

Politecnico di Milano

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING

Master Degree – Computer Engineering



NetQuack: a distributed system for collecting and searching radio signals

Supervisor
Prof. Stefano ZANERO

Co-Supervisor
Dr. Federico MAGGI

Candidate
Alessandro FIORILLO – 905701

Academic Year 2019 – 2020

Acknowledgements

I would like to thank my thesis advisor Prof. Stefano Zanero for introducing me to this project and for his charisma and expertise in the fascinating field of computer security. I would like to acknowledge also my co-supervisor Dr. Federico Maggi for his continuous support during this long journey, from the first days of the project until the drawing up of this thesis. Then I would like to thank all the people who somehow accompanied me in these years of study. Thanks to the professors of Politecnico di Milano, who taught me the knowledge of computer science with passion. Thanks to Roobi and Cristian, who always helped me during difficulties and I shared many funny moments with during these years. Thanks also to Andrea, who has been so helpful and enjoyable to work with while we shared part of the thesis work. Thanks to all my friends I spent time with, who made this journey more joyful also when tons of books and exams were waiting for me. Finally, I would like to thank my parents, who continuously supported me to reach this day.

Sommario

Le trasmissioni radio sono diventate parte importante del modo attuale di comunicare. In origine un metodo intelligente per trasportare semplici segnali e la voce umana, sono diventati un mezzo per trasportare dati digitali e controllare ogni tipo di dispositivo. I protocolli più famosi sono Wi-Fi e Bluetooth, ma ci sono molti protocolli personalizzati per controllare dispositivi nell'ambito della domotica e dell'industria: cancelli, automobili, droni, allarmi, robot, gru... Un comportamento inatteso in questi dispositivi potrebbe causare problemi anche in termini di sicurezza e privacy, e questo ha portato i ricercatori a studiare possibili vulnerabilità e attacchi nei protocolli radio.

Con questo obiettivo, è stato sviluppato uno strumento flessibile chiamato RFQuack. Esso rende possibile per i ricercatori ricevere, analizzare, modificare e trasmettere pacchetti radio, così come implementare routine ed exploit attraverso il linguaggio di programmazione Python. Questa tesi presenta un'estensione di RFQuack, chiamata NetQuack, che trasforma questo strumento in un sistema distribuito dove diversi utenti possono contribuirvi caricando pacchetti radio ed eseguendo ricerche su di essi in base al loro contenuto o altri parametri. In questo modo abbiamo creato uno strumento utile per svolgere ricerche partendo da una grande raccolta di segnali.

La principale sfida nell'implementazione di questo sistema risiede nell'affidabilità e nella scalabilità e questi obiettivi sono stati raggiunti usando l'infrastruttura cloud di Amazon Web Services. NetQuack è stato testato con diversi carichi di lavoro e i risultati mostrano che può gestire un flusso continuo di migliaia di pacchetti per ora.

Abstract

Radio transmissions have become an important part of the modern way of communicating. Originally a clever way to transport simple signals and the human voice, they became a mean to transport digital data and control all kinds of devices. The most famous protocols are Wi-Fi and Bluetooth, but there are many custom protocols used to control devices in home automation and industrial environments: gates, car lockers, drones, alarms, robots, cranes... An unintended behavior in these devices could cause problems also in terms of safety and privacy, and this brought researchers into studying possible vulnerabilities and attacks inside radio protocols.

With this objective, a flexible tool called RFQuack has been developed. It makes it possible for a researcher to collect, analyze, modify, and transmit radio packets, as well as to implement routines and exploits through the Python programming language. This thesis presents an extension of RFQuack, called NetQuack, which transforms this tool into a distributed system where different users can contribute to it by uploading radio packets and searching for them according to their content or other parameters. This allows researchers to conduct surveys on a large collection of signal captures.

The main challenge in the implementation of such a system resides in reliability and scalability and these goals have been reached using the cloud infrastructure of Amazon Web Services. NetQuack has been tested with different workloads and results show that it can handle a continuous stream of thousands of packets per hour.

Contents

Sommario	3
Abstract	4
Contents	6
List of Figures	7
List of Tables	8
Listing	9
Introduction	10
1 Background and motivation	12
1.1 Radio signals and protocols	14
1.1.1 Physical level	14
1.1.2 Protocol level	16
1.2 MQTT: a protocol for the Internet of Things	17
1.3 Radio signal analysis	20
1.3.1 Software Defined Radios	20
1.3.2 Hardware Dongles	22
1.3.3 RFQuack	23
1.4 Applications based on data collection	26
1.4.1 Shodan and Censys	26
1.4.2 Wigle and pcapr	27
1.4.3 Amazon Web Services	28
1.5 Goals and challenges	31
2 Implementation	32
2.1 Data collection	34
2.2 Data storage	37
2.3 Query	40

2.3.1	Partitioning	40
2.3.2	Caching	41
2.4	API description	43
2.5	Web interface	45
2.5.1	Structure	46
2.5.2	Design	49
2.6	Summary	51
3	Experimental analysis	52
3.1	Cost analysis	53
3.2	Scalability analysis	55
4	Conclusions and future works	62
4.1	Future works	63
4.2	Conclusions	64
A	Appendix	65
	Acronyms	68
	Bibliography	71

List of Figures

Figure 1.1	Modulation examples	15
Figure 1.2	Structure of a radio packet	16
Figure 1.3	MQTT Communication	18
Figure 1.4	R2832U and HackRF One	20
Figure 1.5	GNU Radio	21
Figure 1.6	Universal Radio Hacker	22
Figure 1.7	GQRX	22
Figure 1.8	Yard Stick One and PandwaRF	23
Figure 1.9	RFQuack dongle	24
Figure 1.10	RFQuack architecture	25
Figure 1.11	Pcapr home page	28
Figure 2.1	NetQuack architecture	33
Figure 2.2	NetQuack: register a device	46
Figure 2.3	NetQuack: homepage	47
Figure 2.4	NetQuack: perform a query	48
Figure 2.5	Component architecture	49

