Inside 5G: Monitoring & Fast Chaining

R&D-II Project Report

Submitted by:
Arghyadip Chakraborty

Guide:
Prof. Mythili Vutukuru

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai – 400076

May 2025

Abstract

As 5G networks evolve in complexity and scale, the need for observability, programmability, and efficient data processing becomes increasingly critical. This report presents a comprehensive exploration of techniques and systems developed to enhance monitoring, measurement, and network function deployment in modern 5G infrastructures. We begin by constructing a fully functional 5G testbed using OpenAirInterface (OAI) and integrating it with FlexRIC, a RAN Intelligent Controller platform. On top of this setup, we develop a custom xApp to monitor critical RAN metrics, including Radio Link Control (RLC) and Key Performance Measurements (KPM), showcasing how real-time telemetry can be extracted and visualized from the RAN layer.

To extend user-level insight into 5G control-plane protocols, we adapt and upgrade MobileInsight, a diagnostic tool traditionally built for 3G and 4G LTE networks. This effort required reverse engineering Qualcomm's undocumented diagnostic packet structures — typically accessible only via proprietary tools — and modifying parser logic to support newer 5G NR message formats. Our reverse engineering methodology is detailed, highlighting both technical challenges and successful parser updates.

Finally, we explore fast packet processing in 5G and edge environments through FLASH, a novel in-kernel chaining framework built on AF_XDP. FLASH enables efficient zero-copy network function (NF) chaining directly in the Linux kernel, offering low-latency and high-throughput benefits. We demonstrate FLASH's flexibility by reimplementing its NF library from scratch in Rust, thereby ensuring memory safety and process-level isolation while preserving performance. This validates the language-agnostic promise of FLASH and illustrates how safe systems programming can coexist with high-performance networking.

Together, these contributions provide a foundation for future work in building programmable, observable, and secure 5G platforms.

Contents

1 Introduction

The deployment of 5G networks marks a major step forward in mobile communications, enabling ultra-low latency, high throughput, and connectivity at unprecedented scale. However, this shift brings significant complexity across the Radio Access Network (RAN) and Core Network. As infrastructure becomes increasingly disaggregated, software-defined, and edge-integrated, achieving real-time observability, programmability, and high-performance packet processing becomes critical for effective research, development, and deployment.

In this work, we present a practical and systematic approach to addressing these challenges by building a programmable and observable 5G testbed, extending user-level visibility into 5G control-plane activity, and improving network function performance using high-speed chaining frameworks. These contributions span three tightly coupled themes:

- ❒ **Monitoring and Telemetry via xApps and FlexRIC:**
  We deployed an open 5G testbed using OpenAirInterface (OAI) and integrated it with FlexRIC, a near-real-time RAN Intelligent Controller (RIC). On top of this, we implemented a custom xApp capable of collecting fine-grained RAN-layer statistics such as RLC and KPM metrics. This component demonstrates the feasibility and value of telemetry extraction directly from the RAN, enabling applications like closed-loop control, real-time diagnostics, and performance analysis.
- ❒ **Extending MobileInsight for 5G NR:**
  MobileInsight is a user-level tool for cellular protocol tracing, widely used for LTE but with limited support for 5G NR due to missing parsers. We extended its capabilities by reverse engineering Qualcomm diagnostic packets — typically readable only with proprietary tools like QXDM. Through raw packet analysis and custom parser development, we enabled decoding of modern 5G messages, providing real-time visibility into control-plane signaling.
- ❒ **Rust-Native FLASH NF Library:**
  As network functions move closer to the edge, performance and security become paramount. We examine FLASH, an in-kernel zero-copy chaining framework for `AF_XDP` sockets that enables high-speed packet processing in Linux. To validate FLASH's portability and language-agnostic promise, we reimplement its NF (network function) library from scratch in Rust. This Rust-based version maintains the same performance while offering stronger memory safety and NF isolation — key benefits for multi-tenant or untrusted environments.

Through these efforts, we demonstrate a practical architecture that combines visibility, control, and performance in 5G networks. This work bridges the gap between theoretical advances in software-defined networking and real-world deployments of programmable mobile infrastructure. By building on open platforms and first-principles design, we show how researchers and developers can gain deeper insight into, and control over, the evolving 5G ecosystem.

2 5G Testbed with OpenAirInterface

OpenAirInterface (OAI) [7] is a comprehensive open-source software platform that provides a full-stack implementation of both the 5G Radio Access Network (RAN) and the Core Network (5GC). Designed to support research, development, and academic experimentation, OAI serves as a powerful tool for advancing the understanding and deployment of next-generation wireless communication systems.

In this chapter, we describe our deployment of a fully functional 5G testbed using OAI. This setup enabled us to observe control-plane and data-plane interactions, experiment with flexible RAN configurations, and implement custom telemetry via RAN Intelligent Controller (RIC) integration. The deployment process involved the following key steps:

- Deploying the containerized OAI 5G Core Network using Docker
- Setting up the OMEC gNB simulator to emulate a gNB and connect to the Core
- Running gNBSim with different configuration profiles to generate and capture 5G call flow traces
- Deploying a functional OAI gNB with two simulated user equipments (UEs) and integrating the FlexRIC platform
- Generating traffic between UEs and the Internet to evaluate end-to-end behavior
- Developing a custom xApp using the FlexRIC SDK to monitor RAN-layer metrics, including Radio Link Control (RLC) statistics and Key Performance Measurements (KPM)

The primary objective of this exercise was to establish a fully operational 5G testbed that enabled in-depth exploration of interactions across both the RAN and the Core Network. By integrating advanced RAN monitoring and control through a RAN Intelligent Controller (RIC), the setup demonstrated the potential for real-time observability, telemetry collection, and programmability in modern cellular networks. This testbed provided a practical foundation for studying 5G architecture, understanding protocol behavior, and evaluating performance under realistic conditions.

2.1 Deploying OAI 5G CN

To begin the practical setup of our 5G testbed, we deployed the essential components of the OpenAirInterface (OAI) 5G Core Network using containerized services. The architecture of this deployment is illustrated in Figure 2.1.

The 5G Core (5GC) typically comprises multiple network functions that collectively support the full range of 5G capabilities. However, for the purpose of this study — which focuses on establishing end-to-end connectivity and analyzing basic call flows — we deployed only the components necessary to support a complete data path from the RAN to the Internet. The deployed core network components include:

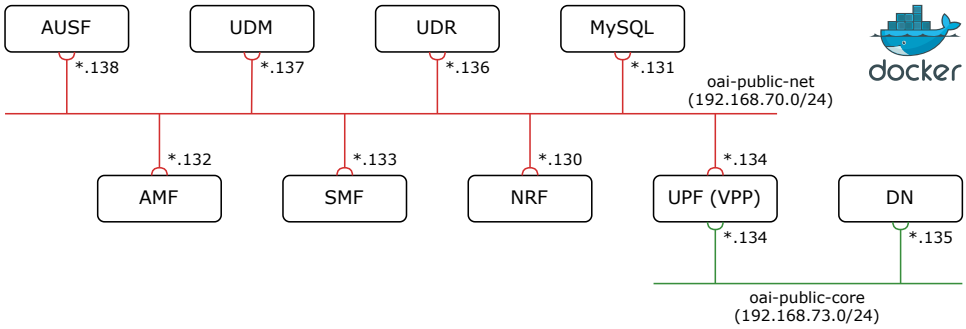

Figure 2.1: OAI 5G Core Network Setup

- **Access and Mobility Management Function (AMF)**
  Manages registration, connection, and mobility of UE
- **Session Management Function (SMF)**
  Handles session establishment, IP address allocation, and policy enforcement
- **User Plane Function (UPF)**
  Forwards user traffic between the RAN and the external data network
- **Unified Data Management (UDM)**
  Maintains subscriber profiles and handles authentication-related data
- **Unified Data Repository (UDR)**
  Serves as the central repository for subscriber and policy information
- **Network Repository Function (NRF)**
  Enables service discovery across network functions
- **Authentication Server Function (AUSF)**
  Provides authentication services for UEs

To support the UDR, we deployed a MySQL (v8.0) database, pre-populated with subscriber information for simulated UEs used in later exercises. For optimal performance in the user plane, we utilize OAI's VPP-based UPF, which leverages the Vector Packet Processing framework for high-throughput, low-latency packet forwarding. Additionally, a Data Network (DN) container is included to emulate Internet connectivity. It uses `iptables` for packet routing and NAT, enabling end-to-end data sessions from the UE through the Core to the Internet.

