Open Source 5G Networks, Spoofing Attacks, and Proof-of-Concept Multi-Node Continuous Spectrum Analysis

A Major Qualifying Project Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the Degree of Bachelor of Science

SUBMITTED BY:

Kena Dudac, Electrical and Computer Engineering

Eric Heinemann, Computer Science

John Matthews, Electrical and Computer Engineering

Conor McDonough, Computer Science

John Max Petrarca, Computer Science

Matt Zoner, Electrical and Computer Engineering

ADVISED BY:

Project Advisors:

Professor Alexander Wyglinski,

Electrical and Computer Engineering

Professor Robert Walls

Computer Science

Worcester Polytechnic Institute

Submitted on: March 24, 2023

Abstract

As 5G networks increase in popularity, so do their security concerns. This report will be focused primarily on 5G Spoofing Attacks and proposes a way to prevent this. This is done by taking advantage of the Global User Temporary Identifier (GUTI), Random Access Control, and other aspects of 5G specifications to create a connection to a User Equipment (UE) that would accept 5G data packets and hold them from an adversary gNB. The original plan for this project was to create a 5G network and design spoofing and targeted attacks; however, several issues with the OAI software prevented this plan from materializing. Because of this, the latter part of this project shifted its focus to creating a network of sensor nodes to collect data from 5G networks that would allow for further analysis.

Acknowledgments

This project was successful through the help and support of other Worcester Polytechnic Institute community members and the Air Force Research Lab. Our team would like to thank Professor Alexander Wyglinski, the faculty advisor for this project, for his continuous guidance and support over the past academic year. Our team would also like to acknowledge Maya Flores, Adriyel Nieves, and Mitchell Jacobs

Table of Contents

List of Figures

List of Tables

List of Acronyms

Abbreviation	**Term**
3GPP	3rd Generation Partnership Project
4G	4th Generation
4IR	4th Industrial Revolution
5G	5th Generation
AF	Application Function
AKA	Authentication & Key Agreement
AMF	Access & Mobility Management Function
AUSF	Authentication Server Function
BPSK	Binary Phase Shift Keying
CBRA	Contention Based Random Access
CFRA	Contention Free Random Access
CN	Core Network
COTS	Commercial Off the Shelf
CP-OFDM	Cyclic-Prefix Orthogonal Frequency Division Multiplexing
CSI-RS	Channel State Information Reference Signal
CU	Centralized Unit
DM-RS	Demodulation Reference Signal
DN	Data Network
DNS	Domain Name Spoofing
DU	Distributed Unit
EPC	Evolved Packet Core
F1AP	F1 Application Protocol
DoS	Denial of Service

GNAP	Grant Negotiation and Authorization Protocol
gNB	Next Generation Node B
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communication
GTP	GPRS Tunneling Protocol
GUTI	Global Unique Temporary Identifier
HARQ	Hybrid Automatic Repeat Request
IMSI	International Mobile Subscriber Identity
IoT	Internet of Things
LDPC	Low Density Parity Check
LTE	Long-Term Evolution
MAC	
MIB	Master Information Block
MIMO	Multiple-Input and Multiple-Output
MITM	Man-In-the-Middle
NAS	Network-Attached Storage
NEF	Network Exposure Function
NF	Network Functions
NG	Next Generation
NR	New Radio
NSSF	Network Slicing Selection Function
OAI	OpenAirInterface
OFDM	Orthogonal Frequency Division Multiplexing
OSA	OpenAirInterface Software Alliance
PBCH	Physical Broadcast Channel

PCF	Policy Control Function
PDCCH	Physical Downlink Control Channel
PDCP	Packet Data Convergence Protocol
PDSCH	Physical Downlink Shared Channel
PHY	5G Physical Layer
PRACH	Physical Random Access Channel
PSS	Primary Synchronization Signal
PUSCH	Physical Uplink Shared Channel
QAM	Quadrature Amplitude Modulation
QoS	Quality of Service
QPSK	Quadrature Phase Shift Keying
RACH	Random Access Control
RAN	Radio Access Network
RF	Radio Frequency
RLC	Radio Link Control
RRC	Radio Resource Control
SBA	Service-Based Architecture
SDR	Software-Defined Radio
SMF	Session Management Function
SMS	Short Message Service
SQL	Structured Query Language
SRS	Sounding Reference Signal
SSB	Secondary Synchronization Block
SSS	Secondary Synchronization Signal
TMSI	Temporary Mobile Subscriber Identity

UDM	Unified Data Management
UDR	Unified Data Repository
UE	User Equipment
UPF	User Plane Function
X2AP	X2 Application Protocol

Executive Summary

This report outlines the team's efforts in setting up a 5G network to create a spoofing attack that would expose 5G vulnerabilities and the subsequent shift to developing a scalable spectrum sensing analysis network. The initial approach involved using OpenAirInterface (OAI) and srsRAN to establish a 5G network, but both attempts faced various challenges and ultimately did not yield a functional network.

The team encountered multiple issues with OAI, including the lack of the AVX2 instruction set in the initial computers, difficulties with virtual machines, and communication problems between the User Equipment (UE) and the network. Although the team managed to get a UE registered and connected to the gNB, it did not receive an IP address from the core network, preventing it from effectively communicating with the network.

Similarly, the team faced difficulties with srsRAN, including unreliability and inconsistent connections between the UE and the gNB. The team attributed the lack of success with OAI and srsRAN to outdated documentation and limited resources.

As a result, the focus shifted to a new project: developing a scalable spectrum sensing analysis network. The goal was to create a network of hardware nodes capable of collecting 5G spectrum data and making it available for predictive analysis. The sensor nodes would gather data using a software-defined radio (SDR) and send it to a server node for analysis.

The project had two primary milestones. The first was establishing a single sensor node communicating with a local server. This allowed the team to concentrate on adequately acquiring and transmitting data with minimal errors. It also helped to ensure correct server-client connections and error-free data transmission. The second milestone was to expand the system to include multiple sensor nodes, all sending information to the same server. The

server could aggregate the data and analyze the 5G channel by receiving samples from multiple sensor nodes.

To implement the project, the team used an Ettus Research USRP X310 connected to a high-performance server via high-speed ethernet for the initial test. As the number of nodes increased, four Ubuntu 20.04 desktop computers, each equipped with a 7th Generation Intel Core i7 and an Ettus Research USRP B210 connected via USB 3.0, were deployed.

Two primary Python programs were developed on the software level, one for the sensor side (client side) and one for the server side. The sensor side focused on data acquisition from an SDR and sent the acquired data to the server side using the "socket" Python module, facilitating the creation of a Berkeley socket. The server side received the packaged data continuously from all active sensor nodes, decoded it, and stored it with a timestamp for future analysis.

The team initially aimed to expose 5G vulnerabilities through spoofing attacks but faced several challenges with OAI and srsRAN. The focus shifted to developing a scalable spectrum sensing analysis network, which involved creating sensor nodes to collect and transmit 5G spectrum data for predictive analysis. The project was implemented in two milestones, first by establishing a single sensor node and then expanding the system to include multiple sensor nodes.

In conclusion, despite the initial challenges with OAI and srsRAN, the team successfully pivoted to develop a scalable spectrum sensing analysis network. By implementing a robust system capable of analyzing the 5G spectrum through data aggregation from multiple sensor nodes, the team has laid the groundwork for future research and development in 5G communication and analysis.

[1] Introduction

[1.1] Motivation

In a world full of growing demand for speed, reliability, and global communication via 5G networks, our team aims to tackle the neglected question of network and information security. Recent forecasts suggest that by 2030, more than 25 billion Internet of Things (IoT) devices will be used worldwide. As a result, there is an ever-growing concern about whether current security procedures can provide the necessary protection for users [1].

Our team focused on the Transmission Layer of the 5G architecture, commonly referred to as the Transportation or Network Layer, where communication channels serve to route data to the correct final destination. Protocols within the Transmission Layer complete data processing tasks such as mining, aggregation, and encoding.

The scalable spectrum sensing network project provided an opportunity to improve communications analysis by adding additional sensing nodes. This primarily concerns the Physical, or PHY, layer of the 5G architecture, where carrier frequencies and resource block lengths are more relevant.

[1.2] Technical Challenges

The 5G standard provides various features that make security attacks more difficult to perform compared to its predecessor, 4G. Our goal was to test 4G LTE vulnerabilities in a 5G environment, specifically spoofing or man-in-the-middle (MITM) attacks, so that future improvements can be made to 5G architecture. Toward the end of the project, the team pivoted to another 5G-focused challenge. We aimed to create a scalable sensor network to analyze 5G data. Having multiple sensors send data to a server to be analyzed will be difficult, given the amount of data collected from the sensors [2].

[1.3] Notable Hardware

Ettus Research USRP B210 SDRs Implementation

Two USRP B210 SDRs were used to construct the gNB and UE. The software OpenAirInterface was used to develop a 5G cellular network connecting the gNB SDR and UE SDR. Additionally, these SDRs were used to add nodes to the sensor network after the project switch.

Ettus Research USRP X310 SDR and Sensor Nodes Implementation

The X310 was primarily used in the second project, where the team used it to test run initial sensor collection code. It was chosen for its superior performance over the B210 to remove any bottlenecks during development.

[1.4] Report Organization

This report consists of five chapters. Chapter 1 describes the project’s motivation, an overview of technical challenges, and the most utilized hardware devices, as seen above. The Chapter 2 will cover details on the overall 5G architecture, components, and vulnerabilities while describing OpenAirInterface. Chapter 3 will present how the team tried to implement the open-source 5G base station. Chapter 4 will explain how the team implemented a spectrum sensing analysis network. These chapters include the problem statement, proposed solution, implementation, and a description of the results. Finally, the report will conclude by summarizing the project and presenting a few recommendations and research possibilities for the future.

[2] Overview of 5G Technology

This chapter outlines the structure of 5G networks. It begins by explaining the overall 5G architecture, detailing the components of a 5G network, and comparing 5G networks to 4G networks. The chapter then provides an overview of 5G vulnerabilities and the security principles that they compromise. Finally, the chapter describes OpenAirInterface and how it is used to set up a 5G network for testing purposes.

[2.1] 5G Architecture

5G cellular networks are designed to provide faster connectivity, increased bandwidth, and better security compared to 4G. It consists of the same three components as the previous generation: 5G Core Network (5GCN), Next Generation Radio Access Network (NG-RAN), and User Equipment (UE). The 5GCN uses a cloud-aligned service-based architecture (SBA) to support authentication, security, session management, and traffic aggregation from connected devices. The 5GCN is composed of multiple components that make up its architecture, including the User Plane Function (UPF), Data network (DN), operator services, Internet access or third-party services, Core Access and Mobility Management Function (AMF), Authentication Server Function (AUSF), Session Management Function (SMF), Policy Control Function (PCF), Unified Data Management (UDM), and Application Function (AF). These components can be seen in Figure 2.1.1 and Table 2.1 [3].

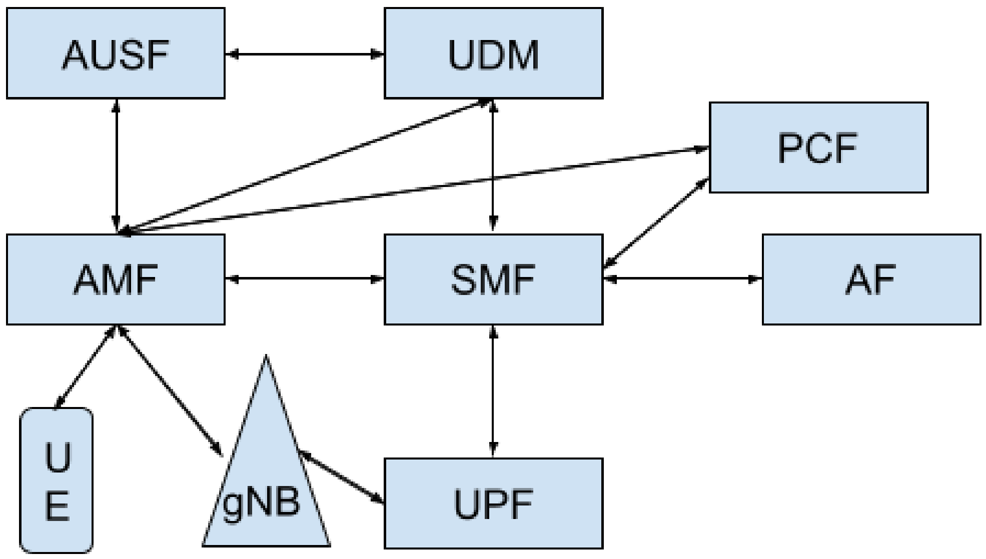


Figure 2.1.1 5GCN System Architecture. A more detailed visual of the system utilized in the project is found in Section 2.4, with divisions of the types of Network Functions

The NG-RAN comprises base stations, also known as access nodes or gNodeBs (gNBs), which are connected to the same 5GCN. The purpose of the NG-RAN is to connect individual devices to other parts of a network, which may include any type of UE [11]. A UE is any piece of equipment designed for use by an end user, such as cellphones, laptops, tablets, or any other remotely controlled machine. The primary difference between 4G and 5G architecture is that 5G architecture uses service-based communications between user plane and control plane services. This enables more effortless scalability and flexible deployments [1]. The service-oriented architecture in 5G technology allows components to request a new network entity to discover and communicate with each other over application programming interfaces. The architecture enables each network function to communicate with every other function [12].

Table 2.1 5G architecture components with descriptions

Architecture Component	Description
Authentication Server Function (AUSF)	Receives authentication requests from the AMF, interacts with the UDM to obtain authentication vectors for processing 5G authentication, validates network responses to determine if authentication was successful, and handles re-synchronization procedures [4]
Unified Data Management (UDM)	A centralized way to control network user data [5]
Access and Mobility Management Function (AMF)	Receives all connection and session related information from the UE Also handles connection and mobility management tasks [6]
Policy Control Function (PCF)	A platform that governs the implementation of policy control and charging rules. Also enables end-to-end policy management based on network parameters, implements slice-based policies for applications, and offers advanced analytics for improved services [7]
Session Management Function (SMF)	Responsible for interacting with the decoupled data plane, creating, updating, and removing PDU sessions, and managing session context with the UPF [8]
Application Function (AF)	A control plane function within the 5G core network that provides application services to the subscriber [9]
User Plane Function (UPF)	Responsible for routing and forwarding user plane gNB packets to the external data network, and also handles downlink packet buffering and downlink data notification triggering [9]
Data Network (DN)	Identifies Service Provider services, Internet access, or 3rd party services [10]

In contrast to its predecessor LTE, which used the MME for both, the 5GCN separates mobility and session management. The AMF is a control plane function responsible for mobility management and provides UE location information. It can retrieve information such as status changes, time zone shifts, and location shifts, which can be narrowed down using filters. Additionally, it frequently interacts with subscriber databases to determine whether the subscriber is allowed on the network. When a device is signaled to the network, the AMF provides a temporary identity. It also manages security context, access authentication, connection management, and registration management [13]. The SMF is another control plane function that creates and manages many sessions based on network regulations sent by the PCF. This includes allocating IP addresses to UEs using information received from the UPF. The UPF is always running to and from the gNB and Core Network, making it an ideal point to enforce QoS (Quality of Service) and policy enforcement. The policy control function regulates the AMF and SMF by checking whether the network meets the active conditions. To have an effective 5G Core Network, all these functions must run together on a designated COTS server or device [14].

An overall example of 5G architecture can be seen in Figure 2.1.1. 5G architecture considers indoor and outdoor scenarios and prevents signal loss by restricting penetration through walls. Using an array of antennas, 5G addresses can be evenly distributed across all antennas. Users indoors can communicate using indoor access points, while outdoor users connect via large antenna arrays. Although very complicated, the 5G architecture has been designed to provide higher speeds, less latency, capacity for a more significant number of connected devices, less interference, and better efficiency than that of the 4G architecture.

There are two ways to implement current 5G technology: standalone and non-standalone. The non-standalone implementation allows for a smoother transition to 5G technology, as a previously used 4G core can serve as a 5G headend. This configuration was designed to facilitate a seamless transition from the previous generation. The standalone version of 5G is a completely independent network configuration that does not rely on a 4G EPC. The industry has found significant use for both versions of 5G technology.

[2.2] 5G Physical Layer

The 5G physical (PHY) layer serves as the foundation for a 5G network. This layer can support a broad range of frequencies, ranging from under 1 GHz to 100 GHz, and supports various deployment methods such as pico cells, micro cells, and macro cells. The New Radio (NR) protocol has developed constantly since April 2016, with the standardized name for the protocol being 3GPP. The first public release of 3GPP NR (Release 15) was in June 2018. The protocol is designed so that any future release of the NR is backward compatible, referred to as forward compatibility of NR.

NR supports a wide range of different modulation schemes, including quadrature phase shift keying (QPSK), 16 quadrature amplitude modulation (QAM), 64 QAM, 256 QAM, and π/2-BPSK. These modulation schemes work with both downlink and uplink, except for π/2-BPSK, which only works with uplink [15].

