Technology
Arts Sciences
TH Köln

Comparative Analysis of Open-Source Security Monitoring Tools for Kubernetes-Based 5G Core Networks

Sharyu Rajiv Chavan

Research Project
MSc. Communications Systems and Networks

Lecturer: Prof. Dr. Andreas Grebe

March 10, 2026

Abstract

In this research, two open-source security monitoring tools—Falco and Kubescape—are compared in a Kubernetes-based 5G core network architecture. On a K3s cluster, a containerized 5G core with Open5GS and UERANSIM is set up. Falco is used for runtime threat detection, and Kubescape is used to evaluate configuration compliance and security posture. The tools are assessed according to their integration effort, performance overhead, coverage, and detection capacity. The findings show how runtime behavioral monitoring and preventive configuration analysis differ when it comes to protecting cloud-native 5G core deployments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

The introduction of 5G technology has greatly accelerated the evolution of mobile communication networks. 5G is intended to offer a variety of services, such as ultra-reliable low-latency communication, improved mobile broadband, and massive machine-type connectivity, in contrast to earlier generations that primarily concentrated on raising data rates. The 5G core network has been developed utilizing cloud-native principles to accommodate these cutting-edge services.

Containerization and orchestration technologies like Kubernetes are being used more and more in the deployment of modern 5G core networks. Flexible scaling, quicker upgrades, and better resource usage are made possible by this transition from hardware-based infrastructure to software-based, cloud-native deployment. Microservices operating within containers are being used to implement network functions that were previously implemented on dedicated hardware.

However, new security issues are brought about by this architectural change. Environments that are containerized are complicated and dynamic. Vulnerabilities can be caused by exposed APIs, insecure container settings, and Kubernetes misconfigurations. Critical elements of the 5G core network may also be compromised by runtime attacks such privilege escalation, illegal shell access, and malicious process execution.

Securing this environment is crucial since the 5G core controls network connectivity and sensitive subscriber data. For cloud-native architectures, conventional perimeter-based security measures are insufficient. The efficacy of specialist Kubernetes security monitoring solutions in safeguarding such deployments must thus be assessed.

In this project, a realistic Kubernetes-based 5G core testbed is used to compare two open-source technologies, Kubescape and Falco.

1.2 Problem Statement

A dual security challenge that has not received sufficient attention in the literature arises from the deployment of 5G core networks on Kubernetes. Cloud-native deployments inherit all of Kubernetes' security weaknesses — misconfigured pods, overly permissive RBAC roles, missing network policies, and exposed service account tokens. In the 5G context, these weaknesses carry domain-specific consequences: an exposed service account token in an AMF pod gives an attacker direct API access to MongoDB subscriber credentials, while the absence of NetworkPolicy objects allows a compromised UPF to reach every other network function in the cluster without restriction.

Previous studies treat these issues in isolation. Research on 5G security concentrates on protocol-level threats such as NAS vulnerabilities and signaling manipulation, ignoring the Kubernetes infrastructure on which modern 5G cores run. Research on Kubernetes security assesses tools against generic cloud workloads, without accounting for the domain-specific behaviour of 5G network functions — such as the UPF requiring NET_ADMIN capabilities for TUN interface management or the AMF spawning child processes during registration bursts — which can cause legitimate activity to resemble attack signatures.

No empirical, side-by-side comparison of configuration-based and runtime-based security monitoring tools in a Kubernetes-hosted 5G core environment currently exists. It is unknown which tool addresses which threat category, where each tool's blind spots lie, what performance overhead each introduces to live 5G core operations, and whether the two tools are redundant or complementary in this specific domain. This study directly fills that gap.

1.3 Research Objective

Given the absence of empirical evidence comparing these tools in the 5G domain, the main objective of this project is to conduct a comparative evaluation of Kubescape and Falco in securing a Kubernetes-based 5G core network.

The project aims to:

- Deploy a containerised 5G core using Open5GS and evaluate its security posture.
- Simulate RAN and UE behaviour using UERANSIM to generate realistic 5G traffic.
- Assess Kubescape's ability to identify Kubernetes misconfigurations in a 5G deployment.
- Evaluate Falco's runtime threat detection capability against controlled attack scenarios.

- Measure the performance overhead introduced by each tool during live 5G core operation.
- Determine whether the two tools are redundant or complementary in this specific context.

1.4 Research Questions

The study addresses the following research questions:

- How accurately does Kubescape identify Kubernetes misconfigurations in a 5G core deployment?
- To what extent does Falco detect runtime threats using its default rule set?
- Which attack categories remain undetected by each tool, and why?
- What performance overhead does each tool introduce to live 5G core operations?
- Does combined deployment provide detection coverage that neither tool achieves individually?

1.5 Scope and Limitations

The security monitoring layer of a 5G core network hosted by Kubernetes is the subject of this study. The scope is defined by the following boundaries:

In Scope:

- Open5GS (AMF, SMF, UPF, UDM, UDR, PCF, AUSF, NRF, SCP) deployment and configuration on a Kubernetes cluster.
- Data plane traffic, PDU session establishment, and UE registration are all simulated using UERANSIM.
- Kubescape checks for network policy gaps, RBAC problems, and misconfigurations.
- Falco runtime monitoring for ten specific attack scenarios.
- Performance measurements are collected and analyzed using Prometheus and Grafana.

Out of Scope:

- Physical radio access network (RAN) security is outside the scope.
- 5G signaling-layer attacks (such as NAS protocol exploits).
- Mesh security and multi-cluster.

- Anomaly detection using machine learning.

Limitations:

- The testbed is a three-node K3s cluster; results at production scale with hundreds of nodes and thousands of concurrent UE registrations may differ.
- Attack simulations are scripted and controlled; real-world attackers may use more varied techniques or multi-stage approaches.
- Open5GS is an open-source implementation; commercial 5G core deployments may have additional hardening requirements.
- Results are specific to Kubescape 1.30.4 and Falco 0.43.0; subsequent releases may alter detection rates or introduce new false positives.

1.6 Ethical Considerations

All attack simulations conducted in this research were performed exclusively within a self-contained, isolated three-node K3s cluster with no connection to any production network or live subscriber data. No real user equipment, real SIM credentials, or production 5G infrastructure was involved at any stage.

The subscriber IMSI, authentication keys, and operator codes used in the testbed are test values defined by 3GPP for research purposes (MCC=999).

All attack tools were deployed and executed solely within the open5gs-vulnerable and open5gs namespaces of the isolated testbed. The research follows responsible disclosure principles: no vulnerabilities discovered during this study were exploited outside the controlled environment, and all findings are reported to assist the security community in improving cloud-native 5G deployments.

Chapter 2

Related Work

2.1 Security in 5G Core Networks

Numerous perspectives have been used to examine the security of 5G core networks. According to research by [2], the normative security architecture for 5G systems specifies network slice isolation, mutual authentication using 5G-AKA, and subscriber identity protection using SUCI. Nevertheless, the security of the cloud-native infrastructure that the 5G core is installed on is not covered by these protocol-level security measures.

Vulnerabilities in 5G signaling have been the subject of several investigations. Research on NAS protocol exploitation has shown that faulty registration, impersonation attacks, and denial of service Even on 5G networks, request messages and subscriber location tracking are still possible [15]. These attacks are not covered by Kubernetes-level security mechanisms like Falco and Kubescape since they operate at the protocol layer.

In order to analyze 5G behavior in controlled conditions, academic research has looked at deploying 5G core networks utilizing open-source platforms as Open5GS [3] and free5GC. Nevertheless, there hasn't been a thorough assessment of these installations' security posture, particularly with regard to the Kubernetes configuration vulnerabilities brought about by their Helm charts and default manifests.

2.2 Kubernetes Security Monitoring

Following high-profile breaches, the security environment of Kubernetes has drawn significant attention. A thorough framework addressing pod security, network policies, RBAC, logging, and supply chain security is offered by the NSA-CISA Kubernetes Hardening Guidelines [7]. The CIS Kubernetes Benchmark [8] offers comprehensive, prescriptive rules for hardening each cluster component. Kubescape's control library is built upon

these frameworks.

The threat model for containerized workloads, the use of Linux security primitives (namespaces, capabilities, seccomp), and the significance of least-privilege configuration are all covered in detail by Rice and Hausenblas [11] .The theoretical foundation for the misconfiguration scenarios assessed in this study is established by their work.

2.3 Runtime Security with eBPF

The industry standard for runtime security monitoring is the usage of eBPF. The fundamental treatment of eBPF and BPF performance tools is given by Gregg [10], who shows that eBPF-based instrumentation adds very little overhead in comparison to previous methods like kernel modules or strace. By adding a rule engine to the richer syscall stream created by eBPF, Falco [5] expands on this foundation and makes it possible to identify container escape attempts, sensitive file access, and unusual network activity with high confidence.

Previous evaluations of Falco have mostly concentrated on general Kubernetes workloads. These studies have confirmed detection accuracy and rule coverage for common attack scenarios including shell spawning inside containers and credential leakage. Nevertheless, no previous research has assessed Falco in the context of 5G network function behavior, where normal processes like the SMF and AMF display syscall patterns that might coincide with attack signatures.

2.4 Gap Identified

The literature currently in publication treats Kubernetes security and 5G security as distinct fields. By implementing and assessing configuration-based and runtime-based security monitoring tools in a Kubernetes-hosted 5G core environment, no previous empirical study integrates both. This study closes that gap by offering a precise, quantitative comparison of Kubescape and Falco in this particular setting, with findings linked to actual 5G network function behavior and subscriber registration activities.

Chapter 3

Background and Fundamentals

3.1 5G Core Architecture

3.1.1 5G Core Networks

The 3rd Generation Partnership Project (3GPP) defines the 5G Core (5GC) in Release 15 and later versions. The 5GC differs significantly from the 4G evolved packet core (EPC) by adopting a service-based architecture (SBA), where all network functions (NFs) communicate via a common framework using HTTP/2 and RESTful APIs. The main network functions are:

- **AMF** (Access and Mobility Management Function): Handles NAS signalling, mobility management, and UE registration.
- **SMF** (Session Management Function): Manages the creation, modification, and release of PDU sessions, and controls the UPF via PFCP.
- **UPF** (User Plane Function): Forwards user data packets and enforces QoS policies.
- **UDM** (Unified Data Management): Stores subscriber data and authentication credentials.
- **UDR** (Unified Data Repository): Provides a unified storage backend for policy and subscription data.
- **PCF** (Policy Control Function): Makes policy decisions for sessions and traffic flows.
- **AUSF** (Authentication Server Function): Handles 5G-AKA and EAP-AKA authentication.
- **NRF** (Network Repository Function): Acts as a registry for service discovery across all NFs.

- **SCP** (Service Communication Proxy): An optional proxy for indirect communication between NFs.

A key architectural decision is the separation of the control plane (CP) and user plane (UP), which allows data forwarding capacity to scale independently from control logic.

3.1.2 Service-Based Architecture

In the SBA, each NF exposes its services through a well-defined API and uses the NRF to discover peer services. This enables microservice decomposition, rolling upgrades, and horizontal scaling — a significant shift from the point-to-point interfaces used in 4G. In production deployments, inter-NF communication is secured using mutual TLS (mTLS) and OAuth 2.0 tokens, though many research and open-source deployments skip these protections for simplicity. In Kubernetes, each 5GC network function runs as a separate pod with its own network interface, resource limits, container image, and service account. This maps naturally onto Kubernetes' workload model, but it also means that if an attacker compromises one pod — such as the UPF — they can potentially move laterally to other pods via the shared cluster network.

3.1.3 Open5GS Overview

Open5GS is an open-source C implementation of the 5G Core and 4G EPC specifications. It is widely used in research and prototyping, and provides all the network functions required by 3GPP Release 16. The full 5GC stack can be deployed on a standard Kubernetes cluster using Open5GS's Helm charts and Kubernetes manifests. Open5GS uses a MongoDB database to store subscriber data including the IMSI, authentication key (K), operator code (OPC), and sequence number (SQN). A web-based UI is provided for subscriber provisioning. In this testbed, the subscriber is identified by IMSI `999700000000009`. The PLMN (MCC/MNC), slice configuration (SST/SD), and APN/DNN settings are critical configuration points — any mismatch in these values causes authentication or session failures, as observed during testbed commissioning.

3.1.4 Security Challenges in 5G Core Networks

Compared to traditional EPC deployments, the cloud-native 5GC introduces several additional security challenges:

1. **Expanded attack surface:** Every NF is a networked microservice. Vulnerabilities in any NF, its runtime, or its dependencies can be exploited remotely.
2. **Shared infrastructure:** 5GC pods share Kubernetes nodes with other workloads. A container escape from any workload puts all co-located NFs at risk.
3. **Credential exposure:** Misconfigured pods or RBAC policies can unintentionally expose service account tokens, MongoDB credentials, and inter-NF certificates.
4. **Lateral movement:** Without network controls, a compromised AMF pod can communicate directly with the UDM or MongoDB, enabling subscriber data theft.
5. **Availability attacks:** Crypto mining or resource exhaustion in any pod competes with 5GC pods for CPU and memory, causing dropped sessions and registration failures.

3.2 Kubernetes Fundamentals

3.2.1 Containers and Virtualization

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and lifecycle management of containerised workloads across a cluster of servers. An operator declares the desired state in a YAML manifest, and Kubernetes continuously works to match the actual cluster state to that description — automatically restarting crashed containers, redistributing workloads when a node fails, and scaling replicas based on demand. Containers offer lightweight virtualisation by using Linux namespaces and control groups to isolate applications. They are more resource-efficient than virtual machines because they share the host kernel, but this shared kernel also means that a container escape can expose the entire host system and all co-located workloads.

3.2.2 Kubernetes Architecture

A Kubernetes cluster consists of a control plane that manages the cluster, and worker nodes that run application workloads. The control plane has four main components, the API Server, which is the single entry point for all cluster operations and the component most relevant to security; etcd, a key-value store that holds the entire cluster state including secrets and RBAC policies; the Scheduler, which assigns new pods to worker nodes; and the Controller Manager, which runs reconciliation loops to keep the cluster

in its desired state. Each worker node runs three components: kubelet, which manages container lifecycle on the node; kube-proxy, which maintains network rules for routing service traffic; and the container runtime (containerd in this testbed), which pulls images and runs containers. This namespace separation not only enforces access and network

Table 3.1: Kubernetes Component Summary

Plane	**Component**	**Role**
Control Plane	API Server	Single entry point; enforces RBAC
	etcd	Persistent key-value store for cluster state
	Scheduler	Assigns pods to nodes
	Controller Manager	Reconciliation loops; self-healing
Worker Node	kubelet	Node agent; manages container lifecycle
	kube-proxy	Maintains network rules for Service routing
	Container Runtime	Pulls images and runs containers (containerd)

isolation between production workloads and the attack testbed, but also creates the multi-namespace environment against which Kubescape's network policy and RBAC checks are evaluated.

3.2.3 Kubernetes Networking

By default, Kubernetes uses a flat network model where every pod can communicate with every other pod regardless of namespace. This is implemented by a Container Network Interface (CNI) plugin — Flannel in this testbed. Network isolation can be added using NetworkPolicy resources, which restrict traffic to only explicitly permitted flows. Open5GS pods communicate via Kubernetes Services. The NGAP interface between UERANSIM and the AMF uses a NodePort service, while SBI interfaces use ClusterIP services resolved by the cluster DNS (CoreDNS). The absence of NetworkPolicy objects between namespaces is flagged by Kubescape as a misconfiguration, since it allows any pod in the cluster to reach any other pod without restriction.

3.2.4 Security Concepts in Kubernetes

The key Kubernetes security primitives relevant to this study are:

- **RBAC (Role-Based Access Control):** Controls which actions (get, list, create, delete) each subject (user or service account) can perform on which resources. Overly permissive roles — especially those granting cluster-admin or wildcard (*) permissions — are one of the main findings flagged by Kubescape.

- **Service Accounts:** Every pod is automatically assigned a service account, and its bearer token is mounted at `/var/run/secrets/kubernetes.io/serviceaccount/token`. If stolen, this token can be used to authenticate to the Kubernetes API server and potentially take control of the cluster.
- **Pod Security Context:** Fields such as `runAsNonRoot`, `privileged`, `allowPrivilegeEscalation`, `readOnlyRootFilesystem`, and `capabilities` control what a container is allowed to do. These are often left unset in default Open5GS deployments.
- **Admission Controllers:** Webhooks that intercept pod creation requests, allowing tools like Kubescape to validate or reject manifests before they reach the scheduler.
- **Secrets:** Kubernetes objects for storing sensitive data. A common misconfiguration is placing credentials directly in pod environment variables instead of referencing them via `secretKeyRef`.