List of Tables

Table 3.1	Cost overview	53
Table 3.2	Cost analysis results	54
Table 3.3	Previous cost analysis	55
Table 3.4	Comparison between expected and real results . .	60
Table A.1	Scalability analysis 64 bytes (part 1)	65
Table A.2	Scalability analysis 64 bytes (part 2)	65
Table A.3	Scalability analysis 128 bytes (part 1)	66
Table A.4	Scalability analysis 128 bytes (part 2)	66
Table A.5	Scalability analysis 250 bytes (part 1)	67
Table A.6	Scalability analysis 250 bytes (part 2)	67

Listings

2.1	Policy document for devices in IoT Core	35
2.2	Transformation routine	38
2.3	Table definition	41
2.4	Cache routine in query	42

Introduction

In the last decades, remote-controlled devices through radio communication have become more and more widespread and this has brought many innovations both in home automation and industrial environments. These innovations increased the level of safety in workplaces, since dangerous machinery can be controlled remotely without a human operator physically involved. On the other hand, remote-controlled devices also became a further target for attackers and cybercriminals, since these technologies are also used to protect important assets. For example, an attacker who knows how the device communicates could craft legit radio packets and act as if he were the real operator. This can result in an intruder capable of unlocking a car, turning off an alarm, or controlling an industrial crane [13]. For this reason, radio communications and protocols have been actively analyzed by cybersecurity specialists to find possible vulnerabilities and exploits, while recommended practices have been defined when designing a new radio protocol.

The analysis of radio protocols is possible through many tools, among those the most common are Software Defined Radios and USB Dongles, normally coupled with other software for reverse engineering, such as GNU Radio and Universal Radio Hacker. Each one of these solutions has its own pros and cons, which are quite known for cybersecurity specialists. Software Defined Radios are universal, flexible radios and they make signal analysis and reconnaissance easier, but they require a deep knowledge of how signals work and serious ones are quite expensive. USB Dongles have integrated hardware for the modulation of the signal, simplifying many tasks, however they are not flexible. Each dongle is specialized for a certain set of signals and is a system on its own, using a specific radio chip, its own API, and a different way to communicate with it. With these limitations in mind, a new tool has been developed, namely RFQuack. Through a transceiver-agnostic firmware and a wholly documented API, RFQuack behaves like a USB Dongle, but with way more flexibility. It runs on any Arduino-compatible board and supports the most common transceivers, translating into a cheap solution easy to develop and deploy but as powerful as SDRs. Its flexibility relies

also on the software side, since the firmware is entirely modular and this feature makes the platform easily expandable and ready for new scenarios.

RFQuack proves to be the best solution for a project consisting of the collection of radio signals coming from a network of IoT devices. These signals should be cataloged into a global database and be publicly accessible for research purposes. A similar approach is already adopted by some large scale data collection platforms, such as Shodan, Censys, or WiGLE. As far as we know, there are no similar solutions in the radio frequency domain, and for this reason, we have decided to build such a platform using RFQuack as our ground base. We have designed, implemented, and tested this platform using the enhanced capabilities of cloud computing, which bases its nominal working on a network of RFQuack dongles. Each node acts as a crawler for RF signals: by running an automatic signal detection routine, a node can identify and capture all signals traveling in the air in a matter of milliseconds, demodulate them and send them to the cloud infrastructure, which then takes care of storing these data and making them available for querying. In the path of developing this platform, we always had to keep in mind that it should be scalable in the context of thousands of dongles transmitting data continuously. To demonstrate we reached this goal, we made a scalability analysis on our application and the results are encouraging: our platform can bear almost two million packets incoming per hour by a half-million dongles. These limits are enforced by the cloud infrastructure, but they can be further raised if necessary with the cooperation of the cloud provider.

This thesis presents in a detailed fashion the journey which brought us from the intention of building a distributed database of radio packets until its effective implementation, which manifests itself in a cloud application called NetQuack. In Chapter One we will dive into all necessary background knowledge to understand the topics involved in this work and the motivations around it. In Chapter Two we will present a complete implementation of NetQuack with explanations of each design choice. This is the core of our thesis. In Chapter Three we will see the results of experiments aimed at measuring the cost and the efficiency of NetQuack, then we will draw our conclusions on this new tool in the cybersecurity landscape.

1. Background and motivation

The goal of this work is the building and deployment of a large-scale database of digital radio signals to be used for security purposes in the radio-frequency domain. To better understand the motivations which brought us to this idea and the tools we used to implement it, we start showing the theoretical knowledge required to know how radio communications work and how IoT devices communicate in the context of a network. Then we will discuss several tools and solutions currently used by researchers.

In the first section, we will present the basic properties of a radio signal and how they are used to encode and transport information. Then we will show how these signals can be composed to build more high-level objects, such as packets and protocols. This will ensure the necessary knowledge to understand the topic and it will also clarify where security aspects reside in the field of radio communications.

In the second section, we will discuss the communication of devices in a distributed network. In particular, we will describe the inner workings of MQTT, a popular lightweight protocol suited for this purpose. It is a protocol easy to explain, but its working should be known to understand some implementation choices that we will present later.

The third section will describe the tools commonly used for analyzing radio packets and detecting vulnerabilities, in particular Software Defined Radios and hardware dongles, along with their differences, advantages, and disadvantages. This presentation will bring us discussing RFQuack, an innovative framework which exploits the advantages of both solutions and it is specifically designed with security research in mind: one of its features, the automatic detection of frequency and bitrate of an unknown signal, is the basis for the development of NetQuack, a central database to store radio packets detected by a net of dongles controlled by users.