All network functions were deployed as Docker [6] containers using official images provided by the OAI Software Alliance (`oaisoftwarealliance`) on DockerHub. To manage network connectivity between these containers, we define two custom Docker network bridges:

- `oai-public-net` (subnet `192.168.70.0/24`): Connects the core components internally
- `oai-public-core` (subnet `192.168.73.0/25`): Connects the UPF to the DN and facilitates external data routing

Each container was assigned a static IP address within its respective subnet, eliminating the need for DNS resolution and enabling consistent, direct referencing between components. All core components shared a common configuration file, which ensured uniform parameter settings across services. The deployment was orchestrated using a single `docker-compose` file, simplifying service management and promoting consistent, reproducible setups.

This deployment formed the foundational layer of our 5G testbed and enabled the subsequent integration of simulated and real RAN elements for in-depth analysis and experimentation.

2.2 Deploying gNB Simulator

With the core network in place, the next step in our testbed setup was to simulate a gNodeB (gNB) and user equipment (UE) to interact with the 5G Core. For this purpose, we utilize the gNB Simulator (gNBSim) [8], a powerful open-source tool developed under the Aether SD-Core project (also known as OMEC).

gNBSim is designed to emulate the control plane functionality of a 5G gNB and UE by generating Non-Access Stratum (NAS) and Next Generation Application Protocol (NGAP) messages. This allows for comprehensive testing of the 5G Core Network (5GC) through various signaling scenarios such as registration, session establishment, and deregistration — without the need for physical radio hardware. Figure 2.2 illustrates the internal architecture and message flow of gNBSim.

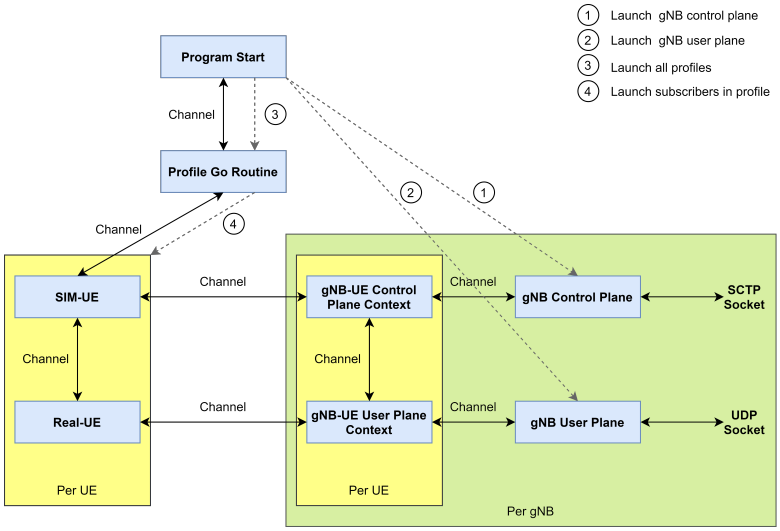

Figure 2.2: gNBSim Flow Diagram, from [8]

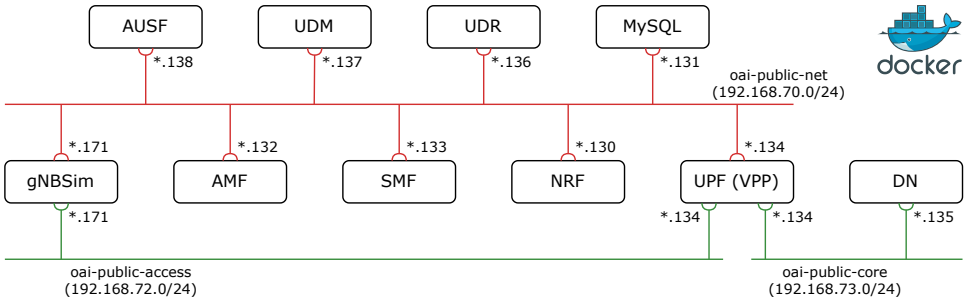

Figure 2.3: gNB Simulator Setup

In our deployment (Figure 2.3), we built the gNBSim Docker image directly from the official GitHub repository and deployed it as a container within the same environment as the 5G Core.

The gNBSim container is connected to the AMF via the existing `oai-public-net` Docker network bridge, enabling NAS and NGAP signaling between the simulated UE and the core. To forward user data to the UPF, we created an additional Docker network bridge named `oai-public-access` (subnet `192.168.72.0/24`). This separation of signaling and user-plane paths, although not strictly compliant with 3GPP's specified architecture, is sufficient for functional testing and analysis in our controlled environment.

2.3 Tracing Call Flows

With the gNBSim environment in place and connected to the 5G Core Network, we proceeded to simulate various UE-driven and network-triggered procedures in order to generate realistic 5G call flows. This step is essential for validating the behavior of the Core Network components and for gaining deeper insight into the signaling interactions defined by 3GPP specifications.

gNBSim supports a wide range of control plane procedures, including:

- UE Registration
- UE-Initiated PDU Session Establishment
- UE-Initiated De-registration
- Access Network (AN) Release
- UE-Initiated Service Request
- Network-Triggered PDU Session Release
- UE-Requested PDU Session Release
- Network-Triggered UE Deregistration

These procedures can be grouped into configurable profiles, each representing a specific call flow scenario. Profiles may be executed sequentially or in parallel, enabling us to simulate diverse network behaviors and interaction patterns.

To capture these signaling flows, we used the `tshark` tool to monitor the `oai-public-net` Docker network bridge. Each session was recorded as a `.pcapng` file, which was later inspected using Wireshark [4]. This allowed us to study the message exchanges in detail and validate the compliance and performance of the deployed Core Network. Figure 2.4 presents an example call flow, showing the initial part of the UE registration procedure as visualized in Wireshark.

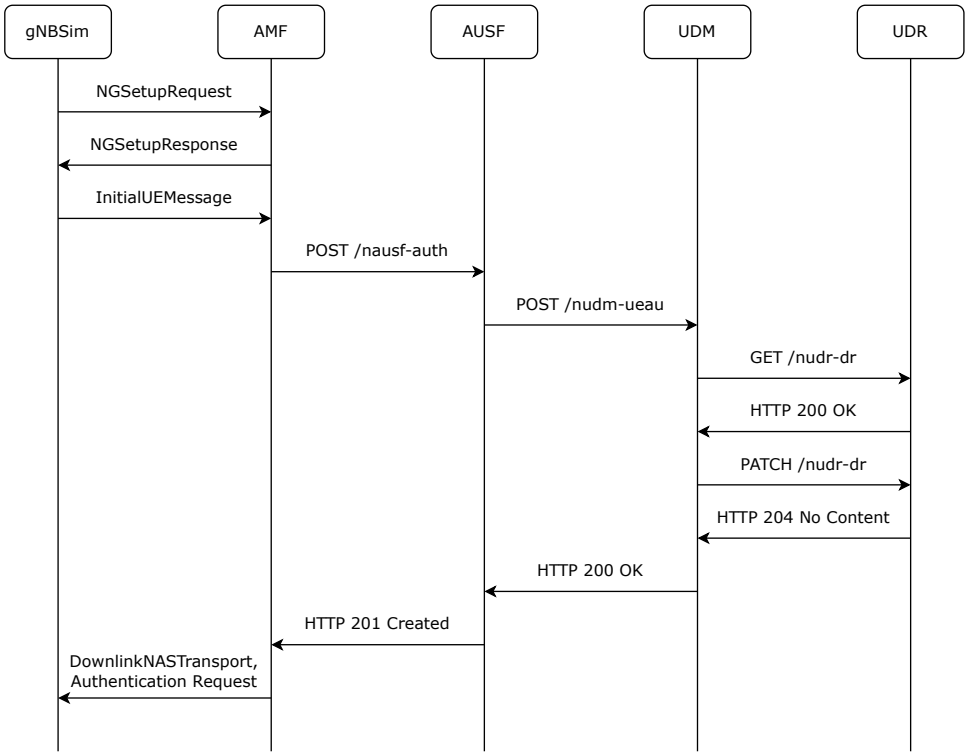

Figure 2.4: Initial part of UE Registration Call Flow

This approach provides a valuable means of observing and understanding how the Core Network processes fundamental operations, offering insight into the signaling behavior, protocol structure, and timing characteristics of 5G Core communication.