3.3 Security Monitoring Approaches

3.3.1 Configuration-Based Security

Configuration-based security — also called static analysis or posture management — evaluates the declared state of the cluster (YAML manifests, RBAC policies, Kubernetes API objects) against established security benchmarks. This approach is proactive: it finds vulnerabilities before they are exploited and can be integrated into CI/CD pipelines to prevent insecure configurations from reaching production. The main frameworks used to define these benchmarks are:

- **NSA-CISA Kubernetes Hardening Guidelines:** Published by the US National Security Agency, these guidelines provide 24 policies for securing Kubernetes clusters.
- **CIS Kubernetes Benchmark:** The most widely used standard for Kubernetes configuration security, covering the API server, etcd, controller manager, scheduler, nodes, and pod security.
- **MITRE ATT&CK for Containers:** Maps attacker tactics and techniques to Kubernetes and container environments, helping security teams understand what each misconfiguration enables.

The limitation of this approach is that it cannot monitor runtime behaviour or detect attacks that exploit correctly configured systems.

3.3.2 Runtime-Based Security

Runtime security monitors what workloads are actually doing, rather than what they are declared to do. This is achieved through kernel-level instrumentation that observes Linux system calls (syscalls). The main events tracked include process execution (`execve`), file access (`open`, `read`, `write`), network connections (`connect`, `accept`), privilege changes (`setuid`, `capset`), and memory operations (`mmap`, `ptrace`). The standard tool for this purpose is eBPF (extended Berkeley Packet Filter), which allows safe, sandboxed programs to run inside the Linux kernel and intercept specific events with very little overhead. Falco uses eBPF probes to capture syscalls and then applies its rule engine to the resulting event stream. The limitation of runtime monitoring is that it can only detect exploitation after it happens — it cannot prevent a misconfiguration from existing in the first place.

3.3.3 Preventive vs Reactive Security

Preventive security (Kubescape) aims to remove vulnerabilities before they can be exploited. Reactive security (Falco) detects and alerts on active exploitation. Neither approach is enough on its own: without configuration hardening, attackers have an unnecessarily wide attack surface; without runtime detection, a correctly configured system can still be compromised by zero-day exploits or stolen credentials. A simple analogy makes this clear: Kubescape is a building inspector who checks that all doors have locks and fire exits are unblocked. Falco is the CCTV system that raises an alarm when someone breaks through a window. You need both.

Chapter 4

System Design and Testbed Setup

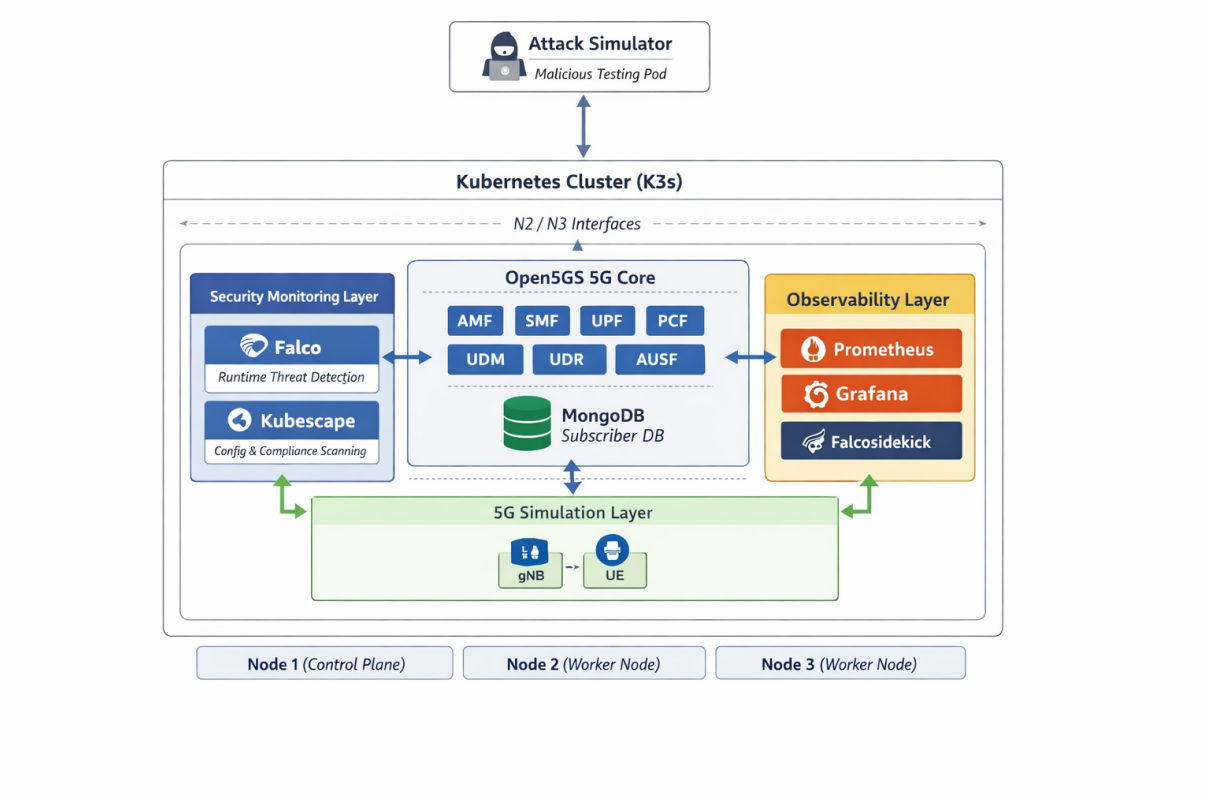

Figure 4.1: Overall testbed architecture showing the K3s cluster, Open5GS 5G Core, UERANSIM simulation layer, security monitoring tools (Falco and Kubescape), observability stack, and attack simulator pod.

4.1 Why Kubernetes for 5G

The 5G core architecture is built as a collection of independent microservices — the AMF, SMF, UPF, UDM, and other network functions all communicate via HTTP/2 REST APIs. This maps naturally onto Kubernetes, which was designed to orchestrate microservice-based applications. Each network function runs as its own pod and can be scaled, updated, or restarted independently without affecting the others. The entire 5G core can be deployed from YAML manifests or Helm charts, making deployments repeatable and version-controlled.

Containers are also significantly more efficient than virtual machines. Because containers share the host operating system kernel, they start faster and consume fewer resources. The trade-off is that this shared kernel introduces the risk of a container escape vulnerability, where an attacker who breaks out of a container gains access to the host node and every other container running on it. This is a central concern throughout this study.

4.2 Kubernetes Cluster Setup

4.2.1 K3s Cluster

K3s [12], a lightweight CNCF-certified Kubernetes distribution created by Rancher Labs, is used in this testbed. K3s has a reduced binary footprint, incorporates etcd, and is built for resource-limited environments, making it appropriate for a three-node virtual machine research cluster. K3s is fully compatible with the Kubernetes API; Helm charts and kubectl commands function exactly as in standard Kubernetes.

The cluster consists of three Ubuntu 24.04 LTS nodes:

```
NAME       STATUS   ROLES                  AGE   VERSION        INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION     CONTAINER-RUNTIME
sharyu01   Ready    control-plane,master   57d   v1.33.5+k3s1   139.6.19.201    <none>        Ubuntu 24.04.3 LTS   6.8.0-88-generic   containerd://2.1.4-k3s1
sharyu02   Ready    <none>                 57d   v1.33.5+k3s1   139.6.19.202    <none>        Ubuntu 24.04.3 LTS   6.8.0-88-generic   containerd://2.1.4-k3s1
sharyu03   Ready    <none>                 57d   v1.33.5+k3s1   139.6.19.203    <none>        Ubuntu 24.04.3 LTS   6.8.0-88-generic   containerd://2.1.4-k3s1
```

Figure 4.2: All Open5GS network function pods running in the `open5gs` namespace, distributed across sharyu01, sharyu02, and sharyu03

4.2.2 Flannel CNI and Namespace Setup

The default CNI plugin for K3s is Flannel. By employing VXLAN encapsulation to build a flat overlay network, Flannel gives each pod a distinct IP address in the `10.42.0.0/16` range and by default enables any pod to communicate with any other pod across all nodes. Because network services must communicate directly with one another via Kubernetes DNS names, this flat network paradigm is crucial for the 5G core. Six namespaces are created to logically separate components:

Table 4.1: Kubernetes Namespace Allocation

Namespace	Contents
`open5gs`	Production 5G Core: AMF, SMF, UPF, UDM, UDR, PCF, AUSF, NRF, SCP, MongoDB
`open5gs-vulnerable`	Intentionally misconfigured 5G core for attack simulation
`falco`	Falco DaemonSet and Falcosidekick
`kubescape`	Kubescape operator, scanner, node-agents, storage
`monitoring`	Prometheus, Grafana, cAdvisor, Falcosidekick UI
`kube-system`	Kubernetes core: CoreDNS, Flannel CNI, traefik

Additional node-level security configuration is implemented: cAdvisor is deployed as a DaemonSet to expose per-container CPU and RAM metrics to Prometheus, anonymous authentication on the Kubernetes API server is deactivated, and audit logging is enabled to `/var/log/kubernetes/audit.log`.

4.3 Open5GS Deployment

4.3.1 What is Open5GS

Compliant with 3GPP Release 16, Open5GS [3] is an open-source C implementation of the 5G Core and 4G EPC specifications. All necessary 5G standalone (SA) network functions—AMF, SMF, UPF, UDM, UDR, PCF, AUSF, NRF, and SCP—are implemented.It is widely used in academic research and experimentation because it provides a complete, functional 5G core that can be installed on commodity hardware.

The Gradiant OpenVerso Helm charts, a community-maintained set for deploying Open5GS and UERANSIM on Kubernetes, are used in this testbed. Each network function has its own chart, enabling independent scaling and configuration.

4.3.2 Installation via Helm

Listing 4.1: Installing Open5GS via Helm

```
# Add the Gradiant OpenVerso Helm repository
helm repo add openverso \
  https://gradiant.github.io/openverso-charts/
helm repo update

# Install the full Open5GS stack
helm install open5gs openverso/open5gs \
  --namespace open5gs \
  --create-namespace \
  -f open5gs-working-values.yaml

# Verify all pods are running
kubectl get pods -n open5gs -o wide
```

The `open5gs-working-values.yaml` file sets the MongoDB connection string, activates 5G standalone mode, and configures the PLMN using MCC=999 and MNC=70.

4.3.3 Installed Helm Releases

Table 4.2 shows all Helm releases installed in the testbed.

Table 4.2: Helm Releases Installed in the Testbed

Release	**Namespace**	**Version**	**Purpose**
open5gs	open5gs	chart N/A	5G Core network functions
falco	falco	8.0.0	Runtime security (Falco 0.43.0)
falcosidekick	monitoring	0.12.1	Alert forwarding (v2.31.1)
kubescape	kubescape	1.30.4	Configuration security
monitoring	monitoring	81.2.2	kube-prometheus-stack

4.3.4 Running Pods and Services

After installation, all Open5GS network function pods reach Running status across the three nodes.

```
sharyu@sharyu01:~$ kubectl get pods -n open5gs -o wide
NAME                                READY   STATUS    RESTARTS      AGE   IP            NODE       NOMINATED NODE   READINESS GATES
mongodb-784fd6bfff-ds8rn            1/1     Running   2 (8h ago)    11h   10.42.0.127   sharyu01   <none>           <none>
open5gs-amf-84c6659b4-g9hr7         1/1     Running   0             10h   10.42.0.150   sharyu01   <none>           <none>
open5gs-ausf-7bdbbcd67f-msz9d       1/1     Running   0             11h   10.42.1.154   sharyu02   <none>           <none>
open5gs-bsf-877c5fc5d-q6pgg         1/1     Running   0             11h   10.42.1.164   sharyu02   <none>           <none>
open5gs-hss-7b6b97ff9-876kt         1/1     Running   9 (11h ago)   11h   10.42.1.166   sharyu02   <none>           <none>
open5gs-mme-6bcd8944b5-c26nv        1/1     Running   0             11h   10.42.1.155   sharyu02   <none>           <none>
open5gs-nrf-7b665d5779-9tvtx        1/1     Running   0             11h   10.42.1.156   sharyu02   <none>           <none>
open5gs-nssf-58fb9bb8b5-t9bvl       1/1     Running   0             11h   10.42.1.158   sharyu02   <none>           <none>
open5gs-pcf-788bb557f4-gggqk        1/1     Running   9 (11h ago)   11h   10.42.2.186   sharyu03   <none>           <none>
open5gs-pcrf-b877449b6-s8hfx        1/1     Running   9 (11h ago)   11h   10.42.0.124   sharyu01   <none>           <none>
open5gs-populate-6b78f57ff7-k9xqs   1/1     Running   0             11h   10.42.0.121   sharyu01   <none>           <none>
open5gs-scp-7666d7fbf9-cz5fs        1/1     Running   1 (8h ago)    11h   10.42.1.161   sharyu02   <none>           <none>
open5gs-sepp-7d7cf9d787-dkcfv       1/1     Running   0             11h   10.42.1.159   sharyu02   <none>           <none>
open5gs-sgwc-6fc456964-7vl7h        1/1     Running   0             11h   10.42.1.160   sharyu02   <none>           <none>
open5gs-sgwu-69c49cb68b-twwpv       1/1     Running   0             11h   10.42.1.163   sharyu02   <none>           <none>
open5gs-smf-5ff666f57c-khvtk        1/1     Running   0             11h   10.42.2.185   sharyu03   <none>           <none>
open5gs-udm-7d646dd74c-c4924        1/1     Running   0             10h   10.42.0.147   sharyu01   <none>           <none>
open5gs-udr-84b58dbb7-sth6h         1/1     Running   0             10h   10.42.0.148   sharyu01   <none>           <none>
open5gs-upf-5c487c8df7-g5djq        1/1     Running   0             11h   10.42.0.120   sharyu01   <none>           <none>
open5gs-webui-7f6dffdf8b-tng2c      1/1     Running   0             11h   10.42.2.187   sharyu03   <none>           <none>
ueransim-gnb-59dbd6799b-bnklr       1/1     Running   0             10h   10.42.0.152   sharyu01   <none>           <none>
ueransim-ue-5679cf9d57-jd6wk        1/1     Running   0             9h    10.42.0.177   sharyu01   <none>           <none>
```

Figure 4.3: All Open5GS network function pods running in the `open5gs` namespace, distributed across sharyu01, sharyu02, and sharyu03

```
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP                          PORT(S)          AGE   SELECTOR
gnb-service       ClusterIP      10.43.210.245   <none>                               4997/UDP         11h   app=ueransim-gnb
mongodb           ClusterIP      10.43.38.233    <none>                               27017/TCP        11h   app=mongodb
open5gs-amf-ngap  ClusterIP      10.43.157.218   <none>                               38412/SCTP       11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=amf
open5gs-amf-sbi   ClusterIP      10.43.167.29    <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=amf
open5gs-ausf-sbi  ClusterIP      10.43.175.20    <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=ausf
open5gs-bsf-sbi   ClusterIP      10.43.247.182   <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=bsf
open5gs-hss-frdi  ClusterIP      10.43.50.162    <none>                               3868/SCTP        11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=hss
open5gs-mme-frdi  ClusterIP      10.43.139.107   <none>                               3868/SCTP        11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=mme
open5gs-mme-gtpc  ClusterIP      10.43.50.229    <none>                               2123/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=mme
open5gs-mme-s1ap  ClusterIP      10.43.212.15    <none>                               36412/SCTP       11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=mme
open5gs-mongodb   ExternalName   <none>          mongodb.open5gs.svc.cluster.local    <none>           11h   <none>
open5gs-nrf-sbi   ClusterIP      10.43.91.74     <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=nrf
open5gs-nssf-sbi  ClusterIP      10.43.215.71    <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=nssf
open5gs-pcf-sbi   ClusterIP      10.43.171.154   <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=pcf
open5gs-pcrf-frdi ClusterIP      10.43.233.194   <none>                               3868/SCTP        11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=pcrf
open5gs-scp-sbi   ClusterIP      10.43.2.92      <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=scp
open5gs-sepp-n32  ClusterIP      10.43.181.185   <none>                               7443/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=sepp
open5gs-sepp-sbi  ClusterIP      10.43.214.67    <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=sepp
open5gs-sgwc-gtpc ClusterIP      10.43.48.135    <none>                               2123/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=sgwc
open5gs-sgwc-pfcp ClusterIP      10.43.164.77    <none>                               8805/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=sgwc
open5gs-sgwu-gtpu ClusterIP      10.43.142.97    <none>                               2152/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=sgwu
open5gs-sgwu-pfcp ClusterIP      10.43.128.172   <none>                               8805/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=sgwu
open5gs-smf-frdi  ClusterIP      10.43.139.162   <none>                               3868/SCTP        11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=smf
open5gs-smf-gtpc  ClusterIP      10.43.205.88    <none>                               2123/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=smf
open5gs-smf-gtpu  ClusterIP      10.43.45.212    <none>                               2152/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=smf
open5gs-smf-pfcp  ClusterIP      10.43.51.6      <none>                               8805/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=smf
open5gs-smf-sbi   ClusterIP      10.43.15.246    <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=smf
open5gs-udm-sbi   ClusterIP      10.43.82.74     <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=udm
open5gs-udr-sbi   ClusterIP      10.43.56.86     <none>                               7777/TCP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=udr
open5gs-upf-gtpu  ClusterIP      10.43.19.249    <none>                               2152/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=upf
open5gs-upf-pfcp  ClusterIP      10.43.137.207   <none>                               8805/UDP         11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=upf
open5gs-webui     NodePort       10.43.140.213   <none>                               9999:30694/TCP   11h   app.kubernetes.io/instance=open5gs,app.kubernetes.io/name=webu
```