NR uses a cyclic prefix OFDM (CP-OFDM) for both uplink and downlink. Using the same waveform for both uplink and downlink simplifies the design, particularly for device-to-device communications. The CP-OFDM operation is similar to the OFDM used in 4G LTE, but 4G LTE can only use a fixed 15 kHz subcarrier. The CP-OFDM numerology supported by NR is scalable, allowing it to access a wide range of frequencies and deployments. The subcarrier spacing is specified by a range of $15 * 2^n$ kHz, with n being an integer. Each CP-OFDM signal contains data payloads inside OFDM symbols and cyclic prefixes [15]. These prefixes are created to eliminate intersymbol interference from the previous symbol. Usually, two types of cyclic prefixes are used: normal or extended. A third extended prefix is also available but rarely used. The 3GPP Release 17 supports seven subcarrier spacings [16], as shown below in Table 2.2.

Table 2.2.1 Subcarrier spacings available as of 3GPP Release 17

n	$\Delta f = 15 * 2^n\ kHz$	Cyclic Prefix
0	15	Normal
1	30	Normal
2	60	Normal, Extended
3	120	Normal
4	240	Normal
5	480	Normal
6	960	Normal

The scalable nature of NR subcarrier spacings means that it would be possible to add much higher mmWave frequencies in future releases [16].

5G requires advanced error correction techniques for channel coding [17]. To fulfill the 5G communication requirements, NR uses low-density parity check (LDPC) codes for data transmission and polar codes for control signaling [15]. The LDPC codes implemented for NR utilize a rate-compatible structure. This structure allows for HARQ operation using incremental redundancy and transmission at different code rates. The use of LDPC code also allows for higher throughput, low power dissipation, and low latency [17]. The encoding and decoding of both LDPC and polar codes are too detailed to cover in this report. Overall, the schemes chosen offer a favorable tradeoff of performance and complexity.

The 5G architecture’s physical layer can be broadly separated into two types of elements: channels and signals. First among the channels are the Physical Downlink Shared Channel (PDSCH) and Physical Uplink Shared Channel (PUSCH), both of which work to carry user data between the UE and the base station. Next are the Physical Downlink Control Channel (PDCCH) and Physical Uplink Control Channel (PUCCH). The former is used for transmitting control information to the UE, such as the scheduling of uplink and downlink transmissions and their modulation and format. The latter allows the UE to send control information to the base station, such as scheduling requests and channel state information. The Physical Broadcast Channel (PBCH) contains the Master Information Block (MIB), the crucial system information that is broadcasted to the UE so it can access the network. Last among the PHY layer channels is the Physical Random Access Channel (PRACH). The UE uses this to send identification information about itself to the base station it is attempting to connect to.

Table 2.2.2 5G physical channels with descriptions

5G Physical Channels	Description
Physical Downlink Shared Channel (PDSCH)	The main physical channel used for unicast data transmission. Also transmits paging information, random-access response messages, and delivers system information [18]
Physical Uplink Shared Channel (PUSCH)	Used to carry the user data and optionally the Uplink Control Information (UCI) [19]
Physical Downlink Control Channel (PDCCH)	Used for downlink control information, scheduling decisions, and scheduling grants enabling transmission on the PUSCH [18]
Physical Uplink Control Channel (PUCCH)	Used to carry the UCI with the Channel State Information (CSI) reports, the HARQ feedback, and the Scheduling Requests (SR) [19]
Physical Broadcast Channel (PBCH)	Carries part of the system information, required by the device to access the network [18]
Physical Random Access Channel (PRACH)	Transmits an initial random access preamble consisting of sequences used for channel access [20]

In addition to channels in the PHY layer, various signals are utilized. These can be delineated into the reference signals and the synchronization signals. The synchronization signals consist of the Primary Synchronization Signal (PSS) and the Secondary Synchronization Signal (SSS). These two signals, along with the PBCH described above, make up the Synchronization Signal Block (SSB). This block is used for synchronizing the timing and frequency of communication between base stations and mobile devices.

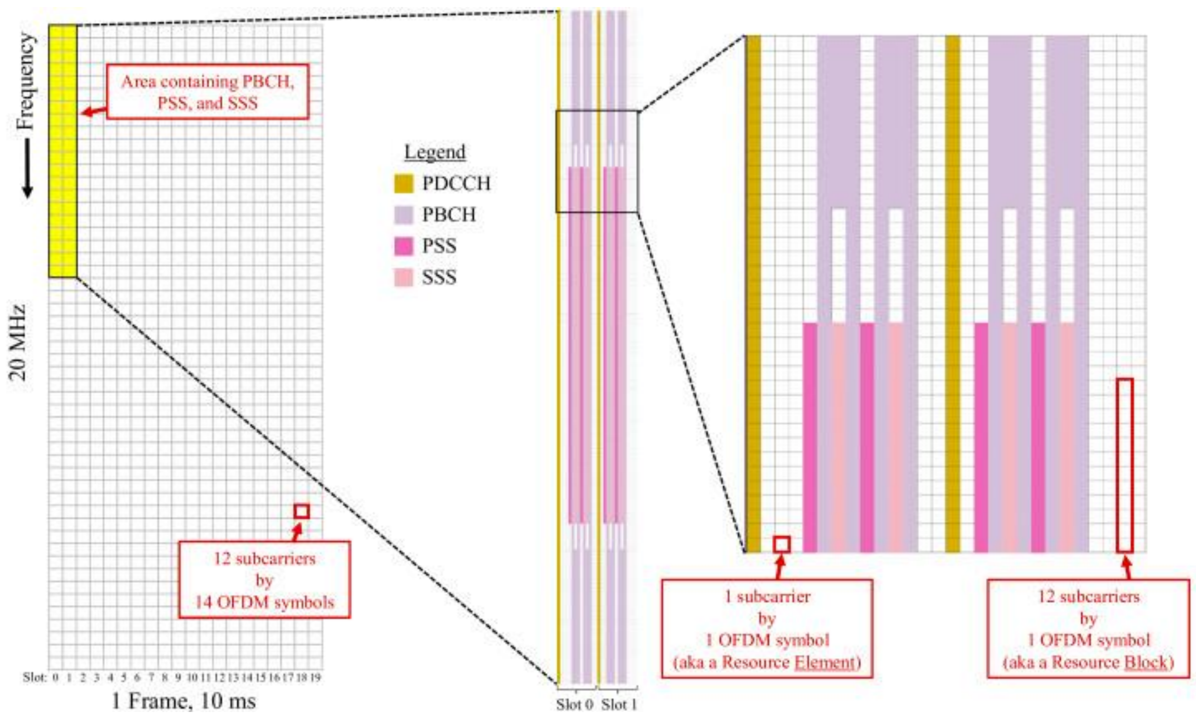

Frequency
20 MHz
Area containing PBCH, PSS, and SSS
12 subcarriers by 14 OFDM symbols
Slot: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1 Frame, 10 ms
Legend
PDCCH
PBCH
PSS
SSS
Slot 0
Slot 1
1 subcarrier by 1 OFDM symbol (aka a Resource Element)
12 subcarriers by 1 OFDM symbol (aka a Resource Block)

Figure 2.2 The Synchronization Signal Block (SSB), divided into various subcarriers. Each OFDM symbol, or Resource Block, is composed of 12 subcarriers defined during the gNB-UE connection process. These Resource Blocks are used to send information such as the synchronization information in the PSS and SSS or the scheduling defined in the PDCCH.

Other than synchronization signals, various reference signals are integral to the 5G PHY layer. The first of these is the Demodulation Reference Signal (DM-RS), which the UE uses to demodulate incoming transmissions from the base station and read their contents, operating in both uplink and downlink connections. Next is the Channel State Information Reference Signal (CSI-RS), which is only used for downlink communication. It allows a UE to acquire information on the channel state, track time/frequency, perform beam management, and uplink reciprocity-based precoding. The Sounding Reference Signal (SRS) is only used for uplink communication and is utilized by the base station primarily to estimate the quality of the channel over a wide bandwidth.

[2.3] 5G Vulnerabilities

With the massive push for Internet of Things connectivity comes an increase in devices such as smart homes, smart cars, and smart cities. These devices are connected to the same network, which can potentially increase the number of eavesdropping devices, denial of service attacks, and unwanted private data collection devices. Furthermore, since older devices often do not have the same security protocols, it endangers the privacy and security of those connected to that network.

The PHY layer comprises multiple channels and signals that offer varying levels of susceptibility to spoofing and jamming attacks. The PSS and SSS, which were described in the previous section, are used together to transmit the Physical Cell ID and signal timing. These signals also contain certain criteria for the handover process, such as power levels [21]. If an adversary wishes to cause a denial of service (DoS) attack on a particular UE, they may broadcast false PSS/SSS signals at a higher power level than the base station being spoofed. This can capture the UE and prevent its signals from reaching the base station. An attack targeting this part of the PHY layer can cause a denial of service while the UE is still performing its cell search.

Another vulnerability that exists in 5G technology is related to device capability information. Altaf Shaik, a graduate student at the Technische Universität Berlin and Kaitiaki Labs, conducted testing on 5G networks and was able to capture device capability information. This is because device capability information is sent to the base station before any security measures are implemented. This information is unencrypted and can be viewed as plain text [22]. This information can enable mobile network mapping, bidding down, and battery draining. Shaik used a fake base station to determine certain characteristics of a 5G device, such as the baseband processor and modem. One can draw conclusions about the device with the model number of the modem and its features, such as voice codecs for phones or a lower power mode for IoT devices. This may include identifying security vulnerabilities prone to a specific radio component, device type, or device model. With this information, an attacker can plan a tailored strike [22].

Table 2.3.1 The taxonomy of different cellular vulnerabilities

Device Type	Threat
IoT	Denial of Service, Data Breach, SQL/Code Injection, Brute Force Attack
Cellular Device	Phishing, Data Breach, Man in the Middle Attacks, Malware, DNS Tunneling
Applications	Denial of Services, Data Breach, SQL/Code Injection

Unlike mobile network mapping, bidding down and battery drain attacks can be executed by using a man-in-the-middle relay [22]. Bidding down involves altering the RAN, which determines the speed at which the UE can receive information. For example, an iPhone that can receive 1 GB/s can be altered to only receive 2 MB/s. Carrier Aggregation and MIMO enablement can be removed to worsen the performance of fast devices, or the frequency band information can be changed to prevent device roaming. In battery drain attacks, the power-saving mode parameters are altered so that a device is never in sleep mode and is constantly searching for a connection. This attack works on devices that send little data over an extended period. Disabling power-saving mode abilities was found to decrease the battery life of devices by a factor of five [22].

Table 2.3.2 The different security principles of the 5G vulnerabilities

Security Principle	Threat	Impact
Confidentiality	AKA Attack, Unsecured DNS Paging Broadcast	Spoofing, Malware dropping MITM Location Determination
Integrity	Silent Downgrade AKA Attack	Phone/SMS snooping, Subscriber Impersonation
Availability	Spectrum Slicing Attack, Botnet Attack Paging Attack	Performance Degradation, Denial of Service

In the paper "Overview of 5G Security and Vulnerabilities," Shane Fonyi breaks down 5G vulnerabilities as shown in Table 2.5. The first category Fonyi describes is confidentiality. Confidentiality refers to the privacy of information and the authorization of people to access it [23]. In terms of networking, this could include text messages, phone calls, files, emails, and any other type of internet traffic [24]. Ensuring confidentiality is an essential part of any functional network. To ensure confidentiality, authentication and key agreements are used. Authentication validates that the network or device is who it claims to be through certificates. Key agreements occur when the network and device put in place the necessary cryptographic keys so that only they can decrypt traffic between each other. These techniques help prevent information from being sent and read by the wrong device [25]. Research conducted on authentication and key agreement has revealed weaknesses in the system that allow for "false base station attacks and IMSI catchers through non-protected identity request mechanisms and authentication failure messages" [24]. These protocols allow attackers to connect to a network while pretending to be a different valid user. This is possible because the keys in the key exchange are transported insecurely when the device is roaming [24].

Attackers can also learn about a user's cellular usage through an attack on the predictability of the sequence number of a device. The sequence number grants access to private information, and with it, an attacker can find out information like the time spent on the phone, the number of texts sent, the user's location, and the schedule. While some of these security issues originated in 4G networks, they still exist in 5G because backward compatibility is necessary. Man-in-the-middle attacks are one such issue that carried over from 4G networks. In this type of attack, an adversary device can listen to the conversation between two devices to obtain private information like personal data, banking information, or passwords [26]. Fonyi states that this is possible in 5G because base stations can reuse old keys from previous sessions that appear legitimate to various UE devices. The hole in the authentication and key agreement protocols allows for StringRay or other IMSI catchers [24]. DNS traffic was also found to be insecure in 5G networks, which can lead to an array of issues like stolen credentials, the deployment of remote malware, and other problems [24].

Fonyi also asserts that the location of a user can be discovered through the broadcasting of the Temporary Mobile Subscriber Identity (TMSI) assigned to a user by the Mobile Management Entity [24]. The TMSI is given to users to support subscriber identity confidentiality. In theory, a user's TMSI should be changed frequently to prevent security breaches; however, this does not happen in reality. Fonyi argues that when there are multiple services waiting for the UE, the UE requests a base station to broadcast a paging message that contains the TMSI. An attacker can then sniff traffic on the network to determine the paging interval of the UE, which further allows the attacker to track where the UE is located. This is a serious flaw in 5G technology, as it is a privacy and safety concern for each user.

The second area of concern described in Fonyi's paper deals with integrity. Integrity in computer networks ensures that data is not modified or deleted by unauthorized users. Data integrity can be preserved by backing up files and encrypting data when it is being sent [27]. Message signatures are a method that ensures data is not altered after being sent. The consequences of data being changed from source to destination can vary greatly. Fonyi asserts that message alteration is possible in 5G networks. He describes that in "the current model, message authentication provides the verification of the source; however, there is no protection against the duplication or modification of the message" [24]. An attack that deals with integrity is message spoofing. Message spoofing is the process of sending a message to a user while claiming to be a friendly source. These attacks include DNS, website, IP address, and email spoofing [28]. Fonyi claims that spoofing attacks are possible through the authentication and key agreement holes described earlier. The last attack on integrity Fonyi asserts is possible in 5G networks is a silent downgrade attack where a "malicious base station may be able to force the UE to downgrade to GSM, an older and less secure communication protocol, exploiting the pre-authentication messages" [24].

The last security principle of 5G vulnerabilities is availability. Availability is a principle that guarantees that a service, application, or data will be available to a user when needed [29]. To avoid attacks on availability, redundancy paths and failover strategies must be incorporated into a system. Availability is often considered the most important ability of an application because, without availability, there is no application at all. A real-life example of availability would be cell phone service. Without service, a user cannot communicate with anyone, which can be a safety concern if help is needed. Within 5G networks, there is the possibility of distributed denial of service (DDoS) attacks. A DDoS attack, as described above, is when an adversary attempts to bring down a service by overloading the system with a large amount of data from a large number of devices. These attacks are hard to prevent and exist in 4G networks as well [24]. The rise of IoT devices allows for these attacks to become more devastating.

As described above, there are 5G vulnerabilities in all three principles of information security. An array of attacks are possible, and there is still a lot of work to be done in 5G to fix the various issues. The current 5G infrastructure allows for attacks on confidentiality, where an adversary can obtain private data or information. Attacks on integrity, where an attacker can change data being sent that may lead to catastrophic results, and attacks on availability, where a botnet of devices can send a large amount of data to a service to overload the system and bring it down. The lack of foolproof security opens the door for attackers to create and plan new attacks that could have a global impact.

[2.4] OpenAirInterface (OAI)

OpenAirInterface (OAI) is an open source platform developed by the OpenAirInterface Software Alliance (OSA) to support mobile telecommunication systems like 4G and 5G and their development. OSA's goal is to gather a community of developers and work together to build Radio Access Network (RAN) and Core Network (CN) technologies [30]. OSA provides a GitLab repository containing all former and current projects and software.

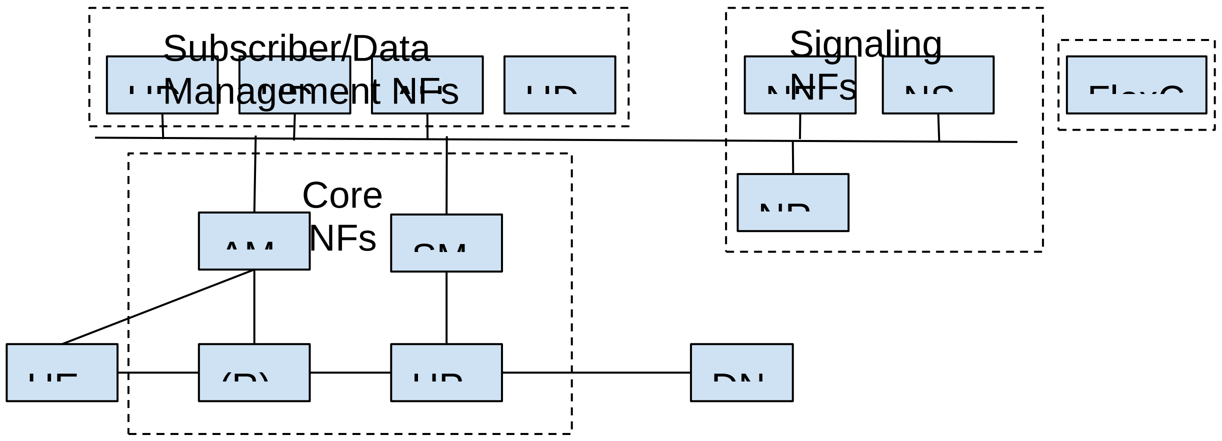


Figure 2.4.1 OAI's 5G CN's non-roaming reference architecture, divided into the primary Network Functions (NFs). The Core NFs are the components of the 5G CN which have the most direct interaction with a user. For example, the AMF is used to validate a user's enrollment with a network provider as it connects to different gNBs. The Subscriber/Data Management NFs contain more security and enrollment information. This includes the AUSF, which receives authentication requests from the AMF and communicates with other portions of the CN for security purposes. Lastly, the signaling NFs allow for a network provider to connect the rest of the NFs and handle headend functions.