2.4 Deploying OAI RAN with UEs and FlexRIC

After successfully deploying and validating the 5G Core and gNB simulator, we advanced to setting up a more realistic RAN environment. This involved deploying the OpenAirInterface (OAI) gNB, two simulated UEs, and integrating the RAN Intelligent Controller (RIC), FlexRIC [10]. FlexRIC is an open-source platform that enables near-real-time RAN control and monitoring via modular xApps. With FlexRIC in place, we could build a complete 5G testbed capable of both data-plane communication and comprehensive RAN monitoring.

In typical 5G deployments, the RAN is disaggregated into a Central Unit (vCU) and Distributed Unit (vDU), separating high- and low-layer processing. For simplicity and manageability, our testbed used a monolithic gNB implementation provided by OAI, which integrates both vCU and vDU functionalities into a single container. While this approach does not provide full disaggregation flexibility, it is sufficient for end-to-end testing and RIC integration.

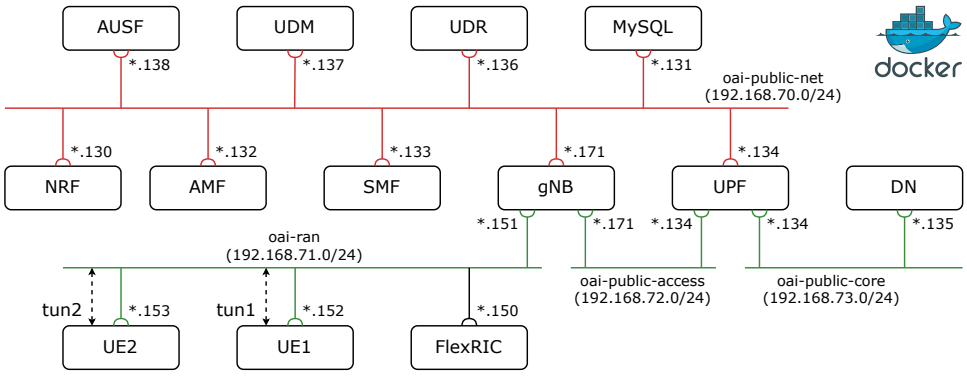

Figure 2.5: OAI 5G RAN Setup

As shown in Figure 2.5, the OAI gNB container is connected to both the `oai-public-net` (for control-plane communication with the AMF) and `oai-public-access` (for user-plane data traffic via UPF). Additionally, we deployed two UE containers and the FlexRIC container, all of which are linked to a new Docker bridge network `oai-ran` (subnet `192.168.71.0/24`) with static IP assignments. Each UE is configured with a tunnel interface (`oaitun_ueX`) through which data is routed to the gNB and subsequently to the Internet.

To test data connectivity between UEs, we ran `iperf3` tests across the tunnel interfaces. Figure 2.6 shows a server running on UE1, while Figure 2.7 illustrates a client on UE2 initiating the test. The iperf results confirm successful UE-to-UE communication over the 5G stack, with throughput reaching up to 17.5 Mbps. To evaluate end-to-end Internet connectivity, we also run the Ookla Speedtest utility on UE1 via its tunnel interface.

As shown in Figure 2.8, UE1 achieved a peak download speed of approximately 54.47 Mbps and an upload speed of 22.20 Mbps, with a latency of 48.51 ms. These results confirm that the deployed RAN and Core Network are functioning as intended, enabling both inter-UE and external Internet connectivity — providing a realistic and testable 5G environment.

This setup also served as the foundation for the integration of FlexRIC, enabling real-time monitoring and control of the RAN through xApps, which is covered in the next phase of our testbed exploration.

2.5 Building xApp for RAN Monitoring

With the FlexRIC platform integrated into our RAN, the final step in our testbed setup was to develop a custom xApp to monitor RAN-layer metrics in real time. This xApp acts as a client to

```
oai@turing06:~$ docker exec -it rfsim5g-oai-nr-ue iperf3 -s -B 10.0.0.3
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 10.0.0.2, port 41804
[  5] local 10.0.0.3 port 5201 connected to 10.0.0.2 port 33288
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec   403 KBytes  3.30 Mbits/sec
[  5]   1.00-2.00   sec  1001 KBytes  8.20 Mbits/sec
[  5]   2.00-3.00   sec  1.61 MBytes  13.5 Mbits/sec
[  5]   3.00-4.00   sec  2.44 MBytes  20.5 Mbits/sec
[  5]   4.00-5.00   sec  2.49 MBytes  20.9 Mbits/sec
[  5]   5.00-6.00   sec  2.60 MBytes  21.8 Mbits/sec
[  5]   6.00-7.00   sec  2.47 MBytes  20.7 Mbits/sec
[  5]   7.00-8.00   sec  2.42 MBytes  20.3 Mbits/sec
[  5]   8.00-9.00   sec  2.54 MBytes  21.3 Mbits/sec
[  5]   9.00-10.00  sec  2.64 MBytes  22.1 Mbits/sec
[  5]  10.00-10.42  sec  1.10 MBytes  22.2 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.42  sec  21.7 MBytes  17.5 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 2.6: Iperf3 Server running on UE1

```
oai@turing06:~$ docker exec -it rfsim5g-oai-nr-ue2 iperf3 -c 10.0.0.3
Connecting to host 10.0.0.3, port 5201
[  5] local 10.0.0.2 port 33288 connected to 10.0.0.3 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   700 KBytes  5.73 Mbits/sec    0   53.7 KBytes
[  5]   1.00-2.00   sec  1.30 MBytes  10.9 Mbits/sec    0    103 KBytes
[  5]   2.00-3.00   sec  1.86 MBytes  15.6 Mbits/sec    0    187 KBytes
[  5]   3.00-4.00   sec  3.04 MBytes  25.5 Mbits/sec    0    314 KBytes
[  5]   4.00-5.00   sec  3.11 MBytes  26.1 Mbits/sec    0    441 KBytes
[  5]   5.00-6.00   sec  3.04 MBytes  25.5 Mbits/sec    0    574 KBytes
[  5]   6.00-7.00   sec  2.43 MBytes  20.4 Mbits/sec    0    699 KBytes
[  5]   7.00-8.00   sec  2.50 MBytes  21.0 Mbits/sec    0    823 KBytes
[  5]   8.00-9.00   sec  2.50 MBytes  21.0 Mbits/sec    0    953 KBytes
[  5]   9.00-10.00  sec  2.50 MBytes  21.0 Mbits/sec    0   1.06 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  23.0 MBytes  19.3 Mbits/sec    0             sender
[  5]   0.00-10.42  sec  21.7 MBytes  17.5 Mbits/sec                  receiver

iperf Done.
```

Figure 2.7: Iperf3 Server running on UE2

the FlexRIC controller and subscribes to telemetry events from the gNB, specifically focusing on Radio Link Control (RLC) statistics and Key Performance Measurements (KPM).

Our xApp implementation declares two callback functions to handle RLC and KPM report messages. These are triggered by FlexRIC as updates are received from the gNB. The xApp then aggregates the statistics and outputs them to `stdout` at one-second intervals. As shown in Figure 2.9, the xApp is deployed directly on the host system and communicates with the FlexRIC container using its static IP address, forming a lightweight and efficient monitoring setup.

To observe xApp behavior during active user traffic, we ran a Speedtest from UE1 while capturing RLC and KPM stats. Figure 2.10 shows the real-time output generated by the xApp. Metrics such as receive and transmit buffer occupancy, PDU byte counts, and latency for each UE are displayed and updated every second.

While our testbed setup with FlexRIC enabled detailed monitoring and control within the RAN, observability at the UE layer remains limited in such infrastructure-centric approaches. In the next chapter, we shift our focus to the UE itself by exploring MobileInsight, a tool that enables protocol-level tracing directly on mobile devices — offering a complementary perspective

```
oai@turing06:~$ docker exec -it rfsim5g-oai-nr-ue /root/speedtest -I oaitun_ue1

