastructure
Objectives	Exploit the lack of native authorization	Use co-residency attacks against XRF server/client
Capabilities	Has access to the information in RMR	Can break virtual sandbox boundaries

of this adversary is that they can craft attacks strong enough to break virtual sandboxes.

3.3 Attack Vectors

Attack vectors are highlighted in Figure 7. Using the first attack vector, malicious xApp2 can access the information in the RMR. It can retrieve information intended for the other xApps and perform rogue signaling to disrupt the messages sent between the RAN entities and legitimate xApps.

Ordinarily, such an attack would be thwarted by leveraging the XRF framework. However, using the second attack vector, the actor can compromise the integrity of the XRF client overseeing the authorization of xAppN or that of the XRF server. This can lead to the malicious xAppN being falsely treated with trust, thereby making the XRF framework redundant.

To address these two attack vectors, we present the XRF system design in Section 4 and the hardening of the critical XRF infrastructure in Section 5 respectively.

4 Framework Design

This section introduces the primary functional design of the XRF framework followed by the client-server message exchange and the implementation details.

4.1 Functional Blocks

The XRF framework consists of client and server modules, where the former is designed as a SCP, offloading the security flow illustrated in Figure 8, for a given xApp container. Server-to-client and client-to-client communication is facilitated by Representational State Transfer (REST) APIs described in Table 4. The XRF server is a multi-threaded HTTP server with different endpoint handlers to manage the lifecycle phases of XRF clients. Once an XRF client is instantiated, it loads a metadata profile for the xApp with the parameters in Table 5.

Following the xApp profile creation, the XRF client initiates a sequence of exchanges with the server to become service-ready. The client and server mutually authenticate each other through the `Authentication` endpoint. After parametrizing the profile into JSON key-value pairs, the client sends the xApp profile to the server using the `Registration` endpoint where the latter stores it in the xApp database using an in-memory key-value storage. The server tracks changes to the XRF client profiles of service provider xApps and notifies associated consumers through their `ProfileUpdate` endpoint.

To consume services, clients send a discovery request to the `xAppDiscovery` endpoint of the server with the desired `xAppOffering` code from Table 5 to designate a target functionality. The server responds with a list of profiles after querying the database for eligible xApps.

Acting as an OAuth 2.0 authorization server, the XRF server will distribute access tokens to clients that request one through the `TokenRequest` endpoint to consume services from their desired providers. As a policy enforcer, it will control the distribution of these access tokens based on a permissions matrix to prevent conflicting access requests. The permissions matrix contains the scope of access and read/write permission information cataloged by `xAppOffering` codes. The codes not only indicate the xApp functionality but also carry the predetermined access rights allowed by service providers. Therefore, if the server receives an access token request that asks for write access whereas the service endpoint is read-only then it will deny the token request. At the same time, it will prevent any service conflicts i.e., when two consumers try to write the same data to the same provider simultaneously.

TABLE 4: XRF server and client APIs

System	**API**
Server	`POST`/Authentication(Challenge) → Counter `PUT`/Registration(xAppProfile) → HTTP::OK `PUT`/ProfileUpdate(xAppData) → HTTP::OK `GET`/xAppDiscovery(targetProfile) → List `POST`/AccessTokenRequest(Func, Loc) → JWT `POST`/TokenIntropsection(JWT) → HTTP::OK `GET`/JWKSRequest(keyID) → token pubkey
Client	`PUT`/ProfileUpdate(xAppData) → HTTP::OK `X`/ServiceRequest -H Bearer "JWT" → desired

TABLE 5: xApp profile metadata

Data	**Description**
xAppInstanceID	Universally Unique Identifier (UUID)
xAppInstanceName	instance name for the xApp
xAppOffering	*ServiceRequest* endpoints offered
xAppStatus	availability status of the xApp
xAppLocation	physical deployment location
xAppLoad	number of serviced xApps

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2024.3522218

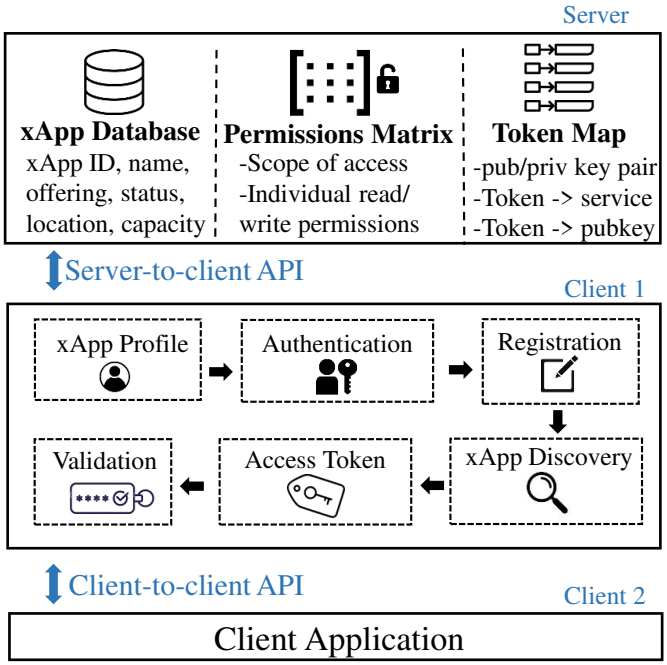


Fig. 8: XRF functional system design overview with server-client internals and entity APIs

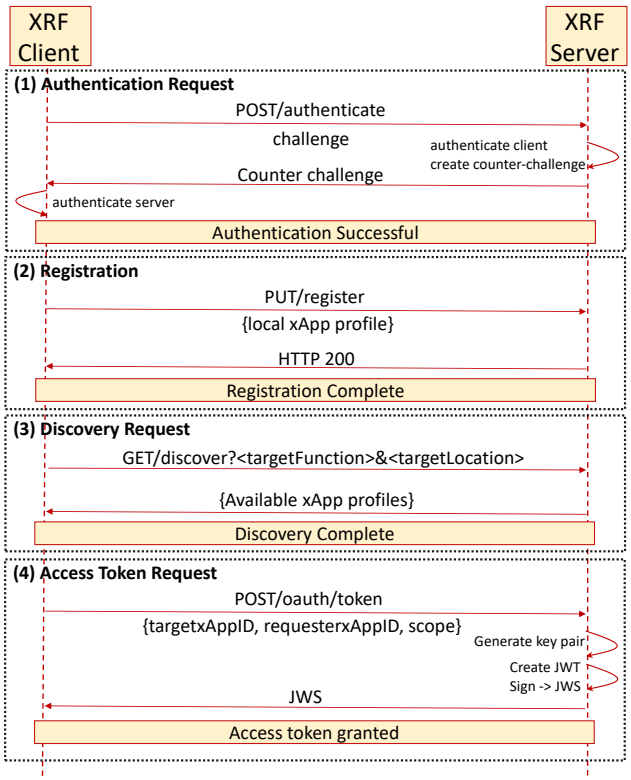


Fig. 9: Initial exchanges between XRF client and server showing ❶ initial authentication, ❷ registration, ❸ discovery request, and ❹ access token request

Once an access token request is received, the server will generate an RSA key pair to derive the associated JWT (discussed in Section 2.4) by signing it with the private key. The client receiving the token can use the public key to verify the token in one of two methods. It can obtain the public key from the server through the `JWKSRequest` endpoint and validate the token itself or verify the token through the `TokenIntrospection` endpoint of the server.

4.2 XRF Client-Server Message Exchange

Initial exchanges between a newly instantiated client and the XRF server are illustrated in Figure 9. After creating the xApp profile, the client goes through mutual authentication ❶ with the server. Upon success, it registers ❷ with the server by forwarding the profile to be stored in the xApp database. Once these steps have concluded, the client can join the microservice chain by sending a discovery request ❸ to the server with the target `xAppOffering` and `xAppLocation` codes. Once the server responds with a list of suitable xApp profiles, the client selects a provider xApp based on the parameters in Table 5.

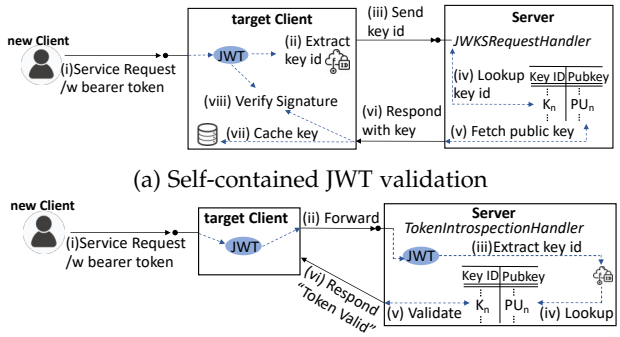
To start consuming services from other xApps, the next step is to obtain an access token from the XRF server. After choosing the provider xApp, the client generates an access token request ❹ containing its own UUID, the target xApp UUID, the scope of access (e.g., read/write) and sends it to the `TokenRequest` endpoint. By checking the permissions matrix, the server decides whether the incoming request is allowed or not. If the request is valid then the server generates a unique RSA key pair and signs the token with the private key and forwards it to the client. The server stores the key pair in the token map for future verification requests for the designated client UUID. This JWS is tied to the client, the target, and the associated scope of access, therefore cannot be used with a different target and/or a different permission scope. The client can request service from another xApp by attaching the token in the header of the API calls. Any client can verify the token either by itself or with the assistance of the server.