The team used the development branch of the OAI repository to build and create a 5G RAN system containing both the physical (PHY) and high layers for a gNodeB (gNB) and User Equipment (UE). The gNB PHY layer contains the PSS, SSS, PDCCH, PDSCH, PUCCH, PUSCH, and PRACH. The gNB high layer contains the MAC, RLC, PDCP, RRC, X2AP, GNAP, F1AP, and GTP-U. The UE PHY layer contains software to receive and decode the PSS, SSS, PDCCH, and PDSCH information and generates data for the PUCCH, PUSCH, and PRACH. The UE high layer contains the MAC, RLC, PDCP, RRC, and NAS. These two systems are connected using a software-defined radio (SDR). The CN for the 5G RAN system was also created using repositories developed and provided by OSA (Figure 2.4). The CN currently supports the following network elements: Access and Mobility Management Function (AMF), Session Management Function (SMF), User Plane Function (UPF), Network Repository Function (NRF), Authentication Server Function (AUSF), Unified Data Management (UDM), Unified Data Repository (UDR), Network Slicing Selection Function (NSSF), and Network Exposure Function (NEF) [31]. All three systems are run on Linux machines, with the CN and gNB being connected by an ethernet switch and gNB and UE being connected by SDR. OAI supports both a non-standalone mode and a standalone mode for the gNB (Figure 2.5) [30]. This project used the standalone method and bypassed the need for an EPC or LTE interface to connect the UE and CN.

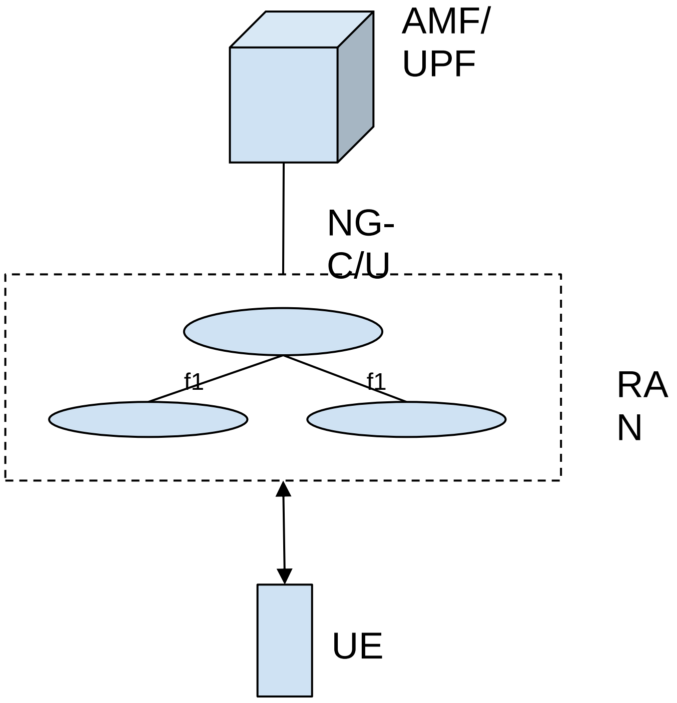


Figure 2.4.2 OAI’s 5G RAN operating in standalone mode. As shown, OAI supports the Centralized Unit/Distributed Unit (CU/DU) split. The CU provides RRC and other protocols, while the DU deals with the MAC layer and portions of the PHY layer. The RAN then connects with the AMF and other authentication CN components.

OpenAirInterface provides three important directories: openair1, openair2, and openair3. Each of these directories has its own significance to the system. Openair1 is the location of the code that handles UE scheduling processes and the PHY layer of our network. Functional code for X2AP, RRC, driver, PHY UE interface, and in-depth documentation can be found in the openair2 directory.

[2.5] Chapter Summary

This chapter outlined the structure of 5G networks, discussed 5G’s architecture, the physical layer, the vulnerabilities that 5G networks currently have, and the current implementation methods we used (OAI). The architecture and physical layer were designed for higher speeds, less latency, the capacity for a more significant number of connected devices, less interference, and better efficiency [32]. Although 5G provides advancements, new vulnerabilities have been exposed, which a man-in-the-middle attack could exploit. Due to the increased flexibility and autonomous features added to 5G, an adversary base station can easily replicate signals sent out by “real” base stations. In doing so, the adversary base station becomes a new node in the network, controlling traffic routing and possibly accessing private information. To assist in researching the possibility of this attack, the team planned to use OAI to implement the needed RAN, CN, and UE. The 5G network would use OAI’s standalone functionality to minimize confusion between 4G and 5G vulnerabilities.

[3] Open Source 5G Base Station Implementation

[3.1] Problem Statement

The project's original goal was to demonstrate security vulnerabilities in 5G base stations. The team had to construct a 5G cellular network utilizing free, open-source programs. The two most notable options were OpenAirInterface (OAI) and srsRAN. OAI was initially favored in A term due to the previous experience and expertise held by multiple members in WiLAB. The team then used part of B term to experiment with srsRAN due to suggestions based on its improved documentation and stability.

There are several complexities involved in tricking a UE into connecting to the adversarial gNB. For example, the initial uplink connection can involve either “Contention Based Random Access” (CBRA) or “Contention Free Random Access” (CFRA), depending on the implementation. In CBRA, the UE first sends a preamble to the gNB, which response with the same preamble and additional timing and identifier information. In CFRA, the UE is assigned a preamble by the gNB, and it must send a request for the same timing and identifying information. After the preambles are sent back and forth identically, frequency and time resources are allocated.

Several steps later, the Global Unique Temporary Identifier (GUTI) is sent from the UE. This allows for UEs to connect more anonymously. It can be changed to a new random identifier periodically, depending on the implementation by the carrier. This could have added complexity as the adversarial node may need to update on a similar cadence to maintain the base station ruse. Additional problems can arise when the gNB is required to send a security header containing the K-gNB key, an identifier defined by the Access and Mobility

Management Function (AMF). Due to this key coming from the carrier, it is among the primary concerns when fooling a UE.

[3.2] Implementation

Implementation of OAI began on three older (prior to 2013) desktop computers. These computers were going to be used as an initial test for installing and running a CN, gNB, and UE. The computers were unable to install the most recent version of OAI, as they did not have Advanced Vector Extensions 2 (AVX2) functionality. AVX2 is a vectorization extension added to the Intel x86 instruction set that allows for single instruction multiple data instructions over vectors of 256 bits [33]. When installing OAI with a processor that does not support AVX2, the installation will stop when attempting to install the soft-modem package. The AVX2 requirement was discovered after posting a question on the openair5g-devel mailing list from OAI [34].

After the AVX2 issues, the team made the switch to virtual machines that had the required instruction set. Following OAI's documentation, the team was able to successfully install and build the UE, gNB, and core network on the virtual machines. The team continued to follow the documentation, and the core network, gNB, and UE were all set up and made ready to run. A test bench was set up with two USRP B210s, a Bulipu GPS Clock for a PPS signal, a signal generator for a 10 MHz wave, and an OctoClock Clock Distribution Module to connect the entire system together.

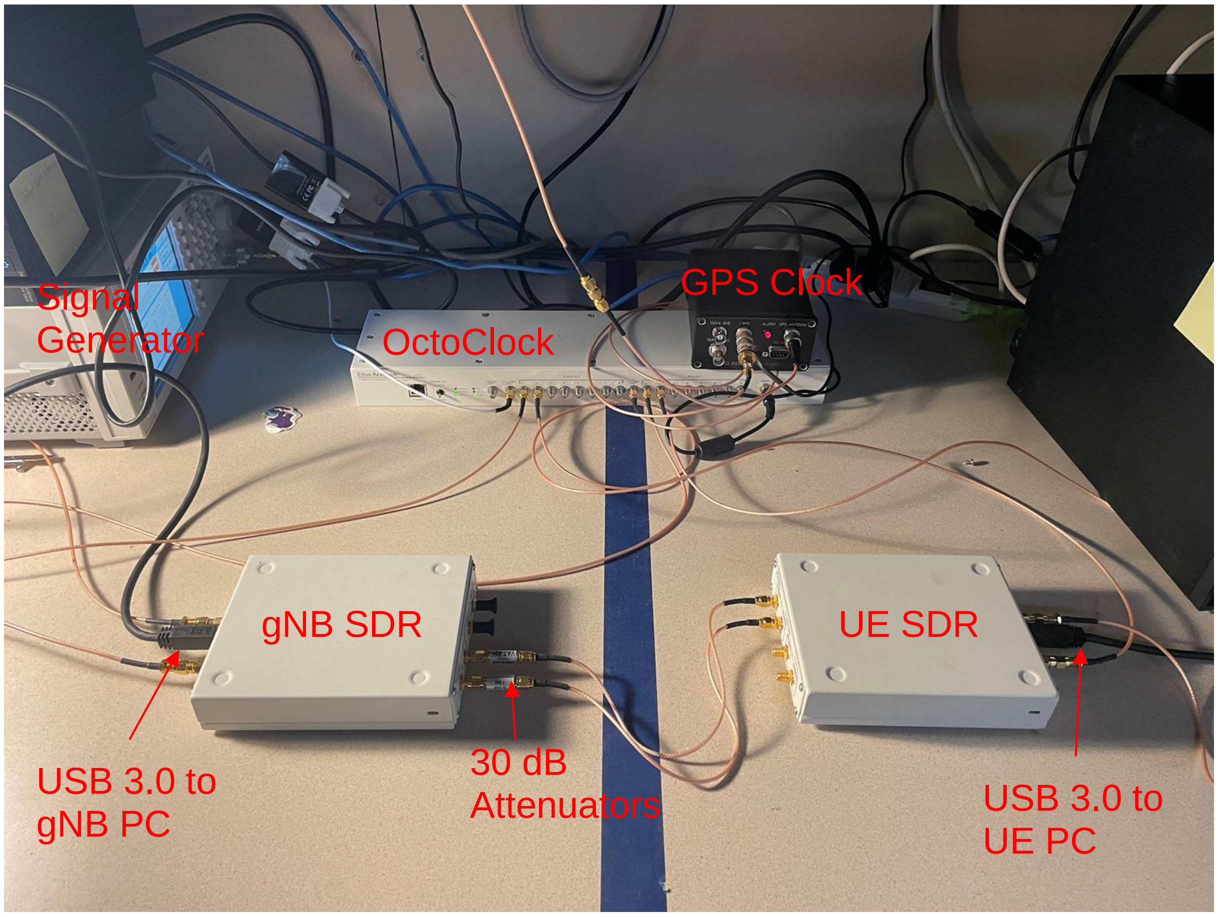


Figure 3.2.1 The test bench setup includes two B210 SDRs connected to the gNB PC and UE PC with USB 3.0 cables. The SDRs are connected to an OctoClock that supplies a PPS signal from the GPS clock and a 10MHz reference clock from the signal generator.

The Bulipu GPS Clock generator was inconsistent; the GPS lock would randomly disengage, so the team used an already established roof antenna with a Jackson Labs GPS clock to fix the issue. With everything ready, the team began testing and tried to connect the core network to the UE. Unfortunately, the team was not able to connect the two. After this disappointing discovery, the team pivoted to trying srsRAN, a different open-source software suggested by a graduate student.

To maintain momentum with OAI, the team was divided in two: one team working with OAI and another with srsRAN. Like OAI, the team attempted to get srsRAN functional and running on virtual machines. By following srsRAN’s documentation, the core network and gNB were successfully set up.

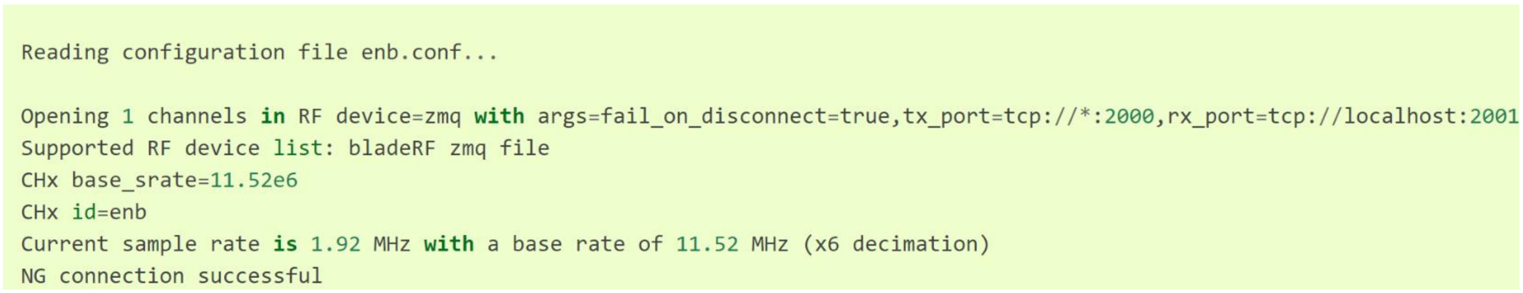
There were several steps taken to get the core network functional. The first step was downloading Open5GS, an open-source core network program. This software contained many of the same components that OAI’s core network had, such as NRF, AMF, SMF, AUSF, UPF, AUSF, UDM, UDR, and NSSF containers. After the containers were set up, WebUI was installed to add and edit subscriber information. Upon completing the installation, the configuration files of Open5GS were edited to match the use case of the team. The team then entered SIM card information into the WebUI by visiting http://localhost:3000 and logging into an admin account. The last step before testing was enabling IP forwarding and adding a route to the IP table so the UE could connect.

Once Open5GS was configured and ready to run, the gNB was the new focus. For simplification purposes, the team decided to implement the gNB using ZMQ virtual radios. Edits were made to the “enb.conf” file so that it would be able to find and connect to the core network. This was done by changing the MCC, MNC, and MME parameters to match that of the core network. To enable ZMQ, the “device_name” argument was changed to “zmq.” After changing these parameters, the LTE cells in the “rr.conf” file were commented out and the 5G NR cell was added from srsRAN’s documentation.

The Open5GS install allowed the core network functions to run automatically on startup. Once the core network was confirmed to be working, the gNB was started. Unfortunately, the first attempt proved to be unsuccessful. The first step taken to solve the issue was to double-check the instructions and the parameters that we needed to change. After doing so and still being unsuccessful, the team looked towards other resources and the srsRAN mailing list to further investigate what the issue was. After substantial research, the team decided to switch out the edited configuration files with example files from srsRAN. After switching these files out, the team saw immediate success.

Figure 3.2.2 The message displayed after successful connection of the gNB and core network. This was achieved by using the example configuration files provided by srsRAN [35].

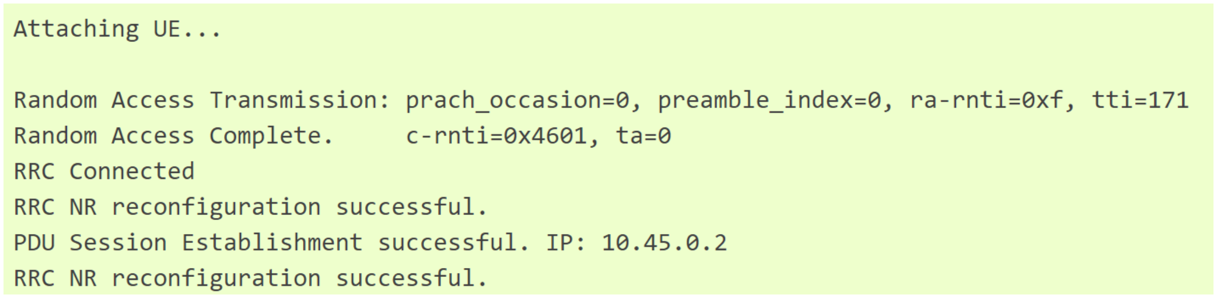
Once the core network and gNB were connected, the team shifted its focus to the UE. Similarly to the gNB, the UE was installed using a simple command and configured in a



```
Reading configuration file enb.conf...

Opening 1 channels in RF device=zmq with args=fail_on_disconnect=true,tx_port=tcp://*:2000,rx_port=tcp://localhost:2001
Supported RF device list: bladeRF zmq file
CHx base_srate=11.52e6
CHx id=enb
Current sample rate is 1.92 MHz with a base rate of 11.52 MHz (x6 decimation)
NG connection successful
```

separate file. ZMQ was enabled by changing the “device_name” argument, exactly as described for the gNB. The namespace was then set to “ue1,” as instructed by srsRAN’s documentation. Next, the LTE carriers were disabled, the NR bands were enabled, the release version of the application was set to 15, and the USIM credentials were set. Lastly, the network namespace was created.