Speedtest by Ookla

    Server: Hostycare Data Center - Mumbai (id: 63121)
      ISP: Powai
Idle Latency:    48.51 ms   (jitter: 6.13ms, low: 42.78ms, high: 51.11ms)
  Download:    54.47 Mbps (data used: 68.4 MB)
               348.23 ms   (jitter: 72.23ms, low: 48.13ms, high: 829.32ms)
    Upload:    22.20 Mbps (data used: 10.9 MB)
               288.05 ms   (jitter: 74.67ms, low: 45.45ms, high: 535.87ms)
Packet Loss: Not available.
Result URL: https://www.speedtest.net/result/c/0f7ddfa6-eac9-4985-8467-e44632a93bb5
```

Figure 2.8: Ookla Speedtest running on oaitun_ue1 in UE1

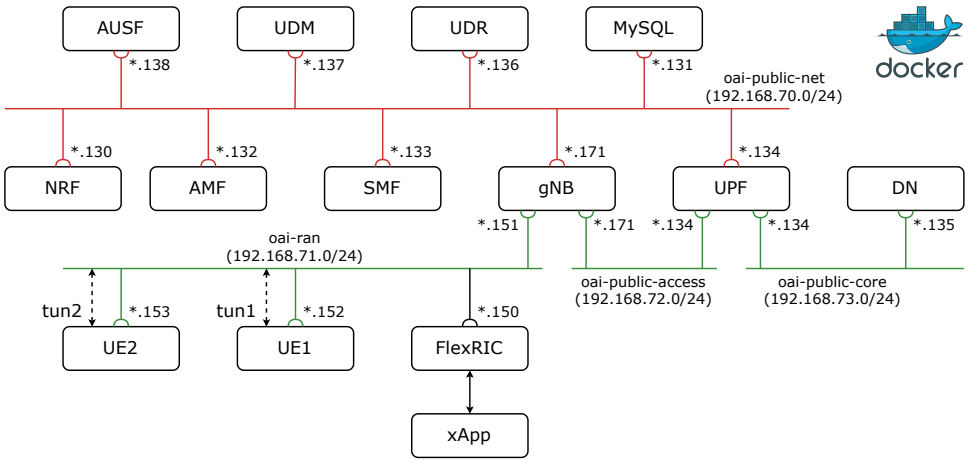

Figure 2.9: xApp Monitor

on 5G behavior from the network's edge.

```
Avg Latency :: RLC = 364 micros | KPM = 531 micros

============================================= RLC and KPM Stats =============================================
Index | PLMN_ID |   RNTI   | LCID | RAN_UE_ID | AMF_UE_NGAP_ID | RXBUF_OCC | RXPDU_BYTES | TXBUF_OCC | TXPDU_BYTES
------+---------+----------+------+-----------+----------------+-----------+-------------+-----------+------------
    0 |   00101 |     9fcc |    4 |         1 |              1 |         0 |    76408113 |         0 |     1525946
    1 |   00101 |     56c7 |    4 |         2 |              2 |         0 |    39190403 |         0 |   310372126
============================================================================================================

Avg Latency :: RLC = 341 micros | KPM = 719 micros

============================================= RLC and KPM Stats =============================================
Index | PLMN_ID |   RNTI   | LCID | RAN_UE_ID | AMF_UE_NGAP_ID | RXBUF_OCC | RXPDU_BYTES | TXBUF_OCC | TXPDU_BYTES
------+---------+----------+------+-----------+----------------+-----------+-------------+-----------+------------
    0 |   00101 |     9fcc |    4 |         1 |              1 |         0 |    76408113 |         0 |     1525946
    1 |   00101 |     56c7 |    4 |         2 |              2 |         3 |    39191819 |      4694 |   310377360
============================================================================================================

Avg Latency :: RLC = 352 micros | KPM = 920 micros

============================================= RLC and KPM Stats =============================================
Index | PLMN_ID |   RNTI   | LCID | RAN_UE_ID | AMF_UE_NGAP_ID | RXBUF_OCC | RXPDU_BYTES | TXBUF_OCC | TXPDU_BYTES
------+---------+----------+------+-----------+----------------+-----------+-------------+-----------+------------
    0 |   00101 |     9fcc |    4 |         1 |              1 |         0 |    76408113 |         0 |     1525946
    1 |   00101 |     56c7 |    4 |         2 |              2 |         3 |    39200061 |      3351 |   310390111
============================================================================================================

Avg Latency :: RLC = 391 micros | KPM = 677 micros

============================================= RLC and KPM Stats =============================================
Index | PLMN_ID |   RNTI   | LCID | RAN_UE_ID | AMF_UE_NGAP_ID | RXBUF_OCC | RXPDU_BYTES | TXBUF_OCC | TXPDU_BYTES
------+---------+----------+------+-----------+----------------+-----------+-------------+-----------+------------
    0 |   00101 |     9fcc |    4 |         1 |              1 |         0 |    76408113 |         0 |     1525946
    1 |   00101 |     56c7 |    4 |         2 |              2 |         3 |    39258284 |    374058 |   313258624
============================================================================================================
```

Figure 2.10: xApp during Speedtest

3 Extending MobileInsight

MobileInsight [5] is a cross-platform software toolkit designed to collect, analyze, and exploit runtime network information directly from commercial cellular devices. It enables below-IP, fine-grained mobile network analytics by providing access to lower-layer protocol events and messages. Figure 3.1 illustrates its core architecture, which consists of low-level protocol monitors and extensible event-driven analyzers for key cellular protocols such as Radio Resource Control (RRC), Mobility Management (EMM), and Session Management (ESM).

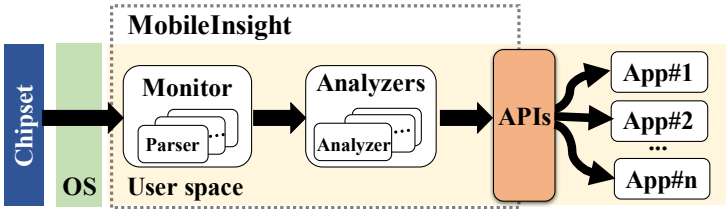

Figure 3.1: MobileInsight Architecture

While MobileInsight is well-established for 3G and 4G networks, its support for 5G NR remains incomplete. In our attempt to build a consolidated 5G measurement platform [1] leveraging MobileInsight, we encountered several challenges. Chief among them was the lack of up-to-date parsers for 5G NR protocols. Most existing parsers were incompatible with newer devices, and extending them proved difficult due to limited public documentation.

In this chapter, we describe our efforts to extend MobileInsight with enhanced support for 5G NR by upgrading and adapting the existing parsing infrastructure. We outline our attempts to adapt and upgrade the internal parser framework, highlight the technical challenges faced, and propose directions for improving 5G support in MobileInsight.

3.1 Reverse Engineering 5G NR Messages

MobileInsight accesses low-level cellular protocol messages by leveraging Qualcomm's diagnostic mode interface. However, the structure of these messages is complex and largely undocumented due to Qualcomm's proprietary architecture. Qualcomm provides a commercial tool — QXDM Professional™ Tool [9] — to capture and interpret these messages, but it is closed-source and not publicly extensible.

Fortunately, we were able to obtain raw 5G NR diagnostic messages along with their decoded counterparts from QXDM/CAT, such as the sample shown in Figure 3.2. This enabled us to explore the possibility of reverse engineering these messages to repair or extend MobileInsight's existing 5G NR parsers.

```
2025 Feb 19  13:52:25.309  [1C]  0xB97F  NR5G ML1 Searcher Measurement Database Update Ext
Subscription ID = 1
Misc ID         = 0
Major.Minor Version = 2. 9
System Time
   Slot Number = 0
   SubFrame Number = 1
   System Frame Number = 6
   SCS = DEFAULT
Num Layers = 1
SSB Periodicity Serv Cell = INVALID
Frequency Offset = 2.281 PPM
Timing Offset = 182108951
Component Carrier List[0]
   Raster ARFCN = 643296
   CC_ID = NA
   Num Cells = 1
   Serving Cell PCI = NA
   Serving Cell Index = NA
   Serving SSB = NA
   ServingRSRP Rx23[0]
      ServingRSRP_Rx23 = NA
   ServingRSRP Rx23[1]
      ServingRSRP_Rx23 = NA
   Serving RX Beam = { NA, NA }
   Serving RFIC ID = NA
   ServingSubarrayId[0] {
      SubArray ID = NA
   }
   ServingSubarrayId[1] {
      SubArray ID = NA
   }
   Cells
      ---------------------------------------------------------------------------------------------------------------------------------------
      |   |      |      |     |            |            |Detected Beams                                                                    |
      |   |      |      |     |            |            |     |RX Beam Info       |NR2NR       |NR2NR       |L2NR        |L2NR        |
      |   |      |PBCH  |Num  |Cell Quality|Cell Quality|SSB  |RX Beam |          |Filtered Tx |Filtered Tx |Filtered Tx |Filtered Tx |
      |#  |PCI   |SFN   |Beams|RSRP        |RSRQ        |Index|Id      |RSRP      |Beam RSRP L3|Beam RSRQ L3|Beam RSRP L3|Beam RSRQ L3|
      ---------------------------------------------------------------------------------------------------------------------------------------
      | 0|    35|   436|    1|     -105.44|      -10.41|    0|       0|   -105.44|     -105.44|      -10.41|          NA|          NA|
      |   |      |      |     |            |            |     |       0|   -107.69|            |            |            |            |