Figure 4.4: Kubernetes ClusterIP services exposing Open5GS network function endpoints, including AMF NGAP on 38412/SCTP, SMF PFCP on 8805/UDP, and UPF GTP-U on 2152/UDP

4.3.5 Subscriber Provisioning via WebUI

Subscriber data is managed through the Open5GS WebUI, accessible by port-forwarding to the `open5gs-webui` service:

Listing 4.2: Accessing the Open5GS WebUI

```
kubectl port-forward svc/open5gs-webui \
  9999:9999 -n open5gs
# Open browser: http://localhost:9999
# Login: admin / 1423
```

The subscriber is provisioned with the following parameters, which must exactly match the UERANSIM UE configuration:

- **IMSI:** 999700000000009
- **Subscriber Key (K):** 465B5CE8B199B49FAA5F0A2EE238A6BC
- **Operator Code (OPc):** E8ED289DEBA952E4283B54E88E6183CA
- **USIM Type:** OPc
- **AMF:** 8000
- **Slice:** SST=1 (eMBB), Default S-NSSAI
- **DNN/APN:** internet, IPv4v6
- **SQN:** Reset to zero before each test run to prevent sequence number synchronisation failures

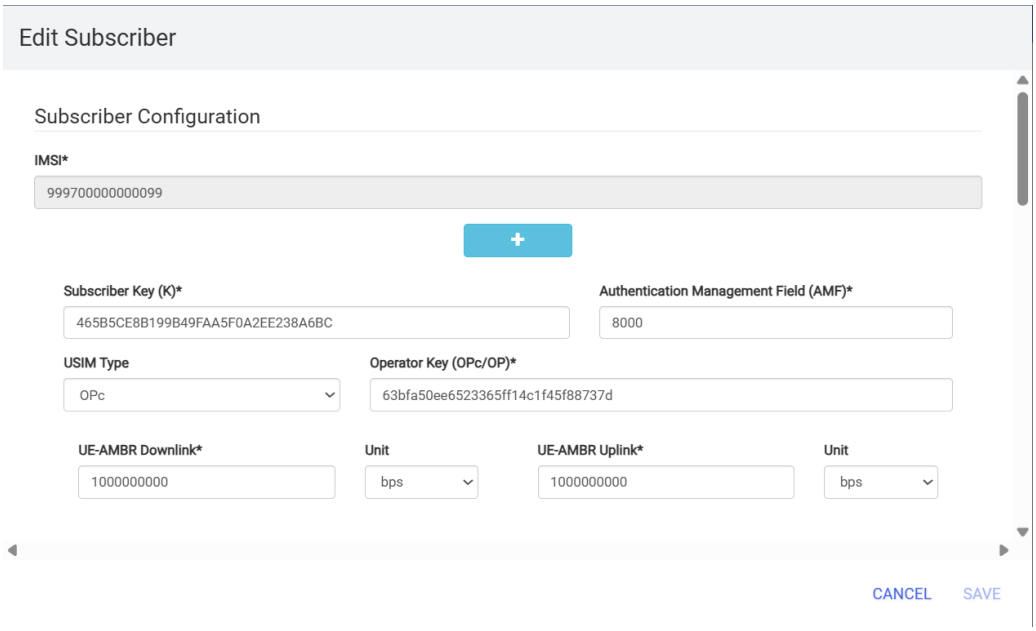

Figure 4.5: Open5GS WebUI showing subscriber IMSI 999700000000001, key K, and USIM type OPc

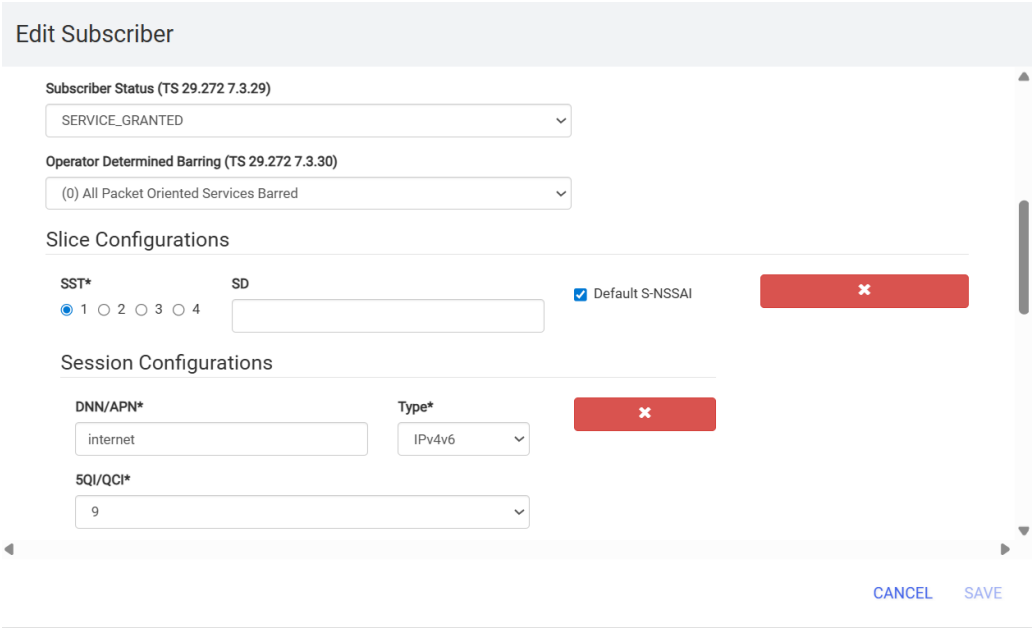

Figure 4.6: Open5GS WebUI slice configuration: SST=1 as default S-NSSAI and DNN `internet` with IPv4v6 session type

4.4 UERANSIM Deployment

4.4.1 What is UERANSIM

UERANSIM [4] (UE and RAN Simulator) is an open-source C++ implementation of a 5G gNodeB and user equipment. It replicates the entire NAS and RRC protocol stacks, enabling a software UE to perform valid 5G authentication, registration, and PDU session creation against an actual 5G core network. The Gradiant UERANSIM container image `docker.io/gradiant/ueransim:3.2.6` is used in this testbed.

4.4.2 Combined Pod Design

The gNB and UE are deployed as two containers inside a single combined pod (`ueransim-combined`) rather than two distinct pods. Both containers share the same pod network namespace, meaning the UE connects to the gNB's radio interface via `127.0.0.1` (loopback) without any additional Kubernetes services.

Both containers operate with `privileged: true` because the UE must establish the `uesimtun0` TUN network interface for the PDU session data plane, which requires the `NET_ADMIN` Linux capability. This privileged mode is a directly significant security finding in the evaluation—Kubescape identifies it as a critical misconfiguration (control C-0002), though it represents a legitimate operational trade-off rather than a correctable error in this context.

4.4.3 Pod Manifest: ueransim-fixed.yaml

Listing 4.3: ueransim-fixed.yaml: Combined gNB+UE pod manifest

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ueransim-combined
  namespace: open5gs
  labels:
    app: ueransim
spec:
  nodeSelector:
    kubernetes.io/hostname: sharyu01
  containers:
  - name: gnb
    image: docker.io/gradiant/ueransim:3.2.6
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true  # Required for TUN interface
    command: ["/entrypoint.sh", "gnb"]
    env:
    - name: MCC
      value: "999"     # Test PLMN
    - name: MNC
      value: "70"
    - name: TAC
      value: "1"       # Tracking Area Code
    - name: SST
      value: "1"       # eMBB slice
    - name: SD
      value: "0xffffff"
    - name: AMF_IP
      value: "open5gs-amf-ngap.open5gs.svc.cluster.local"
    - name: N2_BIND_IP
      value: "0.0.0.0"
    - name: N3_BIND_IP
      value: "0.0.0.0"
    - name: N3_ADVERTISE_IP
      value: "0.0.0.0"
    - name: RADIO_BIND_IP
      value: "0.0.0.0"

  - name: ue
    image: docker.io/gradiant/ueransim:3.2.6
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true  # Required for TUN interface
```

```
    command: ["/bin/bash", "-c"]
    args:
    - |
      envsubst < /etc/ueransim/ue.yaml > /tmp/ue.yaml
      sed -i '/sd:/d' /tmp/ue.yaml
      nr-ue -c /tmp/ue.yaml
    env:
    - name: MCC
      value: "999"
    - name: MNC
      value: "70"
    - name: MSISDN
      value: "0000000099"
    - name: KEY
      value: "465B5CE8B199B49FAA5F0A2EE238A6BC"
    - name: OP
      value: "63bfa50ee6523365ff14c1f45f88737d"
    - name: OP_TYPE
      value: "OPC"
    - name: APN
      value: "internet"
    - name: SST
      value: "1"
    - name: GNB_IP
      value: "127.0.0.1"  # Loopback: shared pod network
```

gNB Configuration: The `gnb` container launches `nr-gnb` by executing the `/entrypoint.sh gnb` script. `MCC=999, MNC=70` defines the test PLMN identity; `TAC=1` sets the Tracking Area Code; `AMF_IP` is resolved by CoreDNS to the AMF ClusterIP `10.43.157.218` on port `38412/SCTP`.

UE Configuration: The `ue` container uses a customised starting command to address a known slice differentiator (SD) matching issue in UERANSIM 3.2.6. `sed -i '/sd:/d'` removes all SD lines from the configuration, avoiding a slice mismatch error. `GNB_IP=127.0.0.1` directs the UE to connect to the gNB via loopback.

4.5 Automated Registration: fix-open5gs.sh

4.5.1 The Problem It Solves

Race conditions frequently caused manual UERANSIM deployment to fail during testbed development. If the UE pod started before the AMF and SMF had finished initialising their security contexts, registration would fail with an authentication error or session management failure. Furthermore, stale security contexts from earlier registrations would

cause 5G-AKA authentication problems on subsequent runs. `fix-open5gs.sh` automates the complete restart and registration process in the correct order with appropriate readiness checks at each stage.

4.5.2 Script Walkthrough

Listing 4.4: fix-open5gs.sh: Automated registration and verification script

```
#!/bin/bash

# Step 1: Restart SMF and AMF to clear security contexts
echo "=== Restarting Open5GS Core ==="
kubectl rollout restart deployment/open5gs-smf -n open5gs
kubectl rollout restart deployment/open5gs-amf -n open5gs

echo "Waiting for SMF and AMF..."

# Step 2: Wait for both pods to reach Ready status
kubectl wait --for=condition=ready pod \
  -l app.kubernetes.io/name=smf \
  -n open5gs --timeout=120s
kubectl wait --for=condition=ready pod \
  -l app.kubernetes.io/name=amf \
  -n open5gs --timeout=120s

# Step 3: Delete and recreate the UERANSIM pod
echo "=== Launching UERANSIM ==="
kubectl delete pod -n open5gs ueransim-combined \
  2>/dev/null || true
kubectl apply -f ~/ueransim-fixed.yaml

# Step 4: Wait 30 seconds for registration to complete
echo "Waiting 30 seconds..."
sleep 30

# Step 5: Show the registration result
echo "=== Status ==="
kubectl logs -n open5gs ueransim-combined -c ue \
  | grep -E "successful|TUN|error" | tail -5
```

```
timed out waiting for the condition on pods/open5gs-smf-d8f794577-b78js
Restarting AMF to clear security contexts...
deployment.apps/open5gs-amf restarted
pod/open5gs-amf-55f9b8d8bf-s6xxc condition met
pod/open5gs-amf-c567d4d65-hwbzz condition met
=== Core network components ready ===

Restarting UERANSIM...
pod "ueransim-combined" deleted
pod/ueransim-combined created
Waiting for UERANSIM to start and register (30 seconds)...

=== Registration Status ===
[2026-02-17 10:37:32.829] [nas] [info] Initial Registration is successful
[2026-02-17 10:37:35.595] [nas] [info] PDU Session establishment is successful PSI[1]
[2026-02-17 10:37:36.031] [app] [info] Connection setup for PDU session[1] is successful, TUN interface[uesimtun0, 10.45.0.2] is up.

=== Recent UERANSIM Logs ===
[2026-02-17 10:37:32.705] [rrc] [info] UE switches to state [RRC-CONNECTED]
[2026-02-17 10:37:32.705] [nas] [info] UE switches to state [CM-CONNECTED]
[2026-02-17 10:37:32.707] [ngap] [debug] Initial NAS message received from UE[1]
[2026-02-17 10:37:32.737] [nas] [debug] Authentication Request received
[2026-02-17 10:37:32.737] [nas] [debug] Received SQN [000000001E1]
[2026-02-17 10:37:32.737] [nas] [debug] SQN-MS [00000000000]
[2026-02-17 10:37:32.772] [nas] [debug] Security Mode Command received
[2026-02-17 10:37:32.772] [nas] [debug] Selected integrity[2] ciphering[0]
[2026-02-17 10:37:32.828] [ngap] [debug] Initial Context Setup Request received
[2026-02-17 10:37:32.829] [nas] [debug] Registration accept received
[2026-02-17 10:37:32.829] [nas] [info] UE switches to state [MM-REGISTERED/NORMAL-SERVICE]
[2026-02-17 10:37:32.829] [nas] [debug] Sending Registration Complete
[2026-02-17 10:37:32.829] [nas] [info] Initial Registration is successful
[2026-02-17 10:37:32.829] [nas] [debug] Sending PDU Session Establishment Request
[2026-02-17 10:37:32.830] [nas] [debug] UAC access attempt is allowed for identity[0], category[MO_sig]
[2026-02-17 10:37:33.032] [nas] [debug] Configuration Update Command received
[2026-02-17 10:37:35.594] [ngap] [info] PDU session resource(s) setup for UE[1] count[1]
[2026-02-17 10:37:35.595] [nas] [debug] PDU Session Establishment Accept received
[2026-02-17 10:37:35.595] [nas] [info] PDU Session establishment is successful PSI[1]
[2026-02-17 10:37:36.031] [app] [info] Connection setup for PDU session[1] is successful, TUN interface[uesimtun0, 10.45.0.2] is up.