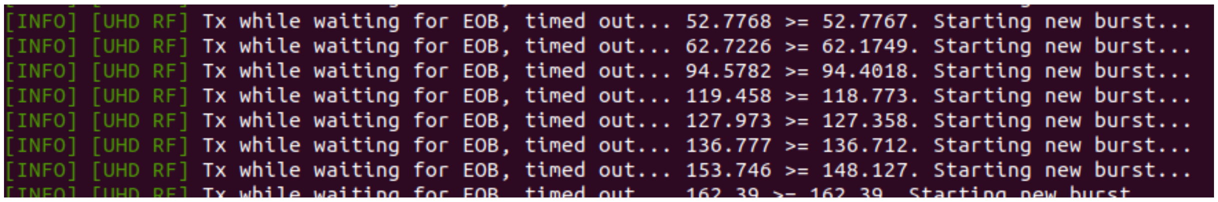
As with the gNB, the UE did not see success upon initial testing. Despite following the directions closely, the team was unable to get the expected output. The expected output depicted by srsRAN is shown below. After realizing that the UE was not connecting to the gNB, the team began searching for solutions to the issue. One of the first actions the team took, learning from the gNB experience, was switching out the “ue.conf” file entirely with the one from the srsRAN documentation. Unfortunately, this did not resolve the issue as it did previously. Troubleshooting continued by searching online for similar issues discovered by other users. Some relevant articles were found, but the changes that were suggested did not fix the error. Many articles suggested changing the gain values, which did not seem to have any impact. The installation process was repeated multiple times in the hope that a small detail was overlooked. YouTube tutorials were also followed diligently. After taking all of these steps, the team concluded that the configuration and setup were not the issues.



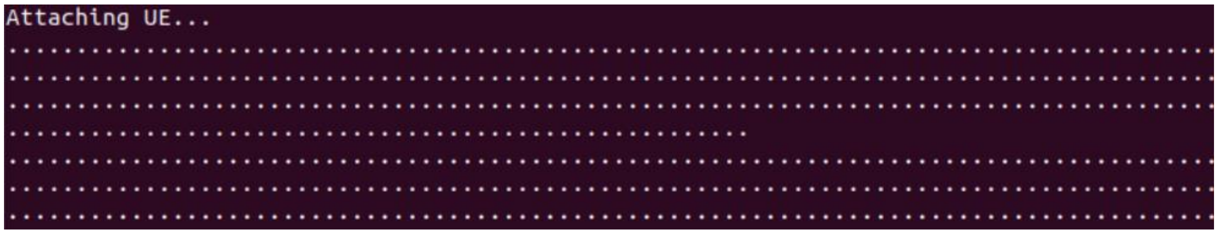
```
Attaching UE...

Random Access Transmission: prach_occasion=0, preamble_index=0, ra-rnti=0xf, tti=171
Random Access Complete.     c-rnti=0x4601, ta=0
RRC Connected
RRC NR reconfiguration successful.
PDU Session Establishment successful. IP: 10.45.0.2
RRC NR reconfiguration successful.
```

Figure 3.2.3 The correct output when the UE is attached to the core network and gNB. This output should appear just below the dialogue of the gNB connected to the core network [35].

Due to the lack of experience with the virtual radio and its minimal relevance to a fully functioning project, the team retried the connection using SDRs. The SDR used for both the UE and gNB was the Ettus Research USRP B210, which required USRP Hardware Drivers (UHD). These were then connected via coaxial cables with SMA connectors. A clean install of the core network, gNB, and UE was completed, and the ZMQ steps were replaced with arguments to specify the UHD device instead. After switching to the B210s, the team witnessed sporadic success. From time to time, the UE would fully connect to the gNB. During the times when the UE was not able to connect to the gNB, the gNB had an output similar to the one shown below in Figure 3.2.4. The console also printed "Attaching UE," followed by an increasing sequence of periods. After reading the log files from the UE and gNB, the team noticed a large number of late packet sequences. After further researching the cause of this online, the team discovered that the root cause of the issue was computing power.



```
[INFO] [UHD RF] Tx while waiting for EOB, timed out... 52.7768 >= 52.7767. Starting new burst...
[INFO] [UHD RF] Tx while waiting for EOB, timed out... 62.7226 >= 62.1749. Starting new burst...
[INFO] [UHD RF] Tx while waiting for EOB, timed out... 94.5782 >= 94.4018. Starting new burst...
[INFO] [UHD RF] Tx while waiting for EOB, timed out... 119.458 >= 118.773. Starting new burst...
[INFO] [UHD RF] Tx while waiting for EOB, timed out... 127.973 >= 127.358. Starting new burst...
[INFO] [UHD RF] Tx while waiting for EOB, timed out... 136.777 >= 136.712. Starting new burst...
[INFO] [UHD RF] Tx while waiting for EOB, timed out... 153.746 >= 148.127. Starting new burst...
[INFO] [UHD RF] Tx while waiting for EOB, timed out... 162.39 >= 162.39. Starting new burst
```



```
Attaching UE...
..........................................................................................
..........................................................................................
..........................................................................................
.......................................................
..........................................................................................
..........................................................................................
..........................................................................................
```

Figure 3.2.4 The terminal output when the UE failed to connect. Both of these outputs would continue to print out to the console until the program was shut down [36].

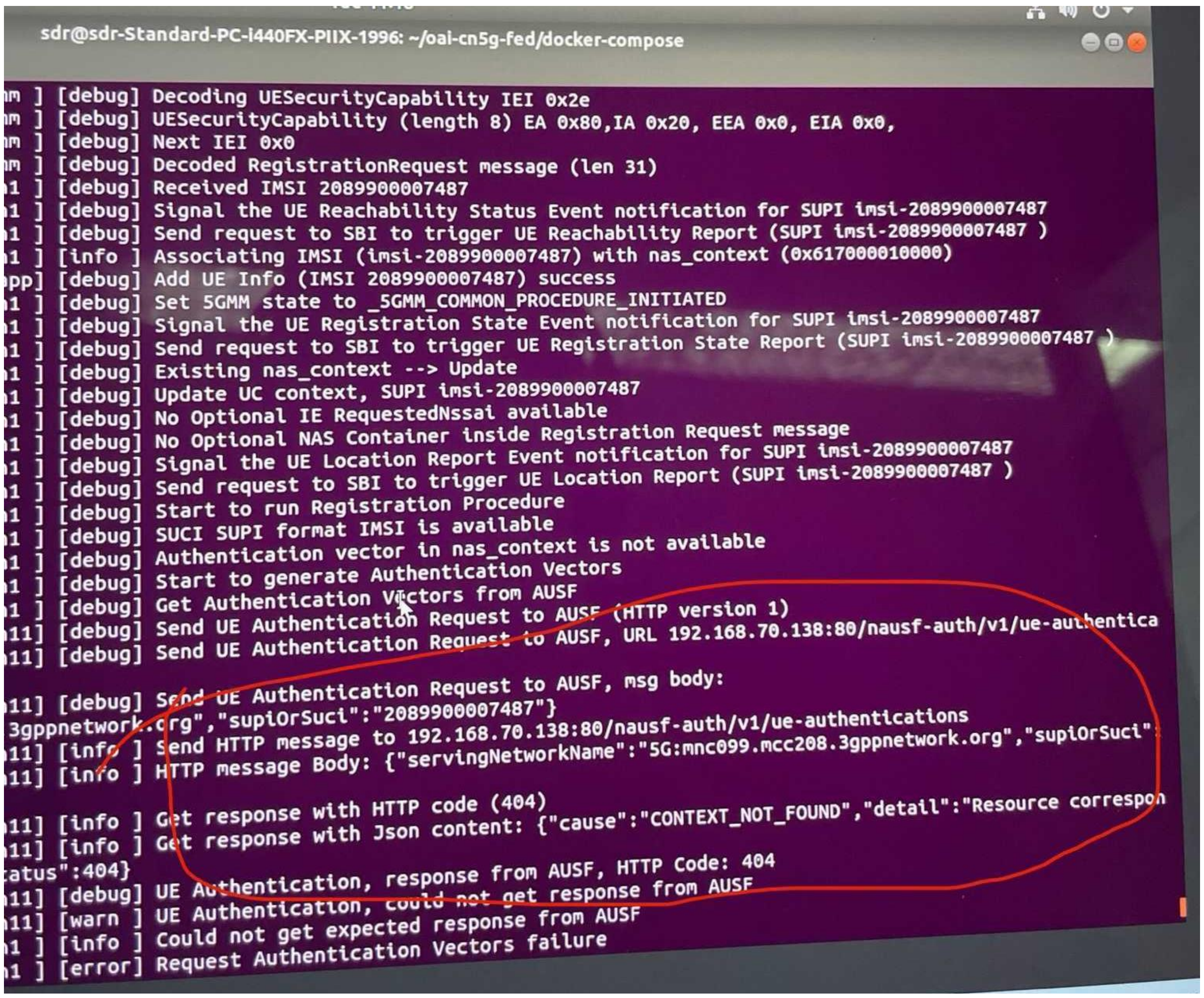
Once the team decided to focus on computing power, the team double-checked that the CPU governor was set to performance mode and that C states were disabled. When the issue persisted, the team reassessed the hardware in use. The team discovered that the virtual machines were only utilizing two CPU cores each. To address this issue, other virtual machines on the same computer were deleted or had some CPU cores removed. These cores were then added to the virtual machines for the core network, gNB, and UE. After making these changes, the core network and gNB had four CPU cores and the UE had two. The gNB and CN required more cores due to their greater computational requirements. While there was a decrease in the number of late packet sequences, there was still no consistent connection between the UE and gNB. After consulting with advisors and AFRL, the team decided to try to switch from virtual machines to “bare-metal” hardware. The team checked out laptops from WPI’s ATC department with quad-core 8th generation Intel Core i7 processors. The core network, gNB, and UE software were installed on the devices, and the configuration files were replaced by those provided by srsRAN. The team also made sure C-States were disabled and that the performance governor was enabled. Using B210s, the gNB and UE still would not connect. Adjusting the receiving and transmitting gains did not prove successful. With no success with srsRAN, the team decided to switch focus back to OAI on taking advantage of prior experience within the WiLab.

The two laptops, one running the OAI gNB and the other the OAI UE, were connected to separate B210s. The B210s were then connected with 30 dB attenuators. This setup was able to connect the core network to the gNB and the gNB to the UE by adjusting the Rx and Tx gain values in the UE configuration file. Once the gNB and UE were connected, another issue emerged; the hardware was once again not powerful enough to run the system. The log files were filled with “late” messages, meaning there was an overflow of information that the computer could not keep up with. The team ensured that the SDRs were operating on the correct channel using a spectrum analyzer and validated the signals by using an oscilloscope. Once both of those were determined to be correct, we believed it was once again a computing power issue. To resolve this, we switched from the laptops to more powerful desktops that were well above the recommended minimum CPU specifications. The OAI gNB and UE were then successfully installed on the desktops, and the CN was once again connected to the gNB. The gNB and UE were also able to connect with few or no late messages.

[3.3] Results

After implementing the techniques described above, we were unable to get OAI fully operational. Originally, we could not get the gNB connected to the core network. Fortunately, after enabling IP forwarding and adding the IP route to the table, we connected the two. The AMF logs on the core network indicated that the gNB was connected. Once that was connected, we worked to get the UE attached to the gNB. This process proved to be much more difficult. The UE and gNB were able to detect each other, but the UE was never fully registered with the core network.

Initially, the gNB detected the UE, but they did not connect. This issue was fixed by changing the Rx and Tx gain values systematically, allowing us to find the optimal values to enable a gNB to UE connection. The core network was put into debug mode to uncover more errors. The AMF displayed that the UE had initialized registration, indicated by "5GMM-REG-INITIATED". The SMF logs revealed that the core network could not authenticate the UE, and the subscriber information was reassessed and corrected to match the USIM and the SMF database.



```
sdr@sdr-Standard-PC-i440FX-PIIX-1996: ~/oai-cn5g-fed/docker-compose
m ] [debug] Decoding UESecurityCapability IEI 0x2e
m ] [debug] UESecurityCapability (length 8) EA 0x80,IA 0x20, EEA 0x0, EIA 0x0,
m ] [debug] Next IEI 0x0
m ] [debug] Decoded RegistrationRequest message (len 31)
1 ] [debug] Received IMSI 208990007487
1 ] [debug] Signal the UE Reachability Status Event notification for SUPI imsi-208990007487
1 ] [debug] Send request to SBI to trigger UE Reachability Report (SUPI imsi-208990007487 )
1 ] [info ] Associating IMSI (imsi-208990007487) with nas_context (0x617000010000)
pp] [debug] Add UE Info (IMSI 208990007487) success
1 ] [debug] Set 5GMM state to _5GMM_COMMON_PROCEDURE_INITIATED
1 ] [debug] Signal the UE Registration State Event notification for SUPI imsi-208990007487
1 ] [debug] Send request to SBI to trigger UE Registration State Report (SUPI imsi-208990007487 )
1 ] [debug] Existing nas_context --> Update
1 ] [debug] Update UC context, SUPI imsi-208990007487
1 ] [debug] No Optional IE RequestedNssai available
1 ] [debug] No Optional NAS Container inside Registration Request message
1 ] [debug] Signal the UE Location Report Event notification for SUPI imsi-208990007487
1 ] [debug] Send request to SBI to trigger UE Location Report (SUPI imsi-208990007487 )
1 ] [debug] Start to run Registration Procedure
1 ] [debug] SUCI SUPI format IMSI is available
1 ] [debug] Authentication vector in nas_context is not available
1 ] [debug] Start to generate Authentication Vectors
1 ] [debug] Get Authentication Vectors from AUSF
11] [debug] Send UE Authentication Request to AUSF (HTTP version 1)
11] [debug] Send UE Authentication Request to AUSF, URL 192.168.70.138:80/nausf-auth/v1/ue-authentica

11] [debug] Send UE Authentication Request to AUSF, msg body:
3gppnetwork.org","supiOrSuci":"208990007487"}
11] [info ] Send HTTP message to 192.168.70.138:80/nausf-auth/v1/ue-authentications
11] [info ] HTTP message Body: {"servingNetworkName":"5G:mnc099.mcc208.3gppnetwork.org","supiOrSuci":

11] [info ] Get response with HTTP code (404)
11] [info ] Get response with Json content: {"cause":"CONTEXT_NOT_FOUND","detail":"Resource correspon
atus":404}
11] [debug] UE Authentication, response from AUSF, HTTP Code: 404
11] [warn ] UE Authentication, could not get response from AUSF
1 ] [info ] Could not get expected response from AUSF
1 ] [error] Request Authentication Vectors failure
```

Figure 3.3.1 SMF logs displaying an authentication error with the AUSF. This was resolved by reviewing the OPC and key, two important identifiers for connecting a UE to a network.

This progressed the connection to "5GMM - REGISTERED". This was one step before a proper connection and receiving an IP address. The final output should have read "5GMM-Connected," and the core network would have then assigned an IP address. The team struggled with finding solutions and even went so far as to attempt an identical install as the previous year's MQP team. After discovering missing components in the year-old commits, the team decided to pivot to the new project.

[3.4] Summary

The initial goal for the team was to get a 5G network operational and create a spoofing attack that would expose 5G vulnerabilities. In order to do so, the team first attempted to use OAI to create a 5G network. Unfortunately, the team ran into a variety of issues that prevented OAI from becoming functional. For example, the initial computers that were used lacked the AVX2 instruction set as described above. Vague and undescriptive errors elongated the time it took for the team to discover the cause of the issue. After switching back to laptops, the team was still unable to get the UE and the network to communicate. Explored in parallel with OAI, srsRAN did not provide results either. Despite following the documentation provided by srsRAN exactly, the team was unsuccessful with both ZMQ and physical SDRs. After this, the team realized that computing power might be the issue. However, increasing the number of CPU cores on the virtual machines did not make much of a difference. Finally, the team switched to physical laptops and desktops. The main reason for this switch was that virtual machines often cause issues that are difficult to debug and add performance overhead, as described by members of AFRL.

For OAI, the team was able to get a UE registered and connected to the gNB, but did not receive an IP address from the core network. This prevented the UE from actually being able to communicate with the network. For srsRAN, the team was unable to get the UE to attach to the gNB consistently. Despite rarely connecting, srsRAN was not reliable nor operational. The team was unable to get either software working due to a lack of updated documentation and resources. As a result, the team switched gears to a different focus: a spectrum sensing analysis network.

[4] Proposed Spectrum Sensing Analysis Network

[4.1] Problem Statement: Aggregating data into one place

The goal of the new project was to develop code for a scalable network of hardware nodes to collect data about the 5G spectrum and make it available for predictive analysis. As a brief overview, a sensor node collects data using a software-defined radio (SDR) and then sends that data to a server node for analysis. The server node could also be responsible for sending commands to the sensor nodes to change their behavior.

The sensor node is capable of continuously receiving data from specified frequencies and bandwidths using the UHD library. The data is then segmented into time chunks of 1 ms as that is the length of one radio subframe. The analysis will be performed on the data to reduce the amount of data to need to be transported. The final operation of the sensor node is to package data for transport and send it to the server node. The data is encoded into a series of bytes so that the sensor nodes can efficiently send data.

The server node then receives the packaged data continuously from all active sensor nodes and decodes the data. The decoded data is then stored and time stamped so that it may be analyzed and combined between nodes later.