The first method of token verification is **self-contained JWT validation** shown in Figure 10a. Upon receiving a service request with the JWS as a bearer, the target client extracts the key ID and forwards it to the `JWKSRequest` endpoint of the server. The server internally queries the token map and responds with the public key pair based on the incoming key ID. The client uses this key to verify the token itself and caches the key for processing future requests. This method requires the token verification functionality to be implemented in the client.

In the **remote token introspection** method, depicted in Figure 10b, the client forwards the token to the `TokenIntrospection` endpoint. The server goes through the process of verifying it on behalf of the client and responds with the result. The remaining processes are identical to the self-contained validation approach. This method offloads the verification process to the server, making the client lighter. However, the server may become a bottleneck for large hyperactive networks. The target client responds to the consumer client with the requested resources after verifying the token using either method.

4.3 Implementation Details

The proposed XRF server and client are implemented as HTTP servers written in C++17 on Ubuntu 20.04. The HTTP



(a) Self-contained JWT validation

(b) Remote JWT introspection with server

Fig. 10: Implemented access token validation mechanisms

servers expose the REST API endpoints in Table 4, created with an OpenAPI [48] compatible C++ library called Pistache [49]. For access token handling, a JWT library [50] was used for the generation and verification processes.

The XRF server and client are pods in the Kubernetes deployment model depicted in Figure 11. The client pod consists of the xApp container, the XRF client application SCP, a Linkerd mutual TLS (mTLS) SCP, and a Linkerd monitoring SCP. The mTLS and monitoring SCPs are connected using the Linkerd [45] service mesh which serves as a centralized monitoring tool and certificate manager. The XRF server also follows the same deployment model, minus the xApp application.

The XRF client application is exposed to the network as an HTTPS server through the Linkerd SCP, where the SCP functions as a Secure Socket Layer (SSL) proxy, encrypting the traffic to and from the client application with TLS 1.3. The monitoring SCP collects metrics and reports to the Prometheus server in the service mesh control plane for analysis. Separating security functionalities from the main application in this manner increases maintainability and offloads the responsibility of updating security primitives.

Finally, Figure 12 demonstrates how our proposed XRF framework integrates with the O-RAN 5G architecture. In our framework, the xApp container can only communicate with the XRF client SCP as a back-end function. The pod network sandbox will bounce every HTTP request destined for the xApp from the attached XRF client to use as a

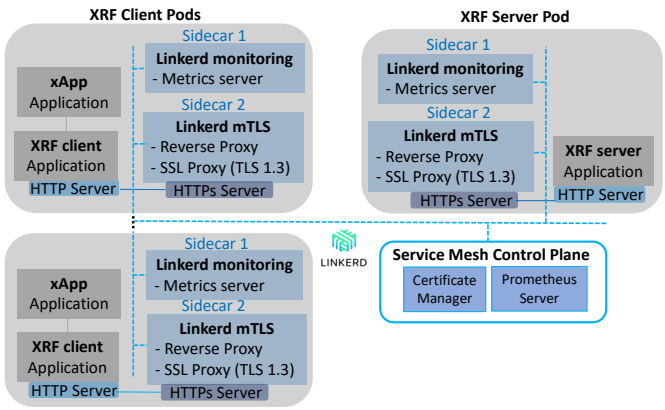


Fig. 11: Kubernetes deployment model with functional and non-functional SCPs connected with a service mesh

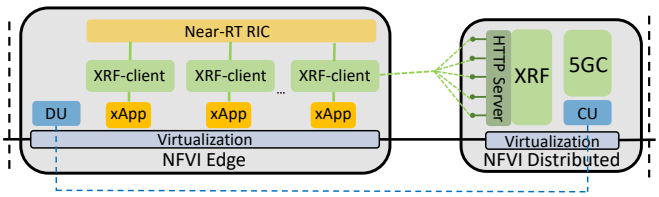


Fig. 12: XRF integration into the O-RAN ecosystem with XRF-client proxy between xApp and near-RT RIC

forwarding proxy. During this redirection, the client of the sender xApp will attach the access token to the header of the HTTP request, and the client of the receiver xApp will intercept and verify it. The client serves as the front-end proxy to the xApp back-end, providing essential security and microservice chain-related functionalities originally lacking in the O-RAN specifications. The XRF server functions as the central control plane for managing the life-cycle of clients and enforcing access control policies. The server can be deployed at the edge for lower latency requirements or positioned more centrally to serve a broader management scope. The XRF clients facilitate secure and structured xApp-to-xApp and xApp-to-near-RT RIC communication in conjunction with the XRF server.

5 Refactoring and Isolation of XRF

This section presents the refactored and isolated design of the critical XRF server modules using SGX enclaves. Finally, the deployment details of the chosen modules are discussed inside the enclaves.

5.1 Refactored Design and Isolation

The design of the XRF framework presented in Section 4 can thwart the type 1 attacker described in Section 3. Namely, it will stop the attackers that are seeking to exploit the lack of authentication and authorization mechanisms in the current O-RAN architecture.

However, for the type 2 attackers exploiting the weaknesses in the host virtualization boundaries of the XRF server and the client, this design is not sufficient. It leaves

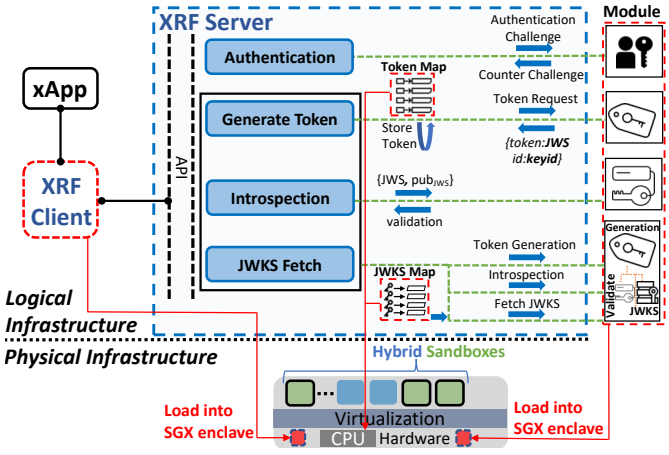


Fig. 13: Overview of the XRF server refactoring for hardware-based isolation of critical modules loaded into Intel SGX enclaves for processing

TABLE 6: The variables and functions loaded in an SGX enclave for each of the isolated modules shown in Figure 13

	Initial Authentication	Token Request	Token Introspection	All Token Handlers
Variables	pub_{client} pub/prv_{server} $challenge_{client\text{-}server}$	$profile_{client}$ $token_{client}$ $signature_{client}$	JWT_{client} JWS_{RSApub}	$JWS\text{-}profile_{client}$ $JWKS_{map}$ $JWS_{RSApair}$
Functions	RSA_{decode} RSA_{enc} $verify_{chall.}$ $create_{chall.}$	gen_{JWT} $sign_{JWT}$ $gen_{RSApair}$	JWT_{decode} JWT_{verify}	$gen\text{-}sign_{JWT}$ $gen_{RSApair}$ $fetch_{JWKS}$ $JWT_{decode\text{-}verify}$

the critical modules of the XRF server and the client vulnerable to the second attack vector shown in Figure 7.

To address this issue, we first determine the critical security services offered by the modules inside the XRF server and then refactor them into separate entities as shown in Figure 13. This figure highlights the refactored design of the XRF server modules, which are deployed inside the Intel SGX enclave for hardware-based isolation to protect against the attack vector 2 in Section 3. Furthermore, it shows an overview of the additional message exchanges between the main server and the external modules for the relevant operations.

The analysis conducted by the 3GPP regarding the security impact of virtualization in 5G [32] highlights the vulnerability of specific VNFs and microservices that deal with security primitives. The XRF server modules carrying out cryptographic and other security primitive processing operations are determined as the most critical. These are the `InitialAuthentication` and `AccessTokenHandler` subsystems. We further consider the `AccessTokenHandler` module as three sub-modules, namely the `TokenGeneration`, the `TokenIntrospection`, and the `JWKSfetch` functions.