  Length:  124
  Header:  7C 00 7F B9 1D 2F 17 5F 02 3E 09 01
  Payload: 09 00 02 00 00 80 30 A0 01 00 00 00
           20 09 00 00 17 C3 DA 0A E0 D0 09 00
           FF 01 FF FF FF FF 00 00 00 00 00 00
           00 00 00 00 FF FF FF FF FF FF 00 00
           FF FF FF FF 23 00 B4 01 01 00 00 00
           48 CB FF FF CC FA FF FF 00 00 00 00
           00 00 00 00 00 00 00 00 B9 BA 7C 68
           DC 74 18 00 48 CB FF FF 28 CA FF FF
           48 CB FF FF CC FA FF FF 00 00 00 00
           00 00 00 00
```

Figure 3.2: Sample Message: NR5G ML1 Searcher Measurement Database Update

To decode new message types or update outdated parser versions in MobileInsight, we followed a systematic process:

- Identify message type, version, and header from QXDM output
- Locate the hex header in the raw (little-endian) dump
- Group similar dumps to infer field structure and variation
- Define or update the C `struct` in the parser accordingly
- Cross-check struct size with total packet length for validation
- Modify parsing logic and log field outputs for verification

3.2 Successful Parser Updates

Despite the inherent complexity of Qualcomm diagnostic messages, we achieved partial success in extending MobileInsight's capabilities for 5G NR. By applying the reverse engineering methodology described earlier, we were able to update two parsers:

- `NR_ML1_Search_Meas_Database_Update` v2.9
- `NR_RLC_DL_Stats` v3.0

Figure 3.3 illustrates the changes made to the `NR_RLC_DL_Stats` parser to support version 3.0.

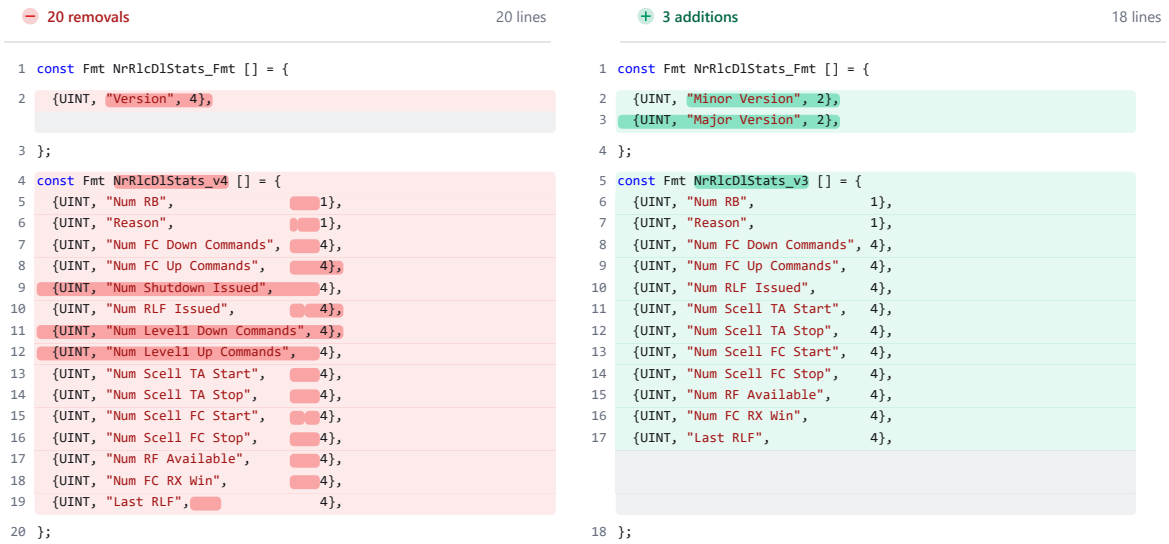

Figure 3.3: NR_RLC_DL_Stats Parser $v4 \rightarrow v3.0$

3.3 Challenges

While the parser update methodology proved effective in certain cases, it encountered significant limitations when applied to other messages. Some of the key challenges include:

- ➢ Message fields are not always byte-aligned, making it difficult to map raw offsets to struct definitions
- ➢ Certain bytes are shared across multiple fields, requiring intricate bit-level parsing logic
- ➢ Some parts of the packet dump are merely padding or unused, complicating payload length validation
- ➢ Many message formats are highly version-specific and may include conditional fields or variable-length sections
- ➢ Lack of official documentation forces reliance on trial-and-error and manual inspection
- ➢ Structural changes across versions are not always backward-compatible, necessitating distinct parsers for each version
- ➢ Misalignment or incorrect assumptions can silently result in partially decoded data, which is harder to detect than complete failures

These challenges highlight the fragile nature of extending parsers without formal specifications or toolchain support. Nonetheless, the successful updates demonstrate that with careful reverse engineering, valuable extensions to MobileInsight are still possible.

4 Rust-Native NF Library for FLASH

Modern cloud-native applications increasingly rely on high-performance packet processing capabilities within the Linux kernel. To meet these demands, FLASH [2] introduces a novel in-kernel chaining mechanism designed specifically for `AF_XDP` sockets. FLASH enables efficient composition of Network Functions (NFs) with minimal overhead and robust performance characteristics. The architecture of FLASH is illustrated in Figure 4.1.

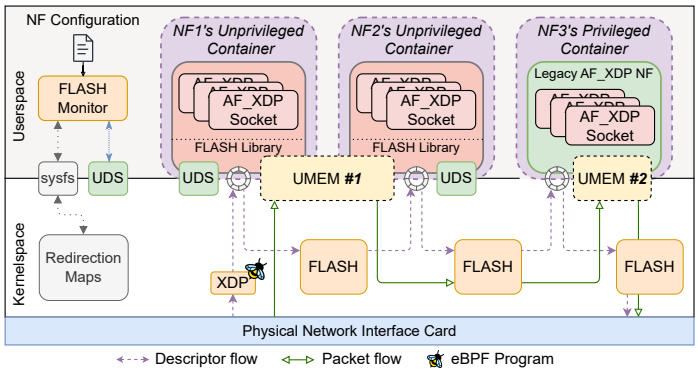

Figure 4.1: FLASH Architecture

Key features of FLASH include:

- In-kernel chaining for `AF_XDP`-based NFs
- Native support for zero-copy and single-copy processing
- Backward-compatible, language-agnostic, and cloud-native
- No elevated privileges required with FLASH Monitor

FLASH provides a lightweight C library for implementing FLASH-native NFs on `AF_XDP` sockets. These NFs benefit from zero-copy chaining, achieving low latency and high data plane efficiency. However, the shared user memory (UMEM) used for zero-copy forwarding poses safety risks. Any NF with UMEM access can potentially read or modify another NF's packets, jeopardizing isolation and reliability.

To address these issues and explore FLASH's language-agnostic design, we undertook a complete, from-scratch reimplementation of the FLASH NF library in Rust. This Rust-native library avoids any bindings to the original C codebase and preserves the same high-performance characteristics. In addition, it leverages Rust's memory safety guarantees to enforce strict packet isolation, reducing the risks associated with shared memory usage.

This chapter details our efforts to port FLASH to Rust, evaluates its performance, and demonstrates how leveraging Rust's memory safety features enhances the isolation and reliability of network functions using FLASH.

4.1 Why Rust?

FLASH's core design principle is language agnosticism — the ability to implement NFs in any language that can interface with `AF_XDP`. Given this flexibility, a natural question arises: Why choose Rust for reimplementing the FLASH NF library?

The decision was driven by Rust's unique balance of system-level control and modern safety guarantees, which align closely with FLASH's goals: low-latency, high-performance, and secure packet processing. Specifically, Rust offers the following advantages:

- ❒ **Memory Safety Without Garbage Collection**:
  Rust enforces memory safety through its ownership model and borrow checker, catching issues like null dereferencing and use-after-free at compile time. Importantly, Rust achieves this without a garbage collector, allowing for deterministic memory management — critical for low-latency, high-throughput packet processing pipelines
- ❒ **Strong Static Typing**:
  Rust's expressive type system enables compile-time validation of many common errors, increasing code correctness and reducing runtime faults
- ❒ **Safe Concurrency**:
  The language's concurrency model ensures that data races are eliminated by design, making it easier to write multi-threaded NFs safely and efficiently
- ❒ **Performance Comparable to C/C++** :
  With zero-cost abstractions and fine-grained control over memory and execution, Rust provides performance on par with traditional system languages — without compromising safety
- ❒ **Explicit Error Handling**:
  Rust's approach to error handling, using `Result` and `Option` types, encourages developers to account for failure cases explicitly, reducing hidden bugs and improving robustness
- ❒ **Modern Ecosystem and Tooling**:
  The Rust ecosystem offers mature libraries (crates), tooling (like cargo), and excellent documentation, helping streamline development while maintaining quality

In essence, Rust provides the low-level access needed for packet manipulation while introducing guarantees that make such systems safer and more maintainable. In the following sections, we explore how we applied these features to develop a complete, standalone Rust-native FLASH NF library from scratch — validating FLASH's language-agnostic promise and raising the bar for safety in high-performance networking.

4.2 Understanding Rust's Ownership Model

Before diving into the implementation details, it's essential to understand Rust's ownership model, which forms the cornerstone of its memory safety guarantees — a key feature that we leveraged in our NF library.

Ownership In Rust, every value has a single owner, and when the owner goes out of scope, the value is automatically dropped. This ensures that memory is freed without the need for a garbage collector. For example, in the code snippet shown in Figure 4.2, the string `s` is valid only within its scope (lines 2 to 4). Once the scope ends (line 5), `s` is dropped, and its memory is released. This is different from stack unwinding in other languages, as `s` is a heap-allocated `String`.