[4.2] Proposed Solutions

The project had two primary milestones, with the first comprising the bulk of the work. For the first milestone, there needed to be a single sensor node communicating with a local server. With this simplified path, the team could focus on properly acquiring data and transmitting it with few errors. The data, 5G Band 13, had to be divided into single radio subframes and converted to the frequency domain. For all cases, a radio subframe has a length of 1 millisecond but is composed of multiple slots, which vary based on the numerology [40]. On the transportation layer, the server had to be correctly connected to the client and ensure that the data sent between them had no errors. To reduce the network bandwidth and server storage requirements, the frequency data were downsampled. Lastly, the server only needed to store packets from a single client, allowing sensor node identifiers to be momentarily deprioritized.

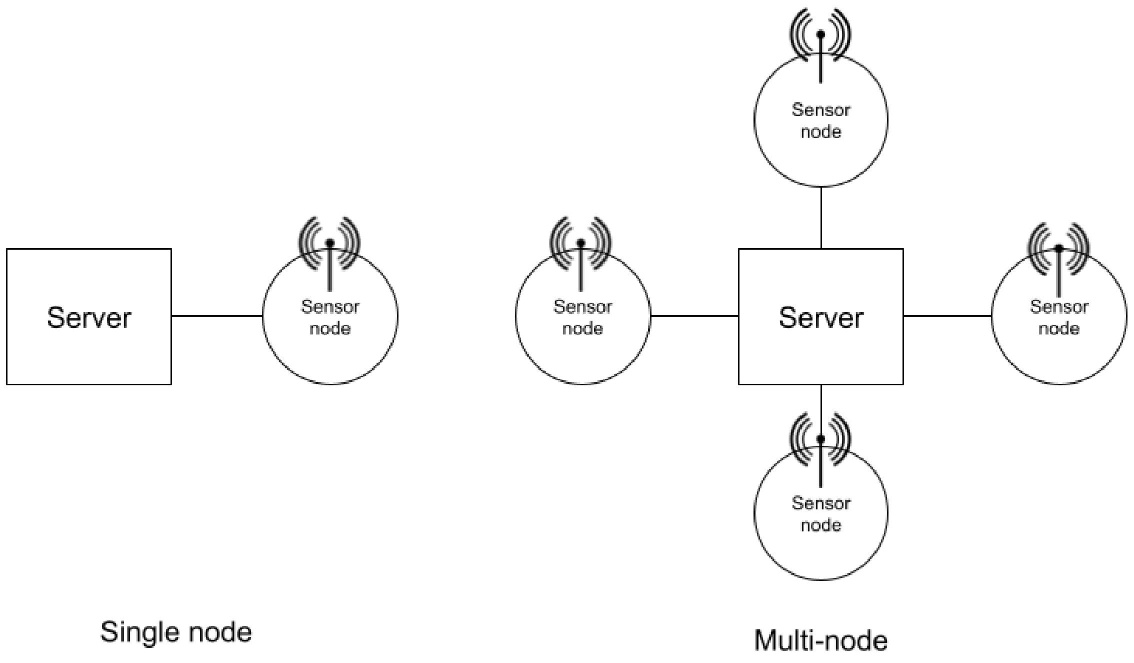


Figure 4.2: Block diagram for single and multi node spectrum sensing

The second milestone was to expand this system to multiple sensor nodes, all sending information to the same server. After receiving samples from multiple sensor nodes, the server will be able to aggregate the data to analyze the 5G channel. Average power will be sent to prove this concept, but this could theoretically be expanded to other communication characteristics. Combining data from multiple sources requires that the samples must be taken at the same time, thus requiring sent data to be timestamped. To reduce the scope of the project, the team will not be trying to combine any data. Rather, the team will just send and store data on a server.

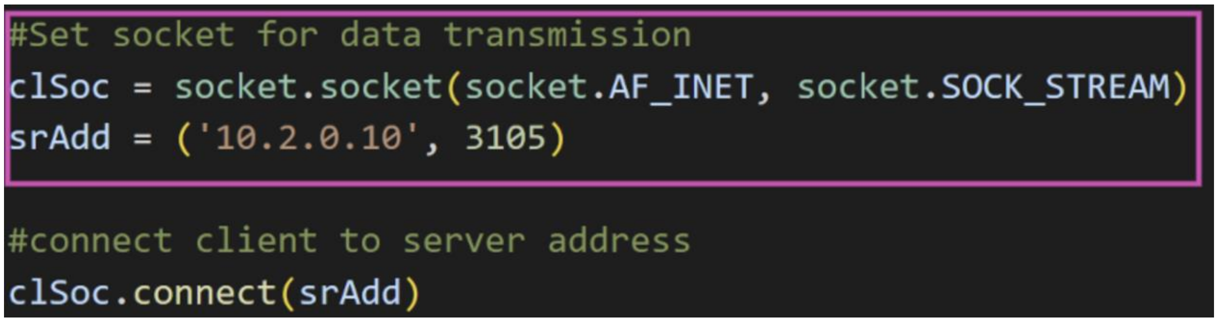
[4.3] Implementation

The project required specific hardware in increasing quantities as the solution expanded to scale. For the initial test, an Ettus Research USRP X310 was connected to a high-performance server via high-speed ethernet. To avoid confounding variables, the system remained on a shelf in WiLab. Increasing the number of nodes required additional PCs. One Ubuntu 20.04, 7th Generation Intel Core i7 equipped desktop computer was deployed with an accompanying Ettus Research USRP B210 connected via USB 3.0.

The software level consisted of two primary Python programs, one for the sensor side, also referred to as the client side, and one for the server side. Connecting the two nodes was the “socket” Python module, which enabled the creation of a Berkeley socket. A Berkeley Software Distribution (BSD) socket is an API developed at the University of California, Berkeley, in the 1980s and later incorporated into TCP/IP standards [37]. These sockets allow both the server and client to send and receive information [38]. The sensor side's primary function is data acquisition from an SDR, which it then sends via socket. For data acquisition, the team utilized the most configurable method featured on the PySDR website to tune the radio to the targeted 5G channel [39]. For this project, Band 13 was chosen. This Frequency Division Duplex (FDD) band uses the 751 MHz channel for downlink and the 782 MHz channel for uplink with a 10 MHz bandwidth. According to the 5G specification, each radio subframe is 1 ms long [40]. After being sampled, these downlink packets were stored in a

NumPy array using the maximum SDR buffer size. They were then sent continuously to the server storage program. The two python scripts used ChatGPT as a reference when building portions of the code [41].

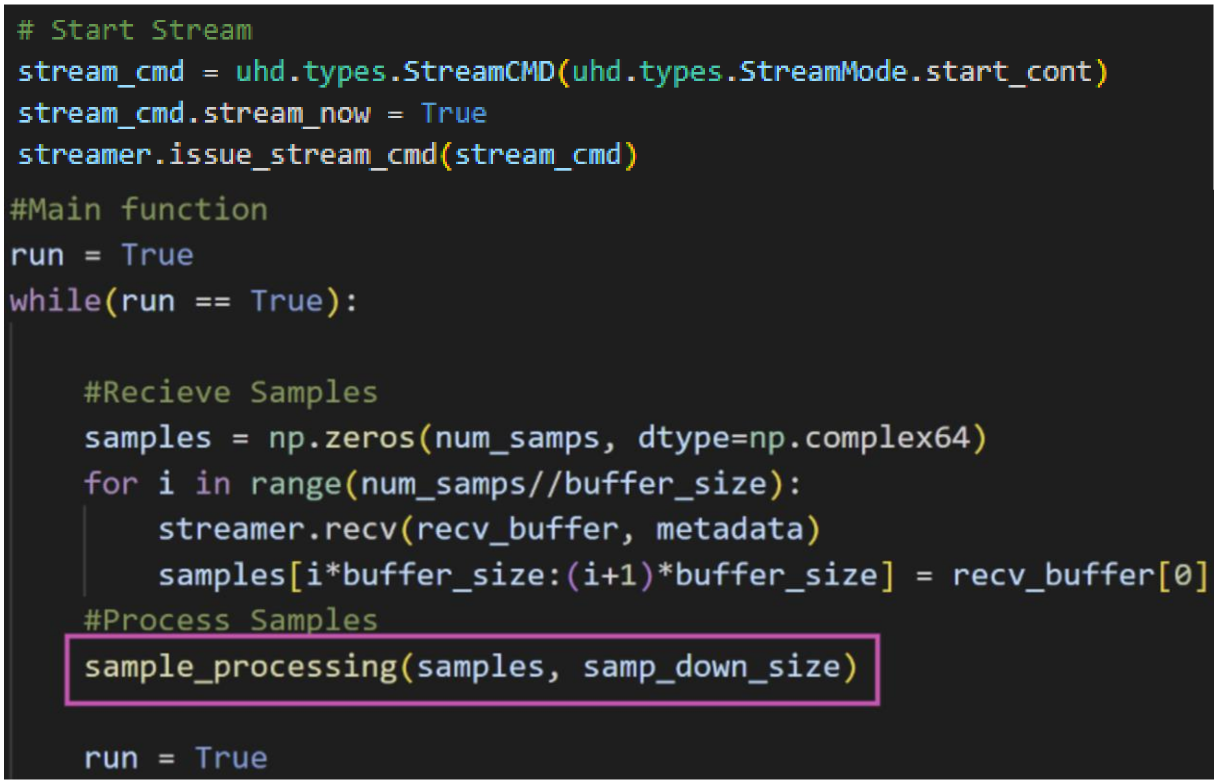
The sensor code uses a simple yet effective implementation of socket programming in order to send the data to the server. The first step in this process was to create the client socket and connect to the server address. This step can be seen below:



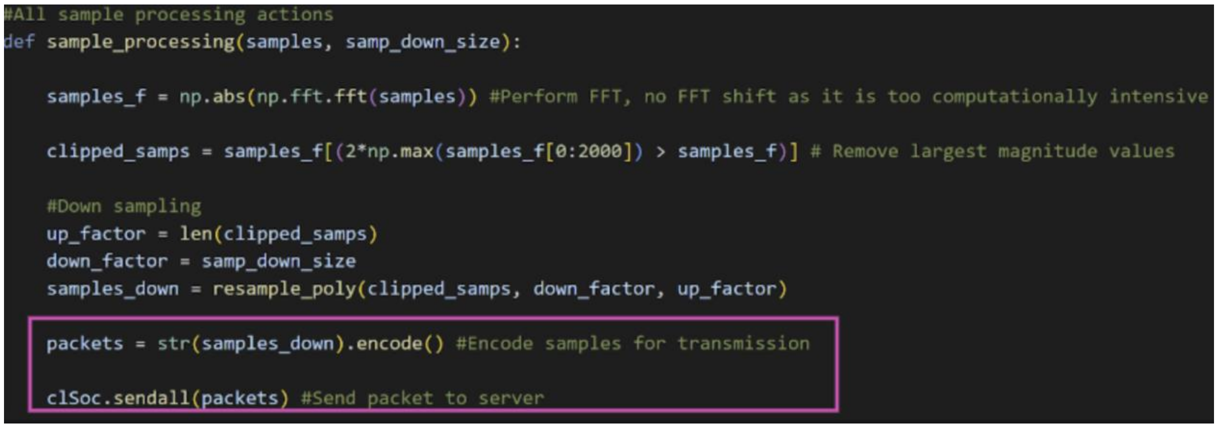
```
#Set socket for data transmission
clSoc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srAdd = ('10.2.0.10', 3105)

#connect client to server address
clSoc.connect(srAdd)
```

Figure 4.3.1 The client socket connecting to the server. This is an essential step for sending the data.

The next step in the process was sending the data in a continuous fashion. It was important to ensure that data was being received by the client and sent to the server in real-time. In order to accomplish this, an endless while loop was used that continuously collected data and sent it to the server. Initially, the team erroneously placed the commands to start and stop the data stream inside this while loop, causing unnecessary overruns. This corrected portion of the code can be seen below:



```
# Start Stream
stream_cmd = uhd.types.StreamCMD(uhd.types.StreamMode.start_cont)
stream_cmd.stream_now = True
streamer.issue_stream_cmd(stream_cmd)
#Main function
run = True
while(run == True):

    #Recieve Samples
    samples = np.zeros(num_samps, dtype=np.complex64)
    for i in range(num_samps//buffer_size):
        streamer.recv(recv_buffer, metadata)
        samples[i*buffer_size:(i+1)*buffer_size] = recv_buffer[0]
    #Process Samples
    sample_processing(samples, samp_down_size)

    run = True
```

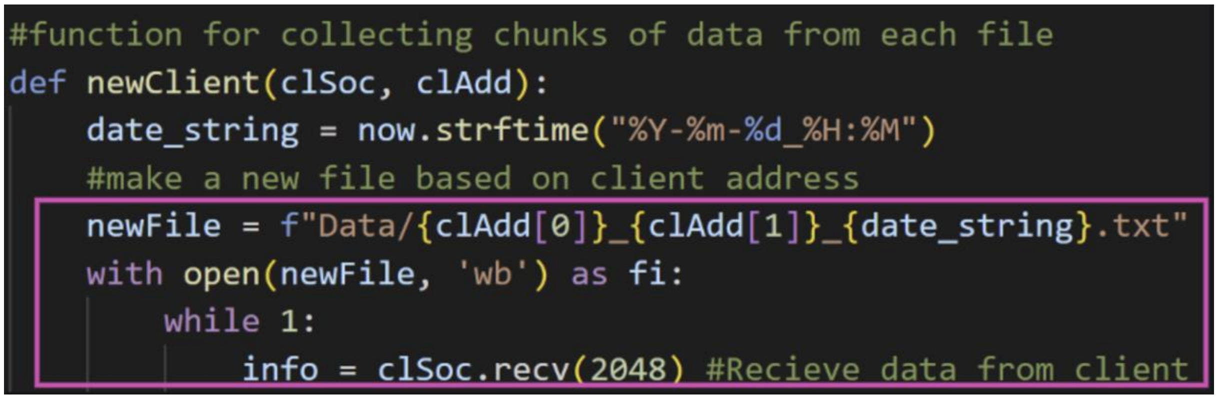


```
#All sample processing actions
def sample_processing(samples, samp_down_size):

    samples_f = np.abs(np.fft.fft(samples)) #Perform FFT, no FFT shift as it is too computationally intensive

    clipped_samps = samples_f[(2*np.max(samples_f[0:2000]) > samples_f)] # Remove largest magnitude values

    #Down sampling
    up_factor = len(clipped_samps)
    down_factor = samp_down_size
    samples_down = resample_poly(clipped_samps, down_factor, up_factor)

    packets = str(samples_down).encode() #Encode samples for transmission

    clSoc.sendall(packets) #Send packet to server
```

Figure 4.3.2 The data is streamed and encoded as a byte string. The first boxed line shows the function being called and the second demonstrates data being sent to the server.

An important aspect of the code above is the encoding aspect. Encoding the information as a byte string allows the information to be accurately sent to the server. As depicted above, the code streams the data to the server in real-time until manually stopped.

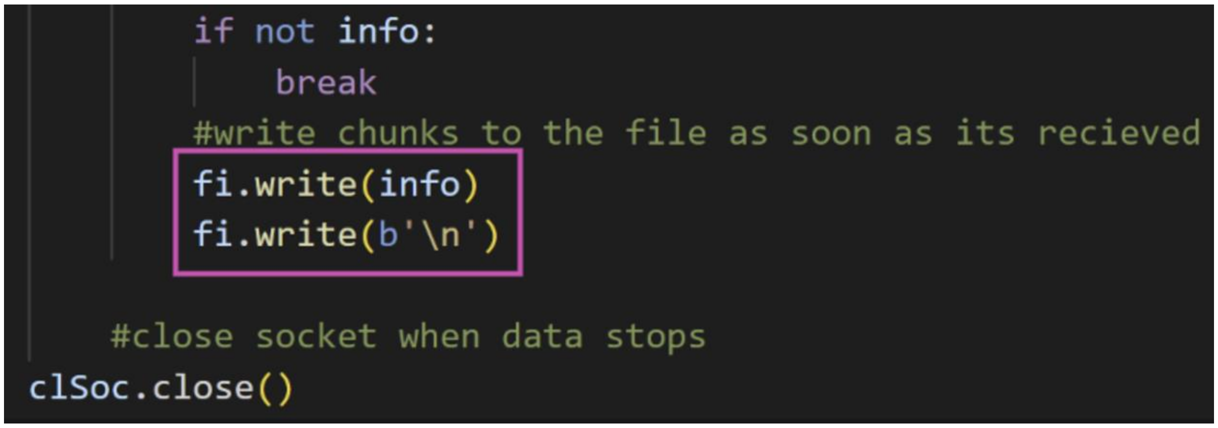
The server code is slightly more complex than the sensor socket code because it needs to support multiple clients. The server code consists of one main function called “newClient.” The function takes in a client socket and a client address. The first action the function takes is making a file based on the address of the client. The file is then opened, and data is received.



```
#function for collecting chunks of data from each file
def newClient(clSoc, clAdd):
    date_string = now.strftime("%Y-%m-%d_%H:%M")
    #make a new file based on client address
    newFile = f"Data/{clAdd[0]}_{clAdd[1]}_{date_string}.txt"
    with open(newFile, 'wb') as fi:
        while 1:
            info = clSoc.recv(2048) #Recieve data from client
```

Figure 4.3.4 Function for each new client that creates a file to write to based on their address.