To realize the isolation of these critical components, we first separate the source code of these modules from the monolithic XRF server and design them in a way so that they can operate as independent programs. We then deploy these in different isolation environments (e.g., Container and SGX). The separate modules still directly communicate with the XRF server to make the isolation seamless from the perspective of the XRF client and the xApp. For each module, the relevant functions and variables loaded inside an SGX enclave are given in Table 6.

For the `InitialAuthentication` module shown in Figure 14, the authentication challenge is directly forwarded to the enclave. The processing of the challenge as well as the creation of the counter challenge requires the full RSA key pair of the XRF server as well as the public key of the XRF client to be loaded into the enclave. The module will then generate the challenge inside the enclave and forward the response back to the XRF server which in turn will respond to the XRF client. This entire process will execute the RSA encryption and decryption functions inside SGX, as well as the verification of the incoming challenge and the creation of a new challenge.

After the authentication is handled by the external module, the registration and discovery operations inside the XRF server are processed outside SGX enclaves in regular virtualized environments. Following the discovery request

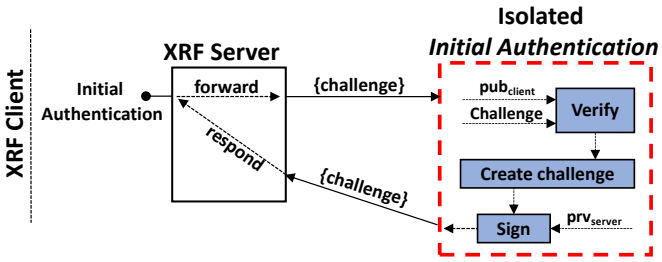


Fig. 14: Isolation of the `InitialAuthentication`

of the XRF client, the next sub-system which is engaged by the XRF client in its lifecycle is the `AccessTokenHandler` block. To isolate the sub-modules of this block, we carry out the refactoring shown in Figure 15.

The `AccessTokenGeneration` module receives the token request and creates the RSA key pair inside the enclave. Afterward, the JWT encoding and the subsequent signing to create the JWS are all carried out under SGX isolation. The JWS is forwarded to the XRF server which then sends it back to the client.

For the isolation of the `TokenIntrospection`, the validation request is forwarded directly to the external handler module. Upon receiving the token, the module loads it into the SGX enclave, decodes the JWT and verifies the signature, sending back a validation acknowledgment.

Finally, to fully secure the `AccessTokenHandler` system, the JWKS map is loaded into the enclave, where the received `JWKSFetch` request is forwarded to the enclave module for the JWKS lookup operation. The key ID is sent back to the XRF server which responds to the client.

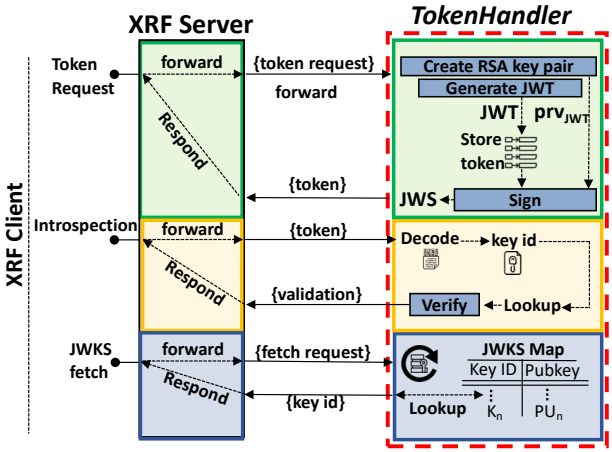


Fig. 15: Isolation of the `AccessTokenHandler`

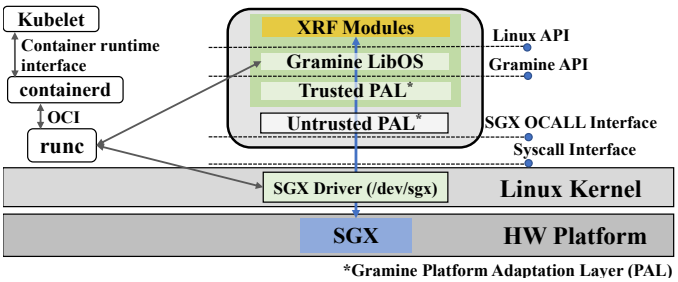


Fig. 16: Graminized container deployment with Kubernetes

The `AccessTokenHandler` enclave stores the JWKS/token map, and the RSA private keys used to sign the tokens. The computations carried out inside the enclave are the RSA key pair generation, JWT encoding, and the signing of the JWT to obtain the JWS for the `AccessTokenGenration` module. The `TokenIntrospection` has the token decoding and verification carried out under SGX isolation. Finally, the lookup function of the relevant key pair for the `JWKSFetch` of the JWKS map is carried out inside the enclave.

In addition to the XRF server modules that were discussed, the monolithic XRF client is also placed inside an enclave. Given that the operations of the client are not intensive, this is an easy design option with no refactoring requirement. In Section 7.2, we investigate the implications of different XRF module isolation options which are designated for additional security. Our goal is to understand the computational overhead that is introduced for hardened security using SGX enclaves as opposed to simply using virtual sandbox boundaries.

5.2 Isolated Module Deployment in SGX Enclave

Applications cannot be directly executed inside Intel SGX enclaves due to the presence of strict limitations. These restrictions, such as the prohibition of system calls from within the enclaves, necessitate the use of an additional layer of indirection for enclave code. A significant deterrent to SGX adoption is the engineering expertise and effort required for refactoring existing applications to run inside an enclave [51], [52].

To alleviate the development cost and effort, researchers have proposed and implemented shim layers that can run unmodified binaries on SGX such as Gramine (formerly GrapheneSGX) [52], Panoply [53], and SCONE [54]. Research has shown that employing well-designed shim layers for a partitioned application running on SGX can introduce a certain level of computational overhead. However, this overhead is deemed acceptable considering the reduction in development and verification effort achieved through such an approach [52], [54].

In this case, we have opted to use Gramine for running binaries of our refactored modules inside the SGX enclave since it is suited for a microservice-oriented framework and is completely open-source. The Gramine Shielded Container (GSC) tool is responsible for repackaging, or "graminizing," the application container image by incorporating the required LibOS [55] components and other intermediate layers. This enables the execution of the dedicated Docker container within the SGX enclave.

The enclave properties (e.g., enclave size, threads, trusted files) were configured through a manifest file during the repackaging process and the original image was re-built with an entry point instruction pointing to the application binary. The deployment model is depicted in Figure 16. The SGX driver and associated sockets were included in the standard Kubernetes deployment file for the graminized container to run inside the enclave.

6 Experimental Setup

6.1 Scalability Testing Setup

For scalability testing, we have set up an OpenStack infrastructure made up of multiple VMs which are hosting a HA Kubernetes cluster. The XRF server and clients are instantiated concurrently at a large scale with each client having its own containerized environment.

For OpenStack, we use two hosts, one serving as a controller, networking, and block storage node, and the other a compute node with the KVM hypervisor. The nodes we use are two Precision 7920 Tower servers with 2 x Intel Xeon Gold 5218R 2.1GHz CPUs, 512GB RAM, 1TB disk space, running the Ubuntu 20.04 operating system.

The OpenStack compute node is running 11 VMs to form a HA-Kubernetes cluster, constructed through the Ranchers Kubernetes Engine (RKE) [56]. Three of the VMs serve as control nodes and the remaining eight are workers with all of them running Ubuntu 18.04. For the XRF server, we have a dedicated L-worker (large worker) with higher compute resources to accurately test the multi-threading capabilities of our framework. The flavors of the OpenStack VMs are given in Table 7.

6.2 Isolation Testing Setup

Since the OpenStack host nodes don't support SGX, for isolation testing we use a smaller scale setup with a single Optiplex 7080 which has an Intel i7-10700 CPU, 32GB RAM, and 1TB disk space on Ubuntu 20.04.

To maintain the fidelity of the measurements between different isolation strategies, we use a vagrant VirtualBox setup on the same machine for VM isolation and a single-node baremetal Kubernetes cluster for deploying containers. For physical isolation, we use our baremetal node, the Poweredge T640 server with 2 x Intel Xeon Gold 6240R 2.4GHz CPUs, 768GB RAM, 3TB disk space, and the Ubuntu 20.04 operating system. The entire infrastructure is connected with a NETGEAR 28-Port 10G Ethernet switch.