```rust
{
  let s = String::from("hello");  // s is allocated on the heap
  println!("{s}");                // use s within its scope
}                                 // s is dropped when the scope ends
```

Figure 4.2: Rust Ownership Example

Ownership Transfer In Rust, ownership can be transferred when a value is **moved** to another variable. After the transfer, the original owner can no longer access the value, which prevents memory issues like double freeing. In the code snippet in Figure 4.3, `s` is moved into the function `takes_ownership`. After the move, accessing `s` in `main` results in a compile-time error.

```rust
fn main() {
  let s = String::from("hello"); // s comes to scope
  takes_ownership(s);            // s is moved into the function
  println!("{s}");            // error[E0382]: borrow of moved value: `s`
}

fn takes_ownership(some_string: String) { // some_string comes into scope
  println!("{some_string}");
} // some_string goes out of scope and memory is freed
```

Figure 4.3: Rust Ownership Transfer Example

References In Rust, references allow access to data without transferring ownership, similar to pointers in C/C++ but with added safety. In the code snippet in Figure 4.4, a reference to `s` is passed to `calculate_length` using `&`, allowing the ownership to remain with `main` while enabling access to `s`. This prevents ownership transfer and avoids issues like double freeing, ensuring memory safety.

```rust
fn main() {
  let s = String::from("hello");
  let len = calculate_length(&s);       // reference to s is passed
  println!("Length of '{s}' is {len}");
}

fn calculate_length(s_ref: &String) -> usize {
  s_ref.len()
}
```

Figure 4.4: Passing References Example

```rust
fn main() {
  let mut s = String::from("hello");
  append_exclamation(&mut s);          // mutable reference to s is passed
  println!("{s}");                     // s is modified by the function
}

fn append_exclamation(s_ref: &mut String) {
  s_ref.push_str("!");
}
```

Figure 4.5: Passing Mutable References Example

Mutable References Rust also allows mutable references, which enable modification of data through a reference. In the code snippet in Figure 4.5, `s` is passed as a mutable reference to the `append_exclamation` function. This allows the function to modify `s` in place without transferring ownership. To enable mutability, the variable `s` must be marked as `mut`, indicating that it can be modified.

```rust
fn main() {
  let mut s = String::from("hello");

  let r1 = &mut s;
  let r2 = &mut s;        // error[E0499]: cannot borrow `s` as mutable more than
                          // once at a time
  println!("{r1}, {r2}");
}
```

Figure 4.6: Multiple Mutable References Example

However, multiple mutable references to the same data are not allowed at the same time. In the code snippet below, the second mutable reference `r2` to `s` results in a compile-time error, as multiple mutable references are disallowed. This ensures that no two parts of the program can modify the same data concurrently, which could lead to unpredictable behavior and bugs, especially in multithreaded contexts.

4.3 Rust NF Library Design

The Rust NF library provides a minimal and ergonomic API for implementing FLASH-native NFs. Its main entry point is the `connect` function (Figure 4.7), which first establishes a UNIX Domain Socket (UDS) connection to the FLASH Monitor. Through this channel, it requests the creation of an `AF_XDP` socket, allocates a UMEM region, and retrieves routing metadata. Once this information is received, the library maps the UMEM into the process's virtual address space and populates the `Fill Queue` with usable packet descriptors. By abstracting these steps, the library enables NFs to start processing packets with minimal boilerplate.

```rust
pub fn connect(config: &FlashConfig) -> Result<(Vec<Socket>, Route), FlashError> {}
```

Figure 4.7: Rust Library: `connect` function

Configuration is supplied via the `FlashConfig` struct, which can either be constructed programmatically or parsed from CLI arguments using the optional `clap` integration. On success, the function returns a tuple containing a vector of `Socket` objects and a `Route` object, which holds forwarding information relevant to NF chaining.

```rust
pub struct Socket {
  // no pub fields
}

impl Socket {
  pub(crate) fn new(...) -> io::Result<Self> {}
  pub fn poll(&mut self) -> io::Result<bool> {}
  pub fn recv(&mut self) -> io::Result<Vec<Desc>> {}
  pub fn send(&mut self, descs: Vec<Desc>) {}
  pub fn drop(&mut self, descs: Vec<Desc>) {}
  pub fn read<const SIZE: usize>(&mut self, desc: &Desc)
              -> io::Result<&mut [u8; SIZE]> {
    // runtime size check
  }
...
}
```

Figure 4.8: Rust Library: `Socket`

The `Socket` struct (Figure 4.8) encapsulates an `AF_XDP` socket and exposes a minimal, safe API for packet I/O. To preserve safety and encapsulation, all internal fields are private, and

interaction is restricted to a set of carefully designed methods. This prevents NF developers from accidentally or maliciously modifying internal state. The public API includes:

- `poll`: Blocks until packets are available using the `poll` syscall
- `recv`: Retrieves packet descriptors from the `Rx Queue`
- `send`: Enqueues packet descriptors to the `Tx Queue` for transmission
- `drop`: Releases descriptors to the `Completion Queue` when packets are discarded
- `read`: Provides safe access to raw packet data in UMEM with bounds checks