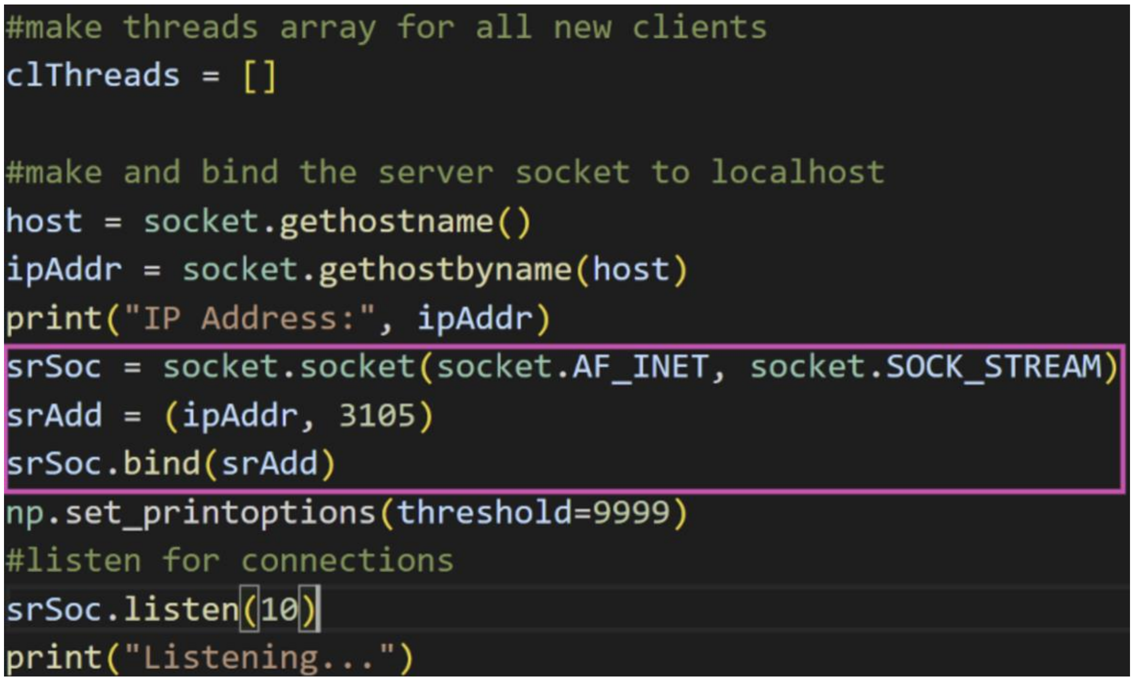
The data received from the client is then written to the file that was created and opened. If the data inflow ever stops, the code then breaks out of the loop and the client socket is finally closed.



```
        if not info:
            break
        #write chunks to the file as soon as its recieved
        fi.write(info)
        fi.write(b'\n')

    #close socket when data stops
clSoc.close()
```

Figure 4.3.5 The server writes the data to the file. Once data is no longer received, the loop breaks and the socket is closed.

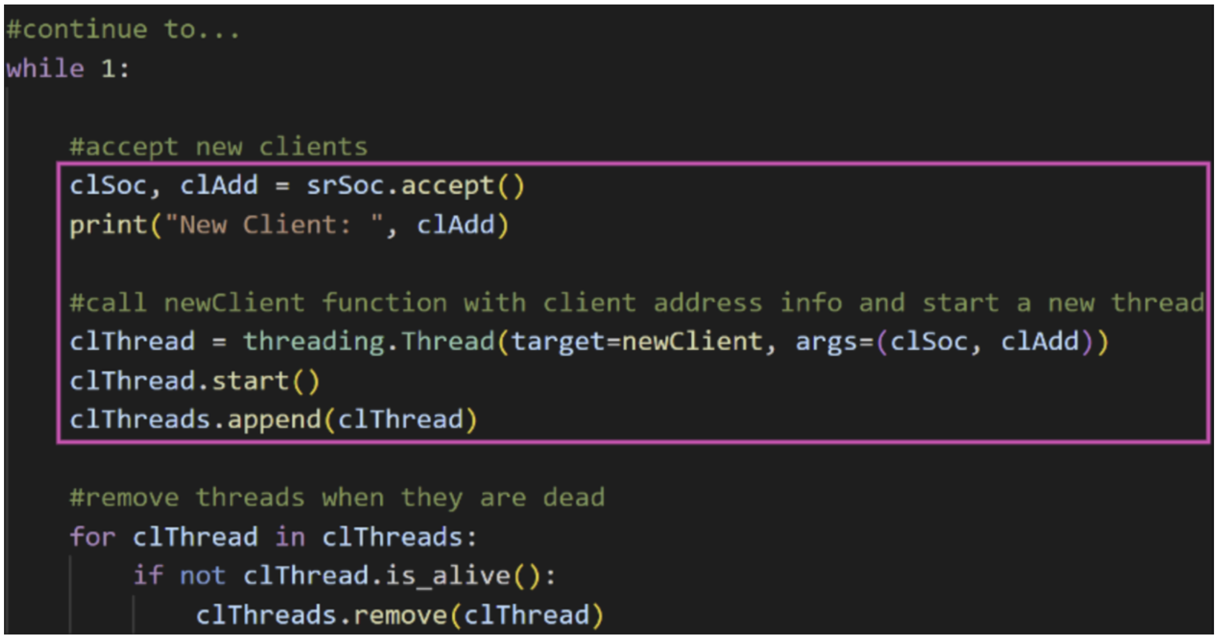
Before the function for the new clients could be effectively used, the server itself had to have been set up. Similar to the client code, the socket was first created and bound to an address. The server was configured to listen for up to ten connections, and once everything was ready, “Listening” was printed out to the console as an indication that clients could start connecting.



```
#make threads array for all new clients
clThreads = []

#make and bind the server socket to localhost
host = socket.gethostname()
ipAddr = socket.gethostbyname(host)
print("IP Address:", ipAddr)
srSoc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srAdd = (ipAddr, 3105)
srSoc.bind(srAdd)
np.set_printoptions(threshold=9999)
#listen for connections
srSoc.listen(10)
print("Listening...")
```

Figure 4.3.6 The server creates a socket and binds it to an address. Up to ten clients or sensors are currently supported by the server.

Once the server socket was set up and ready for clients, thread support had to be implemented. First, an infinite while loop was introduced so the server could continuously search for new clients. When a new client does try to connect, the server accepts the connection and prints the address of the client to the console. This is helpful because it allows the team to study how many clients are connecting and watch their output files in real-time. Finally, threading is started using the newClient function. The client socket and client address are passed into the function as described above. The thread is started and appended to the list of threads created in the prior figure. Lastly, at the end of the while loop, the code goes through each thread in the list of threads, checking if it is still alive. If the thread is not alive, it is terminated.

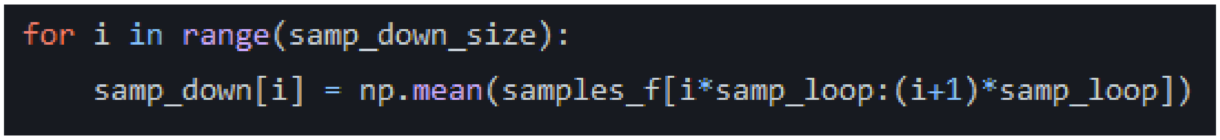


```
#continue to...
while 1:

    #accept new clients
    clSoc, clAdd = srSoc.accept()
    print("New Client: ", clAdd)

    #call newClient function with client address info and start a new thread
    clThread = threading.Thread(target=newClient, args=(clSoc, clAdd))
    clThread.start()
    clThreads.append(clThread)

    #remove threads when they are dead
    for clThread in clThreads:
        if not clThread.is_alive():
            clThreads.remove(clThread)
```

Figure 4.3.7 The server looks and connects to new clients continuously. Each client gets its own thread to run the "newClient" function on.

The team encountered various obstacles when constructing the code above. For example, sending data between sockets requires encoding to a byte string. The initial tests revealed truncated arrays stored in the server. Initially, the problem was misdiagnosed as a problem with the encoder, resulting in the testing of several alternatives. The team first tried "Pickle," a module that converts Python objects to a byte stream. This introduced performance issues, resulting in an overrun on the sensor node and truncated data after unpickling on the server. Additional solutions were explored, with JSON being unsuitable due to its inability to send complex data and MessagePack having issues with the default numpy array data type. It was later noticed that printing the output of the samples array on the sensor node itself resulted in a visually truncated array in the console. Despite the visual appearance, the length of the array was still the correct value. Exploring all our options, "np.set_printoptions(threshold=9999)" was employed to remove any visual truncation. This ultimately solved our problem, correctly displaying the entire array on the server.

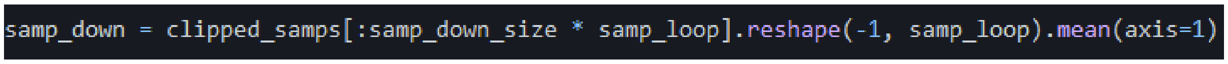
The team worked to implement a more efficient way to process the signal before sending it to the server node. The most computationally intensive piece of code was the downsampling portion, the final code for downsampling can be seen in figure 4.3.2. Initially, an inefficient for loop was used for downsampling.



```
for i in range(samp_down_size):
    samp_down[i] = np.mean(samples_f[i*samp_loop:(i+1)*samp_loop])
```

Figure 4.3.8 The initial method used to downsample frequency domain samples. This method was found to be inefficient.

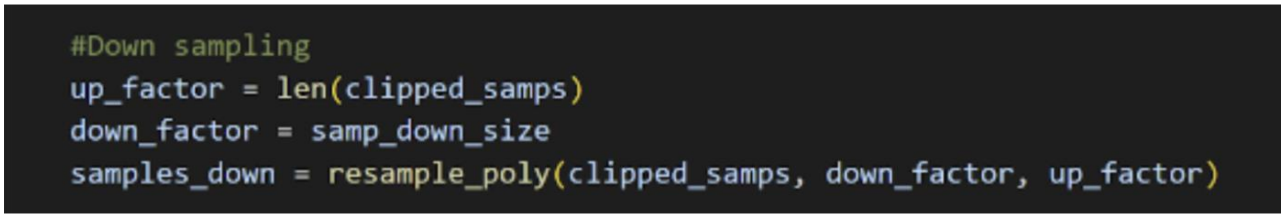
The next method the team used was the reshape function from the NumPy library. This method allowed us to manipulate the array so that a for loop was unnecessary for downsampling.



```
samp_down = clipped_samps[:samp_down_size * samp_loop].reshape(-1, samp_loop).mean(axis=1)
```

Figure 4.3.9 The next method used to downsample frequency domain samples. This method was also found to be inefficient.

The team's final method to downsample was using the resample_poly command supplied by the SciPy library. This command upsamples clipped_samps by the up_factor, a zero-phase low-pass FIR filter is applied, and then it is downsampled by the down_factor. This was the most efficient method the team could find, but was still too inefficient for its intended purpose.



```
#Down sampling
up_factor = len(clipped_samps)
down_factor = samp_down_size
samples_down = resample_poly(clipped_samps, down_factor, up_factor)
```

Figure 4.3.10 The final method used to downsample frequency domain samples. This method was found to be the most efficient.

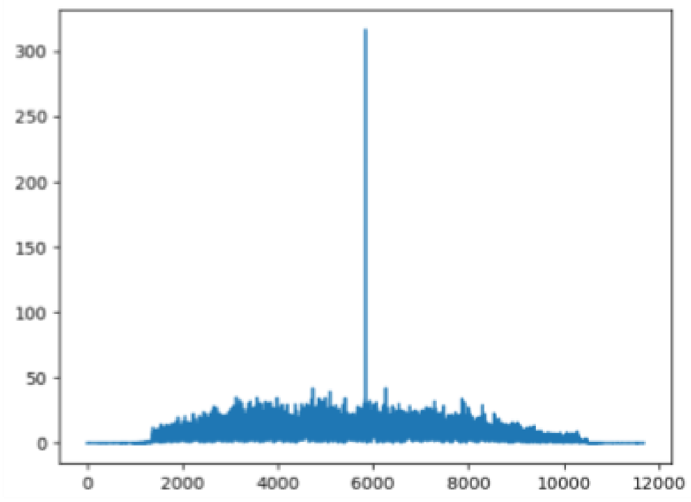
The team also found that the fftshift command used during signal processing was also computationally intensive. Fftshift is a command that shifts the zero frequency component to the center of the array. To reduce overruns, this operation was removed and could be run in the future to shift the array.

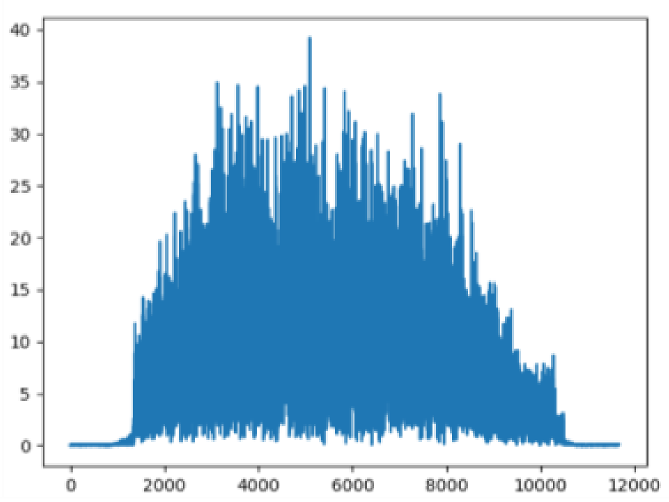
Last to be implemented was the ability to increase the number of attached nodes. To determine the source of each data collection, the team settled on using each sensor node's IP address. This provided information on which computer was sending data, assisting in troubleshooting and data verification.

[4.4] Results

The fundamental goals of the projects were achieved. Each sensor node is able to capture one millisecond of data, the time length for a single resource block on 5G Band 13.

This time domain data was then converted to the frequency domain using the NumPy FFT function. The progression of the frequency data can be seen in Figure 4.4. From left to right, the frequency data was captured, then refined by removing the spike at the center frequency. To reduce storage and network bandwidth bottlenecks, the array was then downsampled by repeatedly taking averages of approximately 20 data point chunks. Unfortunately, this aspect of the program resulted in overruns, as discussed later.


1. FFT of 1ms of samples


2. FFT with largest spikes removed

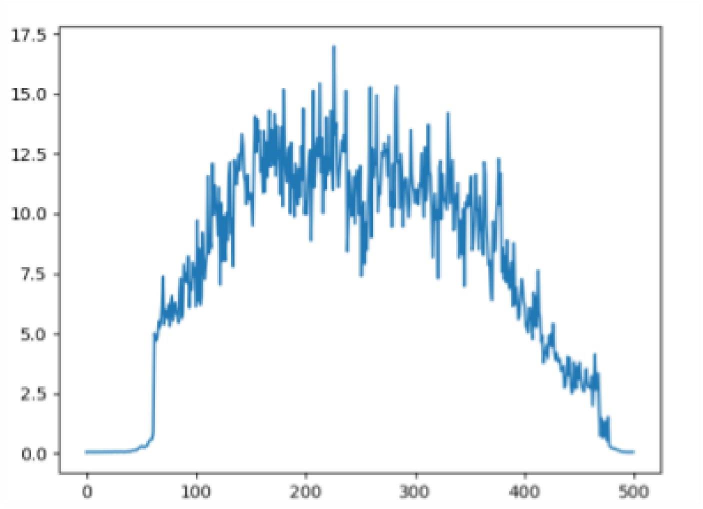

3. FFT with downsampling

Figure 4.4 Progression of signal processing. (1) The program begins by taking an FFT of the 1 ms of samples, the previously defined length of a resource block. (2) It then continues by removing the unnecessary middle spikes. (3) The final step takes several samples, approximately 20 at a time, and creates an array with those averages.

Each node can then continuously send data to a storage program via a BSD socket. This process was executed with an SDR locally and from a different computer simultaneously, thus accomplishing the multiple node goal. Each node is identified on the server by utilizing the node’s IP address, with a port number and time stamp added to discern different sessions.

While the core functionality is largely operational, there are a few ways the team fell short. The most notable is a performance issue. On each run of the continuous while loop, a for loop is required to downsample the frequency array. Omitting this step results in only occasional overruns on the SDR, but the added repeated processing of the downsampling loop causes constant overruns on both B210 and X310 systems. This could be resolved with a more appropriate process or algorithm. Another aspect of the project that is partially working is a couple of lines used to rework the array. Without it, the array is converted to a string with multiple columns and opening and closing brackets, making conversion back to a NumPy array more difficult. This was removed to reduce processing on the node side to improve the number of overruns.

Other goals were not reached. Ideally, each sample should be accompanied by a timestamp so the data can be properly combined on the server. Should this add too much overhead, the time could be recorded every several samples. This was deprioritized to improve the overrun problem. Additionally, the team did not create server control of the nodes. Ideally, this would include the ability to start and stop the sensor and to set the sampling and center frequencies. Other options could be added, like how much the array would be downsampled.

[4.5] Chapter Summary

The new goal of the project was to develop code for a scalable network of hardware nodes to collect data about the 5G spectrum and make it available for predictive analysis. The server node is able to accept multiple nodes at once and receive data simultaneously through these nodes. The sensor node acquires data constantly on a specified frequency and bandwidth, converts the data to the frequency domain, downsamples the data, and then sends the downsampled data to the server.