=== Complete! ===
```

Figure 4.7: Output of `fix-open5gs.sh`: AMF and SMF restarted, core ready, UERANSIM recreated, registration successful, PDU session established, and `uesimtun0` at `10.45.0.2` confirmed up

4.6 UE Registration and Session Establishment

This section documents the full end-to-end registration process as observed in the testbed logs. Understanding this baseline behaviour is important for the attack evaluation — Kubescape and Falco must be configured so that normal 5G core activity does not produce false positive warnings.

Before going into each step in detail, the overview below shows how all the components work together from the moment the UE sends its first message to the moment it has a working data connection:

1. The UE sends a Registration Request to the gNodeB over the radio interface.
2. The gNodeB forwards the request to the AMF over the N2 (NGAP) interface as an `InitialUEMessage`.
3. The AMF initiates authentication by contacting the AUSF, asking it to verify the UE's credentials.
4. The AUSF queries the UDM, which looks up the subscriber record in MongoDB and returns the authentication vectors.
5. The AMF completes mutual authentication with the UE and sends a Registration Accept. The UE replies with Registration Complete.
6. The UE requests a PDU session. The AMF forwards this request to the SMF.

7. The SMF assigns the UE an IP address and programs the UPF via PFCP to open a GTP-U data tunnel.

8. The UE receives PDU Session Establishment Accept and is now fully connected with a working data path through the 5G core.

Throughout this process, Falco monitors all syscalls generated by the AMF, SMF, and UPF containers in real time, while Prometheus records CPU and memory usage across all network function pods. Grafana displays this data as live dashboards for the duration of the evaluation.

4.6.1 Step 1: gNB Connects to AMF

The gNB immediately establishes an SCTP connection to the AMF at `10.43.157.218:38412` as the `ueransim-combined` pod starts. The gNB and AMF exchange the supported PLMN list, tracking regions, and slices during the NG Setup procedure. 4.8

```
sharyu@sharyu01:~$ kubectl logs -n open5gs $(kubectl get pods -n open5gs | grep "gnb.*Running" | awk '{print $1}')
UERANSIM v3.2.6
[2026-01-07 23:21:11.895] [sctp] [info] Trying to establish SCTP connection... (10.43.157.218:38412)
[2026-01-07 23:21:11.901] [sctp] [info] SCTP connection established (10.43.157.218:38412)
[2026-01-07 23:21:11.902] [sctp] [debug] SCTP association setup ascId[18537]
[2026-01-07 23:21:11.902] [ngap] [debug] Sending NG Setup Request
[2026-01-07 23:21:11.927] [ngap] [debug] NG Setup Response received
[2026-01-07 23:21:11.929] [ngap] [info] NG Setup procedure is successful
[2026-01-07 23:21:12.416] [rrc] [debug] UE[1] new signal detected
[2026-01-07 23:21:12.419] [rrc] [info] RRC Setup for UE[1]
[2026-01-07 23:21:12.420] [ngap] [debug] Initial NAS message received from UE[1]
[2026-01-07 23:21:12.563] [ngap] [debug] UE Context Release Command received
[2026-01-07 23:21:12.564] [rrc] [info] Releasing RRC connection for UE[1]
[2026-01-07 23:21:45.509] [rls] [debug] UE[1] signal lost
[2026-01-07 23:21:45.714] [rrc] [debug] UE[2] new signal detected
[2026-01-07 23:21:45.717] [rrc] [info] RRC Setup for UE[2]
[2026-01-07 23:21:45.719] [ngap] [debug] Initial NAS message received from UE[2]
[2026-01-07 23:21:45.787] [ngap] [debug] UE Context Release Command received
[2026-01-07 23:21:45.787] [rrc] [info] Releasing RRC connection for UE[2]
[2026-01-07 23:25:12.449] [rrc] [debug] UE[3] new signal detected
[2026-01-07 23:25:12.453] [rrc] [info] RRC Setup for UE[3]
[2026-01-07 23:25:12.456] [ngap] [debug] Initial NAS message received from UE[3]
[2026-01-07 23:25:13.468] [ngap] [debug] Initial Context Setup Request received
[2026-01-07 23:25:13.670] [rls] [debug] UE[2] signal lost
[2026-01-07 23:25:22.047] [ngap] [info] PDU session resource(s) setup for UE[3] count[1]
```

Figure 4.8: gNB log: SCTP connection established to AMF at `10.43.157.218:38412`, NG Setup Request sent, NG Setup Response received, and NG Setup procedure successful

4.6.2 Step 2: UE Registration

The UE starts the NAS registration process as soon as the gNB is connected. The registration sequence follows the 3GPP NAS procedure:

1. UE selects PLMN 999/70 and enters state MM-DEREGISTERED/PLMN-SEARCH.

2. UE sends an Initial Registration via RRC and enters MM-REGISTER-INITIATED.

3. An RRC connection is established; UE enters RRC-CONNECTED and CM-CONNECTED.
4. AMF sends an Authentication Request; the 5G-AKA challenge begins.
5. Security Mode Command is received; the NAS security context is built using integrity algorithm IA2.
6. Registration Accept is received; UE transitions to MM-REGISTERED/NORMAL-SERVICE.
7. Initial Registration is successful.

Refer to 4.9

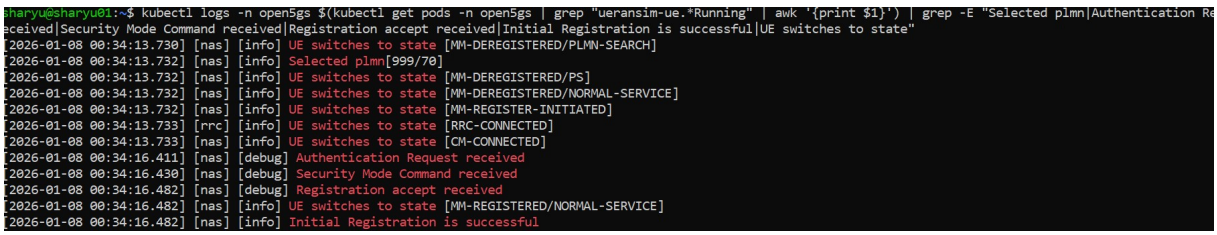

Figure 4.9: Filtered UE log showing the four key registration milestones: Authentication Request received, Security Mode Command received, Registration Accept received, and Initial Registration successful

4.6.3 Step 3: AMF Processes the Registration

When a UE wants to connect to the 5G network, the first message it sends is called an `InitialUEMessage`. This travels from the gNodeB to the AMF over the NGAP interface, where it is processed by the **GMM (GPRS Mobility Management)** layer — the part of the AMF responsible for handling registration requests. GMM checks the UE's identity, verifies the subscriber in the database, runs authentication, and either accepts or rejects the registration.

The confirmation appears in the AMF log as:

```
[GMM] Registration complete
```

This single line confirms that the UE has been accepted by the 5G core. For IMSI `999700000000009`, this means the simulated UE is live and ready to establish a PDU session for data traffic. Refer to 4.10

```
sharyu@sharyu01:~$ kubectl logs -n open5gs open5gs-amf-5694c97d75-8k6kx | grep -E "Authentication|Registration|NAS|UE|NGAP"
03/07 15:10:30.086: [amf] INFO: InitialUEMessage (../src/amf/ngap-handler.c:437)
03/07 15:10:30.087: [amf] INFO: [Added] Number of gNB-UEs is now 1 (../src/amf/context.c:2789)
03/07 15:10:30.087: [amf] INFO:     RAN_UE_NGAP_ID[1] AMF_UE_NGAP_ID[1] TAC[1] CellID[0x100] (../src/amf/ngap-handler.c:598)
03/07 15:10:30.087: [amf] INFO: [suci-0-999-70-0000-0-0-0000000099] Unknown UE by SUCI (../src/amf/context.c:1906)
03/07 15:10:30.088: [amf] INFO: [Added] Number of AMF-UEs is now 1 (../src/amf/context.c:1682)
03/07 15:10:30.088: [gmm] INFO: Registration request (../src/amf/gmm-sm.c:1339)
03/07 15:10:30.701: [gmm] INFO: [imsi-999700000000099] Registration complete (../src/amf/gmm-sm.c:2698)
```

Figure 4.10: AMF GMM log: Registration Complete confirmed for IMSI `999700000000001`

4.6.4 Step 4: PDU Session Establishment

After registration, the UE immediately requests a **PDU session** for internet access using network slice `SST=1`. A PDU session is simply the data connection that allows the UE to send and receive traffic through the 5G core.

The **SMF** handles this request by assigning the UE an IP address (`10.45.0.2`) and setting up a **GTP-U tunnel** — the tunnel that carries the actual user data between the gNodeB and the UPF. It does this by talking to the UPF over the **PFCP** interface, which is the protocol the SMF uses to program the UPF with forwarding rules.

The three figures below confirm the session was established at each layer: the UE reports session establishment success, the SMF logs show the IP address being allocated, and the UPF logs show the GTP-U bearer being created. Refer to 4.11 4.12 4.13

```
sharyu@sharyu01:~$ kubectl logs -n open5gs $(kubectl get pods -n open5gs | grep "ueransim-ue.*Running" | awk '{print $1}') | grep -E "Sending PDU Session Establishment Requ
est|PDU Session Establishment Accept received|PDU Session establishment is successful"
[2026-01-08 00:34:16.482] [nas] [debug] Sending PDU Session Establishment Request
[2026-01-08 00:34:19.255] [nas] [debug] PDU Session Establishment Accept received
[2026-01-08 00:34:19.256] [nas] [info] PDU Session establishment is successful PSI[1]
```

Figure 4.11: UE log: PDU Session Establishment Request sent, PDU Session Establishment Accept received, PDU Session establishment successful PSI[1]

```
sharyu@sharyu01:~$ kubectl logs -n open5gs open5gs-smf-f5f4c446-slq98 | grep -E "Session|PDU|UE"
03/07 15:10:41.095: [smf] INFO: [Added] Number of SMF-UEs is now 1 (../src/smf/context.c:1033)
03/07 15:10:41.095: [smf] INFO: [Added] Number of SMF-Sessions is now 1 (../src/smf/context.c:3203)
03/07 15:10:41.261: [smf] INFO: UE SUPI[imsi-999700000000099] DNN[internet] IPv4[10.45.0.2] IPv6[] (../src/smf/npcf-handler.c:586)
```

Figure 4.12: SMF log: UE session created for SUPI `imsi-999700000000009` on DNN `internet` with IPv4 address `10.45.0.2` allocated

```
sharyu@sharyu01:~$ kubectl logs -n open5gs $(kubectl get pods -n open5gs | grep "upf.*Running" | awk '{print $1}') | grep -E "Session|GTP|UE"
Defaulted container "open5gs-upf" out of: open5gs-upf, tun-create (init)
01/07 23:25:21.991: [upf] INFO: [Added] Number of UPF-Sessions is now 1 (../src/upf/context.c:209)
01/07 23:25:21.992: [upf] INFO: UE F-SEID[UP:0xec4 CP:0xe43] APN[internet] PDN-Type[1] IPv4[10.45.0.3] IPv6[] (../src/upf/context.c:495)
01/08 00:15:20.087: [upf] INFO: [Added] Number of UPF-Sessions is now 2 (../src/upf/context.c:209)
01/08 00:15:20.088: [upf] INFO: UE F-SEID[UP:0xd3e CP:0x8a1] APN[internet] PDN-Type[1] IPv4[10.45.0.4] IPv6[] (../src/upf/context.c:495)
01/08 00:16:01.525: [upf] INFO: [Added] Number of UPF-Sessions is now 2 (../src/upf/context.c:209)
01/08 00:16:01.525: [upf] INFO: UE F-SEID[UP:0xefd CP:0x545] APN[internet] PDN-Type[1] IPv4[10.45.0.5] IPv6[] (../src/upf/context.c:495)
01/08 00:16:49.508: [upf] INFO: [Added] Number of UPF-Sessions is now 2 (../src/upf/context.c:209)
01/08 00:16:49.508: [upf] INFO: UE F-SEID[UP:0xe06 CP:0x5c6] APN[internet] PDN-Type[1] IPv4[10.45.0.7] IPv6[] (../src/upf/context.c:495)
01/08 00:17:31.314: [upf] INFO: [Added] Number of UPF-Sessions is now 2 (../src/upf/context.c:209)
01/08 00:17:31.314: [upf] INFO: UE F-SEID[UP:0x957 CP:0x515] APN[internet] PDN-Type[1] IPv4[10.45.0.8] IPv6[] (../src/upf/context.c:495)
01/08 00:18:18.530: [upf] INFO: [Added] Number of UPF-Sessions is now 2 (../src/upf/context.c:209)
01/08 00:18:18.530: [upf] INFO: UE F-SEID[UP:0x24c CP:0xf3b] APN[internet] PDN-Type[1] IPv4[10.45.0.9] IPv6[] (../src/upf/context.c:495)
01/08 00:19:02.289: [upf] INFO: [Added] Number of UPF-Sessions is now 2 (../src/upf/context.c:209)
01/08 00:19:02.290: [upf] INFO: UE F-SEID[UP:0x81f CP:0x664] APN[internet] PDN-Type[1] IPv4[10.45.0.11] IPv6[] (../src/upf/context.c:495)
01/08 00:19:35.311: [upf] INFO: [Added] Number of UPF-Sessions is now 2 (../src/upf/context.c:209)
01/08 00:19:35.311: [upf] INFO: UE F-SEID[UP:0xf2b CP:0x3ee] APN[internet] PDN-Type[1] IPv4[10.45.0.12] IPv6[] (../src/upf/context.c:495)
01/08 00:20:51.881: [upf] INFO: [Added] Number of UPF-Sessions is now 3 (../src/upf/context.c:209)
01/08 00:20:51.881: [upf] INFO: UE F-SEID[UP:0x86d CP:0x5fe] APN[internet] PDN-Type[1] IPv4[10.45.0.19] IPv6[] (../src/upf/context.c:495)
```

Figure 4.13: UPF log: GTP-U bearer sessions established with F-SEID assignments and IPv4 addresses allocated from the `10.45.0.0/16` pool for the `internet` APN

4.6.5 Common Registration Failures

During testbed commissioning, several failures were encountered. These are documented in Table 4.3 and explain why the `fix-open5gs.sh` automation script was necessary.Once registration and PDU session are confirmed, live traffic is observable in Prometheus via cAdvisor.

Table 4.3: Registration Failures and Resolutions

Error	Cause	Resolution
PLMN_NOT_ALLOWED	IMSI not in MongoDB	Add subscriber via WebUI
Auth Failure (SQN)	SQN out of sync	Reset SQN to zero in MongoDB
Re-synch MAC failed	OP/OPC mismatch	Set both to opType: OPC
NGAP refused	AMF not ready	Run fix-open5gs.sh to restart AMF
uesimtun0 missing	PDU session rejected	Check SMF/UPF logs and PFCP
Slice mismatch	SD value not matched	Remove SD lines with sed -i '/sd:/d'

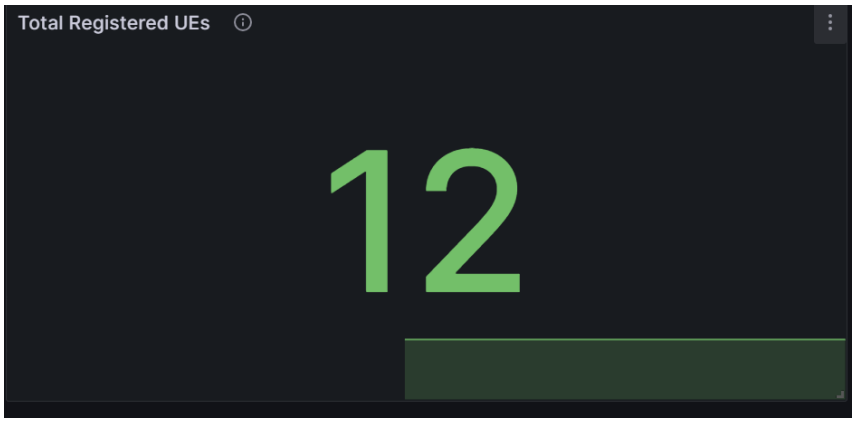

Figure 4.14: Grafana panel showing 12 successfully registered UEs during the evaluation window. The non-zero count confirms the 5G core was actively serving subscribers throughout the attack simulation.

Figure 4.15: AMF Registration Success Rate panel. The sharp rise at 23:00 corresponds to the UERANSIM pod becoming active after `fix-open5gs.sh` restarted the core components. The rate stabilises at approximately 0.2 registrations/s

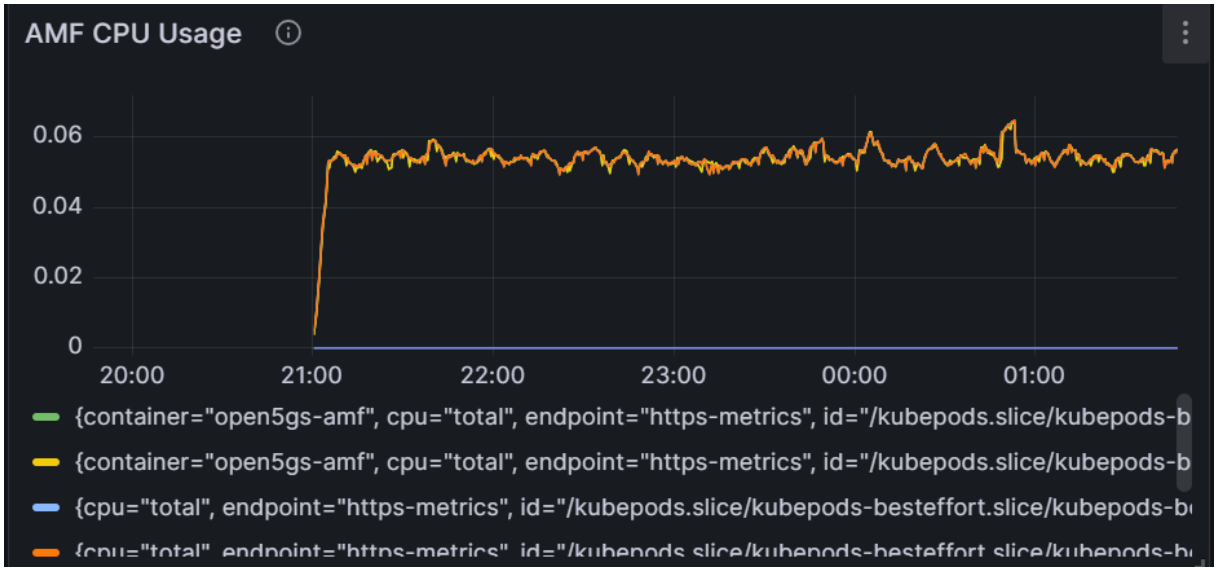

Figure 4.16: AMF CPU usage over the evaluation period. The rise from zero to a steady ∼0.05 cores at 21:00 confirms the AMF was actively processing NAS messages throughout the entire attack simulation, not just during registration bursts.

Figure 4.17: UE-to-UPF packet flow panel showing UE transmit (TX) and UPF receive (RX) traffic. The matching TX and RX rates confirm that the GTP-U data plane tunnel was active and stable throughout the attack simulation.

4.7 Monitoring Stack

Throughout the security assessment, a full monitoring stack was deployed in the `monitoring` namespace to track system performance. All tools were installed via Helm.

4.7.1 Prometheus

Prometheus (version 81.2.2, Prometheus Operator v0.88.0) is deployed as part of the kube-prometheus-stack chart. It is configured with a 15-second scrape interval and exposed on NodePort 30090. Scrape targets include cAdvisor (per-container CPU and memory), node-exporter (per-node system metrics), Falcosidekick (Falco alert rate counters), Kubescape (compliance scores), and Trivy-operator (vulnerability findings).

4.7.2 Grafana

Grafana version 10.2.3 is installed on NodePort 30888. Two custom dashboards are provided via ConfigMaps:

- **Pod Metrics Dashboard:** Displays CPU and RAM per pod for every open5gs namespace pod. Used to track AMF and SMF CPU spikes during UE registration and to gauge the performance impact of each attack scenario.
- **System Metrics Dashboard:** Displays node-level CPU and memory across all three K3s nodes, used to measure overhead from Falco and Kubescape.

4.7.3 Falcosidekick

Two replicas of Falcosidekick (version 2.31.1) are installed in the monitoring namespace. On port 2801, it receives Falco alerts as JSON; on port 2810, it forwards them to Prometheus as counter metrics; on port 6379, it stores them in a Redis StatefulSet; and on port 2802, it offers a web user interface. This pipeline enables direct correlation between an attack event and its effect on 5G core availability by visualising Falco alert rates in Grafana alongside Open5GS performance indicators.

Figure 4.18 shows the Grafana Kubernetes dashboard during normal testbed operation. The cluster runs 65 pods across 10 namespaces on 3 nodes, with a CPU utilisation of 16.9% and RAM utilisation of 21.91%. All workloads — open5gs (28), monitoring (24), kubescape (16), kube-system (15), and the vulnerable namespace (7) — are active simultaneously, confirming a realistic multi-tenant environment throughout the evaluation.

With the 5G core operational, UE registration verified, and the full monitoring stack in place, the testbed provides a stable and observable environment for the security tool evaluation in the following chapters.

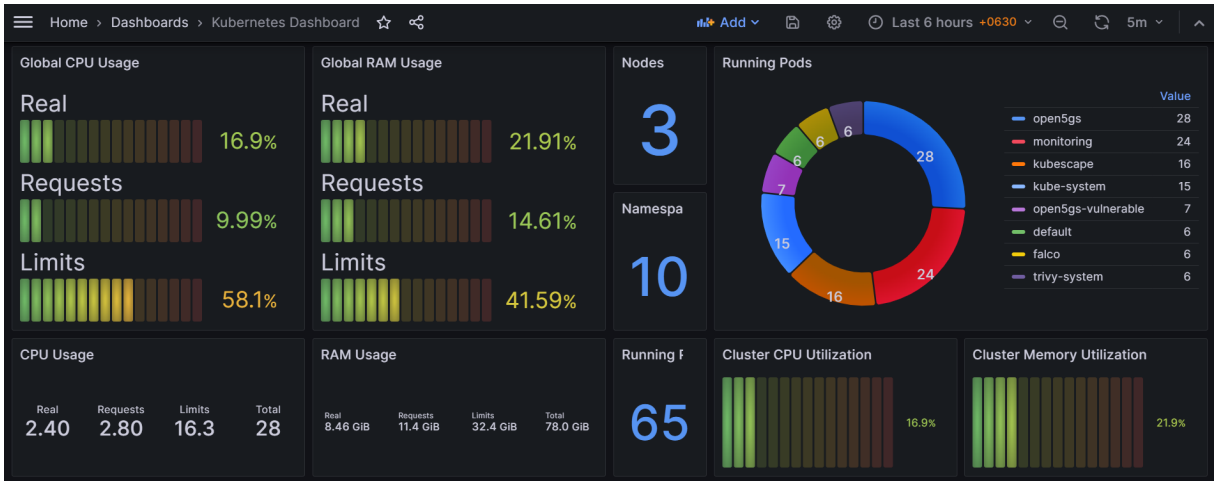

Figure 4.18: Grafana Kubernetes Dashboard showing cluster-wide resource usage and pod distribution across namespaces during normal testbed operation

Chapter 5

Security Monitoring Tools

5.1 Kubescape

5.1.1 Installation

Listing 5.1: Installing Kubescape via Helm