Finally, we will examine some popular applications based on distributed data collection and the contribution of its users. In our analysis, we will show Censys and Shodan, two search engines for the Internet of Things which can perform queries and look for connected devices match-

ing some parameters, and then Wigle and pcapr, two global databases storing Wi-Fi networks and pcap listings entirely fed by the contribution of users. These applications are the inspiration of our project and we investigated a platform suitable for its building and deployment: such a platform is Amazon Web Services, a powerful and flexible set of tools to build scalable applications in the cloud.

At the end of the chapter, we will have all the necessary basic blocks to start developing NetQuack. That is the moment where we need to focus on the goals of the project and the challenges to keep in mind while developing it.

1.1. Radio signals and protocols

Radio communications are a common way to transport information and control remote devices without a physical connection or the presence of a human operator. The connection is given by the possibility of controlling an electromagnetic field and the two endpoints of the communication link, the transmitting device (TX) and the receiving device (RX), must agree on some parameters so that they can recognize their own signal and interpret correctly its content.

1.1.1. Physical level

At the physical level, a radio wave can be described by some parameters, which are also the very first information needed when performing the reverse engineering of an unknown radio protocol. The primary parameter to define is the **carrier frequency**, that is the main frequency used to encode data and to which the two endpoints must tune in. Many devices communicate in sub-GHz bands, such as the popular and free-of-usage 434 MHz and 868 MHz bands, but also the 2,4 GHz band is quite populated and used by relevant protocols.

Then it is necessary to define the method to encode bits inside the chosen frequency: this is called **modulation**. The most common modulations are:

- **Frequency Shift Keying (FSK)** where symbols are encoded as signals at different frequencies very close to the carrier frequency. The most common variant is 2-FSK with two symbols (one for the bit 0 and another for the bit 1), hence the use of two frequencies. However, it is possible to encode more bits in a single symbol, as it happens in the 4-FSK modulation where four frequencies are used to encode the symbols 00, 01, 10, and 11.
- **Amplitude Shift Keying (ASK)** where the different bits are encoded by varying the strength of the signal. Also in this case the most common and simple variant is On-Off Keying (OOK) where bit 1 is represented by the presence of the signal and bit 0 by its lack. It is still possible to find an amplitude modulation based on more than two symbols.
- **Phase Shift Keying (PSK)** represents different symbols with a change of wave phase. It is less common but still a basic

modulation scheme.

There are many variants and more complex modulations, but they are less common and often used only in limited scenarios. The great majority of signals of our interest fall into OOK or 2-FSK modulation which is compatible in all main transceivers.

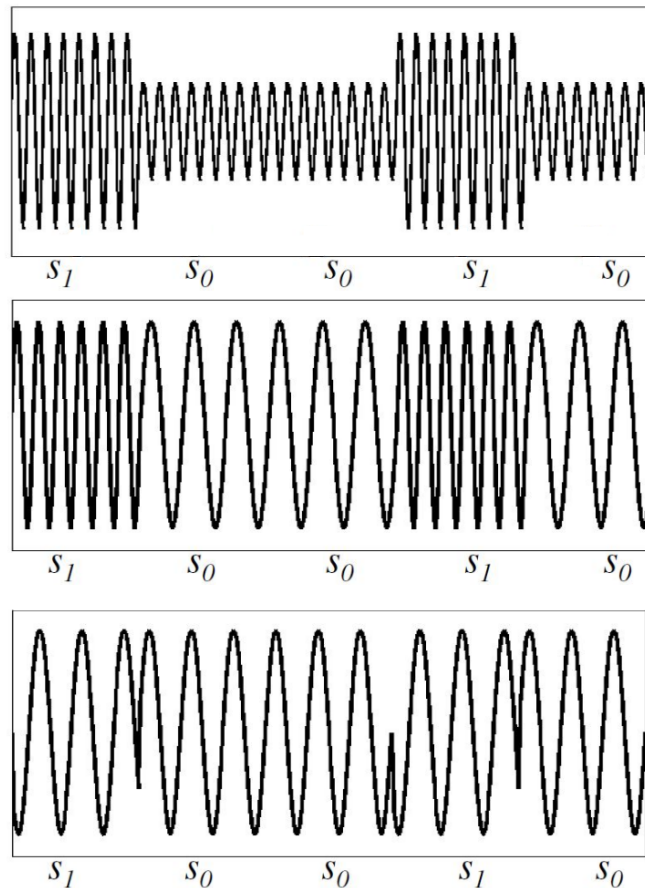


Figure 1.1.. Different types of modulation: in order ASK, FSK, PSK

Finally, we define the **bitrate**, that is the number of bits per second, necessary to synchronize bits with time. This value is in the order of kbps in case of sub-GHz frequencies, while raises to some Mbps only in the 2,4 GHz band. In the case of FSK modulation, another parameter is also required, which is called **frequency deviation** and represents the distance in hertz between the frequencies used to encode symbols. In general, it is a small value in the order of kilohertz. When analyzing an unknown protocol, all these values are initially unknown and they are usually discovered by investigating the datasheet of the transmitting device or by visual inspection of a spectrum analyzer.

1.1.2. Protocol level

Once fixed the carrier frequency, the modulation, and the bitrate, two devices can start sending signals to each other and recognize their content. However, this works only in a scenario where there exist only two devices using radio communication. In the real world, there are thousands of such devices, normally unrelated to each other and working for different purposes. By sending raw signals, there is no way for a device to know if that signal is coming from its corresponding transmitter or an unrelated one. Raw signals also deny the possibility of creating a network of multiple related devices, since each node does not have a way to identify itself to other members of the net. For this reason, once defined the physical level of radio signals, it is necessary to add another level of abstraction, in a similar way to what happens on the TCP/IP stack in the case of Internet communication.