7 Results and Discussion

In this section, we present our scalability and isolation evaluations using the setup described in Section 6. Using

TABLE 7: 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

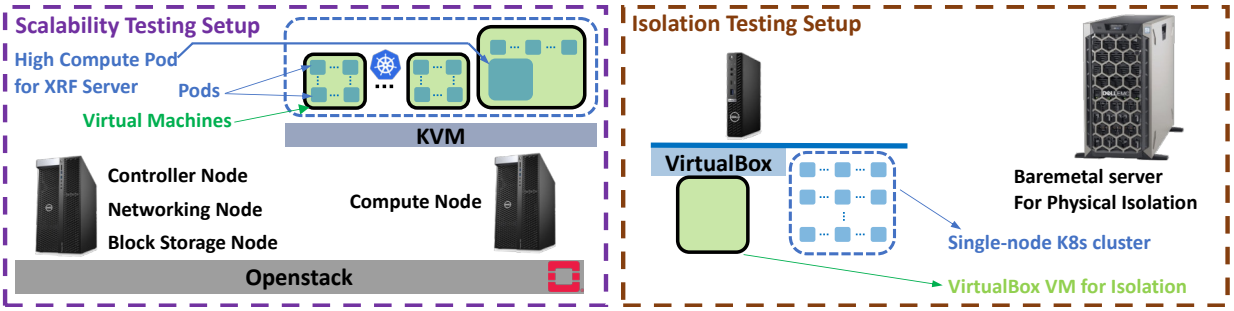


Fig. 17: Openstack VMs hosting a Kubernetes cluster for scalability testing (left) and single-host SGX-capable baremetal Kubernetes cluster with VirtualBox for isolation testing (right)

a multi-threaded environment, we measure the operational throughput and latency, where the base metrics are the CPU and wall time inside our C++ programs. The former corresponds to the total time spent across all the CPUs for a given block while the latter is real-life measured time for the task to be carried out. Our goal is to answer a series of research questions for **Scalability (S)** and **Isolation (I)** evaluations, as listed below.

(S.Q1) What are the operational bottlenecks of the functional XRF workflows? **Methodology:** Conduct stress testing on the XRF pipeline to understand their performance in multi-threaded environments. This includes CPU and wall time, as well as context-switching measurements.

(S.Q2) What are the total execution times spent on the XRF client and XRF server for fulfilling the end-to-end operational flow? **Methodology:** Compare the client/server CPU and wall time to observe which entities do most of the heavy operational lifting.

(S.Q3) What is the latency of individual XRF operations such as initial authentication, registration, discovery, and token processing? **Methodology:** Test the individual modules to understand their latency contribution to the overall pipeline flow.

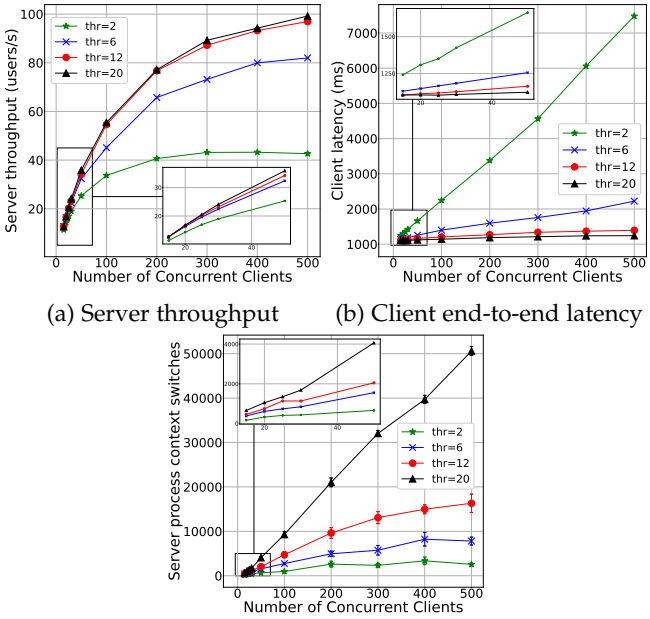
(I.Q1) What is the latency of using different isolation strategies for critical XRF modules? **Methodology:** Evaluate the latency overhead of the XRF pipeline using vanilla containerization, VM encapsulation, physical separation, and Intel SGX enclaves for the sensitive modules.

(I.Q2) What is the effect of isolation topology on the overall latency? **Methodology:** Isolate individual modules separately and together using different deployment topologies to observe the effect on XRF pipeline latency.

7.1 Scalability Testing

For each of the experiments in Figure 18, 10 to 500 users are launched concurrently from individual containers. Typical cloud-native deployments of 5G networks cater to hundreds of concurrent users per cell. By scaling our evaluations from 10 to 500 concurrent users, we aim to cover the spectrum from minimal load conditions to high-load scenarios that can occur in urban deployments. This approach ensures that our framework can be evaluated comprehensively for both under-utilized and heavily-utilized network conditions.

We first evaluate the scalability of the end-to-end system for the combined blocks of initial authentication, xApp registration, xApp discovery, and the access token request.



(a) Server throughput (b) Client end-to-end latency

(c) Server context switching

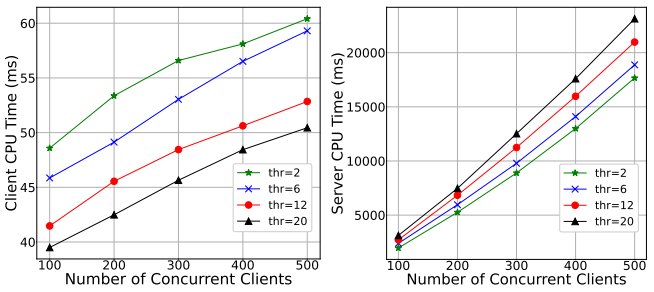
Fig. 18: Operational benchmarking of the XRF server and client in a multi-threaded environment

The results of the operational throughput and latency are given in Figure 18.

The server throughput is measured on the XRF server, where an internal wall timer is started with the arrival of the first client and then stopped with the processing of the final client. The annotation in Figure 18a shows a near-linear increasing pattern for a lower number of clients, which indicates concurrency among the server modules. However, as the client count increases, even higher thread counts start to saturate due to increased scheduling in the processor. This saturation can be observed in the difference between the 12- and 20-threaded XRF servers, especially when compared with the difference in performance between 2- and 6-threaded servers.

We understand this by analyzing the server process context switches in Figure 18c. For 10-50 clients, the total number of context switches across the processors is negligible. With the increasing number of concurrent clients, in the 100-500 range, the context switches start introducing sufficient overhead to the system to degrade the performance.

For the client end-to-end latency in Figure 18b, we can observe the same pattern both with the lower and higher



(a) Total client CPU time (b) Total server CPU time

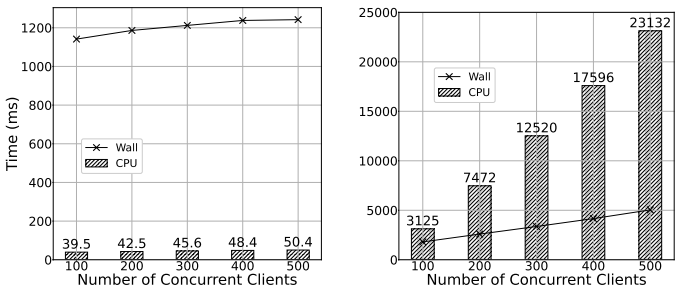
Fig. 19: XRF server and client active CPU time measurements with multiple threads

range of clients. We calculate the latency by measuring the total processing time of a single XRF client's operations served by the same server process. With more threads, fewer clients are suspended in scheduling, and operations are therefore processed faster, resulting in lower latency.

To understand the implications on actual CPU utilization, we report the total CPU and wall times for the XRF server and client in Figure 19. In the case of the XRF client (Figure 19a), we can see that with fewer threads the CPU usage grows marginally. This is because the client HTTP connections remain open slightly longer since the server-side scheduling bottleneck is higher. On the other hand, as the thread count increases, the XRF server's CPU time scales accordingly because the server can accept more concurrent connections and handle multiple clients more efficiently.

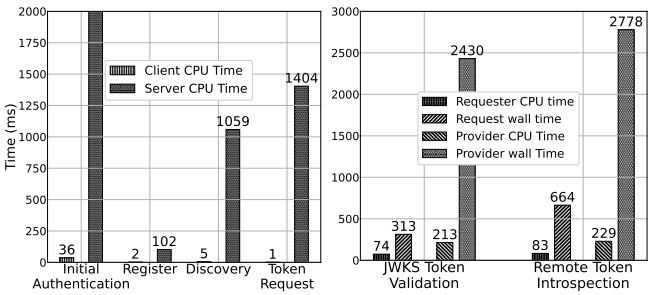
In a multi-threaded system, it is expected that the CPU time should be longer than the wall time when all the CPUs are engaged concurrently. To understand this difference in CPU and wall time for our system, we compared the two for the XRF server with 20 threads in Figure 20.