```
# Add the Kubescape Helm repository
helm repo add kubescape \
  https://kubescape.github.io/helm-charts
helm repo update

# Install Kubescape operator
helm install kubescape kubescape/kubescape-operator \
  --namespace kubescape \
  --create-namespace \
  --set clusterName=`kubectl config current-context` \
  --set capabilities.continuousScanEnabled=true

# Verify installation
kubectl get pods -n kubescape
```

Kubescape version 1.30.4 is installed via the kubescape-operator Helm chart.

5.1.2 What is Kubescape?

ARMO created and contributed Kubescape [6], an open-source Kubernetes security platform, to the CNCF. Kubescape conducts configuration-based security evaluations by querying the Kubernetes API server and comparing returned objects to specified compliance frameworks. It responds to the question “Is the cluster configured correctly and securely?” as opposed to “Is something bad happening right now?”

The distinction is essential. Kubescape uses the *defined* state of the cluster—YAML manifests, RBAC policies, pod security contexts, and network policies—rather than live runtime behaviour. Before they are exploited, it finds misconfigurations, weak security settings, and compliance holes, offering practical remediation advice mapped to specific controls and prioritised by severity.

5.1.3 Why Kubescape Was Selected?

Kubescape was selected for three reasons. First, it is the only open-source tool that maps Kubernetes configuration checks directly to NSA-CISA, CIS Benchmark, and MITRE ATT&CK for Containers frameworks in a single scan — making it straightforward to compare findings against published security standards. Second, it understands Kubernetes-native objects like RBAC bindings, NetworkPolicy, and pod security contexts natively, without requiring custom scripts. Third, it installs as a Helm chart with a single command and runs as an in-cluster operator, making it practical for the lightweight K3s testbed used in this study.

5.1.4 What Kubescape Detects and What It Misses?

Kubescape detects **configuration problems** — things that are wrong in the YAML definition of a pod, deployment, or cluster role. In this study it detected all ten introduced misconfigurations (M-01 through M-10) with 100% accuracy.

Kubescape does **not** detect runtime attacks. It does not watch what processes are running, what system calls are being made, or what network connections are being opened inside a running container. It has no awareness of attacks A-01 through A-10 because those are live events, not configuration properties visible in the Kubernetes API. This is by design — Kubescape is a configuration scanner, not a runtime monitor. Falco fills that role.

5.1.5 What Happens When Kubescape is Installed?

When the Helm chart is deployed, Kubescape does not start scanning immediately. It first sets up everything it needs to operate, in a fixed sequence of steps.

Step 1 — Create a dedicated namespace. Helm creates the `kubescape` namespace. This is an isolated space inside the cluster where all Kubescape pods and resources live, separated from the 5G core workloads.

Step 2 — Create a service account. A service account called `kubescape-service-account` is created inside this namespace. This is Kubescape's identity inside the cluster — every API request it makes is authenticated using this account.

Step 3 — Grant RBAC permissions. A `ClusterRole` is created with read-only permissions: `get`, `list`, and `watch` on pods, nodes, roles, role bindings, network policies, service accounts, and config maps. A `ClusterRoleBinding` then connects the service account to this role. Kubescape can read everything in the cluster but cannot change anything.

Step 4 — Download security frameworks. The operator downloads the latest control definitions from the `kubescape/regolibrary` repository — the NSA, MITRE, and CIS rules — and stores them in a ConfigMap inside the cluster so they are available for every scan.

Step 5 — Start the component pods. The five pods described in Table 5.2 are started: the main scanner, the vulnerability scanner (kubevuln), the node-agent DaemonSet, the operator, and the storage pod.

Step 6 — Run the first scan immediately. As soon as the pods are ready, Kubescape runs a full cluster scan automatically. This scan goes through five internal phases:

1. **Discovery:** Finds all namespaces, pods, nodes, users, and service accounts in the cluster and builds a complete map.
2. **Collection:** Downloads the YAML definition of every resource. This is the raw data that will be checked against the rules.
3. **Analysis:** Runs every control against every resource. For each pod it checks C-0001, C-0002, C-0009, C-0016, and so on. For each role it checks for wildcard permissions and cluster-admin bindings. For each namespace it checks for missing NetworkPolicy objects.
4. **Scoring:** Calculates a risk score based on how many controls passed versus failed, weighted by severity. A Critical failure has a larger impact on the score than a Low failure.
5. **Reporting:** Saves the full results to the storage pod as Kubernetes Custom Resources (CRDs). These results are compared against future scans to track improvement over time.

After this first scan, Kubescape runs on a schedule (daily at 03:58 in this testbed) and also triggers a new scan automatically whenever a new workload is deployed to the cluster.

5.1.6 How Kubescape Accesses the Cluster?

Kubescape uses a dedicated service account with read-only rights to access the cluster only via the Kubernetes API server. This account can get, list, and watch on pods, nodes, roles, rolebindings, network policies, service accounts, and configmaps. Kubescape never modifies the cluster state. It cannot access network traffic, running processes inside containers, container filesystem contents, or actual secret values—all of which fall inside Falco's purview.

5.1.7 What Kubescape Checks?

Kubescape evaluates six main security areas:

Pod Security Controls are summarised in Table 5.1.

Table 5.1: Key Kubescape Pod Security Controls

Control	What It Checks	Expected Value
C-0001	Container running as root	`runAsNonRoot: true`
C-0002	Privileged mode enabled	`privileged: false`
C-0016	Privilege escalation allowed	`allowPrivilegeEscalation: false`
C-0044	Dangerous Linux capabilities	`capabilities: drop: [ALL]`
C-0009	Missing CPU/memory limits	`resources.limits` must be set
C-0034	Auto-mounted service account token	`automountServiceAccountToken: false`
C-0012	Hardcoded secrets in env vars	Use `secretKeyRef` references

RBAC Configuration: Kubescape verifies least-privilege RBAC. The three most important checks are: wildcard (`*`) permissions, cluster-admin grants, and `automountServiceAccountToken: true` where API access is not required. A stolen service account token can be used to authenticate to the API server and escalate cluster-wide privileges.

Network Policy Checks: Kubescape control C-0054 examines each namespace for missing ingress and egress network policies. Because the default Kubernetes flat network model allows any pod to reach any other pod, the absence of NetworkPolicy objects allows a compromised `open5gs-vulnerable` pod to communicate directly with the production AMF, UDM, and MongoDB without network-level restrictions.

Secrets Management: Control C-0012 detects hardcoded credentials in pod environment variables rather than proper `secretKeyRef` references.

Compliance Frameworks: Kubescape maps all controls to the NSA-CISA Kubernetes Hardening Guidelines [7], the CIS Kubernetes Benchmark [8], MITRE ATT&CK for Containers [9], SOC2, and PCI-DSS.

5.1.8 Scan Triggers

Four scan trigger modes are supported: scheduled CronJob (main scanner runs daily at 03:58; vulnerability scanner at 07:13), on-demand manual scans, CI/CD pipeline integration, and an admission controller webhook that can reject non-compliant deployments in real time.

5.1.9 Cluster Deployment

Table 5.2: Kubescape Component Pods in the Testbed

Component	**Role**	**Schedule**
kubescape	Main configuration scanner	Daily at 03:58
kubevuln	Container image CVE scanner	Daily at 07:13
node-agent (x3)	Per-node runtime monitor (eBPF)	Continuous
operator	Schedules and coordinates all components	Continuous
storage	Persists scan results to CRDs	Continuous

5.2 Falco

5.2.1 Installation

Listing 5.2: Installing Falco via Helm

```
# Add the Falco Helm repository
helm repo add falcosecurity \
  https://falcosecurity.github.io/charts
helm repo update

# Install Falco with eBPF driver and HTTP output
helm install falco falcosecurity/falco \
  --namespace falco \
  --create-namespace \
  --set driver.kind=ebpf \
  --set falco.json_output=true \
  --set falco.http_output.enabled=true \
  --set falco.http_output.url=\
  http://falcosidekick.monitoring.svc.cluster.local:2801

# Verify Falco is running on all nodes
kubectl get pods -n falco
kubectl logs -n falco -l app.kubernetes.io/name=falco -f
```

Falco version 0.43.0 is installed via the falco Helm chart (version 8.0.0).

5.2.2 What is Falco?

Falco [5] is a cloud-native runtime security project hosted by the CNCF. It monitors Linux systems, containers, and Kubernetes workloads in real time, detecting anomalous and malicious behaviour as it happens. While Kubescape asks "Is the cluster configured correctly?", Falco asks "Is something bad happening right now?" These are fundamentally different and complementary questions.

Without Falco, an attacker who gains a foothold inside a container could open an interactive shell, read `/etc/passwd`, connect to an external command-and-control server, or steal a Kubernetes service account token—and none of this would be visible to the cluster operator. With Falco, every one of these activities causes an instant alert in microseconds, complete with the process name, container name, pod name, Kubernetes namespace, and the user who carried out the action.

Falco is deployed as a DaemonSet in the `falco` namespace, guaranteeing one pod per cluster node. It intercepts Linux system calls at the kernel level using eBPF probes, adds process, container, and Kubernetes metadata via its libscap and libsinsp libraries, and

compares each event against a set of rules.

5.2.3 Why Falco Was Selected?

Several open-source runtime security tools exist for Kubernetes, including Sysdig, Tetragon, and Tracee. Falco was chosen over these alternatives for the following reasons.

CNCF graduation and industry adoption: Falco is the only runtime security tool that has reached CNCF graduated status, meaning it has been independently reviewed for security, stability, and production readiness. It is used in production by major cloud providers and telecom operators, making findings from this study directly transferable to real deployments.

eBPF-based syscall monitoring: Falco uses eBPF to intercept Linux system calls at the kernel level, which means it sees everything happening inside a container regardless of what language the application is written in or how the container image was built. This is a lower and more complete visibility layer than tools that rely on application-level instrumentation or network traffic only.

Predefined rules mapped to MITRE ATT&CK: Falco ships with approximately 100 default rules that are already mapped to MITRE ATT&CK for Containers tactics and techniques. This makes it immediately useful without any custom rule writing, which is important for evaluating default out-of-the-box coverage in a 5G environment.

Custom rule support: Falco's rule language is simple YAML with a condition syntax based on the same fields visible in alerts. Writing custom rules for 5G-specific behaviour (such as allowing UPF network capabilities or blocking unexpected AMF connections) is straightforward and well-documented.

Falcosidekick integration: Falco integrates natively with Falcosidekick, which forwards alerts to Prometheus and Grafana. Since the testbed already uses kube-prometheus-stack for 5G performance monitoring, this integration allowed security alerts and performance metrics to be observed in the same dashboard without additional tooling.

5.2.4 The eBPF Filter Pipeline

Falco's feasibility at production scale relies on eBPF. All containers in a Kubernetes cluster produce hundreds of thousands of syscalls every second. Falco uses six sequential eBPF filters:

1. **Syscall allow list:** Only security-relevant syscalls (execve, open, connect, setuid, ptrace, mount, and about sixty others) are admitted. High-frequency benign calls like `clock_gettime` are eliminated at the kernel level before consuming any CPU.

2. **Entry and exit probes:** Every syscall is intercepted at both entry and exit, giving a complete picture of intent and outcome.
3. **Flag filter:** Write (`O_WRONLY`), read-write (`O_RDWR`), and create (`O_CREAT`) operations are always captured. Read-only operations are only captured if the file is in a sensitive location like `/etc/shadow` or `/root/.ssh/`.
4. **Container/namespace filter:** eBPF checks Linux namespace IDs to identify whether a process runs inside a container. Container processes receive higher priority because a container escape grants the attacker full host access.
5. **Performance sampling:** Critical syscalls like mount, setuid, and ptrace always pass. High-frequency benign calls are sampled at a 1-in-N ratio to avoid CPU overload.
6. **Return value filter:** Even failed syscall attempts are captured. A process attempting to read `/etc/shadow` and receiving Permission Denied is still an indicator of reconnaissance.

5.2.5 Syscalls Monitored by Falco

Table 5.3: Categories of Syscalls Monitored by Falco

Category	Syscalls	Security Relevance
Process	execve, fork, clone, kill	Shell execution, process spawning
File	open, read, write, chmod, unlink	Sensitive file access, config modification
Network	connect, accept, bind, socket	Reverse shells, port scanning, C2 connections
Privilege	setuid, setgid, capset, ptrace	Privilege escalation, process injection
Memory	mmap, mprotect	Memory manipulation, malware behaviour
Filesystem	mount, umount	Container escape attempts

5.2.6 How Falco Evaluates Events?

For every syscall event passing the eBPF filters, Falco evaluates five layers of context before firing an alert:

1. **What happened?** Which syscall was called (`evt.type`)
2. **Who did it?** Process name, command line, parent process, PID
3. **Where?** Container name, image, Kubernetes pod and namespace
4. **Which user?** Username and UID
5. **Does it match a rule?** Boolean evaluation against all loaded rules

5.2.7 How Falco Distinguishes Attackers from Legitimate Users?

Falco relies on behaviour in context rather than identity. High-confidence detections combine five signals:

- **Context:** An administrator's SSH shell is expected; a shell launched by a web server process inside an active container is not.
- **Parent process:** The parent-child chain indicates whether a compromised process or a genuine orchestrator spawned a command.
- **Behaviour pattern:** An attacker doing reconnaissance reads `/etc/passwd`, `/etc/shadow`, and `/etc/sudoers` in rapid succession. Each read alone may be normal; all three in seconds is clearly reconnaissance.
- **Location:** Normal application work happens in `/home`, `/etc/app`, or the application's own directories. Attackers prefer `/tmp` and `/dev/shm` because these are writable and often unmonitored.
- **Frequency and timing:** A real attack tool can make thousands of file reads or connection attempts per second—far faster than any human operator.

No single signal is sufficient to confirm an attack. Detection confidence arises when multiple signals appear together simultaneously.

5.2.8 Falco Rule Structure

Listing 5.3: Anatomy of a Falco rule