```
pub struct Desc {
  addr: u64,
  len: u32,
  options: u32,
}

impl Desc {
  pub fn len(&self) -> usize {
    self.len as usize
  }
  pub fn set_next(&mut self, idx: usize) {}
}
```

Figure 4.9: Rust Library: `Desc`

Descriptors representing packets are encapsulated in the `Desc` struct (Figure 4.9). These objects store the memory address, length, and metadata associated with each packet. Similar to the socket, all fields in `Desc` are private, and NFs are not allowed to create or modify descriptors directly. This prevents incorrect memory access or descriptor reuse.

Descriptors are transferred to the NF on receipt and passed back to the library via `send` or `drop`, ensuring strict ownership semantics. This design enforces packet isolation: once the NF has relinquished a descriptor, it can no longer access the corresponding packet data. Figure 4.10 illustrates the flow diagram of a Rust-native NF's lifecycle.

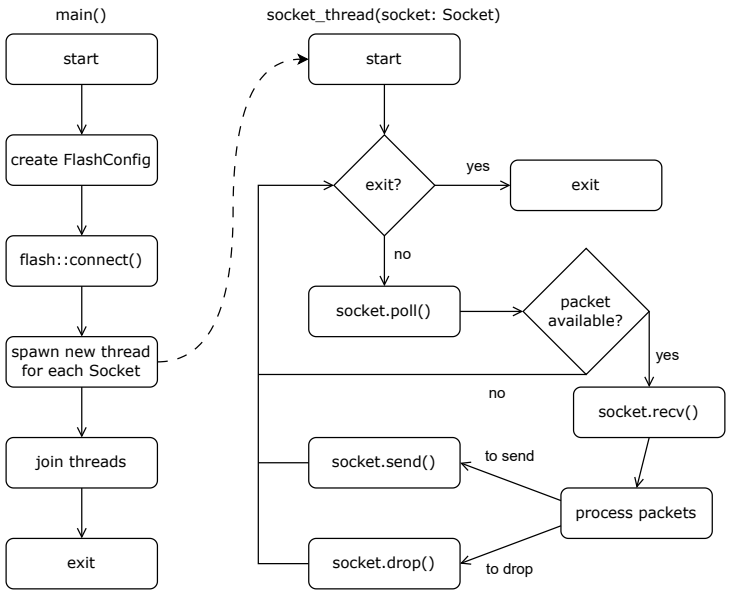

Figure 4.10: Flow Digram of Rust-Native NF's Lifecycle

4.4 Code Evaluation

To validate the effectiveness and completeness of our Rust-native FLASH NF library, we implemented a set of sample Network Functions (NFs) based on those in the original C-based FLASH NF library. These NFs span a variety of functional types — ranging from basic forwarding to stateless filtering and hashing logic — to cover a broad range of use cases. The Rust-based implementations include:

- ➔ `helloworld-rs`: Minimal example NF that connects to the FLASH Monitor
- ➔ `simplefwd-rs`: Forwards all received packets without modification
- ➔ `l2fwd-rs`: Performs Layer 2 forwarding based on MAC address rewriting
- ➔ `ip4ping-rs`: Responds to IPv4 ICMP echo (ping) requests
- ➔ `firewall-rs`: Applies stateless packet filtering using 5-tuple rules
- ➔ `maglev-rs`: Hashing-based load balancer using the Maglev algorithm [3]

All of these NFs were implemented entirely in **safe** Rust, without the use of `unsafe` blocks. This demonstrates that writing network functions in Rust is both practical and expressive, and that low-level packet processing is achievable without compromising memory safety.

To further quantify the benefits, we compared the code size of both implementations. Table 4.1 shows a breakdown of lines of code (LoC) for the core FLASH library and the sample NFs in both C and Rust. Despite Rust's stricter type system and verbosity in some areas, the Rust implementation is generally more concise.

Lang	Library	NFs (in both C & Rust)	Boilterplate code in each NF
C	1849 (approx)	1625	199
Rust	1510	862	109

Table 4.1: Lines of Code for FLASH Library

The Rust NF library itself has fewer lines of code than its C counterpart, owing in part to more expressive abstractions and better reuse of utilities. Even more significant is the reduced boilerplate in individual NFs — Rust's strong type system and safety guarantees allowed for simpler, more compact code with fewer defensive checks and less manual memory handling.

Overall, the Rust reimplementation resulted in cleaner and safer code, while preserving all the performance-critical aspects of FLASH. It also lays a foundation for further tooling and static analysis that can take advantage of Rust's compile-time guarantees.

4.5 Performance Evaluation

To validate that the Rust-based FLASH NF library maintains the performance characteristics of the original C implementation, we conducted a set of benchmarks focused on throughput and latency. Our goal was to assess whether the safety and abstraction benefits of Rust come at any measurable runtime cost.

Experimental Setup

All experiments were conducted using two servers, each equipped with a 24-core Intel Xeon Gold 5418Y CPU and 128 GB of RAM. Both servers ran Ubuntu 22.04.5 with Linux kernel 6.10.6. The system under test (SUT) used FLASH's modified kernel with in-kernel chaining enabled, and hyper-threading was disabled to ensure consistent CPU behavior.

Traffic was generated using Pktgen [11], a high-performance packet generator. The generator server was equipped with an Intel E810 40 Gbps NIC, while the SUT featured a Mellanox ConnectX-4 (MT27700) 40 Gbps NIC. All benchmarks followed RFC 2544 guidelines for evaluating network performance.

Experiment 1: NF Chaining Throughput and Latency

We constructed a simple forwarding pipeline by chaining the `l2fwd` NF multiple times, simulating a common NF chaining scenario.

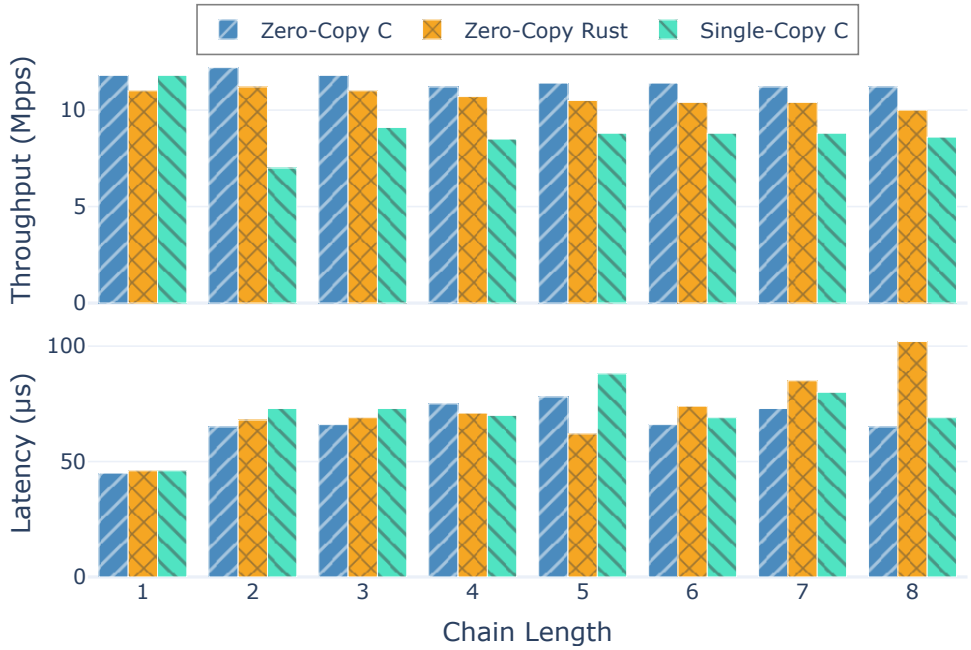

Figure 4.11: NF Performance Comparison

Figure 4.11 compares throughput and latency across three FLASH configurations: Zero-Copy C (ZC), Zero-Copy Rust (ZR) and Single-Copy C (SC). The ZR variant closely tracks the performance of the ZC baseline in both throughput and latency, demonstrating minimal overhead even at higher chain lengths. For instance, at 8 NFs, ZR achieves 10 Mpps compared to ZC's 11.2 Mpps. In contrast, the SC variant shows significantly lower throughput and higher latency, highlighting the cost of unnecessary memory copies. Notably, ZR maintains lower latency than SC at most depths and stays within 10–15% of ZC throughout, confirming that Rust's safety features do not compromise FLASH's performance goals.

Experiment 2: Firewall Scaling

To evaluate multi-threaded scaling, we deployed a topology in which multiple upstream NFs forward traffic to a single `firewall-rs` instance. The firewall was configured to run with 1 to 3 worker threads, each independently processing packets via separate socket queues. This setup simulates a high-fan-in scenario common in real-world middlebox deployments.

As shown in Figure 4.12, throughput increases nearly linearly with the number of threads, indicating that the Rust-based NF effectively utilizes available parallelism under sufficient load. The jump from 1 to 2 threads yields the most significant gain, while scaling to 3 threads continues to improve performance, albeit with diminishing returns likely due to cache contention. These results confirm that `firewall-rs` scales efficiently across cores while maintaining Rust's

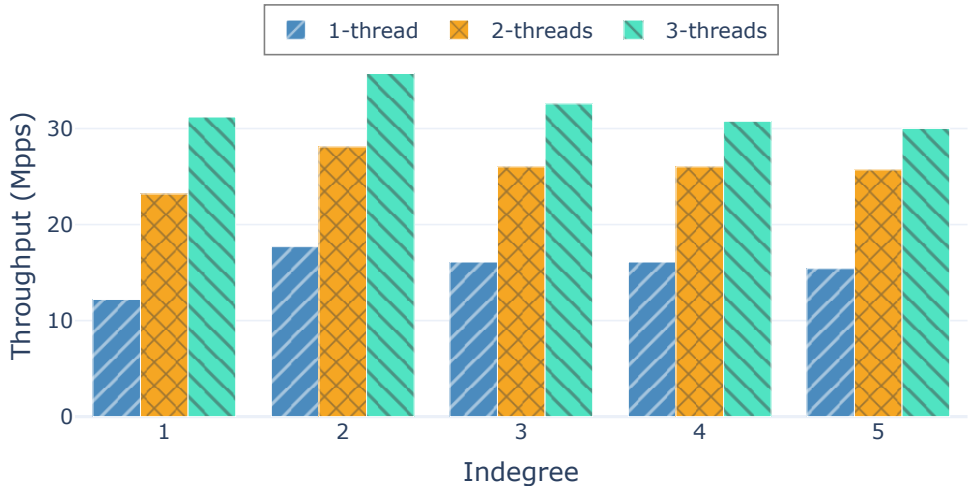

Figure 4.12: Firewall Scaling

memory safety guarantees, demonstrating that the design supports parallel, high-throughput processing.

Experiment 3: Maglev Scaling

We evaluated the scalability of `maglev-rs` in a load-balancing scenario, where a single instance distributes traffic to multiple downstream NFs. The Maglev NF was tested with 1 to 3 worker threads, each processing packets independently.

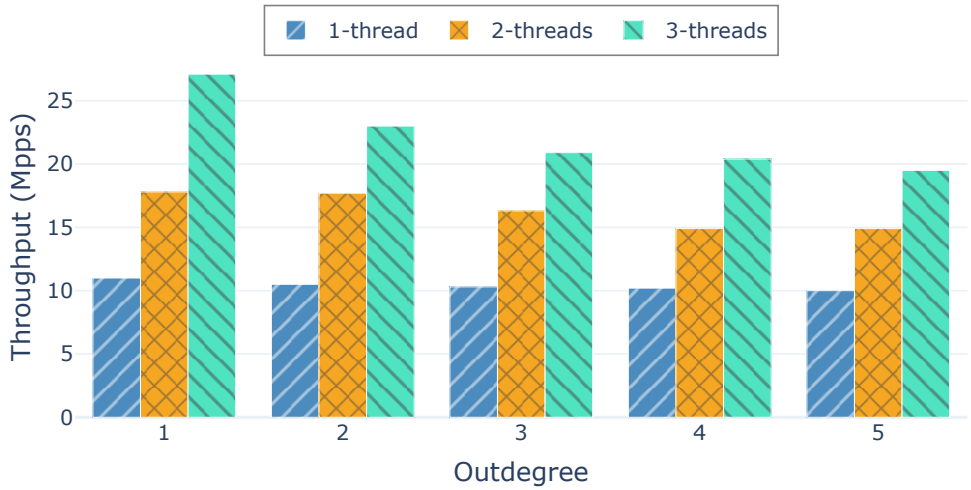