Figure 1.2.. General structure of a radio packet

In radio communication data are sent as packets, where each packet brings some metadata in its header before the actual payload. The exact structure of a packet varies across the different radio chips used, but they generally have a similar structure which can be described as follows:

- **Preamble:** it is a repeating sequence of alternating bits such as 01010101... Its purpose is to warn the receiving device that a valid packet is incoming. Without the preamble, the receiver could not differentiate a real packet from background noise. Normally hardware dongles remove the preamble from packets before delivering them, so it can be seen only by sniffing the radio band with a spectrum analyzer.
- **Sync words:** a sequence of a few bytes (generally between two and five) which identify the device that should receive the packet. This field makes possible the realization of a working network of nodes: each participant in the network should have a way to differentiate its own packets from those addressed to other

participants and sync words solve exactly this problem. If a device receives a packet with a sync word not matching the expected one, the packet is just dropped.

- **Payload:** here is the effective content of the packet. The structure of the payload entirely depends on the particular application and it is here where vulnerabilities reside. There exists a vast range of vulnerabilities involving radio communication [7], starting from simple ones such as replay attack to more sophisticated methods, such as RollJam [15] and MouseJack [17], which captured the interest of researchers.
- **CRC:** a checksum performed by the receiving chip over both the header and the payload to ensure the message arrived without errors. It is calculated automatically by the radio and packets with the wrong checksum are generally discarded.

Many radio chips can be set in promiscuous mode, so that they can receive every packet, also those directed to other nodes. This is useful for research purposes, since a promiscuous node can be used in a similar way done with Wireshark, the famous tool for capturing TCP/IP packets. However, the flexibility of this mode is limited by the hardware design choices of the radio. For example, the popular CC1101 [14] chip can be set in promiscuous mode by disabling the filtering based on sync words, but this will also disable the filtering based on the preamble, causing the radio to interpret background noise as legit packets. This problem can be dammed since the CC1101 has also a filter based on RSSI. But this is not possible with nRF24, a hardware radio for the 2,4 GHz band: despite it does not support promiscuous mode officially, there is a trick to make it behave that way [12]. However, the lack of filtering based on RSSI causes the dongle to receive tons of packets which can not be distinguished from noise in real-time.

1.2. MQTT: a protocol for the Internet of Things

Message Queuing Telemetry Transport (MQTT) is a lightweight network protocol used to transport messages between devices. Its simplicity and scalability make it a suitable protocol in the context of

the Internet of Things, where a variable amount of devices are deployed and continuously send messages.

The architecture of this protocol expects a central node, the **broker**, which handles the routing of messages from the sender to the legit receivers, and it is the element to which devices connect when they want to join the network. The communication protocol follows a public-subscribe architecture: each message is characterised by a **topic** and the effective **payload**. The topic is a string that classifies the content of the message and it has a directory-styled structure. For example in a net of sensors distributed in different buildings and rooms, a sensor which monitors the temperature of a room could send its measures in a message with topic `sensorA/buildingB/roomC/temperature`. The devices interested in receiving this information can subscribe to this topic and it will be the broker's job to forward these messages to them. The strength of this protocol is that each device, be it a publisher or a subscriber, does not need to store any information about other devices.

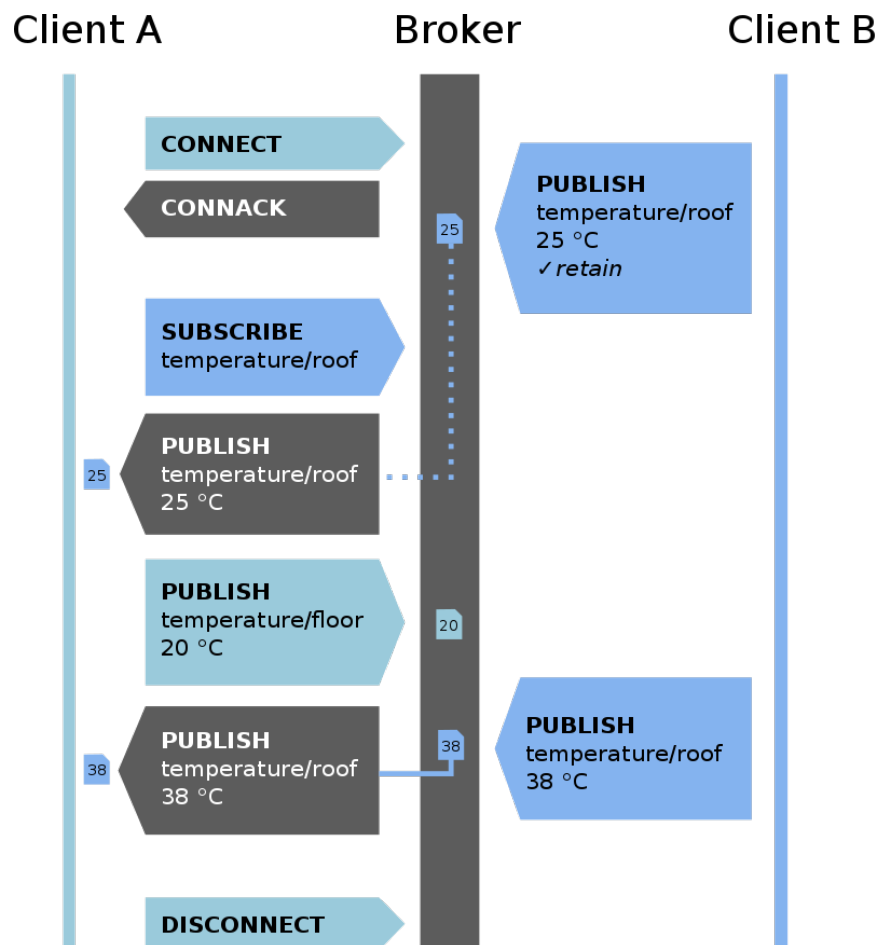


Figure 1.3.. Example of an MQTT communication with QoS = 0 [8]

In MQTT there is a small number of commands used to dispatch messages between the broker and the clients.