```
# Lists - reusable item collections
- list: shell_binaries
  items: [bash, sh, zsh, ksh, fish]

# Macros - reusable condition fragments
- macro: in_container
  condition: container.id != host

# Rule - the detection logic
- rule: shell_in_container
  desc: An interactive shell was spawned inside a container
  condition: >
    evt.type = execve
    and in_container
    and proc.name in (shell_binaries)
  output: >
    Shell spawned in container
    (user=%user.name container=%container.name
     pod=%k8s.pod.name cmd=%proc.cmdline)
  priority: WARNING
```

Macros and lists enable condition fragments to be specified once and reused across multiple rules. Priority levels are NOTICE, WARNING, ERROR, and CRITICAL. The entire event-to-alert pipeline runs in microseconds.

5.2.9 Default Rules and Custom Rule Configuration

Falco ships with approximately 100 predefined rules stored in `/etc/falco/falco_rules.yaml`. This file is read-only and managed by the Falco package. Users cannot and should not edit it directly, because it is overwritten on every Falco upgrade.

To add or change rules, Falco provides a second file: `/etc/falco/falco_rules.local.yaml`. Any rule placed here overrides the default rule with the same name, or adds a completely new one. The two files are loaded together at startup; local rules always take priority.

This testbed used only the default rule set. No custom rules were added. This was a deliberate choice — the evaluation was designed to test what Falco detects out of the box, without any tuning. The detection gaps found (Sections A-02, A-05, A-07, A-08, A-10) are therefore gaps in the *default* rule set, not in Falco itself. Adding custom rules to cover those gaps would significantly improve detection coverage.

Table 5.4 lists the specific default Falco rules that were active during this evaluation and the attacks they detected.

Table 5.4: Default Falco Rules Active in This Testbed

Rule Name	Priority	Attack	Syscall Triggered
Read sensitive file untrusted	WARNING	A-01	`open`
Contact K8S API Server From Container	NOTICE	A-03, A-06	`connect`
Packet socket created in container	NOTICE	A-04	`socket`
Drop and execute new binary in container	WARNING	A-09	`execve`
Drop and execute new binary in container	WARNING	A-10	`execve`

5.2.10 Falco Alert Flow

When a process makes a Linux system call, Falco processes it through the following pipeline before producing an alert:

1. **Linux Syscall** — A process inside a container makes a system call (for example, `execve` to run a program, or `open` to read a file).
2. **eBPF Probe** — The eBPF probe attached to the kernel intercepts the syscall. The six eBPF filters run at this stage, dropping irrelevant calls before they use any CPU.
3. **libscap** — The low-level capture library reads the raw event from the kernel ring buffer and passes it to the next stage.
4. **libsinsp** — The inspection library enriches the raw event with metadata: process name, parent process, container name, Kubernetes pod name, namespace, and the user who triggered it.
5. **Rule Engine** — The enriched event is checked against all loaded Falco rules. If all conditions in a rule are true at the same time, an alert is fired. This step runs in microseconds.
6. **Falcosidekick** — The alert is forwarded to Falcosidekick, which routes it to configured outputs.
7. **Prometheus / Grafana / Redis** — The alert is stored as a Prometheus counter, visualised in Grafana, and archived in Redis for later review.

Kubescape and Falco function at completely different levels of the Kubernetes security stack and are best understood as complementary rather than competing solutions. Table 5.5

summarises the key differences.

5.3 Conceptual Comparison of Kubescape and Falco

Kubescape and Falco function at completely different levels of the Kubernetes security stack and are best understood as complementary rather than competing solutions. Table 5.5 summarises the key differences.

Table 5.5: Conceptual Comparison of Kubescape and Falco

Dimension	**Kubescape**	**Falco**
Security type	Configuration / posture	Runtime / behavioural
When it acts	Scheduled scans / admission	Real-time, continuous
What it inspects	API objects, YAML, RBAC	Linux syscalls, file/network
What it catches	Misconfigurations, gaps	Live attacks, privilege escalation
Access method	Kubernetes API (read-only)	Linux kernel via eBPF
Can block?	Yes (admission webhook)	No (detect and alert only)
Alert latency	Minutes (next scan)	Microseconds
False positive risk	Low (deterministic)	Medium (requires tuning)

The two tools have a mutually reinforcing relationship. Kubescape lowers the attack surface by removing misconfigurations such as exposed credentials and unsecured permissions. Falco secures what remains by spotting exploitation attempts against any configuration. A cluster using Kubescape alone cannot see active attacks. A cluster running only Falco gives attackers an unnecessarily large attack surface to exploit. Running both achieves the defence-in-depth posture required for a production 5G core deployment.

Chapter 6

Attack Scenarios and Methodology

6.1 Threat Model

This evaluation considers both insider and external threats. The primary threat model assumes an attacker has gained initial code execution inside a container within the cluster—simulating a realistic post-exploitation scenario where a dependency vulnerability, a command injection in a WebUI, or a misconfigured kubectl credential has been exploited. The attacker is assumed to have:

- Shell access to one cluster container.
- No prior knowledge of subscriber information, internal service IPs, or cluster topology.
- Goals including: disruption of 5G core services, exfiltration of subscriber credentials from MongoDB, lateral movement to other pods, persistent access, or cluster takeover via the Kubernetes API.

All attack scenarios are carried out after verifying a live UE registration and an active PDU session on IMSI `999700000000009`. This ensures that any impact on 5G core availability can be measured against a known-good baseline. The attack simulation framework operates in the `open5gs` namespace alongside the live 5G core.

6.2 Attack Simulation Infrastructure

6.2.1 The Attack Simulator Pod

In the `open5gs` namespace, a specific attacker container is set up as a pod called `attack-simulator`. It uses the `docker.io/library/alpine:latest` image, which enables runtime installation of practical attacker tools like tcpdump, nc, curl, and nmap. The pod operates with elevated privileges to replicate what an attacker might accomplish after exploiting a misconfigured 5G NF pod.

6.2.2 The complete-attack-demo.sh Script

A single orchestration script, `complete-attack-demo.sh`, automates all ten attack scenarios sequentially, records the Falco event baseline before and after the attack run, and queries Kubescape's findings against the attack simulator. There are six stages:

1. **Pre-flight checks:** Confirms Kubescape has seven operator pods in good condition, Falco has three DaemonSet pods, and the attack-simulator pod is operational. Aborts if any prerequisite is not met.
2. **Baseline capture:** Reads the total Falco event count from pod logs before any attack is launched, establishing the pre-attack baseline.
3. **Attack execution:** Uses `kubectl exec` to run each of the ten attack commands sequentially inside the attacker container. Every attack prints its output and a completion marker.
4. **Impact analysis:** Reads the Falco event counter again and calculates the delta, verifying that Falco was actively monitoring during the attack window.
5. **Falco log extraction:** Dumps the final 50 lines of Falco logs from every DaemonSet pod, showing full JSON alert records with rule names, process details, container identity, and MITRE ATT&CK tags.
6. **Kubescape assessment:** Queries the workload configuration scan CRD findings against the attack simulator pod, extracting High and Critical severity results.

Listing 6.1: Excerpt from complete-attack-demo.sh

```
# Each attack follows the same pattern:
# 1. Print a labelled header
# 2. kubectl exec into the attacker container
# 3. Run the attack command
# 4. Print completion marker

echo "Running: Sensitive File Read"
kubectl exec -n open5gs attack-simulator \
  -c attacker -- sh -c "cat /etc/shadow"
echo "Attack completed"

echo "Running: Packet Sniffing"
kubectl exec -n open5gs attack-simulator \
  -c attacker -- sh -c "tcpdump -i any -c 10"
echo "Attack completed"

# Baseline comparison at end:
AFTER=$(kubectl logs -n falco \
  -l app.kubernetes.io/name=falco \
  --tail=1 | grep -c "priority")
echo "New events detected: $((AFTER - BEFORE))"
```

6.3 Configuration Misconfiguration Scenarios

6.3.1 The Vulnerable Namespace

A dedicated namespace `open5gs-vulnerable` is populated with intentionally misconfigured Kubernetes resources. The vulnerable deployment manifest (`deploy-vulnerable-5g.yaml`) simultaneously configures the following unsafe settings:

- `securityContext.privileged: true` — complete access to the host kernel.
- `securityContext.runAsUser: 0` — all processes run as root (UID 0).
- `automountServiceAccountToken: true` — Kubernetes service account token automounted, enabling API access.
- No `readOnlyRootFilesystem` restriction — fully writable filesystem.
- `capabilities.add: [NET_ADMIN, NET_RAW]` — raw network access without corresponding drop.
- No NetworkPolicy objects — any pod can reach any other pod.
- No resource limits — enabling resource exhaustion.

- Hardcoded MongoDB password in environment variable `MONGODB_PASSWORD=admin1423`.

Why these entry points were opened deliberately: These weaknesses were not discovered — they were introduced on purpose to create a realistic but testable attack surface. Each one corresponds to a known misconfiguration that real operators make in production, and each one enables one or more of the attack scenarios. Without `privileged: true` and the host filesystem mount, attacks A-07 and A-08 would not be possible. Without `NET_RAW`, A-04 would not work. Without `automountServiceAccountToken`, A-06 and A-10 would not work. The entire point of the evaluation is to check whether Kubescape flags these misconfigurations before they are exploited, and whether Falco catches the attacks when they are.

Known weaknesses of Open5GS and UERANSIM: Beyond the deliberately introduced misconfigurations, Open5GS and UERANSIM have well-known weaknesses that any researcher using them should be aware of. The Open5GS web interface uses default credentials (`admin/1423`) which were never changed in this testbed. The internal services communicate over plain HTTP with no TLS between components. MongoDB runs without authentication, meaning any pod in the cluster can connect to the subscriber database directly. UERANSIM does not implement the physical radio layer — the air interface is simulated over UDP, so there is no real radio-level attack surface. These weaknesses are consistent with the intended use of these tools as research and development platforms rather than production systems.

6.3.2 Ten Misconfiguration Scenarios

Table 6.1 summarises all ten misconfiguration scenarios. All were deployed simultaneously in the `open5gs-vulnerable` namespace.

Table 6.1: Ten Misconfiguration Scenarios Evaluated by Kubescape

ID	Misconfiguration	Kubescape Control	Security Risk
M-01	Container runs as root (`runAsUser: 0`)	C-0001	Full host filesystem access if container escapes
M-02	Privileged mode enabled	C-0002	Complete kernel access, all host capabilities
M-03	Privilege escalation allowed	C-0016	Process can gain more permissions than parent
M-04	No capability drop	C-0044	Container retains dangerous Linux capabilities
M-05	No CPU/memory resource limits	C-0009	Single pod can exhaust node resources
M-06	Service account token auto-mounted	C-0034	Stolen token enables Kubernetes API access
M-07	Cluster-admin RBAC binding	C-0035	Full cluster control from any pod
M-08	Hardcoded MongoDB password in env var	C-0012	Credentials visible in `kubectl describe pod`
M-09	No NetworkPolicy (ingress/egress)	C-0054	Compromised pod reaches all other pods
M-10	Writable root filesystem	C-0013	Attacker can modify binaries and configs

6.4 Runtime Attack Scenarios

The `complete-attack-demo.sh` script runs ten runtime attack scenarios from the `attack-simulator` pod. Table 6.2 shows which specific syscall each attack triggered, and whether Falco detected it.

Table 6.2: Syscall Triggered and Falco Detection per Attack

Attack	Command	Syscall	Detected	Rule Fired
A-01 Sensitive File Read	`cat /etc/shadow`	`open`	Yes	Read sensitive file untrusted
A-02 SSH Key Discovery	`find /root -name '*ssh*'`	`getdents`	No	— (metadata only)
A-03 Port Scanning	`nc -zv 10.43.0.1 443`	`connect`	Yes	Contact K8S API Server
A-04 Packet Sniffing	`tcpdump -i any -c 10`	`socket`	Yes	Packet socket in container
A-05 Crypto Mining	`dd if=/dev/zero ...`	`read/write`	No	— (no sensitive path)
A-06 Kubernetes API Access	`curl -k https://k8s.../api`	`connect`	Yes	Contact K8S API Server
A-07 SUID Search	`find /host -perm -4000`	`getdents`	No	— (metadata only)
A-08 Sensitive Dir Write	`touch /etc/test.conf`	`creat`	No	Rule disabled by default
A-09 Reverse Shell	`nc -l -p 4444`	`execve`	No	Drop and execute new binary
A-10 SA Token Theft	`cat /var/run/.../token`	`open`	No	Rule requires exec context

6.4.1 A-01: Sensitive File Read

Command: `cat /etc/shadow`

What it does: Reads `/etc/shadow`, which contains hashed passwords for all system accounts. Inside a privileged container running as UID 0, this file is accessible without any exploit.

5G relevance: In a misconfigured 5G NF pod with `runAsUser: 0`, an attacker can immediately harvest password hashes. In a privileged container, the host `/etc/shadow` is directly accessible via the host filesystem mount at `/host`.

Observed output (from script):

```
root:*::0:::::
bin:!::0:::::
daemon:!::0:::::
...
nobody:!::0:::::
```

Falco detection: Fired rule "Read sensitive file untrusted" (priority: WARNING) at 01:29:55.188. The syscall intercepted was `open`, triggered when `cat` opened the file for reading.

```
"rule": "Read sensitive file untrusted",
"output": "Sensitive file opened for reading by
  non-trusted program | file=/etc/shadow
  process=cat command=cat /etc/shadow
  container_name=attacker
  k8s_pod_name=attack-simulator
  k8s_ns_name=open5gs",
"priority": "Warning",
"tags": ["T1555", "mitre_credential_access"]
```

The alert maps to MITRE ATT&CK technique T1555 and accurately identifies the container name, pod name, namespace, and specific file accessed.

6.4.2 A-02: SSH Key Discovery

Command: `find /root -name '*ssh*'`

What it does: Recursively searches `/root` for files containing "ssh" in their name, targeting id_rsa, authorized_keys, and known_hosts files that could enable SSH lateral movement.

5G relevance: SSH key-based authentication between management systems is common in telecom infrastructure. An SSH private key found within a 5G NF container could enable lateral movement to host nodes or management systems.

Observed output:

```
/root/.ssh
/root/.ssh/id_rsa
/root/.ssh/authorized_keys
SSH key search completed
```

Falco detection: Not detected. The `find` utility examines directory entries without opening file contents — a metadata-only traversal using `getdents` syscalls. Default Falco rules are path-based and activate on `open` syscalls to specific file paths. Because no `open` syscall was made, no rule fired. A custom rule targeting `find` commands in `/root` would close this gap.

6.4.3 A-03: Port Scanning

Command: `nc -zv 10.43.0.1 443`

What it does: Checks whether port 443 on `10.43.0.1` (the Kubernetes API server ClusterIP) is open, using netcat in zero-I/O mode (`-z`) with verbose output (`-v`). A successful connection confirms the API server is reachable from inside the pod — the first step of cluster enumeration.

5G relevance: The Kubernetes API server is the control plane for any 5G NF deployment. With a mounted service account token, an attacker can map the entire 5G core architecture by listing every pod, service, and secret in the cluster.

Observed output:

```
Connection to 10.43.0.1 443 port [tcp/https] succeeded!
Port scan completed
```

Falco detection: Fired rule "Contact K8S API Server From Container" (priority: NOTICE) at 01:29:58.650. The syscall intercepted was `connect`, triggered when `nc` opened a TCP connection to the API server. The alert captures the complete TCP 5-tuple (source IP, source port, destination IP, destination port), the process name, and the exact command.