Figure 4.13: Maglev Scaling

Figure 4.13 shows that throughput increases with the number of threads, with the 3-thread configuration providing the highest throughput, especially with higher outdegree values. For example, with an outdegree of 1, throughput increases from 11 Mpps at 1 thread to 27.1 Mpps at 3 threads. Gains decrease slightly beyond 2 threads, likely due to parallelism overhead.

These results show that `maglev-rs` scales well across threads and efficiently handles load balancing while maintaining high throughput and memory safety guarantees.

5 Conclusion

5.1 Summary of Contributions

This work addresses key challenges in modern 5G network deployments by focusing on visibility, control, and performance optimization. Through three distinct yet interconnected contributions, we have advanced the state of the art in 5G testbed development, network protocol tracing, and high-performance packet processing. Specifically, this work touches upon the deployment of a 5G testbed, the extension of MobileInsight for 5G NR, and the reimplementation of the FLASH NF library in Rust.

In Chapter 2, we explored the development and deployment of a programmable 5G testbed based on OpenAirInterface (OAI) and integrated with FlexRIC, a near-real-time RAN Intelligent Controller (RIC). This testbed was designed to enable real-time observability into 5G control-plane activities. To further enhance its utility, we developed a custom xApp that collects fine-grained RAN-layer telemetry, providing crucial insights into key performance indicators (KPIs) and enabling real-time diagnostics, closed-loop control, and performance analysis.

Chapter 3 focused on enhancing MobileInsight, a powerful tool for cellular protocol tracing. While MobileInsight was already a widely-used tool for LTE networks, its support for 5G NR was limited. To bridge this gap, we reverse-engineered Qualcomm diagnostic packets typically only accessible with proprietary tools like QXDM. By developing custom parsers for these packets, we extended MobileInsight's capabilities, enabling real-time decoding and analysis of 5G control-plane signaling. This work significantly improves the ability to troubleshoot and optimize 5G networks using off-the-shelf, user-level tools.

In Chapter 4, we presented the development of a Rust-native version of the FLASH NF library. FLASH is a high-performance, zero-copy chaining framework for high-speed packet processing in Linux, leveraging `AF_XDP` sockets. By reimplementing the FLASH NF library in Rust, we not only preserved the performance characteristics of the original C-based version but also enhanced it with Rust's memory safety guarantees. Our benchmarking results demonstrated that the Rust-based FLASH library achieves high throughput and low latency while providing stronger guarantees for memory safety and network function isolation. This makes it an ideal solution for environments that require both high performance and robust isolation, such as multi-tenant or untrusted network functions.

Together, these contributions form a cohesive framework that enables deeper visibility, enhanced control, and improved security in 5G networks. By leveraging open-source platforms, extending existing tools, and utilizing Rust's unique capabilities, we have demonstrated that modern 5G infrastructures can be more programmable, observable, and secure. This work provides a solid foundation for future research and development in the evolving landscape of 5G and beyond, offering practical solutions for optimizing network performance and enabling more agile and reliable mobile infrastructures.

5.2 Future Work

This work lays the foundation for a programmable, observable, and performant 5G research infrastructure. Several opportunities remain to enhance and extend its capabilities across all three components:

5G Testbed

- ☞ Extend the OAI testbed with additional Core Network (CN) components (e.g., NSSF, PCF) to support full end-to-end 5G deployments
- ☞ Develop a modular tool for monitoring, control, and formal verification of the 5G Core Network
- ☞ Support diverse RAN configurations (e.g., gNBs from different vendors or simulators) for comparative evaluation and interoperability testing

MobileInsight

- ☞ Extend support for additional 5G NR messages to improve protocol coverage and diagnostic utility
- ☞ Automate the reverse engineering of Qualcomm diagnostic packets using machine learning or pattern mining techniques
- ☞ Finalize the 5G Measurement platform for comprehensive end-to-end user experience analysis

FLASH in Rust

- ☞ Port the FLASH Monitor control plane to Rust to unify the runtime in a memory-safe ecosystem.
- ☞ Implement more complex and stateful NFs (e.g., NAT, tunnel endpoints, DPI, L7 firewalls) to evaluate expressiveness, composability, and safety
- ☞ Evaluate performance under realistic workloads (e.g., QUIC, encrypted tunnels) and under stress conditions (e.g., DDoS, malformed packets)

References

[1] Arghyadip Chakraborty. *Evaluating India's 5G Rollout and Performance*. Tech. rep. Dept of CSE, IIT Bombay, Dec. 2024. URL: https://cdn.arghyac.com/iitb/rnd1-report.pdf.

[2] Debojeet Das et al. "FLASH: Fast Linked AF_XDP Sockets for High Performance Network Function Chains". In Submission: SOSP '25. 2025.

[3] Danielle E. Eisenbud et al. "Maglev: a fast and reliable software network load balancer". In: *NSDI'16*. USA: USENIX Association, 2016, pp. 523–535.

[4] Wireshark Foundation. *Wireshark*. URL: https://www.wireshark.org/docs/man-pages/wireshark.html.

[5] Yuanjie Li et al. "Mobileinsight: extracting and analyzing cellular network information on smartphones". In: *MobiCom '16*. New York, NY, USA: ACM, 2016, pp. 202–215. DOI: 10.1145/2973750.2973751.

[6] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[7] Navid Nikaein et al. "OpenAirInterface: A Flexible Platform for 5G Research". In: *SIGCOMM CCR '14* 44.5 (2014), pp. 33–38. DOI: 10.1145/2677046.2677053.

[8] Aether SD-Core Project. *gNB Simulator*. URL: https://github.com/omec-project/gnbsim.

[9] Qualcomm. *QXDM Professional™ Too*. URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/80-n9471-1_d_qxdm_professional_tool_quick_start.pdf.

[10] Robert Schmidt, Mikel Irazabal, and Navid Nikaein. "FlexRIC: an SDK for next-generation SD-RANs". In: *CoNEXT '21*. New York, NY, USA: ACM, 2021, pp. 411–425. DOI: 10.1145/3485983.3494870.

[11] Keith Wiles. *Pktgen - Traffic Generator powered by DPDK*. URL: https://github.com/pktgen/Pktgen-DPDK.