We can see that on the XRF client side, the wall time is significantly longer than the CPU time since the majority of the processing takes place on the server side. This indicates that the XRF client is a lightweight entity that can be deployed at scale adjacent to the xApps as proposed in our SCP design. For the XRF server, which executes the bulk of the operations, the increase of the CPU time outscales that of the wall time as more concurrent clients are instantiated. This shows that the XRF server modules can be engaged independently from each other for different clients and exchange compute resources for efficiency.



(a) Client CPU vs wall time (b) Server CPU vs wall time

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



(a) Results for the preliminary XRF server flows in Figure 9 (b) Results for token handler of the XRF server in Figure 10

Fig. 21: Benchmarks for individual modules of XRF server and client operations

We next perform the benchmarks on the individual modules given in Figure 21 over 50 concurrent clients to understand the impacts of the XRF microservices. The analysis in Figure 21a shows the results for the initial authentication, xApp registration, xApp discovery, and the access token generation modules. Due to the cryptography operations, the initial authentication and the access token generation have higher utilization on the server side. The access token generation module checks the permissions matrix to ensure that the requested scope of access is allowed on the target xApp. As expected, the XRF client modules have very little active CPU time compared to the server. The discovery consumes a much higher server CPU time since it performs a lookup of all the eligible xApps for a given request to generate a response.

Figure 21b shows the difference between the two token verification methods, which are illustrated in Figure 10. We designate one XRF client as a service requester and the other as a service provider. We then send 100 consecutive service requests from the former to the latter with the access token provided by the XRF server. The overhead of contacting the XRF server each time for the remote introspection causes the requester wall time to be almost twice that of the self-contained validation through JWKS lookup. We can see comparable CPU and wall times on the provider since both methods use the same underlying token verification method but with added decoding overhead in the case of remote introspection.

7.2 Isolation Testing

For isolation testing, we use different combinations of the strategies depicted in Section 5 and compare the computational overhead introduced by each approach. Each strategy shows an alternative microservice breakdown scenario of the XRF framework into separate modules as discussed in Section 5.1. The isolation testing was conducted for the topologies shown in Figure 23 across four isolation strategies depicted in Figure 6. The results were plotted against the monolithic XRF server as end-to-end latency in Figure 22 and Figure 24. The measurements are obtained from the wall time reported by our XRF client application.

The first set of analyses highlights the overhead of individually loading the modules inside the enclave. In the first topology (i.e., Figure 23a), the authentication latency doubles across all four isolation strategies due to network

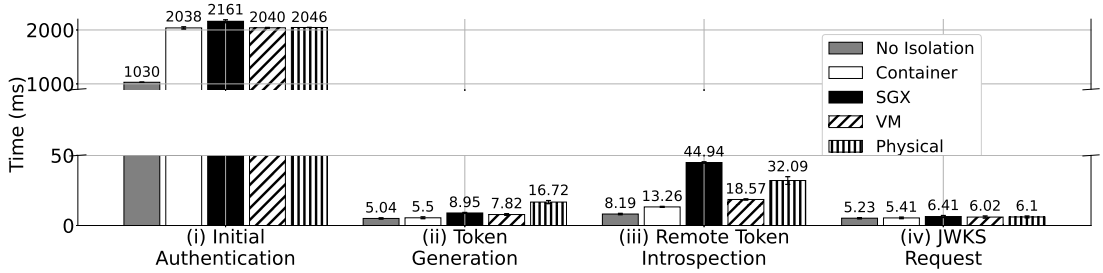
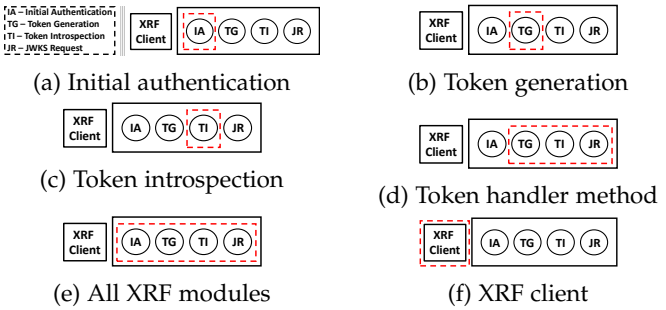


Fig. 22: Comparing the isolation sandboxing methods for individual modules of the XRF server: (i)-Figure23a, (ii)-Figure23b, (iii)-Figure23c, (iv) Measured from the topology in Figure23d

overhead as shown in Figure 22(i). For this use case, the computational overhead of loading the parameters to the enclave is higher than the network overhead of physical isolation. Isolating the token generation module (i.e., Figure 23b) does not have a significant effect as shown in Figure 22(ii) and is only noticeable when the module is physically isolated. This shows that the overhead of transferring the JWS over the network is higher than the overhead introduced by loading it into the enclave.

Using the methodology in our scalability setup, we obtain results for the two token verification mechanisms illustrated in Figure 22(iii) and (iv). The remote token introspection requires more time compared to self-contained validation through JWKS because the token verification process is offloaded to the module in this method. SGX introduces the highest overhead (i.e., 5 times the overhead of the monolithic server) among the isolation strategies. This is because each time the module receives a verification request, it leaves the enclave and incurs performance penalties discussed in Section 2.6. As opposed to token generation, the overhead of loading the variables into the enclave and processing them inside for introspection is higher than that of transferring the relevant parameters over the network to the physically isolated module.

The results in Figure 22 show that, except for the token generation module, SGX sandboxing incurs a higher operational overhead than physical isolation. While physical isolation primarily suffers from network transfer delay, the SGX's enclave loading operations incur higher overhead. Even though physical isolation provides absolute security compared to SGX, it requires additional physical hosts, which increases capital and operational expenditures and may not always be feasible. Thus, physical isolation is less attractive in flexible cloud environments than SGX which can provide hardening services against NFV attack vectors.



(a) Initial authentication (b) Token generation

(c) Token introspection (d) Token handler method

(e) All XRF modules (f) XRF client

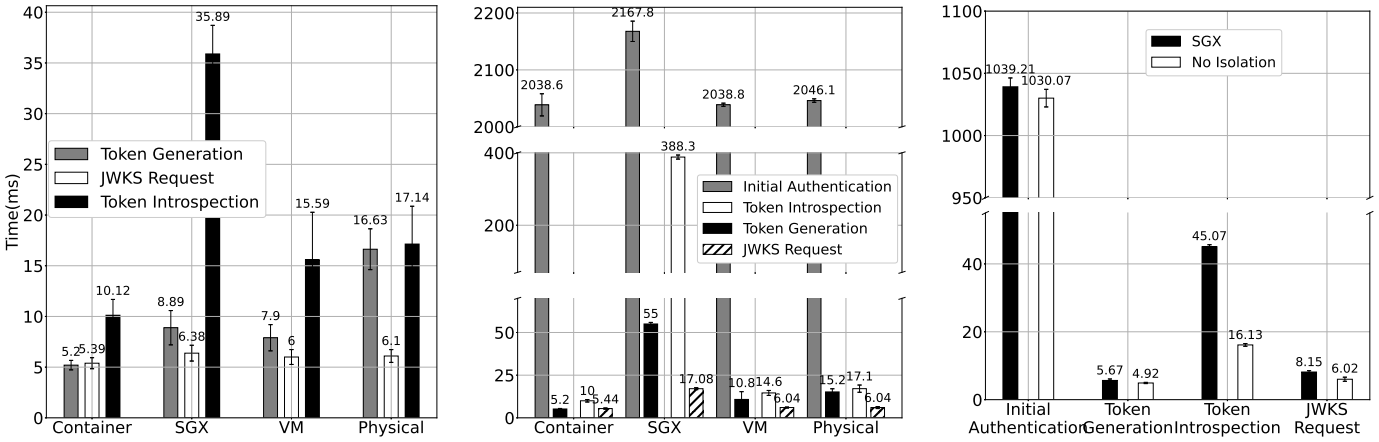
Fig. 23: Topologies used for the evaluation of the isolation strategies of the XRF framework

The results from the topology depicted in Figure 23d are presented in Figure 24a, where token generation, remote introspection, and self-contained validation through JWKS functions are isolated together as proposed in Figure 15. SGX introduces the highest overhead of the four strategies, with a total overhead almost 2.5x higher than container isolation running on the same machine. The results of individual modules are the same as those of the benchmarks in Figure 24b. This indicates that packaging the modules of the `AccessTokenHandler` system into the same enclave does not affect the SGX overhead.