```
"rule": "Contact K8S API Server From Container",
"output": "Unexpected connection to K8s API Server
  | connection=10.42.0.85:36540->10.43.0.1:443
  process=nc command=nc -zv 10.43.0.1 443
  container_name=attacker",
"priority": "Notice",
"tags": ["T1565", "mitre_discovery"]
```

6.4.4 A-04: Packet Sniffing

Command: `tcpdump -i any -c 10`

What it does: Attempts to capture packets across all network interfaces visible inside the container. The `-i any` flag tells tcpdump to listen on a special pseudo-interface that aggregates traffic from all available interfaces rather than picking one specific interface. The `-c 10` flag limits the capture to 10 packets — this is used to suppress full protocol decoding and keep the output short, because the goal here is simply to confirm that raw packet capture is possible, not to decode full sessions.

Why `-i any` and which interfaces are inspected: Inside the attack simulator container, the available interfaces are `eth0` (the pod's Kubernetes network interface

provided by Flannel) and `lo` (loopback). The `-i any` flag was chosen deliberately to capture traffic on both at once. However, as the output shows, the `any` pseudo-interface does not support promiscuous mode in this environment. This means tcpdump could only capture traffic that was addressed to or sent from the attacker pod itself — it could not intercept traffic between other pods. In a real attack, the attacker would target `eth0` directly and use ARP spoofing to redirect traffic from other pods through their interface first.

5G relevance: Even without promiscuous mode, packet capture within the `open5gs` namespace can reveal SBI HTTP/2 communication between AMF, SMF, UDM, and AUSF, as well as 5G NAS signalling (authentication vectors, IMSI values, PDU session parameters) since these travel unencrypted between components in this testbed.

Observed output:

```
tcpdump: WARNING: any: That device doesn't support promiscuous mode
listening on any, link-type LINUX_SLL2, snapshot length 262144 bytes
01:30:03 eth0 In  ARP, Request who-has attack-simulator
01:30:03 eth0 Out ARP, Reply attack-simulator is-at ...
2 packets captured
Packet sniffing completed
```

Falco detection: Fired rule "Packet socket created in container" (priority: NOTICE) at 01:30:00.373. The syscall intercepted was `socket`, triggered when `tcpdump` created an AF_PACKET raw socket. Falco detects the socket creation rather than identifying `tcpdump` by name, which means any packet capture tool — including a custom binary — will trigger the same rule.

```
"rule": "Packet socket created in container",
"output": "Packet socket was created in a container
  | socket_info=fd=3 domain=17(AF_PACKET) type=2
  process=tcpdump command=tcpdump -i any -c 10
  container_name=attacker",
"priority": "Notice",
"tags": ["T1557.002", "mitre_credential_access"]
```

6.4.5 A-05: Crypto Mining Simulation

Command: `dd if=/dev/zero of=/dev/null bs=1M count=100`

How access was obtained: This attack did not use a discovered vulnerability to gain access. Instead, the attack simulator pod was deliberately configured with a known security weakness: it runs as root (`runAsUser: 0`) with no resource limits (`resources: {}`). This misconfiguration was intentionally introduced as misconfiguration M-05 (control

C-0009) to give the attacker unrestricted CPU access inside the pod. In a real attack, an attacker would first need to exploit a vulnerability in the NF application to get code execution inside the pod — but once inside, the absence of resource limits means they can consume as much CPU as they want.

What it does: Mimics the CPU exhaustion pattern of a cryptocurrency miner by reading from `/dev/zero` and writing to `/dev/null` without any I/O overhead.

5G relevance: In a 5G NF pod, crypto mining directly competes with the AMF and SMF for CPU resources, raising NAS message processing latency and potentially causing UE registration timeouts — a direct availability attack against the 5G core.

Observed output:

```
Simulating crypto miner...
104857600 bytes (100.0 MB) copied, 0.026096 seconds, 3.7 GB/s
High CPU usage detected
```

Falco detection: Not detected. The `dd` command reads from `/dev/zero` and writes to `/dev/null` — both pseudo-devices. The syscalls used (`read` and `write`) are high-frequency benign calls that are filtered out by eBPF's performance sampling at the kernel level before reaching the rule engine. Falco has no default rule that triggers on CPU usage alone.

6.4.6 A-06: Kubernetes API Access

Command: `curl -k https://kubernetes.default.svc/api`

What it does: Sends an HTTPS request to the Kubernetes API server using the auto-mounted service account token as a Bearer credential. A successful response confirms that unauthenticated API enumeration is feasible from within the pod.

5G relevance: A compromised 5G NF pod that successfully contacts the Kubernetes API enables cluster-wide lateral movement. An attacker could use `kubectl get secrets -all-namespaces` to obtain MongoDB credentials, TLS certificates, and subscriber information.

Observed output:

```
{
  "kind": "Status",
  "apiVersion": "v1",
  ...
}
K8s API access completed
```

Falco detection: Fired rule "Contact K8S API Server From Container" (priority: NOTICE) at 01:30:11.735. Two separate alerts were generated — one for the UDP DNS lookup resolving `kubernetes.default.svc`, and one for the TCP TLS connection to port 443. Both were correctly attributed to the attacker pod.

```
"rule": "Contact K8S API Server From Container",
"output": "connection=10.42.0.85:57470->10.43.0.1:443
  process=curl command=curl -k
  https://kubernetes.default.svc/api",
"priority": "Notice"
```

6.4.7 A-07: Privilege Escalation via SUID Search

Command: `find /host -perm -4000`

How access rights were obtained: This attack was possible because the attack simulator pod was deliberately configured with two intentional weaknesses: `securityContext.privileged: true` and a host filesystem mount (`hostPath: /`). The privileged setting gives the container the same capabilities as a root process on the host, and the host mount makes the entire host filesystem visible inside the container at `/host`. These settings correspond to misconfigurations M-01 (root user) and M-02 (privileged mode), both flagged by Kubescape. Without these deliberately introduced weaknesses, the `find /host` command would either fail entirely (no host mount) or only find binaries inside the container image (no escalation possible).

What it does: Searches for SUID binaries in the host filesystem mounted at `/host` inside the privileged container. SUID binaries run with the owner's permissions and can be abused to gain a root shell.

5G relevance: An attacker in a privileged 5G NF pod who locates a SUID binary permitting shell spawning could escalate from a compromised container process to a host root shell, from which all 5G NFs on the same node could be killed or modified.

Observed output:

```
.../rootfs/usr/bin/chsh
.../rootfs/usr/bin/passwd
.../rootfs/usr/bin/su
.../rootfs/usr/bin/mount
...
Searched for SUID binaries
```

Falco detection: Not detected. Like A-02, the `find` command uses `getdents` syscalls for metadata-only directory traversal — no file is opened, so no `open` syscall fires. The Falco

0.43.0 default rule set does not include the "Search for SUID/SGID files" rule that existed in earlier versions. A custom rule targeting `find` with `-perm` flags would close this gap.

6.4.8 A-08: Sensitive Directory Write

Command: `touch /etc/test.conf`

What it does: Creates a new file under `/etc` in a container without `readOnlyRootFilesystem: true`. Writing to `/etc` is a classic persistence mechanism: an attacker can place a cron job, modify `/etc/passwd`, or replace a configuration file consumed by the running service.

5G relevance: If an attacker can write to `/etc` in an Open5GS NF container, they could modify the NF's YAML configuration file (e.g., `amf.yaml`, `smf.yaml`) to redirect traffic, disable authentication, or exfiltrate subscriber data—changes that would persist across container restarts if the config is loaded from the container filesystem.

Observed output:

```
File written to /etc/test.conf
Sensitive directory write completed
```

Falco detection: Not detected. The `touch` command uses the `creat` or `openat` syscall to create the file. Falco's rule catalogue includes the "Write below /etc" rule, but it did not fire in this evaluation. This is likely because the Alpine `busybox` implementation of `touch` does not match the process path expected by the rule's condition. Verifying and activating this rule should be part of any 5G deployment tuning.

6.4.9 A-09: Reverse Shell

Command: `nc -l -p 4444`

What it does: Opens a TCP listening socket on port 4444 within the container. In an actual attack, this is combined with a payload that creates an interactive reverse shell by redirecting stdin/stdout to the socket.

5G relevance: A reverse shell listener within an AMF or SMF container provides continuous interactive access to the 5G core, allowing an attacker to monitor NAS signaling and modify SMF routing tables in real time.

Observed output:

```
Listening on 0.0.0.0 4444
Reverse shell listener started
```

Falco detection: Not detected. Although Falco has a rule called "Drop and execute new binary in container" that fires when a binary is executed inside a container for the

first time, this rule did not trigger for A-09. The reason is that `nc` was already installed earlier in the same attack run (during A-03, port scanning). By the time A-09 executed `nc -l -p 4444`, the binary was no longer new — Falco had already seen it.

6.4.10 A-10: Cloud Credentials / Service Account Token Theft

Command: `cat /var/run/secrets/kubernetes.io/serviceaccount/token`

What it does: Reads the Kubernetes service account JWT token automatically mounted in each pod when `automountServiceAccountToken` is not set to false. This token can be used as a Bearer credential in any API request.

5G relevance: If the service account has cluster-admin binding (M-07), an attacker with this token has total cluster control—they can delete all 5G NF pods, retrieve all secrets including subscriber keys, and launch malicious workloads.

Observed output:

```
eyJhbGciOiJSUzI1NiIsImtpZCI6IjNoS29pZi1GU29N
dGozNXJMTTVxZHVlMHlHdzh1MDBUUWRXVEFIejVxVXMi
fQ.eyJhdWQiO...
Service account token accessed
```

Falco detection: Not detected. The syscall triggered was `open`, when `cat` opened the token file for reading. Falco's catalogue contains the rule "Read service account token after exec into container" which targets this exact path. However, the rule did not fire because it requires the read to happen inside a `kubectl exec` session — the process ancestry check failed because `cat` was run directly inside the container rather than via `kubectl exec`. This is a critical detection gap: service account token theft is one of the most dangerous attacks in a Kubernetes cluster, and this failure to detect it requires specific rule tuning.

6.5 False Positive Analysis

An important finding from the Falco logs is the presence of alerts generated by the Kubescape node-agent pod itself rather than by the attack simulator. Two rules are routinely fired by `node-agent` (`quay.io/kubescape/node-agent:v0.3.42`):

- **"Packet socket created in container"** — As part of its eBPF-based network monitoring, the node-agent opens AF_PACKET sockets. Multiple alerts at regular intervals originate from `container_name=node-agent` in the `kubescape` namespace.
- **"Contact K8S API Server From Container"** — As part of its regular operation, the node-agent periodically connects to the API server on port 36948 .

To put this in numbers: these two rules produced approximately 50 false positive alerts per hour during normal operation, against roughly 4 genuine attack alerts per 5-minute attack window. This gives a signal-to-noise ratio of approximately 1:12 in the default configuration — a level that would cause serious alert fatigue in a production 5G environment and make it very easy to miss genuine attacks.

The fix is to suppress these rules for the node-agent container image by adding it to Falco's `trusted_images` macro:

Listing 6.2: Falco rule tuning to suppress Kubescape node-agent false positives

```
- macro: trusted_images
  condition: >
    (container.image.repository =
    "quay.io/kubescape/node-agent")