- **CONNECT**: it is sent by a client to the broker when joining the network. Upon connecting, the client identifies itself through a client ID, which is unique in every moment and is often used as a prefix in the topic of each message sent by that client. The broker answers the request with a CONNACK.
- **SUBSCRIBE**: when a client is interested in receiving messages with a specific topic, the client sends a SUBSCRIBE request to the broker with the topic as a parameter. The topic does not need to be a fixed string, usually subscribe requests are made over a family of topics and this can be done with some wildcards. The character `+` is used as a placeholder for a single level in the topic hierarchy. In the example of a network of sensors, a client interested in the temperatures of all rooms in a building could send a subscribe request for `+/buildingA/+/temperature`. The character `#` is a placeholder for all levels at the end of the hierarchy: the topic `sensorA/#` matches all messages with topic starting with `sensorA`.
- **PUBLISH**: this is the command sent every time a client has a message to deliver. It is sent along with the topic and the payload. It is up to the broker to send back a PUBLISH message to all clients subscribed to the topic.
- **DISCONNECT**: this is sent at the end of a session when a client leaves the network.

This protocol proves to be very simple, yet useful in the context of a distributed network. The protocol provides also three different levels of Quality of Service (QoS), set by the client upon connecting.

- **Level 0**: the message is sent only once and the client and broker take no additional steps to acknowledge delivery (at most once).
- **Level 1**: the message is re-tried by the sender multiple times until acknowledgment is received (at least once).
- **Level 2**: the sender and receiver engage in a two-level handshake to ensure only one copy of the message is received (exactly once).

1.3. Radio signal analysis

In the context of radio signal analysis, a researcher willing to analyze an unknown protocol needs a set of tools both hardware and software to start with. In this section, we are going to present different solutions to capture signals in the air, which are also generally coupled with a related family of software. In the case of Software Defined Radios, the most common software are GNU Radio and Universal Radio Hacker, along with a variety of spectrum analyzers such as GQRX and SDR Sharp. In the case of hardware dongles, there are fewer options and often dependent on the specific kind of hardware: the most common solution is RfCat along with a variety of derived versions.

1.3.1. Software Defined Radios

Software Defined Radio (SDR) are radio systems where all main components used to elaborate a signal, such as filters, converters, modulators, and demodulators, are implemented via software instead of hardware. This simple statement has interesting consequences, since it is possible to receive and transmit a wide variety of signals at different bands without replacing any piece of hardware. Their principle of operation is quite simple: the SDR will capture a sequence of In-phase and quadrature components (IQ) samples of every waveform it is tuned on and these samples are sent to the host computer through USB or Ethernet connection. The sample rate varies across specific models, but it is always in the order of mega samples per second. Then the raw signal must be demodulated and interpreted somehow by software.

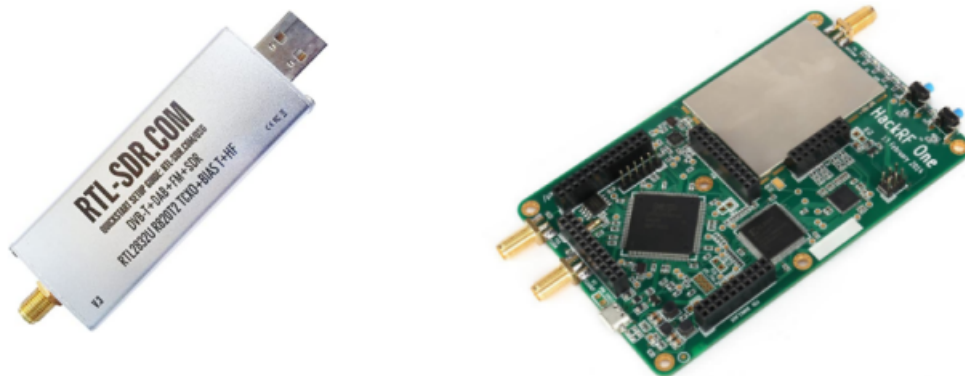


Figure 1.4.. Example of two common SDRs: an R2832U [23] and a HackRF One [10]

One of the most versatile and popular software for signal processing is **GNU Radio**, an open-source development toolkit able to create a digital signal processing pipeline through block diagrams [2]. A radio is implemented by creating a flowgraph, which is a composition of multiple digital signal processing blocks. This job can be done by coding in Python or using a handy graphical user interface. GNU Radio already comes with a range of implemented and useful blocks, such as bandwidth filters and waveform generators, and new blocks can be created by the experienced user. With this application, an SDR can become a receiver for GPS signals, Bluetooth communications, or even LoRa devices [6].