The system was tested using a two-node setup, one on the same device as the server and the other on a different device, both connected to the server through socket programming. Both devices were able to connect successfully to the server and send data to the server. The major limitation of the system is that the signal processing on the sensor node is too computationally intensive, which causes overflows on the node. This overflow causes the data stream to be interrupted and frames to be cut short. The team worked on optimizing the signal processing to try and reduce the number of overflows.

[5] Conclusion & Future Work

[5.1] Conclusion

The project's initial aim was to demonstrate security flaws in 5G base stations by utilizing OAI and srsRAN. The team was able to successfully install OAI gNB and UE, as well as have a CN running on a dedicated server. By the end of this period, the team had managed to link the UE and gNB, but the UE would not fully register with the CN. After browsing through OAI's mailing list and the database file and applying possible fixes, the UE was able to register to the CN but without an IP address. Without the IP address, the team could not continue the project. The team also attempted to use srsRAN. This time the CN would connect to the gNB, but the gNB could not connect to the UE. The team tends to think more computing power would have potentially fixed the issues with srsRAN.

For C-term, the team decided to alter the goal to utilize multiple hardware nodes and an SDR to collect data about the 5G spectrum over a server for predictive analysis. To expand the number of nodes, four Ubuntu 20.04 desktop computers with the USRP B210 SDR were deployed. Additionally, the team used two Python programs, one for the sensor side and the other for the client side.

[5.2] Future Research

Based on the experiences and findings from this project, there are several potential areas of future research and paths of improvement that could be explored. For the open source 5G base station, one could explore different combinations of software. While the Open5GS core network seemed fine on its own, perhaps it would have created a functional system if coupled with OpenAirInterface’s gNB platform. Additionally, exploring the original intent of the project with an already functional open-source 5G network could prove valuable. The continuous data aggregation shown by the team later in the project could be expanded in a few key ways. Firstly, the system could be controlled and tuned from the server. This could allow a user to specify the targeted signal on every remote node from a single interface. Next, improving the efficiency of the sensor node would be helpful in reducing overruns and maintain a more consistent data flow.

APPENDIX A: OpenAirInterface Tutorial

To get to the point we achieved, a precise set of steps were followed consistently. The first step was to image three computers to certain versions of ubuntu. For the gNB and UE, Ubuntu 16.04.07 was used, whereas the core network utilized 18.04. Once the computers were imaged, steps were taken to get the devices prepared to run OAI. The first step taken was to install a low-latency kernel, as described by OAI's documentation. In order to do so, the following command was run:

```
sudo apt-get install linux-image-lowlatency linux-headers-lowlatency
```

After this, the BIOS was accessed on each computer. In the BIOS, hyperthreading, Intel SpeedStep, and C-State were all disabled. P-State and C-State also needed to be disabled in Linux. In order to do so, the grub file was accessed using the following command:

```
sudo nano /etc/default/grub
```

In the grub file, the "GRUB_CMDLINE_LINUX_DEFAULT" line was edited to become:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet intel_pstate=disable"
```

After this line was edited, the grub was updated:

```
sudo update-grub
```

Next, the following command was run and the file was edited in accordance with OAI's documentation:

```
sudo nano /etc/modprobe.d/blacklist.conf
```

The following line was added to the end of that file:

```
blacklist intel_powerclamp
```

Next in the power management process was disabling CPU scaling. Cpufrequtils was installed and edited to put the governor in performance mode by using the commands below:

```
sudo apt-get install cpufrequtils
```

```
sudo nano /etc/default/cpufrequtils
```

This line was added:

```
GOVERNOR="performance"
```

Finally, ondemand daemon was disabled as instructed by OAI’s documentation:

```
sudo update-rc.d ondemand disable
```

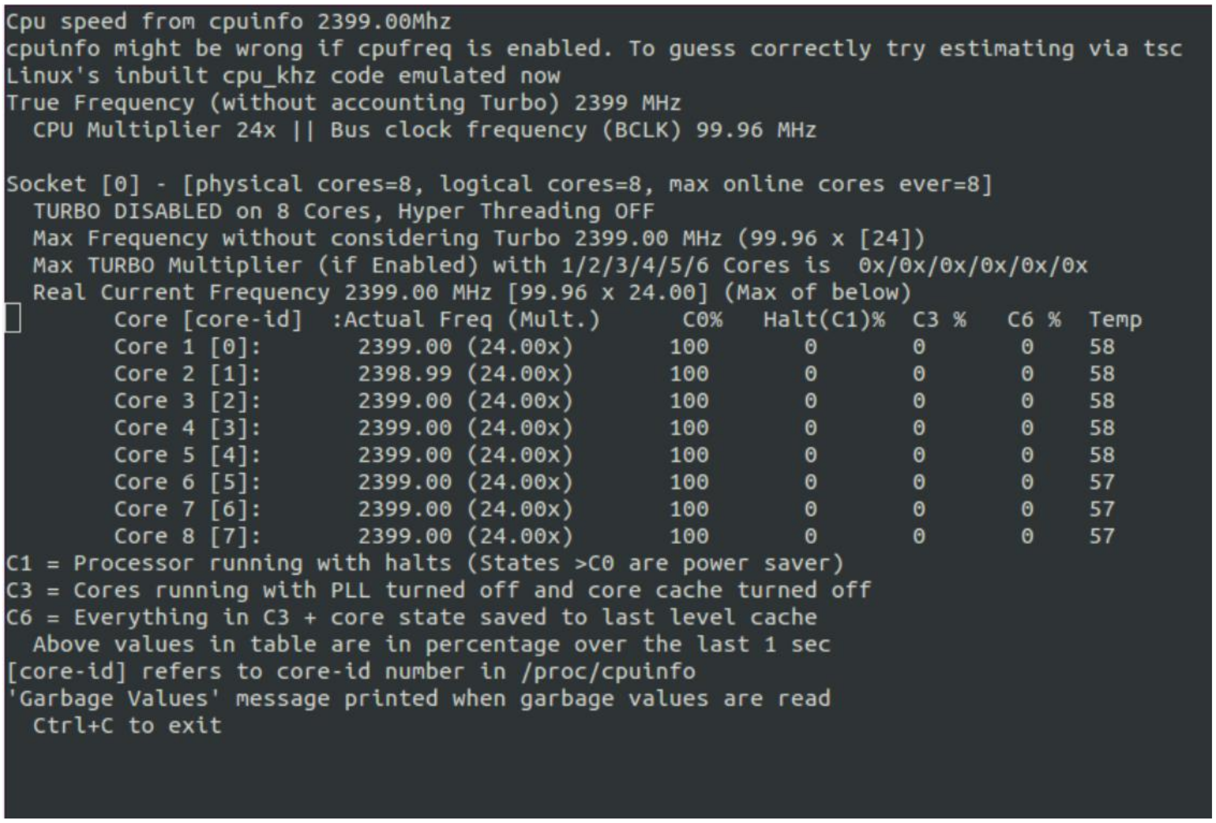
```
sudo /etc/init.d/cpufrequtils restart
```

The team was able to check that C-State was disabled with the following commands:

```
sudo apt-get install i7z
```

```
sudo i7z
```

“sudo i7z” should output something similar to the figure below:

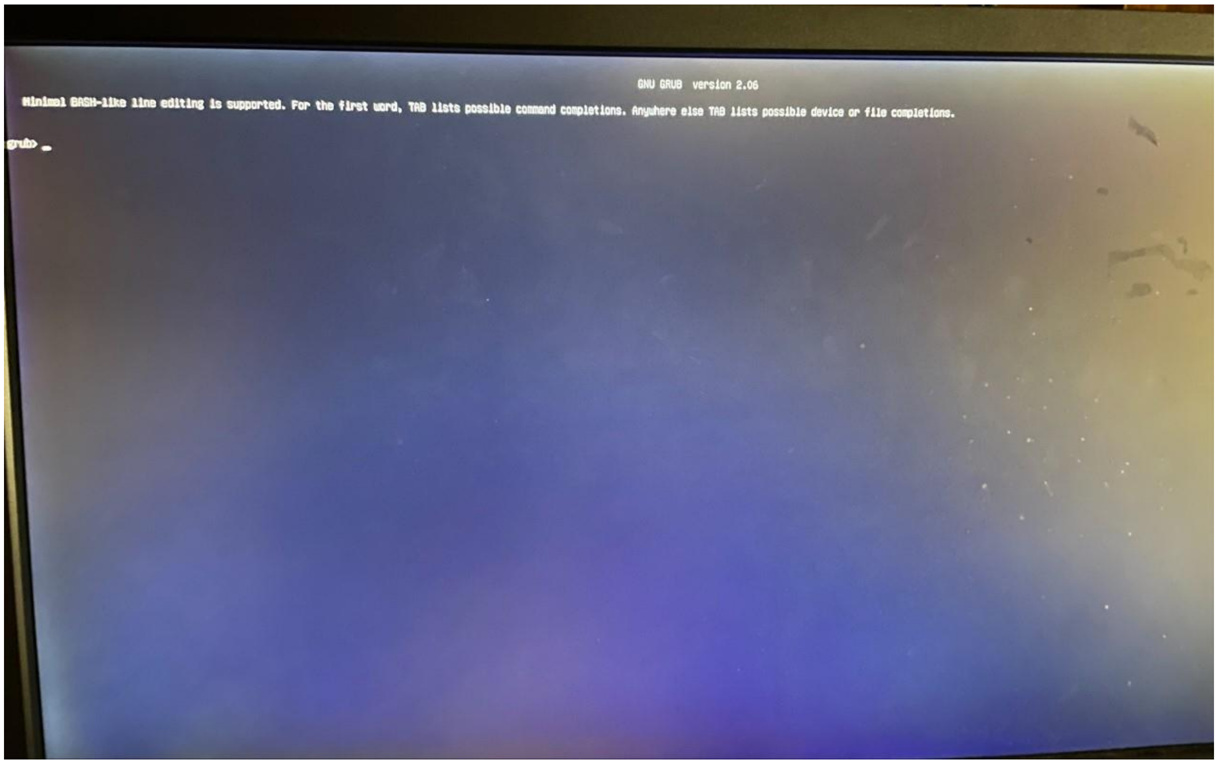


```
Cpu speed from cpuinfo 2399.00Mhz
cpuinfo might be wrong if cpufreq is enabled. To guess correctly try estimating via tsc
Linux's inbuilt cpu_khz code emulated now
True Frequency (without accounting Turbo) 2399 MHz
  CPU Multiplier 24x || Bus clock frequency (BCLK) 99.96 MHz

Socket [0] - [physical cores=8, logical cores=8, max online cores ever=8]
  TURBO DISABLED on 8 Cores, Hyper Threading OFF
  Max Frequency without considering Turbo 2399.00 MHz (99.96 x [24])
  Max TURBO Multiplier (if Enabled) with 1/2/3/4/5/6 Cores is  0x/0x/0x/0x/0x/0x
  Real Current Frequency 2399.00 MHz [99.96 x 24.00] (Max of below)
        Core [core-id]  :Actual Freq (Mult.)      C0%   Halt(C1)%  C3 %   C6 %  Temp
        Core 1 [0]:       2399.00 (24.00x)      100       0       0       0    58
        Core 2 [1]:       2398.99 (24.00x)      100       0       0       0    58
        Core 3 [2]:       2399.00 (24.00x)      100       0       0       0    58
        Core 4 [3]:       2399.00 (24.00x)      100       0       0       0    58
        Core 5 [4]:       2399.00 (24.00x)      100       0       0       0    58
        Core 6 [5]:       2399.00 (24.00x)      100       0       0       0    57
        Core 7 [6]:       2399.00 (24.00x)      100       0       0       0    57
        Core 8 [7]:       2399.00 (24.00x)      100       0       0       0    57
C1 = Processor running with halts (States >C0 are power saver)
C3 = Cores running with PLL turned off and core cache turned off
C6 = Everything in C3 + core state saved to last level cache
  Above values in table are in percentage over the last 1 sec
[core-id] refers to core-id number in /proc/cpuinfo
'Garbage Values' message printed when garbage values are read
  Ctrl+C to exit
```

Figure 6.1 OAI’s depiction of what “sudo i7z" should output. C0% is the only column that should be being used and the CPU should not change much.

After these steps were completed, the team ran into issues after rebooting. Sometimes, the computer would reboot to grub as depicted below. To solve this issue, the team used a command which restarted the GNOME display manager:

```
sudo /etc/init.d/gdm restart
```



```
                                            GNU GRUB  version 2.06
Minimal BASH-like line editing is supported. For the first word, TAB lists possible command completions. Anywhere else TAB lists possible device or file completions.

grub> _
```

Figure 6.2 After power management steps were completed, rebooting sometimes resulted in grub. Restarting the GNOME display manager fixed this issue.

After the power management steps were completed, the team moved onto the actual OAI installation and setup. To install the OAI software for the gNB and UE, the commands below were run. The first step was to install git and use it to clone OAI's repository.

```
sudo apt install git
git clone https://gitlab.eurecom.fr/oai/openairinterface5g.git
cd openairinterface5g
sudo git checkout develop
```

With the software finally installed, the team prepared to build it. The commands below were used to build the software:

```
sudo source oaienv
cd cmake_targets/
```

```
sudo ./build_oai -I -w USRP --eNB --UE --nrUE --gNB
```

A couple of times the build would crash due to a uhd library error. This error took a lot of research and debugging to determine what the cause was. The team was able to find a one-line solution that solved the issue immediately. The line is displayed below and only needs to be pasted into the terminal:

```
sudo apt-get -o Dpkg::Options::="--force-overwrite" install libuhd4.2.0
```

Once the software was successfully built, the team traveled to the build folder to run the software:

```
cd cmake_targets/ran_build/build
```

The team used two separate commands to run the gNB and UE. These commands were used from the year's prior MQP team. They are depicted below:

```
gNB:
sudo ./nr-softmodem -O ../../../targets/PROJECTS/GENERIC-NR-
5GC/CONF/gnb.sa.band78.fr1.106PRB.usrpb210.conf --sa -E --usrp-tx-thread-config 1
UE:
sudo ./nr-uesoftmodem -r 106 --numerology 1 --band 78 -C 3619200000 --sa -E -O
../../../targets/PROJECTS/GENERIC-NR-5GC/CONF/ue.conf --clock-source 1 --time-source
1 --ue-txgain 20 --ue-rxgain 87.5  --ue-fo-compensation  --nokrnmod 1 --dlsch-parallel 8
```

The "--ue-txgain" and "--ue-rxgain" were two parameters we had to experiment with to get the UE and gNB to connect. Note the steps prior to the two commands above were followed for both the gNB and the UE.

To set up the core network, a different set of instructions were followed. First, Docker and Python were installed on the machine. Both items are essential in setting up a core network. The following commands can be used to install them:

```
sudo snap install docker
sudo apt-get install python3.6
```

Once these two items were downloaded, the next action the team took was to set up a docker account. This allowed the team to pull images of the different containers. Once the account was created, docker was logged into using:

```
sudo docker login
```

Next, images were pulled from docker using instructions from OAI’s documentation. The commands below were all run individually:

```
sudo docker pull oaisoftwarealliance/oai-amf:v1.5.0
sudo docker pull oaisoftwarealliance/oai-nrf:v1.5.0
sudo docker pull oaisoftwarealliance/oai-spgwu-tiny:v1.5.0
sudo docker pull oaisoftwarealliance/oai-smf:v1.5.0
sudo docker pull oaisoftwarealliance/oai-udr:v1.5.0
sudo docker pull oaisoftwarealliance/oai-udm:v1.5.0
sudo docker pull oaisoftwarealliance/oai-ausf:v1.5.0
sudo docker pull oaisoftwarealliance/oai-upf-vpp:v1.5.0
sudo docker pull oaisoftwarealliance/oai-nssf:v1.5.0
sudo docker pull oaisoftwarealliance/oai-pcf:v1.5.0
sudo docker pull oaisoftwarealliance/oai-nef:v1.5.0
sudo docker pull oaisoftwarealliance/trf-gen-cn5g:latest
```

Once the images were all pulled, the team cloned the software for the core network using the following command:

```
git clone --branch v1.5.0 https://gitlab.eurecom.fr/oai/cn5g/oai-cn5g-fed.git
```

With the repository cloned, the team entered into the directory and ran the “syncComponents” script.