All four modules are isolated together using the topology depicted in Figure 23e with the designs of Figs. 15 and 14, and the results are given in Figure 24b. Although the results for other isolation strategies show a lower impact than individual benchmarks conducted in Figure 22, SGX isolation introduces significant degradation in performance across all the modules. More specifically, with the additional execution of the `InitialAuhtentication` module inside the enclave, the isolated application binary exceeds the EPC memory size. Applications requiring larger memory than the EPC size are evicted to untrusted memory. This eviction process involves encrypting the page, flushing the translation lookaside buffer (TLB) entries and cache, and thus incurring high-performance costs. The modules that show respectively higher overhead than the other isolation strategies in Figure 22, such as the remote introspection, now incur significantly higher penalties than the more lightweight modules such as the JWKS request handler.

Finally, we show the performance of the XRF client when it is running inside the SGX enclave in Figure 23f and compare it against the vanilla deployment. Except for the token introspection, where loading the token out of the client enclave introduces non-negligible overhead, there is no significant difference. Since the XRF client will be deployed adjacent to each xApp, these results demonstrate that it can remain lightweight in deployment.

Our combined approach of scalability and isolation evaluations provides a comprehensive understanding of the XRF framework's real-world performance. The scalability tests demonstrate how the system can handle varying loads and provide a broad spectrum analysis of concurrent user impacts. The isolation evaluations, focusing on individual process performance under Intel SGX and other methodologies, highlight the security and performance trade-offs of using SGX. By integrating these two sets of results, we offer a complete picture of the system's performance



(a) Isolating `AccessTokenHandler` using the design in Figure 15

(b) Combining isolation of modules in Figure 14 and Figure 15

(c) Isolation of the XRF client without the XRF server

Fig. 24: Performance overhead of the proposed refactoring approaches: (a)-Figure23d, (b)-Figure23e, (c)-Figure23f

under realistic conditions. This methodology reflects typical deployment scenarios, making our findings relevant and applicable to real-world 5G network deployments. This dual analysis approach ensures that the framework not only meets performance requirements under typical user loads but also ensures security in virtualized deployments.

8 Security Analysis

This section describes a three-stage security analysis for the proposed XRF framework by referring to the O-RAN and 3GPP standardization documents. In the first stage of our security analysis, we focus on the security properties introduced by the XRF framework. In the second stage, we present how the XRF framework addresses the relevant Key Issues (KIs) identified by the O-RAN SFG for the near-RT RIC and xApps [36]. In the third stage, the relevant NFV threat vectors are summarized as identified by 3GPP [32] for virtualized 5G deployments. We analyze how these 3GPP KIs are addressed within the context of XRF through the isolation of critical microservices.

8.1 Security Properties

The security of the XRF framework against relevant threats lies in the inherent robustness of the fundamental methods used in the framework. XRF addresses critical security concerns through three primary mechanisms: authentication, authorization, and isolation. Each of them is summarized below.

Authentication: The XRF framework ensures that only legitimate xApps can access services by implementing mutual authentication between xApps and the XRF server. This is achieved through the use of cryptographic tokens that are securely signed by the XRF server. The strength of the cryptographic methods used provides a strong foundation for secure interactions.

Authorization: Role-based access control (RBAC) is used to govern the authorization process. Each xApp is assigned specific permissions, and the XRF server ensures that access to a service is only granted according to these roles. This limits access to only what is necessary, preventing unauthorized actions or privilege escalation. The XRF framework guarantees that no xApp can exceed its assigned capabilities.

Isolation: To defend against co-residency attacks, the XRF framework isolates critical microservices. Leveraging robust isolation (e.g., Intel SGX), XRF ensures that the critical security infrastructure cannot be compromised. Thus, any third-party software deployed next to critical functionality cannot escalate its privileges. This strong isolation model prevents lateral movement between virtualized components, thereby securing the overall XRF system.

Security of Methods: The security properties of the XRF framework are supported directly by the inherent robustness of the cryptographic techniques, RBAC, and isolation strategies employed. These methods have been extensively validated in other security-sensitive environments, and their integration within the XRF framework ensures comprehensive protection against attack vectors.

8.2 O-RAN SFG Key Issues

The development of the XRF framework addresses a set of KIs regarding the security for Near Real-Time RIC and xApps [36] standardization document published by the O-RAN alliance. All the KIs are listed in Table 8.

TABLE 8: The O-RAN standardization KIs addressed by XRF (✓: addressed directly, ▲: addressed with additional configuration, x : out of scope)

KI #	Description	XRF
1	malicious xApps deployed on near-RT RIC	✓
2	compromised xApp with misconfiguration	✓
3	conflicting interests between xApps	✓
4	isolation between xApp consumed resources	x
5	compromise of near-RT RIC ML pipelines	x
6	alteration of near-RT RIC ML pipelines	x
7	data protection and privacy	✓
8	untrusted xApp onboarding	▲
9	exploiting xApps through A1 policies	▲
10	API authentication for near-RT RIC	▲
11	API authorization for near-RT RIC	▲
12	abusing xApp IDs	✓

[KI 1] Malicious third parties can seek to onboard xApps to attain unauthorized access to the features offered by the near-RT RIC. The threat model constructed in this paper considers any third-party xApp provider as an untrusted entity. Each xApp deployment is injected with the XRF client SCP that handles authentication and authorization functionality. This decouples the security features from the near-RT RIC. Furthermore, the XRF framework prevents unauthorized monitoring or manipulation of resources that the xApp is not supposed to access.

[KI 2] An xApp can have faulty configuration or security backdoors. This can lead to weak authentication and authorization for trusted xApps and allow untrusted xApps to compromise information intended for trusted xApps. With the authentication and authorization features, the XRF framework directly addresses this KI.

[KI 3] Different xApps can attempt to modify the same target RAN parameters as a result of conflicting optimization interests. Instead of improving performance, this can further deteriorate or even disrupt user QoS/connection. The permissions matrix maintained inside the XRF server keeps a tight check on the scope of access of individual tokens to make sure that such conflicts are prevented.

[KI 4 5 6] The KIs 4-6 are outside the scope of the XRF framework. For **[KI 4]**, the goal is to make sure that the resource (e.g., virtual CPU, virtual memory) consumption of one xApp does not affect other xApps. The **[KI 5, 6]** are related to securing the machine learning data pipelines used in the near-RT RIC. All these issues fall outside the scope of the XRF framework.

[KI 7] xApps can attempt to perform unauthorized modifications on near-RT RIC databases and extract sensitive data. The XRF framework enforces an XRF client SCP between each xApp and the near-RT RIC. Hence, any data modification or extraction request needs to be properly authorized before being accepted.

[KI 8 9 10 11] The listed KIs require additional security features to be implemented on the near-RT RIC. Currently, the XRF framework deals with the authentication and authorization of xApps. While the near-RT RIC is treated as an untrusted entity in the XRF threat model, no additional configuration is added to the near-RT RIC. Our goal with the XRF framework has been to lay the foundation for a microservice-based security platform for O-RAN. Thus, the XRF pipeline has the necessary foundation for adding additional configurations to address **[KI 8 9 10 11]**. This can be done by adding a custom XRF client SCP to the individual components of the near-RT RIC deployment.

[KI 12] The xApp IDs can be misused by malicious xApps to impersonate legitimate xApps. In the XRF framework, each xApp forwards their ID to the adjacent XRF client SCP. It is indeed true that a malicious xApp can provide the XRF client with a faulty ID, which can lead to unauthorized requests. However, this will be detected within the XRF server when two xApps attempt to register with the same ID.

8.3 3GPP NFV Key Issues

In addition to the O-RAN alliance, the 3GPP has its own set of security KIs published in [32]. This document explores the security impacts of transitioning into a virtual deployment environment for 5G. The chosen KIs that affect the 5G core VNFs are directly applicable to the O-RAN ecosystem. Thus, Table 9 summarizes chosen KIs from [32] that are addressed by the isolation of critical XRF microservices.

[KI 2 5] Isolating the critical microservices of the XRF using Intel SGX makes sure that the sensitive data is protected against attacks aiming to break the virtualization boundaries. The access token map of the XRF server, as well as the JWKS map, are maintained inside secure SGX enclaves to prevent sensitive material from leaking.

[KI 6] One of the main goals of using Intel SGX isolation is to secure the sensitive service chains of the XRF framework. This prevents attackers from targeting virtual boundaries to compromise the functionalities of critical microservices such as access token handling and authentication.

[KI 7] Memory introspection attacks are a fundamental part of the NFV attack surface. The infrastructure managing entities (i.e., container engines, and hypervisors) can inspect and manipulate the memory of services running above them. With Intel SGX isolation, the sensitive data is maintained within the EPC which is encrypted. Thus, the inspection does not reveal any sensitive information.