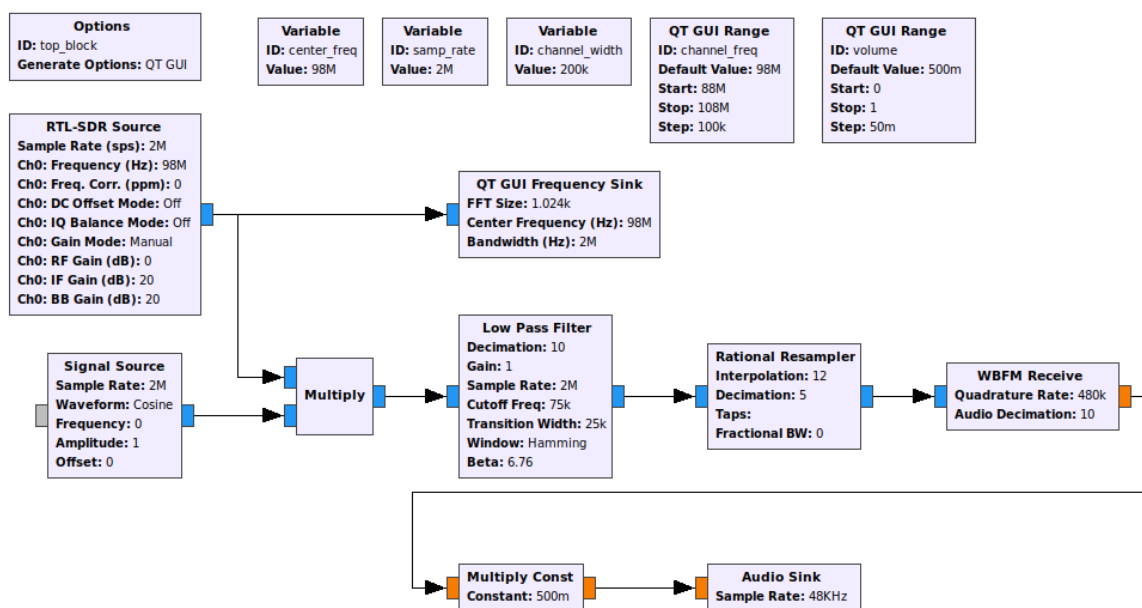


Figure 1.5.. The flowgraph of an FM receiver in GNU Radio

Another common tool aimed for security and reverse engineering of protocols is **Universal Radio Hacker (URH)**. The strength of this tool is that it cares automatically about the digital signal processing part, while giving the user useful features for protocol interpretation. Once a signal has been recorded, a researcher can easily infer the modulation and the bitrate by inspecting visually the raw signal and extract the content as a string of bits. Then the software comes already with features for reverse engineering, such as protocol fields inference, customizable decodings, fuzzing components, and simulation environments [20].

Finally, a fast way to test SDRs and discover signals in the air is given by spectrum analyzers, which are software able to show in real-time the data received by the SDR on a waterfall diagram. They are the primary way to discover the carrier frequency of a signal and they already come

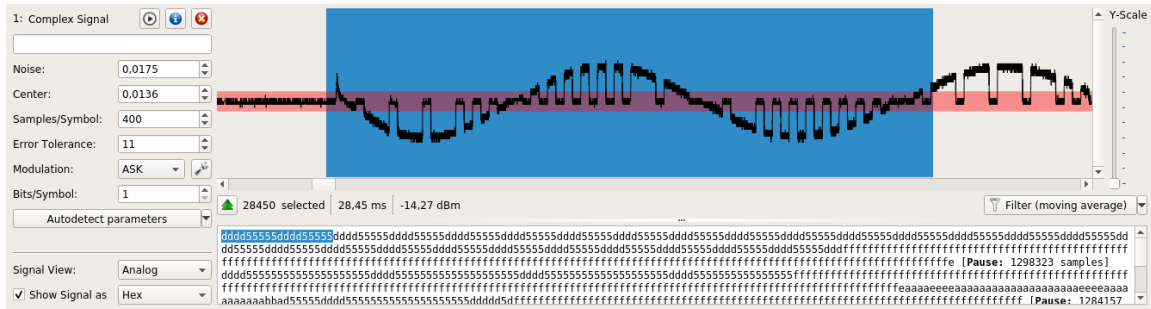


Figure 1.6.. Extracting bits from a raw signal in Universal Radio Hacker

out with some analog demodulation schemes, such as AM, FM, and Narrow FM. Famous examples in this field are **gqrx**, an open-source spectrum analyzer powered by GNU Radio and Qt library, and **SDR Sharp**, an alternative solution for Windows systems.

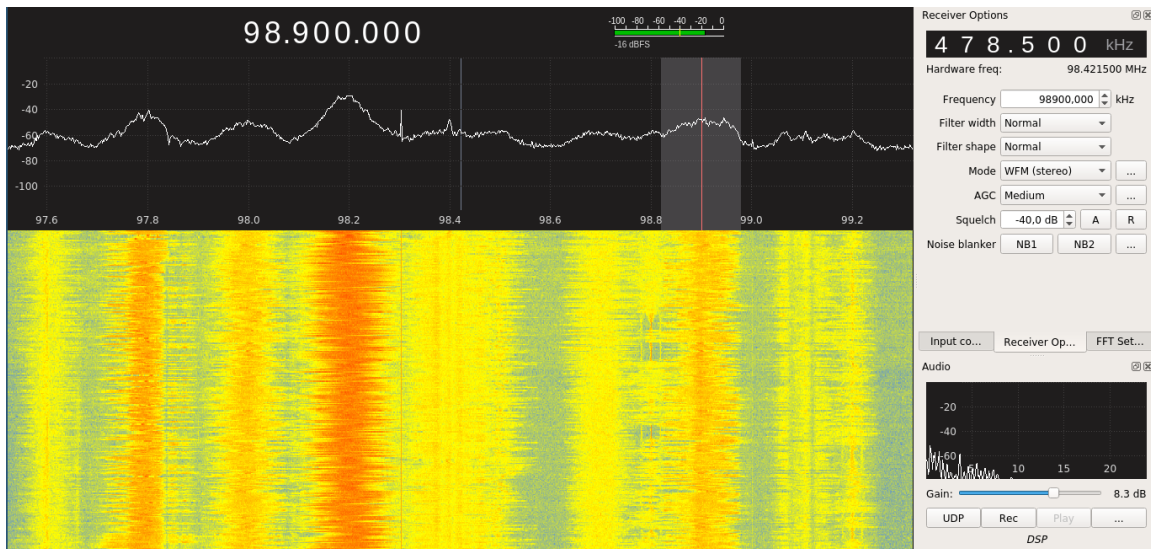


Figure 1.7.. Demodulating audio signals in GQRX

1.3.2. Hardware Dongles

An alternative for radio signal analysis is given by **hardware dongles**: they consist of an Micro Controller Unit (MCU) plus an embedded transceiver chip. The MCU runs a firmware exposing an external API so that a user can interactively configure and change the behavior of the dongle, where this interaction happens through a command-line interface running on a host computer. Unlike SDRs, where the device provides only raw data to be demodulated, in hardware dongles the whole process of capturing and demodulating happens on the chip, which

directly generates binary data. This is ideal in real-time scenarios, for example when performing an attack that requires intercepting a packet, modifying it, and resending it. Using a hardware dongle, the round trip time for this operation is very low, while in an SDR there is a long delay due to the demodulation process made in software. However the potentiality of a hardware dongle is bounded to the transceiver chip, which is fixed and soldered: if the chip can not tune in a certain frequency band, there is no way to capture signals on that band, or if the chip does not support a modulation scheme, there is no way to reverse engineer a protocol using that modulation scheme. Furthermore, when using a hardware dongle, it is necessary to know a priori the characteristics of the signal, such as frequency, bitrate, and modulation, in order to configure correctly the dongle.