```
cd oai-cn5g-fed
sudo git checkout -f v1.5.0
sudo ./scripts/syncComponents.sh
```

Finally, the individual containers were built by running the commands below. The team ensured that the file paths were correct. Errors were thrown because the “ubuntu” file was named “ubuntu18” in some cases:

```
sudo docker build --target oai-amf --tag oai-amf:v1.5.0 \
          --file component/oai-amf/docker/Dockerfile.amf.ubuntu \
          --build-arg BASE_IMAGE=ubuntu:focal \
          component/oai-amf
sudo docker image prune --force
sudo docker build --target oai-nrf --tag oai-nrf:v1.5.0 \
          --file component/oai-nrf/docker/Dockerfile.nrf.ubuntu \
          --build-arg BASE_IMAGE=ubuntu:jammy \
          component/oai-nrf
sudo docker image prune --force
sudo docker build --target oai-spgwu-tiny --tag oai-spgwu-tiny:v1.5.0 \
          --file component/oai-upf-equivalent/docker/Dockerfile.ubuntu \
          --build-arg BASE_IMAGE=ubuntu:20.04 \
          component/oai-upf-equivalent
```

```
sudo docker image prune --force
sudo docker build --target oai-smf --tag oai-smf:v1.5.0 \
               --file component/oai-smf/docker/Dockerfile.smf.ubuntu \
               --build-arg BASE_IMAGE=ubuntu:22.04 \
               component/oai-smf
sudo docker image prune --force
sudo docker build --target oai-ausf --tag oai-ausf:v1.5.0 \
               --file component/oai-ausf/docker/Dockerfile.ausf.ubuntu \
               component/oai-ausf
sudo docker image prune --force
sudo docker build --target oai-udm --tag oai-udm:v1.5.0 \
               --file component/oai-udm/docker/Dockerfile.udm.ubuntu \
               component/oai-udm
sudo docker image prune --force
sudo docker build --target oai-udr --tag oai-udr:v1.5.0 \
               --file component/oai-udr/docker/Dockerfile.udr.ubuntu \
               component/oai-udr
sudo docker image prune --force
sudo docker build --target oai-upf-vpp --tag oai-upf-vpp:v1.5.0 \
               --file component/oai-upf-vpp/docker/Dockerfile.upf-vpp.ubuntu \
               component/oai-upf-vpp
sudo docker image prune --force
sudo docker build --target oai-nssf --tag oai-nssf:v1.5.0 \
               --file component/oai-nssf/docker/Dockerfile.nssf.ubuntu \
               component/oai-nssf
sudo docker image prune --force
sudo docker build --target trf-gen-cn5g --tag trf-gen-cn5g:latest \
               --file ci-scripts/Dockerfile.traffic.generator.ubuntu18.04 \
```

At this point in the process, the team switched to following documentation from Northeastern university:

https://openairx-labs.northeastern.edu/deploying-oai-in-5g-standalone-mode/

Following Northeastern's instructions, parameter "MNC" was set to 99, "SERVED_GUAMI_MNC_0" to 99, "PLMN_SUPPORT_MNC" to 99, "SD_1" to 1, and "OPERATOR_KEY" to c42449363bbad02b66d16bc975d77cc1 in the "docker-compose-basic-nrf.yaml" file under "oai-amf" on the core network. Under "oai-spgwu," parameter "MNC" was also set to 99. In the "openairinterface5g/targets/PROJECTS/GENERIC-NR-5GC/CONF/gnb.sa.band78.fr1.106PRB.usrpb210.conf" file on the gNB, the following changes to parameters were made: tracking area code to 0xa000, "mcc" to 209, "mnc" to 99, "sd" to 0x010203, and the second "sd" to 0x1. Using the command "ifconfig," the team was able to find the IP address of the device as well as the interface name. The parameters "GNB_INTERFACE_NAME_FOR_NG_AMF," "GNB_IPV4_ADDRESS_FOR_NG_AMF," "GNB_INTERFACE_NAME_FOR_NGU," and "GNB_IPV4_ADDRESS_FOR_NGU" were all updated accordingly in the same file. In order for the UE to be able to register with the core network, the following line was added to the "oai_db.sql" file in the docker-compose directory on the core network:

```
INSERT INTO `users` VALUES
```

```
('208990007487','380561234567','55000000000001',NULL,'PURGED',50,40000000,
100000000,47,0000000000,1,0xfec86ba6eb707ed08905757b1bb44b8f,0,0,0x40,
'ebd07771ace8677a',0xc42449363bbad02b66d16bc975d77cc1);
```

The following commands were run on both the gNB and core network in order to allow them to connect:

```
sudo sysctl net.ipv4.conf.all.forwarding=1
sudo iptables -P FORWARD ACCEPT
```

Finally, on the gNB, a route was added to the core network. The command to add the route can be found below:

```
sudo ip route add 192.168.70.128/26 via (core network address) dev (gNB address name) src
(gNB address)
```

The parentheses were filled in with values specific to the team's implementation and will differ for newer iterations of this project. The core network was started and stopped with the following commands:

```
sudo python3 ./core-network.py start-basic

sudo python3 ./core-network.py stop-basic
```

The team checked for gNB and UE connections using the last command below:

```
sudo docker logs oai-amf
```

When attempting to connect the UE to the gNB, the team sometimes received the error “Slot offset K2 (2) cannot be less than DURATION_RX_TO_TX (5).” Searching through the mailing list did not provide a solution to this issue. After further research, the team discovered a change that fixed the issue. In the file, “openairinterface5g/targets/PROJECTS/GENERIC-

NR-5GC/CONF/gnb.sa.band78.fr1.106PRB.usrpb210.conf," another line of code was added below the line "nr_cellid = 12345678L;" Underneath it, the following line was inserted:

```
min_rxtxtime=6;
```

This addition to the code solved the aforementioned error immediately.

APPENDIX B: Multi Node Spectrum Analysis

Sensor Node Program:

```
# import for SDR
import uhd
import numpy as np
import sys
from scipy.signal import resample_poly
# import for server client
import socket

clThreads = [] #keep track of threads

#Set print options for sending the string of samples
np.set_printoptions(threshold=99999)
np.set_printoptions(precision=5)

#All sample processing actions
def sample_processing(samples, samp_down_size):

    samples_f = np.abs(np.fft.fft(samples)) #Perform FFT, no FFT shift as it is too
computationally intensive

    clipped_samps = samples_f[(2*np.max(samples_f[0:2000]) > samples_f)] # Remove
largest magnitude values

    #Down sampling
    up_factor = len(clipped_samps)
    down_factor = samp_down_size
    samples_down = resample_poly(clipped_samps, down_factor, up_factor)

    packets = str(samples_down).encode() #Encode samples for transmission
    clSoc.sendall(packets) #Send packet to server

#Load usrp image
usrp = uhd.usrp.MultiUSRP()

#Set socket for data transmission
clSoc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srAdd = ('10.2.0.10', 3105)

#connect client to server address
clSoc.connect(srAdd)

num_samps = 10000 # number of samples received
```

```
center_freq = 751e6 # Hz
sample_rate = 10e6 # Hz
gain = 30 # dB (max is 31.5dB for x310)

duration = num_samps/sample_rate # seconds
samp_down_size = 200 #num_samps must be divisible by samp_down_size

# Set RX and TX values
usrp.set_rx_rate(sample_rate, 0)
usrp.set_rx_freq(uhd.libpyuhd.types.tune_request(center_freq), 0)
usrp.set_rx_gain(gain, 0)

# Set up the stream
st_args = uhd.usrp.StreamArgs("fc32", "sc16")
st_args.channels = [0]
metadata = uhd.types.RXMetadata()
streamer = usrp.get_rx_stream(st_args)

# Set up receive buffer
buffer_size = streamer.get_max_num_samps() # buffer_size set to maximum possible
recv_buffer = np.zeros((1, buffer_size), dtype=np.complex64)

# Start Stream
stream_cmd = uhd.types.StreamCMD(uhd.types.StreamMode.start_cont)
stream_cmd.stream_now = True
streamer.issue_stream_cmd(stream_cmd)

#Main function
run = True
while(run == True):

    #Recieve Samples
    samples = np.zeros(num_samps, dtype=np.complex64)
    for i in range(num_samps//buffer_size):
        streamer.recv(recv_buffer, metadata)
        samples[i*buffer_size:(i+1)*buffer_size] = recv_buffer[0]
    #Process Samples
    sample_processing(samples, samp_down_size)
    run = True

#When the code is stopped, end the stream for complete packet reception
stream_cmd = uhd.types.StreamCMD(uhd.types.StreamMode.stop_cont)
streamer.issue_stream_cmd(stream_cmd)
print("Exiting Gracefully")
sys.exit(0)
```

Server Side Programming:

```
import socket
import threading
import numpy as np
import datetime

# get the current date and time
now = datetime.datetime.now()

#function for collecting chunks of data from each file
def newClient(clSoc, clAdd):
    date_string = now.strftime("%Y-%m-%d_%H:%M")
    #make a new file based on client address
    newFile = f"Data/{clAdd[0]}_{clAdd[1]}_{date_string}.txt"
    with open(newFile, 'wb') as fi:
        while 1:
            info = clSoc.recv(2048) #Recieve data from client
            if not info:
                break
            #write chunks to the file as soon as its recieved
            fi.write(info)
            fi.write(b'\n')

        #close socket when data stops
    clSoc.close()

#make threads array for all new clients
clThreads = []

#make and bind the server socket to localhost
host = socket.gethostname()
ipAddr = socket.gethostbyname(host)
print("IP Address:", ipAddr)
srSoc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srAdd = (ipAddr, 3105)
srSoc.bind(srAdd)
np.set_printoptions(threshold=9999)
#listen for connections
srSoc.listen(10)
print("Listening...")

#continue to...
while 1:
```

```
#accept new clients
clSoc, clAdd = srSoc.accept()
print("New Client: ", clAdd)

#call newClient function with client address info and start a new thread
clThread = threading.Thread(target=newClient, args=(clSoc, clAdd))
clThread.start()
clThreads.append(clThread)

#remove threads when they are dead
for clThread in clThreads:
    if not clThread.is_alive():
        clThreads.remove(clThread)
```

References

[1] A. Gerodimos, L. A. Maglaras, M. A. Ferrag, N. Ayres, and I. Kantzavelou, “IoT: Communication protocols and security threats,” *Internet of Things and Cyber-physical Systems*, vol. 3, pp. 1–13, Jan. 2023, doi: 10.1016/j.iotcps.2022.12.003.

[2] W. Xiang, K. Zheng, and X. Shen, “5G Mobile Communications,” *Springer eBooks*, Jan. 2017, doi: 10.1007/978-3-319-34208-5.

[3] H. Remmert, “What Is 5G Network Architecture?,” *Digi*. https://www.digi.com/blog/post/5g-network-architecture#:~:text=The%205G%20core%20uses%20a,in%20the%205G%20core%20diagram

[4] Alepo Technologies Inc, “Alepo 5G Core: Authentication Server Function (AUSF),” *Alepo*, Mar. 22, 2023. https://www.alepo.com/products-services/authentication-server-function-5g-ausf/

[5] “Unified Data Management - What is UDM? | Sisense,” *Sisense*, Mar. 11, 2019. https://www.sisense.com/glossary/unified-data-management-udm/

[6] “What is the 5G Access and Mobility Management Function (AMF)?,” *TECHCOMMUNITY.MICROSOFT.COM*, Jan. 05, 2023. https://techcommunity.microsoft.com/t5/azure-for-operators-blog/what-is-the-5g-access-and-mobility-management-function-amf/ba-p/3707685

[7] Alepo Technologies Inc, “5G PCF+5G PCRF | Converged Policy Control for 4G and 5G networks,” *Alepo*, Mar. 21, 2023. https://www.alepo.com/products-services/policy-control-function-5g-pcf/

[8] N. Singh “What is the 5G Session Management Function (SMF)?,” *TECHCOMMUNITY.MICROSOFT.COM*, Feb. 26, 2023. https://techcommunity.microsoft.com/t5/azure-for-operators-blog/what-is-the-5g-session-management-function-smf/ba-p/3693852

[9] Z. Akhundov, “5G Core Network Overview,” *Telecompedia*, Jul. 18, 2020. https://telecompedia.net/5g-core-network-overview/

[10] “DN - Data Network,” *Mpirical*, Mar. 17, 2023. https://www.mpirical.com/glossary/dn-data-network

[11] D. Jones and C. Bernstein, “radio access network (RAN),” *Networking*, Apr. 28, 2021. https://www.techtarget.com/searchnetworking/definition/radio-access-network-RAN

[12] “Private 5G Deployment Guide - OnGo Alliance,” *OnGo Alliance*, May 23, 2022. https://ongoalliance.org/wp-content/uploads/2022/05/OnGo-Private-5G-Deployment-Guide-1.0.0.pdf

[13] S. Sirotkin, *5G Radio Access Network Architecture: The Dark Side of 5G*. John Wiley & Sons, 2020.

[14] D. Fang and Y. Qian, “5G Wireless Security and Privacy: Architecture and Flexible Mechanisms,” *IEEE Vehicular Technology Magazine*, vol. 15, no. 2, pp. 58–64, Mar. 2020, doi: 10.1109/mvt.2020.2979261.

[15] A. N. Zaidi, F. Athley, J. Medbo, U. Gustavsson, G. Durisi, and X.-M. Chen, “NR Physical Layer: Overview,” *5G Physical Layer*, pp. 21–34, Jan. 2018, doi: 10.1016/b978-0-12-814578-4.00007-2.

[16] “5G/NR - Numerology / SCS (Sub Carrier Spacing),” *ShareTechnote*. https://www.sharetechnote.com/html/5G/5G_Phy_Numerology.html

[17] M. V. Patil, S. J. Pawar, and Z. Saquib, “Coding Techniques for 5G Networks: A Review,” *2020 3rd International Conference on Communication System, Computing and IT Applications (CSCITA)*, Apr. 2020, doi: 10.1109/cscita47329.2020.9137797.

[18] Editorial Team, “5G NR Uplink Physical Channels,” Dec. 19, 2021. https://5ghub.us/5g-nr-uplink-physical-channels/

[19] Editorial Team, “5G NR Uplink Physical Channels,” Dec. 19, 2021. https://5ghub.us/5g-nr-uplink-physical-channels/

[20] E. Notes, “5G Data Channels: Physical; Transport; & Logical » Electronics Notes.” https://www.electronics-notes.com/articles/connectivity/5g-mobile-wireless-cellular/data-channels-physical-transport-logical.php

[21] M. Lichtman, R. Rao, V. Marojevic, J. Reed and R. P. Jover, "5G NR Jamming, Spoofing, and Sniffing: Threat Assessment and Mitigation," 2018 IEEE International Conference on Communications Workshops (ICC Workshops), Kansas City, MO, USA, 2018, pp. 1-6, doi: 10.1109/ICCW.2018.8403769.

[22] T. Seals, “Black Hat 2019: 5G Security Flaw Allows MiTM, Targeted Attacks,” *Threatpost*, Aug. 07, 2019. https://threatpost.com/5g-security-flaw-mitm-targeted-attacks/147073/

[23] “Managing data confidentiality,” *University of Delaware*. https://www1.udel.edu/security/data/confidentiality.html

[24] S. Fonyi, “Overview of 5G security and vulnerabilities,” *The Cyber Defense Review*, vol. 5, no. 1, pp. 117–134, Jan. 2020, [Online]. Available: https://cyberdefensereview.army.mil/Portals/6/CDR%20V5N1%20-%2008_%20Fonyi_WEB.pdf

[25] P. K. Nakarmi, “Cheatsheets for Authentication and Key Agreements in 2G, 3G, 4G, and 5G,” *arXiv (Cornell University)*, Jun. 2021, doi: 10.48550/arxiv.2107.07416.

[26]	CrowdStrike, “What is a Man-in-the-Middle Attack? + MITM Attack Prevention,” *crowdstrike.com*, Mar. 08, 2023. https://www.crowdstrike.com/cybersecurity-101/man-in-the-middle-mitm-attacks/
[27]	R. Chang, “Confidentiality, Integrity and Availability in Cyber Security,” *Kobalt.io*, Feb. 14, 2023. https://kobalt.io/blogpost/confidentiality-integrity-and-availability-in-cyber-security/
[28]	Inspired eLearning, “What Exactly is a Spoofing Message? And How do I Prevent It?,” *Inspired eLearning, Part of VIPRE Security Group*, Jun. 29, 2022. https://inspiredelearning.com/blog/spoofing-message/
[29]	A. Gil, “Data Security – Confidentiality, Integrity & Availability,” *kVA by UL*, Dec. 03, 2019. https://www.kvausa.com/data-security-confidentiality-integrity-and-availability/
[30]	“About OSA – OpenAirInterface.” https://openairinterface.org/about-us/
[31]	“oai / openairinterface5G · GitLab,” *GitLab*. https://gitlab.eurecom.fr/oai/openairinterface5g
[32]	H. Vella, “5G vs 4G: what is the difference?,” *Raconteur*, Jan. 30, 2023. https://www.raconteur.net/technology/4g-vs-5g-mobile-technology/
[33]	“Using AVX2 vectorization in Lambda - AWS Lambda.” https://docs.aws.amazon.com/lambda/latest/dg/runtimes-avx2.html
[34]	“EURECOM Mailing List,” *EURECOM*. https://lists.eurecom.fr/sympa/arc/openair5g-devel/2022-09/msg00019.html
[35]	“srsRAN Project Documentation — srsRAN Project documentation.” https://docs.srsran.com/projects/project/en/latest/
[36]	shuimoshusheng, “When I Rrunning a 4G End-to-End System, Tx Wwhile Waiting for EOB, Timed Out...,” *GitHub*, May 14, 2021. https://github.com/srsran/srsRAN_4G/issues/669
[37]	“Berkeley Sockets - Developer Help.” https://microchipdeveloper.com/tcpip:berkeley-sockets
[38]	“Sockets - Developer Help.” https://microchipdeveloper.com/tcpip:tcp-ip-sockets
[39]	“6. USRP in Python — PySDR: A Guide to SDR and DSP using Python.” https://pysdr.org/content/usrp.html
[40]	“5G / NR - Frame Structure,” *ShareTechnote*. https://www.sharetechnote.com/html/5G/5G_FrameStructure.html
[41]	[1] OpenAI. (2021). GPT-3: Language Models are Few-Shot Learners.

	https://arxiv.org/abs/2005.14165