[KI 11 12] Similarly to the 5G core and O-RAN deployments, the XRF framework is fully software-based. This means that the XRF microservices are deployed as VNFs. As a result, it is important to make sure that whenever a VNF is deployed, the physical location of that deployment can be verified. This requires the attestation of VNFs through trusted hardware. While Intel SGX provides this feature, it has not yet been implemented in this work. However, the foundation for supporting it exists and the features can be enabled with additional development.

[KI 21 25 26] The listed KIs raise concerns about an attacker capable of breaking the VM or container boundaries. With the Intel SGX isolation of the XRF server microservices and the XRF client, these attackers are thwarted.

9 Related Work

9.1 O-RAN Security Surveys and Research

Previous studies have investigated the potential risks and vulnerabilities of the O-RAN platform. Some of them have only attempted to demystify the O-RAN attack surface, while others have offered potential solutions. The most comprehensive summary of O-RAN principles, interfaces, and design has been conducted in [61], where the authors

TABLE 9: The addressed security issues from 3GPP [32] through the isolation of XRF critical microservices (✓ : addressed, ▲: addressed with additional configuration)

KI #	Description	XRF Isolation
2	sensitive data confidentiality	✓
5	sensitive data location and lifecycle	✓
6	isolating sensitive functions	✓
7	memory introspection	✓
11	location of keys and confidential data	▲
12	location of sensitive functions	▲
21	VM and hypervisor breakout	✓
25	container security	✓
26	container breakout	✓

TABLE 10: Summary of the O-RAN Security Focus Group Specifications

Sub-group	Focus
[57] Security protocols	Implementation requirements of O-RAN interfaces
[58] Security requirements	Requirements for the O-RAN control mechanisms for interfaces and functions
[36] Near-RT RIC and xApps	Vulnerabilities in the near-RT RIC and xApp interactions
[59] Threat modeling	Description of the attack surface, actors, vulnerabilities, and the target entities
[60] Tests specifications	Proper testing of security protocols and validation of proposed solutions

also briefly discussed the findings of the O-RAN security focus group (SFG) [36], [57]–[60].

A more dedicated analysis of the threat surface has been presented in [62] with a discussion of well-established virtual and physical attack vectors. Although the authors presented a comprehensive overview, no discussion or proposal of a solution regarding near-RT RIC or xApp security can be found. A similar study has been conducted in [63], where the authors focused on the effects of deploying the O-RAN components in a cloud environment and the induced potential vulnerabilities in the platform entities.

In [64], the authors pointed out the basic requirements for synchronization, management, control, and user security. They proposed MACSec as a way of securing the network layer with open APIs in the RAN. There is no discussion regarding the xApps and their interaction with each other or the near-RT RIC.

A practical study is carried out in [65], where authors explored the vulnerabilities within O-RAN deployments, focusing on the interaction between near-RT RIC and xApps. They highlighted the lack of auditing mechanisms for xApp uploads and inadequate permission management protocols. The authors implemented a functional O-RAN platform and identified critical security threats, due to the absence of mutual authentication between services in the Near-Real-Time RIC and improper permission settings for xApps. These vulnerabilities allow malicious xApps to alter the routing table and launch redirection attacks, potentially rendering the entire RAN inoperable. Leveraging our XRF design, such attack vectors can be thwarted as a result of authentication and authorized access to resources.

A pertinent study to O-RAN security is provided in [66], where the 5G-SPECTOR design is proposed for detecting Layer 3 protocol exploits in the O-RAN ecosystem. The work presents an audit framework called MOBIFLOW that transfers fine-grained cellular network telemetry, and a programmable control-plane xApp called MOBIEXPERT. Its evaluation shows that 5G-SPECTOR can detect seven types of cellular attacks in real time. This represents a significant advancement in xApp design for security purposes. Nevertheless, 5G-SPECTOR primarily focuses on leveraging O-RAN for integrating cybersecurity primitives into the 5G ecosystem rather than addressing the inherent attack vectors within the O-RAN platform.

9.2 O-RAN Security Focus Group Specifications

The O-RAN SFG is a dedicated working group within the O-RAN standardization effort to identify the attack surface of the platform and propose solutions to address the identified vulnerabilities. Given that O-RAN is a complementary effort to the 3GPP standardization of 5G mobile networks, its incorporation will expand the existing 5G attack surface with new attack vectors. We are primarily interested in the work carried out by the SFG, which is internally divided into the security protocol [57], security requirements [58], test specifications [60], threat modeling, remediation analysis [59], and near-RT RIC and xApp security [36] sub-groups. The work of these groups is summarized in Table 10.

More specifically for the near-RT RIC and xApps, the potential attack vectors that could arise in the event of malicious xApps are identified in [36]. The SFG proposes basic network layer security such as IPsec and transport layer security (TLS) as well as a basic authorization framework. However, no solution is described detailing the flows or the procedures that would be required to maintain security at a large scale. The xApp repository function [36] that has been currently defined in the standards is a co-located sub-module within the near-RT RIC that facilitates xApp selection for A1 message routing depending on the pre-existing policies. In regards to its functionality, the current proposal in the standards is completely different than the proposed XRF framework in this paper.

An overall summary of the SFG specifications can be found in [67], which takes into account the existing 5G threat model [68]. This work provides a concise summary of the O-RAN SFG effort. Although preliminary steps have been taken towards analyzing the O-RAN attack surface, much of it remains unmapped. No concrete frameworks have been designed, implemented, or evaluated to address the near-RT RIC and xApp interactions. Furthermore, no analysis is provided for the root causes that any security framework in the O-RAN platform will suffer from NFV attack vectors. To the best of our knowledge, this paper is the first to propose an end-to-end, scalable solution for authentication, authorization, and discovery of xApps while also hardening the critical security functionalities against NFV attack vectors.

10 Conclusion

Introducing the O-RAN platform into the 5G ecosystem collaterally introduces new security vulnerabilities. To address these issues, we presented the XRF framework, which orchestrates scalable authentication, authorization, and discovery among xApps. The XRF framework relies on robust security sub-systems to ensure security in inter-xApp communications. Using a microservice approach, the XRF augmentation can be carried out seamlessly to provide security for the O-RAN platform. To harden the security primitives of the XRF framework against the NFV attack surface in the flexible 5G environment, we refactored and isolated the critical modules by using Intel SGX. In our evaluations, we first tested scalability in a HA Kubernetes cluster to demonstrate that the XRF server can balance computing resources to efficiently support concurrent client operations.

We also showed that the XRF client is a lightweight entity that can be deployed adjacent to each xApp as a SCP to provide functional security, while Linkerd SCPs provide non-functional security. In isolation testing, we uncovered the computation overhead necessary for hardware-based isolation in comparison to the virtual isolation of the XRF modules. We showed that even though SGX incurs higher overhead than other isolation strategies, it is still a desirable approach considering the higher level of security provided for the critical functionalities.

REFERENCES

[1] 3GPP, "System architecture for the 5G System (5GS);Stage 2," 3rd Generation Partnership Project (3GPP), TR 23.501 V19.0.0 , Jun. 2024.

[2] "O-ran alliance e.v," https://www.o-ran.org/, (Accessed on 07/16/2024).

[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 v05.00, Feb. 2024.

[5] ——, "Near-Real-time RAN Intelligent Controller E2 Service Model (E2SM) Cell Configuration and Control," O-RAN Alliance e.V., Technical Specification v04.00, 2024.

[6] ——, "Near-Real-time RAN Intelligent Controller E2 Service Model (E2SM) KPM," O-RAN Alliance e.V., Technical Specification v05.00, 2024.

[7] ——, "Near-Real-time RAN Intelligent Controller E2 Service Model (E2SM), RAN Control," O-RAN Alliance e.V., Technical Specification v06.00, 2024.

[8] ——, "Near-Real-time RAN Intelligent Controller and E2 Interface, E2 Application Protocol (E2AP)," O-RAN Alliance e.V., Technical Specification v05.00, Feb. 2024.

[9] 3GPP, "Study of Separation of NR Control Plane (CP) and User Plane (UP) for Split Option 2," 3rd Generation Partnership Project (3GPP), TR 38.806 V15.0.0 , Mar. 2017.

[10] S. D'Oro, L. Bonati, M. Polese, and T. Melodia, "OrchestRAN: Network Automation through Orchestrated Intelligence in the Open RAN," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 270–279.

[11] 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

[12] ——, "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

[13] 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.

[14] 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.

[15] L. Bonati, M. Polese, S. D'Oro, S. Basagni, and T. Melodia, "Intelligent Closed-loop RAN Control with xApps in OpenRAN Gym," in *Proceedings of European Wireless*, 2022.

[16] A. Lacava, M. Polese, R. Sivaraj, R. Soundrarajan, B. S. Bhati, T. Singh, T. Zugno, F. Cuomo, and T. Melodia, "Programmable and Customized Intelligence for Traffic Steering in 5G Networks Using Open RAN Architectures," *IEEE Transactions on Mobile Computing*, 2023.