Figure 1.8.. A Yard Stick One [11] and a PandwaRF dongle [5]

A tool commonly known in the case of hardware dongles is **RfCat**: it is an open-source software which consists of a firmware to flash on the dongle plus an interactive IPython used to control the chip. It offers a vast set of functionalities, both high level (such as tuning to a frequency or setting a modulation) and low level (reading and writing RF registers) [21]. It supports dongles based on the CC1111 chip, such as the Yard Stick One, and there are forks of it adapted for other dongles: an interesting example is given by the PandwaRF project.

1.3.3. RFQuack

Another platform for radio signal analysis is the emerging framework **RFQuack**, which tries to take the advantages of both SDRs and classic hardware dongles in a unique solution. It is a firmware designed to be flexible both in software and in hardware. The hardware flexibility

is given by the exposure of a uniform API which is unaware of the specific radio chip used, since all hardware-specific tasks are handled by a driver. This way, it is possible to use RFQuack both for sub-GHz band and 2,4 GHz band just by swapping the radio module with the one necessary for our needs. Software flexibility is achieved through modules, so that new features can easily be added to the firmware. Since RFQuack is mainly designed for security purposes, it already contains security-related features such as packet manipulation, filtering, and retransmission. Thanks to the efficiency of hardware radios, these functionalities can be used also in real-time contexts to test vulnerabilities and exploits [22].

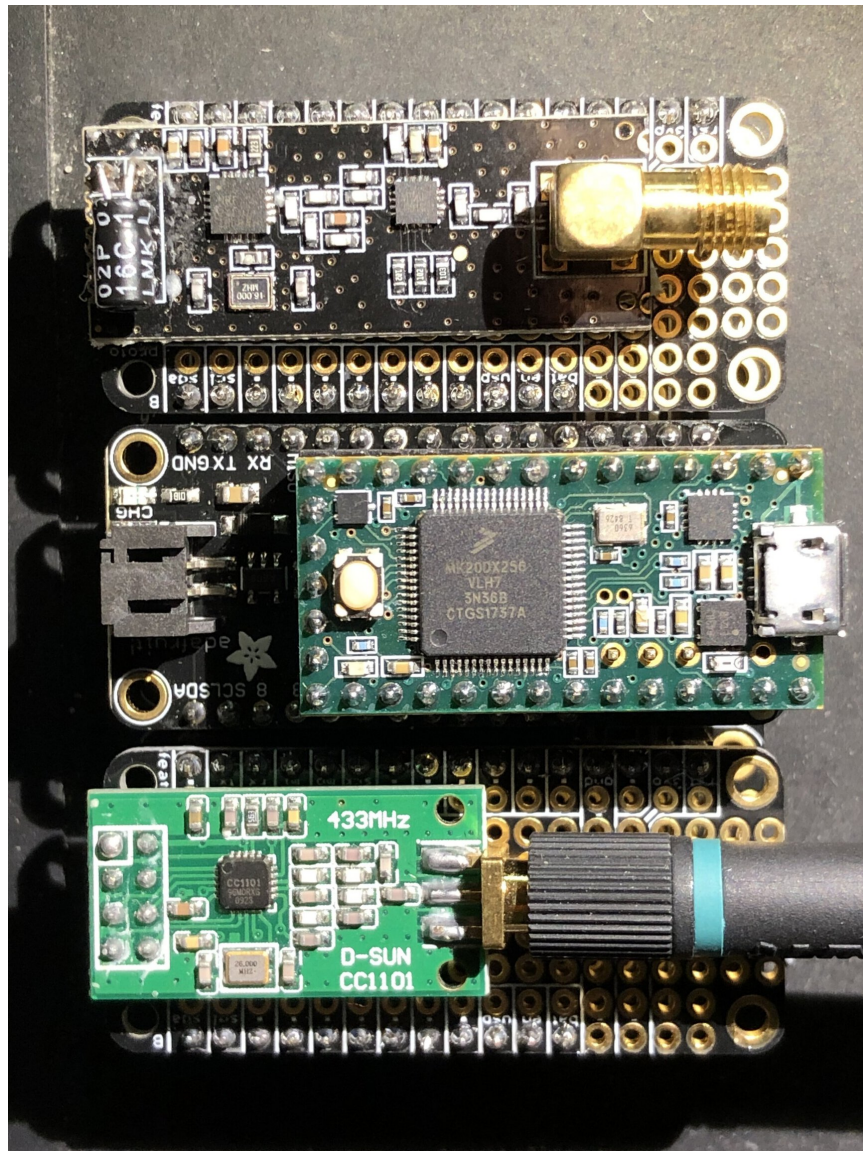


Figure 1.9.. An RFQuack dongle using Teensy as MCU and RF24 + CC1101 as radio chips

A generic RFQuack setup is composed of an MCU which hosts the firmware plus one or more radio chips. The firmware has been actively tested on ESP32 and ESP8266 using CC1101 and nRF24, and it supports up to five different radios connected at the same time. It is controlled by another host, handled by the user, running a command-line IPython interface which enables scripting. The communication between the host and the MCU can happen via serial or through MQTT over a wireless link (cellular or WiFi). In the current version of RFQuack, each exchanged message has a topic matching the structure `rfquack/[in|out]/[set|get|info]/<module_name>/<args>` where `in` is used for messages from the host to the MCU and `out` for the opposite direction.