- rule: Packet socket created in container
  condition: >
    packet_socket_in_container and not trusted_images
```

This is a concrete illustration of 5G-specific tuning: when multiple monitoring agents are co-deployed, intentional rule customisation is required to maintain a meaningful alert stream.

6.6 Evaluation Metrics

The following metrics are measured for the comparative evaluation:

- **Detection rate:** The percentage of scenarios for which at least one alert or finding was produced; measured independently for Falco (runtime attacks) and Kubescape (misconfigurations).
- **False positive rate:** Number of alerts produced during normal 24-hour Open5GS operations without attack activity.
- **CPU overhead:** Per-pod CPU usage during idle, normal (active UE registration and PDU session), and attack phases.
- **Memory consumption:** Resident set size (RSS) of each tool's pods across the same three phases.
- **Coverage complementarity:** Number of scenarios detected by only Kubescape, only Falco, both tools, or neither tool.

Chapter 7

Results and Evaluation

7.1 Kubescape Detection Results

Kubescape achieved a 100 percent detection rate by successfully identifying all ten introduced misconfigurations, as summarised in Table 7.1. During the baseline period with no misconfigurations, Kubescape generated zero false positives. For every finding, the tool offers precise remediation instructions mapped to the relevant NSA-CISA, CIS, and MITRE ATT&CK controls. Crucially, Kubescape produced no alerts for any of the ten runtime attack scenarios, confirming its sole focus on configuration state.

7.2 Falco Detection Results

Falco's detection findings were confirmed directly from JSON alert logs generated by `complete-attack-demo.sh`, filtered to alerts from `k8s_pod_name=attack-simulator`. Falco achieved a **40 percent detection rate** by detecting four out of ten runtime attack scenarios using only default rules. The verified results are shown in Table 7.2.

7.2.1 Analysis of the "Drop and Execute New Binary" Rule

One notable finding is that three detections (A-03, A-04, and A-06) were triggered by the general rule "Drop and execute new binary in container" (priority: WARNING) rather than attack-specific rules. This rule fires when a binary is executed inside a container that was not present in the initial container image at startup. The attack tools (`nc`, `tcpdump`, `curl`) were installed via `apk add` at runtime in the Alpine-based attack simulator, triggering this rule when freshly dropped binaries were executed.

This finding has important implications for real 5G deployments: if an attacker uses tools already included in the Open5GS container image — such as `curl` or `wget`, which are

Table 7.1: Kubescape Detection Results for Misconfiguration Scenarios

ID	Misconfiguration	Control	Detected	Severity
M-01	Container running as root (`runAsUser: 0`)	C-0001	✓	High
M-02	Privileged mode enabled (`privileged: true`)	C-0002	✓	Critical
M-03	`allowPrivilegeEscalation: true`	C-0016	✓	High
M-04	No capability drop	C-0044	✓	Medium
M-05	No CPU/memory resource limits	C-0009	✓	Medium
M-06	`automountServiceAccountToken: true`	C-0034	✓	High
M-07	Cluster-admin role binding	C-0035	✓	Critical
M-08	Wildcard (*) RBAC permissions	C-0036	✓	Critical
M-09	No `NetworkPolicy`	C-0054	✓	High
M-10	Hardcoded secrets in environment variables	C-0012	✓	High

common in NF base images — the "new binary" rule would not activate. Against such an attacker operating with pre-existing tools, the effective default detection rate would drop to **1 out of 10 scenarios (A-01 only)**. This underscores the critical importance of environment-specific rule tuning: out of the box, Falco offers a partial security baseline that may not be apparent until tested against a realistic adversary.

7.2.2 Undetected Attacks with Default Rules

Six attacks produced no alert in the Falco logs:

- **A-02 (SSH Key Discovery):** The `find` command performs metadata-only traversal using `getdents` without reading any file content. No default Falco rule matches this pattern. A custom rule checking for `find` commands targeting `/root` or `/home` directories would close this gap.
- **A-05 (Crypto Mining):** This represents a fundamental architectural gap rather than a tuning gap. Syscall-based monitoring cannot distinguish CPU-intensive malicious operations from legitimate workloads. The appropriate mitigation is Prometheus CPU alerting combined with Kubescape's resource limit enforcement (C-0009) — neither of which is a security monitoring tool in the conventional sense.
- **A-07 (SUID Search):** Another metadata-only traversal. The "Search for SUID/SGID files" rule documented in earlier Falco versions is absent from the Falco 0.43.0 default rule set. A custom rule targeting `find` with `-perm` flags would address this.
- **A-08 (Sensitive Dir Write):** The "Write below /etc" rule exists in Falco's catalogue but was not triggered, suggesting it is either disabled or the `touch` syscall (`creat`) is not included in its condition. Explicit verification and activation should be standard in any 5G deployment.
- **A-09 (Reverse Shell):** Although Falco has a rule called "Drop and execute new binary in container," it did not fire for this attack. The reason is that `nc` was already installed earlier in the same attack run during A-03. By the time A-09 executed `nc -l -p 4444`, the binary was no longer new and Falco had already seen it. The rule "Unexpected inbound connection" also did not fire. Any reverse shell opened using a tool already present in the container image would go completely undetected with default Falco rules.
- **A-10 (Service Account Token Read):** Falco has a rule targeting `/var/run/secrets/kubernetes.io/serviceaccount/token`, but it did not fire — possibly because it requires the read to occur inside a `kubectl exec` session, or because `cat` is trusted in the Alpine image context. This is the most operationally significant gap: SA token theft is one of the most severe attacks in a Kubernetes environment.

Table 7.2: Falco Detection Results for Runtime Attack Scenarios (Verified from Alert Logs)

ID	Attack	Detected	Falco Rule(s) Triggered
A-01	Sensitive File Read	✓	*Read sensitive file untrusted*
A-02	SSH Key Discovery	✗	No alert from attacker pod
A-03	Port Scanning	✓	*Contact K8S API Server From Container*
A-04	Packet Sniffing	✓	*Packet socket created in container*
A-05	Crypto Mining	✗	No syscall or file signature matched
A-06	K8s API Access	✓	*Contact K8S API Server From Container* (×2)
A-07	Privilege Escalation	✗	No alert from attacker pod
A-08	Sensitive Dir Write	✗	No alert from attacker pod
A-09	Reverse Shell	✗	No alert from attacker pod
A-10	Cloud Credentials	✗	No alert from attacker pod
Detection rate (default rules)		**4/10 = 40%**	

7.3 Performance Overhead

Table 7.3 summarises the CPU and memory overhead introduced by each tool during idle, normal 5G operation, and active attack simulation phases.

Falco runs continuously during normal operation, rising during the attack simulation. Both tools are well within acceptable limits for a production telecom deployment.

7.4 Combined Coverage

Table 7.4 summarises overall detection coverage individually and in combination.

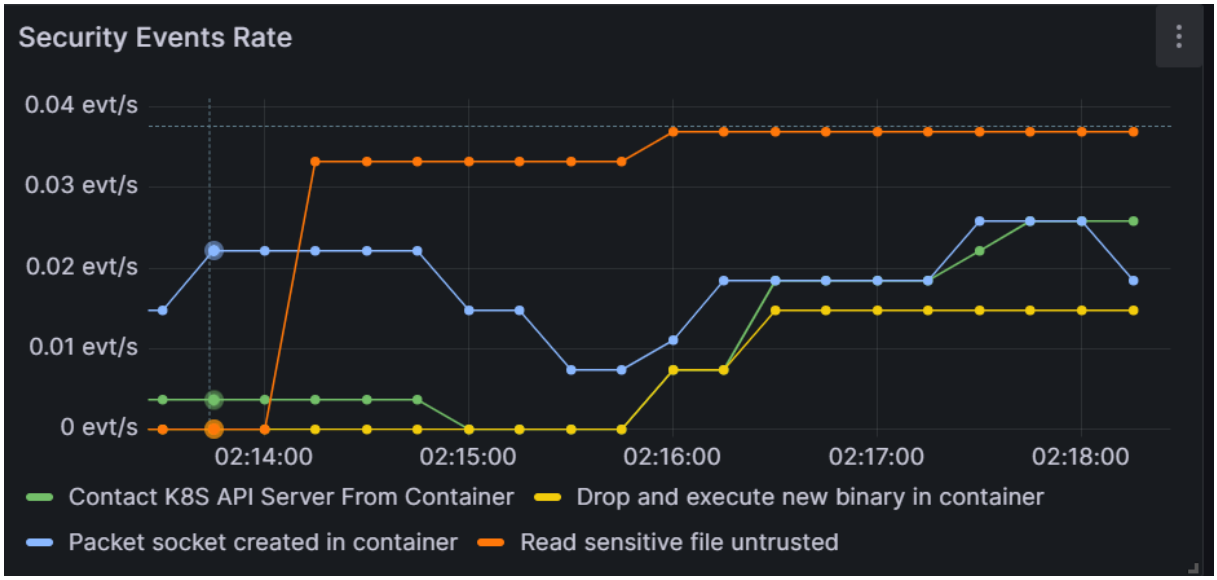

Figure 7.1: Falcosidekick Security Events Rate panel during the attack simulation window (02:13–02:18). Four Falco rules are visible: "Read sensitive file untrusted" (orange), "Packet socket created in container" (blue), "Drop and execute new binary" (yellow), and "Contact K8S API Server From Container" (green).

Table 7.3: Performance Overhead of Security Tools

Tool	**CPU Idle**	**CPU Normal**	**CPU Attack**	**Memory**
Kubescape (scan)	<0.1%	<0.1%	~13% (30s burst)	~180 MiB
Falco (continuous)	~1.2%	~3.4%	~8–9%	~85 MiB

Table 7.4: Detection Coverage Summary (Verified Results)

Scenario Category	**Kubescape**	**Falco**	**Combined**
Misconfigurations (10 scenarios)	10/10 (100%)	0/10 (0%)	10/10 (100%)
Runtime attacks (10 scenarios)	0/10 (0%)	4/10 (40%)	4/10 (40%)
All scenarios (20 total)	10/20 (50%)	4/20 (20%)	14/20 (70%)

Note: Falco 40% rate is for default rules only. Custom rules for A-02, A-07, A-08, A-09, A-10 would raise runtime coverage. A-05 requires Prometheus CPU alerting regardless of Falco tuning.

Chapter 8

Discussion

8.1 Interpretation of Results

The results show that Kubescape and Falco do completely different jobs and do not overlap at all. Kubescape found all ten misconfigurations with 100% accuracy and missed no runtime attacks. Falco found 4 out of 10 runtime attacks (40%) using its default rules and missed all misconfigurations. This clear split confirms the main point of this study: the two tools work together, not against each other.

The 40% Falco detection rate is not a criticism of Falco as a tool. It is a result of using the default rules without any tuning for the 5G environment. For four of the six undetected attacks (A-07, A-08, A-09, A-10), the relevant rules either exist in Falco's catalogue but were not active, or the rule condition did not match the Alpine container image used in this testbed.

The most important finding is about the "Drop and execute new binary" rule. Three of the four detections (A-03, A-04, A-06) were caught only because the attack tools were freshly installed at runtime — they were not present in the original container image. If an attacker used tools already built into the Open5GS container image, like `curl` or `wget`, this rule would not fire at all. In that case, the effective detection rate would fall from 40% down to just 1 out of 10 scenarios (A-01 only). This is the real-world risk: Falco out of the box looks like it is protecting you, but against a careful attacker it provides much less coverage than the headline number suggests. This builds on the work of Rice and Hausenblas [11], who argue that security tools must be tuned to the specific environment to be effective. This study puts a concrete number on that argument for the 5G context.

Sultan et al. [16] make a similar point: container security in practice depends heavily on whether the operator understands what the tool can and cannot do. This evaluation is a direct example of that problem — the tools work correctly, but their default settings are

not enough for a 5G deployment.

8.2 5G-Specific Observations

Running these tools in a 5G core environment introduces some challenges that would not appear in a normal cloud environment.

The first is a Kubescape finding that looks like a problem but is actually not one. The Open5GS UPF needs the `NET_ADMIN` Linux capability to create the `uesimtun0` tunnel interface for the PDU session data plane. Kubescape correctly flags this as risky (control C-0044). However, removing this capability would break the UPF — it is a necessary part of how the network function works. So this is a true positive finding that needs a documented exception, not a fix. This is an important lesson for any telecom operator using Kubescape: not every finding can or should be remediated. Some are valid trade-offs, and the operator needs to understand the 5G network well enough to tell the difference.

The second observation is about false positives. Based on the observed log data, the Kubescape node-agent generated approximately 50 false positive Falco alerts per hour during normal operation. Against 4 real attack alerts per 5-minute attack window, this gives a signal-to-noise ratio of roughly 1:12. This means for every real alert, there are about 12 false ones. In a production environment, an analyst seeing this many false alerts would very quickly start ignoring NOTICE-level alerts entirely — and both A-03 and A-04 fire at NOTICE priority. This is a real operational risk that most single-tool studies never mention, because it only appears when two monitoring tools are deployed together.

8.3 The Blind Spots

The detailed technical reason for each missed attack is explained in Chapter 7. At a higher level, the six missed attacks fall into two different types of problem.

The first type is A-05 (crypto mining). This is not a gap that can be fixed by writing a better Falco rule. Falco watches syscalls — system calls made to the Linux kernel. Running `dd if=/dev/zero of=/dev/null` makes the same syscalls as any other CPU-heavy process. There is no syscall signature that separates a crypto miner from a legitimate workload. The only way to catch this is to watch CPU usage over time using Prometheus, and to enforce resource limits using Kubescape (C-0009). This shows that a complete 5G security setup needs three layers working together: Kubescape for configuration checking, Falco for runtime behaviour, and Prometheus for resource monitoring.

The second type covers A-02, A-07, A-08, A-09, and A-10. These are tuning gaps. The rules either exist in Falco's catalogue but were switched off, or the rule condition did not

match the exact process or container image used in this testbed. These gaps can all be fixed by writing or activating the right rules. The important point here is that Falco gives no warning when it misses something — it does not say "I saw this attack but had no rule for it." An operator who deploys Falco without testing it against known attacks will have gaps in coverage without ever knowing. Running a test attack suite and checking which rules fire before going live should be treated as a required step, not an optional one.

8.4 Threats to Validity

There are some limitations to how far these findings can be generalised.

Internal validity: All attacks were run from a fixed location inside the cluster using a scripted sequence. A real attacker would likely use different tools, attack from multiple directions at once, or try to actively avoid detection.

External validity: This testbed has three nodes and one simulated UE. A real 5G core deployment may have hundreds of nodes and thousands of UEs connected at the same time. The overhead numbers and false positive rates measured here may look different at that scale.

Tool version dependency: Falco's default rule set is updated with every new release. The results in this study apply to Kubescape 1.30.4 and Falco 0.43.0 specifically. A newer version of Falco may detect some of the attacks that were missed here, or may introduce new false positives.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

This study is the first direct comparison of Falco and Kubescape in a Kubernetes-hosted 5G core environment. A full Open5GS 5G core was deployed on a three-node K3s cluster, UERANSIM was used to simulate real UE registration and PDU session traffic, and both tools were tested against twenty controlled security scenarios — ten runtime attacks and ten misconfigurations.

The research questions from Chapter 1 are answered below.

RQ1 — Kubescape misconfiguration detection: Kubescape found all ten misconfigurations with 100% accuracy and zero false positives. The most serious findings in the 5G context were privileged container mode (C-0002), cluster-admin role bindings (C-0035), and missing NetworkPolicy objects (C-0054). Together these three issues would allow an attacker who gets into one pod to move freely across the entire 5G core.

RQ2 — Falco runtime detection: Falco detected 4 out of 10 runtime attacks (40%) using its default rules, with detection time always under 400 ms. This is the real, measured result from this experiment.

However, it is important to understand *why* those four were detected. Three of the four (A-03, A-04, A-06) were caught only because the attack tools were freshly installed at runtime and were not present in the original container image. This triggered the "Drop and execute new binary" rule. The fourth (A-01) was caught by a specific rule for reading sensitive files.

This means that against a more careful attacker — one who only uses tools already built into the NF container image — three of those four detections would not fire. The effective detection rate in that scenario would drop from 40% down to just 1 out of 10 (A-01 only).

This is not what happened in this experiment, but it is a realistic risk that any operator deploying Falco with default rules should be aware of.

RQ3 — Complementarity and blind spots: The two tools have no overlap at all. Together they cover 14 out of 20 scenarios (70%) with default rules. The six missed attacks fall into two groups: tuning gaps (A-02, A-07, A-08, A-09, A-10), which can be fixed by writing or activating the right Falco rules; and one fundamental gap (A-05, crypto mining), which cannot be fixed with Falco rules at all and requires Prometheus CPU alerting as a third layer. The tools are completely complementary — each covers exactly what the other cannot.

RQ4 — Performance overhead: RQ4 — Performance overhead: CPU overhead was measured using `kubectl top nodes`, sampled every 2 seconds using the `watch` command across all three cluster nodes during idle, normal 5G operation, and active scan or attack phases.

Kubescape causes a short CPU spike during its scan window of approximately 30 seconds. On the worker node running the node-agent (sharyu02), CPU rose from a baseline of around 11% to a peak of 24% during the scan — a spike of approximately 13 percentage points — before returning to baseline within seconds of the scan completing. This spike did not affect Open5GS operation. Falco runs continuously at approximately 3–4% CPU overhead during normal operation, rising to around 8–9% during active attack simulation. Both tools are within acceptable limits for a production telecom deployment.

RQ5 — Practical deployment guidance: For a production 5G core, three things should be deployed together: Kubescape in admission controller mode so that misconfigured pods are blocked before they even start; Falco with custom rules for 5G-specific behaviour (especially allowing the UPF `NET_ADMIN` capability) and with the Kubescape node-agent whitelisted to remove false positives; and Prometheus alerting for resource exhaustion, which neither security tool can detect on its own. Most importantly, before going live, operators should run a known-bad test suite against the deployed rules and check which ones fire. This validation step should be treated as required, not optional.

In summary, neither tool is better than the other — they do completely different things. Kubescape prevents problems before they happen by checking configuration. Falco catches attacks as they happen by watching runtime behaviour. Together they provide a layered defence for cloud-native 5G networks, but only when both are set up and tuned correctly for the domain. The main lesson from this study is simple: installing these tools is not enough. The real security work starts after installation.

9.2 Future Work

Several directions would build on this study's findings:

- **5G-specific Falco rule sets:** Writing custom Falco rules for each Open5GS network function (AMF, SMF, UPF) based on their normal syscall behaviour. The six undetected attacks in this study give a direct starting point for what those rules need to cover. These rule sets could then be shared openly for other 5G security researchers to use.
- **Production-scale evaluation:** Running the same evaluation with hundreds or thousands of real UE connections to check whether the overhead and false positive numbers from this three-node testbed hold at realistic scale.
- **Automated remediation:** Connecting Falco alerts to automated response tools (such as Falco Talon or Kubernetes admission webhooks) so that threats can be blocked automatically without waiting for a human to respond.
- **Machine learning-based anomaly detection:** Using per-NF behavioural baselines built with machine learning to catch attacks like crypto mining (A-05) that rule-based tools cannot detect, and to enable detection of completely new attack patterns.
- **Supply chain security:** Evaluating Kubevuln and Trivy for scanning container images for known vulnerabilities, which would add a pre-deployment security layer on top of the runtime and configuration coverage provided by Falco and Kubescape.

Bibliography

[1] 3GPP, "System Architecture for the 5G System," TS 23.501, Release 16, 2020.

[2] 3GPP, "Security Architecture and Procedures for 5G System," TS 33.501, Release 16, 2020.

[3] Open5GS Project, "Open Source 5G Core and EPC," https://open5gs.org, 2023.

[4] UERANSIM, "Open Source 5G UE and RAN Simulator," https://github.com/aligungr/UERANSIM, 2023.

[5] CNCF Falco Project, "Falco: Cloud-Native Runtime Security," https://falco.org, 2023.

[6] ARMO, "Kubescape: Kubernetes Security Platform," https://kubescape.io, 2023.

[7] NSA/CISA, "Kubernetes Hardening Guidance," National Security Agency, Technical Report, 2022.

[8] Center for Internet Security, "CIS Kubernetes Benchmark v1.8," 2023.

[9] MITRE Corporation, "ATT&CK for Containers," https://attack.mitre.org/matrices/enterprise/containers/, 2022.

[10] Gregg, B., "BPF Performance Tools," Addison-Wesley, 2019.

[11] Rice, L. and Hausenblas, M., "Container Security," O'Reilly Media, 2020.

[12] Rancher Labs, "K3s: Lightweight Kubernetes," https://k3s.io, 2023.

[13] CoreOS, "Flannel: A Network Fabric for Containers," https://github.com/flannel-io/flannel, 2023.

[14] Prometheus Authors, "Prometheus Monitoring System and Time Series Database," https://prometheus.io, 2023.

[15] Chlosta, M., Rupprecht, D., Holz, T., and Pöpper, C., "5G SUCI-Catchers: Still catching them all?" in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2021.

[16] Sultan, S., Ahmad, I., and Dimitriou, T., "Container Security: Issues, Challenges, and the Road Ahead," *IEEE Access*, vol. 7, pp. 52976–52996, 2019.