[17] R. Gangula, A. Lacava, M. Polese, S. D'Oro, L. Bonati, F. Kaltenberger, P. Johari, and T. Melodia, "Listen-While-Talking: Toward dApp-based Real-Time Spectrum Sharing in O-RAN," *arXiv preprint arXiv:2407.05027*, 2024.

[18] Y. Chen, Y. T. Hou, W. Lou, J. H. Reed, and S. Kompella, "OM3: Real-Time Multi-Cell MIMO Scheduling in 5G O-RAN," *IEEE Journal on Selected Areas in Communications*, 2023.

[19] L. Bonati, M. Polese, S. D'Oro, S. Basagni, and T. Melodia, "Open-RAN 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.

[20] 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.

[21] 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.

[22] T. O. Atalay, D. Stojadinovic, A. Famili, A. Stavrou, and H. Wang, "Network-Slice-as-a-Service Deployment Cost Assessment in an End-to-End 5G Testbed," in *GLOBECOM IEEE Global Communications Conference*. IEEE, 2022, pp. 2056–2061.

[23] T. O. Atalay, D. Stojadinovic, A. Stavrou, and H. Wang, "Scaling Network Slices with a 5G Testbed: A Resource Consumption Study," in *IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2022, pp. 2649–2654.

[24] T. O. Atalay, A. Famili, D. Stojadinovic, and A. Stavrou, "Demystifying 5g traffic patterns with an indoor ran measurement campaign," in *GLOBECOM 2023-2023 IEEE Global Communications Conference*. IEEE, 2023, pp. 1185–1190.

[25] 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.

[26] A. Esmaeily and K. Kralevska, "Small-Scale 5G Testbeds for Network Slicing Deployment: A Systematic Review," *Wireless Communications and Mobile Computing*, vol. 2021, 2021.

[27] D. Villa, I. Khan, F. Kaltenberger, N. Hedberg, R. S. da Silva, S. Maxenti, L. Bonati, A. Kelkar, C. Dick, E. Baena *et al.*, "X5G: An Open, Programmable, Multi-vendor, End-to-end, Private 5G O-RAN Testbed with NVIDIA ARC and OpenAirInterface," *arXiv preprint arXiv:2406.15935*, 2024.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 3GPP, "Study on Security Impacts of Virtualisation," 3rd Generation Partnership Project (3GPP), TR 33.848 V18.0.0 , 2023.

[33] S. Chamoli *et al.*, "Docker Security: Architecture, Threat Model, and Best Practices," in *Soft Computing: Theories and Applications*. Springer, 2021, pp. 253–263.

[34] A. Saeed, S. A. Hussain, and P. Garraghan, "Cross-VM Network Channel Attacks and Countermeasures Within Cloud Computing Environments," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[35] V. Costan, I. Lebedev, S. Devadas *et al.*, "Secure processors part i: background, taxonomy for secure enclaves and intel sgx architecture," *Foundations and Trends® in Electronic Design Automation*, vol. 11, no. 1-2, pp. 1–248, 2017.

[36] O-RAN SFG, "Study on Security for Near Real Time RIC and xApps," O-RAN Alliance e.V., Technical Specification v02.00, Mar. 2023.

[37] O-RAN WG2, "A1 Interface: Application Protocol," O-RAN Alliance e.V., Technical Specification v04.02, 2024.

[38] O-RAN WG10, "O-RAN O1 Interface Specification," O-RAN Alliance e.V., Technical Specification v13.00, 2024.

[39] S. Sicari, A. Rizzardi, and A. Coen-Porisini, "5G In the Internet of Things Era: An Overview on Security and Privacy Challenges," *Computer Networks*, vol. 179, p. 107345, 2020.

[40] T. Madi, H. A. Alameddine, M. Pourzandi, and A. Boukhtouta, "NFV Security Survey in 5G Networks: A Three-dimensional Threat Taxonomy," *Computer Networks*, vol. 197, p. 108288, 2021.

[41] E. Hardt, D., "The OAuth 2.0 Authorization Framework," RFC Editor, RFC 6749, October 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6749

[42] 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

[43] 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

[44] Microsoft, "Sidecar Pattern - Azure Architecture Center," May 2023, [Online; accessed 16. May. 2023]. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

[45] "What is a service mesh? | Linkerd," https://linkerd.io/what-is-a-service-mesh/, 2023, (Accessed on 05/16/2023).

[46] V. Costan and S. Devadas, "Intel SGX Explained," Cryptology ePrint Archive, Paper 2016/086, 2016, https://eprint.iacr.org/2016/086. [Online]. Available: https://eprint.iacr.org/2016/086

[47] A. Nilsson, P. N. Bideh, and J. Brorsson, "A Survey of Published Attacks on Intel SGX," *arXiv preprint arXiv:2006.13598*, 2020.

[48] "OpenAPI Generator," https://openapi-generator.tech/, 2023, (Accessed on 06/05/2023).

[49] "Github - pistacheio/pistache: A high-performance REST toolkit written in C++," https://github.com/pistacheio/pistache, 2023, (Accessed on 06/02/2023).

[50] "Github - arun11299/cpp-jwt: JSON Web Token library for c++," https://github.com/arun11299/cpp-jwt, 2023, (Accessed on 06/02/2023).

[51] A. Hasan, R. Riley, and D. Ponomarev, "Port or Shim? Stress Testing Application Performance on Intel SGX," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 123–133.

[52] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 645–658.

[53] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux Applications With SGX Enclaves." in *NDSS*, 2017.

[54] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[55] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and Security Isolation of Library OSes for Multi-Process Applications," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592812

[56] "Rancher Kubernetes Engine (RKE)," May 2023, [Online; accessed 20. May. 2023]. [Online]. Available: https://rancher.com/products/rke

[57] O-RAN SFG, "O-RAN Security Protocols Specifications," O-RAN Alliance e.V., Technical Specification v03.00, Jul. 2022.

[58] ——, "O-RAN Security Requirements Specifications," O-RAN Alliance e.V., Technical Specification v03.00, Jul. 2022.

[59] ——, "O-RAN Security Threat Modelling and Remediation Analysis," O-RAN Alliance e.V., Technical Specification v03.00, Jul. 2022.

[60] ——, "O-RAN Security Test Specifications," O-RAN Alliance e.V., Technical Specification v02.00, Jul. 2022.

[61] M. Polese, L. Bonati, S. D'Oro, S. Basagni, and T. Melodia, "Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges," 2022.

[62] 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.

[63] F. Klement, S. Katzenbeisser, V. Ulitzsch, J. KrÃ¤mer, 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.

[64] 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

[65] C.-h. Tseng, C.-F. Hung, B.-K. Hong, and S.-M. Cheng, "On Manipulating Routing Table to Realize Redirect Attacks in O-RAN by Malicious xApp," in *International Symposium on Wireless Personal Multimedia Communications (WPMC)*. IEEE, 2023, pp. 288–292.

[66] H. Wen, P. Porras, V. Yegneswaran, A. Gehani, and Z. Lin, "5G-Spector: An O-RAN Compliant Layer-3 Cellular Attack Detection Service," in *Network and Distributed System Security Symposium (NDSS)*, 2024.

[67] 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.

[68] 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.

Tolga O. Atalay received a PhD degree in computer engineering from Virginia Tech in 2024. He is currently a senior research and development engineer at Kryptowire Labs, a mobile cybersecurity company based in Northern Virginia, USA. His research interests include 5G and beyond networks, cloud computing, and system security.

Sudip Maitra received a BSc degree in electrical and electronic engineering from Ahsanullah University of Science and Technology, Bangladesh in 2016 and MSc degree from Central Michigan University, USA in 2020. He is currently pursuing a PhD degree with the Department of Electrical and Computer Engineering, Virginia Tech, USA. His research interests include hardware-based isolation, 5G networks, and system security.

Dragoslav Stojadinovic received a PhD degree in electrical engineering from Rutgers University in 2022. He is currently a senior research & development engineer at Kryptowire Labs, USA, and an external research consultant with Virginia Tech, USA. His research interests include software-defined radio systems, spectrum usage efficiency, and 5G testing environments.

Angelos Stavrou received a PhD degree in computer science from Columbia University in 2007. He is currently a professor with the Department of Electrical and Computer Engineering at Virginia Tech, USA. His research interests include large systems security & survivability, intrusion detection systems, privacy & anonymity, and security for MANETs and mobile devices.

Haining Wang (Fellow, IEEE) received the PhD degree in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 2003. He is currently a professor with the Department of Electrical and Computer Engineering at Virginia Tech, USA. His research interests include security, networking systems, cloud computing, and cyber-physical systems.