When a user sends a command from the Python client, the message is serialized using Protobuf, a framework by Google which makes it possible to preserve data-type consistency and validation among devices using different programming languages. Since the firmware is written in C and the command-line interface is written in Python, this is the ideal use case for choosing Protobuf. The serialized message is handled by the transport level, which is either a serial connection or wireless communication over MQTT. Finally, when the message reaches the other host, it is deserialized and the firmware performs an action according to its content.

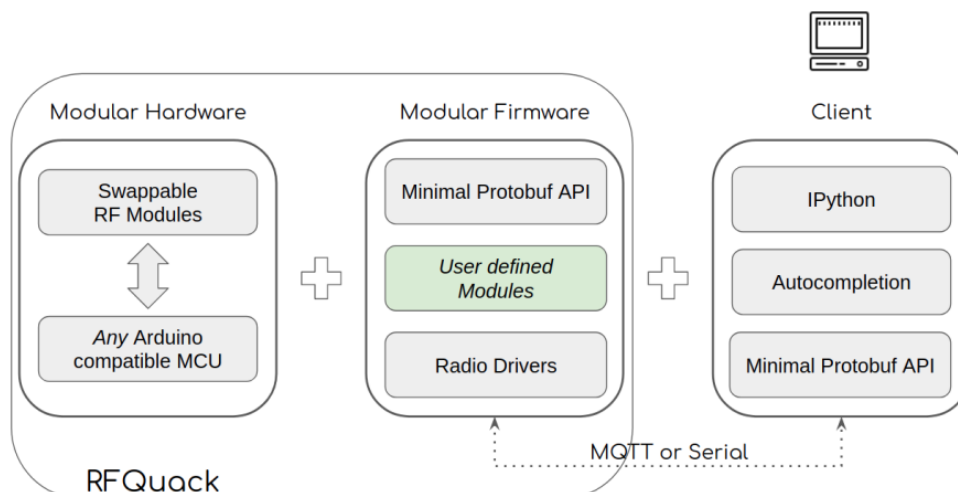


Figure 1.10.. The modular architecture of RFQuack

Inside the firmware, each command traverses a series of user-defined modules in a sort of software pipeline while each module is designed to perform a specific task. For example, the packet filtering module allows receiving packets only when matching one or more regex, while the packet modification module changes the content of a packet according to some rules before retransmitting it. Those two modules can be combined sequentially so that a packet can be first filtered and then modified. The module-based structure of the firmware makes it easy for a developer to add new functionalities.

The firmware comes out with some already implemented modules for common tasks and more specialized attacks, such as Roll Jam and Mouse Jack. A quite interesting module is the Automatic Frequency and Bitrate estimation. This module implements a handy feature consisting of identifying the carrier frequency and the bitrate of an unknown signal [13]. This feature can transform an RFQuack dongle into a sniffer and can be used to crawl radio signals. This module is the base of the distributed system NetQuack and it is the reason we have decided to build this platform as an extension of RFQuack: the automatic detection of signals transforms an RFQuack dongle into a device that autonomously looks for packets and stores them. The next step is defining the technology which will receive these packets and store them in a database.

1.4. Applications based on data collection

In the era of big data, it is becoming easier and easier to find applications relying on distributed data collection to perform their goals, where these data can be obtained through crowdsourcing and the voluntary participation of the users. In this section, we are going to present some examples of applications based on data collection in the field of information security along with their goals and we will present the infrastructure we have used for the implementation of NetQuack.

1.4.1. Shodan and Censys

Shodan is a search engine for finding specific types of computers connected to the Internet, such as webcams, routers, databases, and so on. For this reason, it is often called "the search engine of the Internet of Things". Launched in 2009 by John Matherly, it collects

data about all major protocols used on the Internet, be they web servers, FTP, SMTP, IMAP, or Real-Time Streaming Protocol, by crawling the Internet for publicly accessible devices. Every time a server is contacted by a new client, they often reply with metadata describing the service they provide and this is the information Shodan is constantly searching and cataloging [25].

It is used mainly by cybersecurity researchers and during its existence, it helped find vulnerable systems and security flaws in various areas. For example, a Forbes article in 2013 referenced Shodan claiming it was used to find security flaws in TRENDnet security cameras [16], while in 2015 it was reported that a security researcher used Shodan to identify accessible MongoDB databases on thousands of systems [1].

A similar platform is **Censys**: born in 2013, is conceived for security professionals since it can find devices affected by a specific vulnerability. It maintains three datasets through daily ZMap scans of the Internet and by synchronizing with public certificate transparency logs, in particular with hosts on the Public IPv4 Address Space, websites in the Alexa Top Million Domains, and X.509 Certificates. It can perform queries meeting certain criteria (for example IPv4 hosts in Germany manufactured by Siemens or browser-trusted certificates for github.com), generate reports on how websites are configured (what cipher suites are chosen by popular websites?), and track how networks have patched over time [4].

1.4.2. Wigle and pcapr

Wigle is a website for collecting information about the different wireless hotspots around the world. It can be queried to find WiFi and Bluetooth hotspots by name, location, SSID, MAC address, and so on. In terms of security, it has been useful to measure practically how much encryption schemes are widespread and it boosted the awareness of the need for encryption in wireless networks. Unlike Shodan and Censys, which work by crawling the Internet, Wigle's data are provided by users, who can contribute just by using an app on their smartphone, which constantly searches for hotspots in the neighborhood and uploads the discoveries on Wigle. Currently, it stores information for 688 million Wi-Fi networks, 343 million Bluetooth devices, and 15 million cell towers [26].

Another platform is **pcapr**, an online database storing pcaps with tags and categories, that is captured packets from network communication through sniffing tools such as Wireshark. On this website, users can upload their pcaps with a descriptive name or search for them by content.

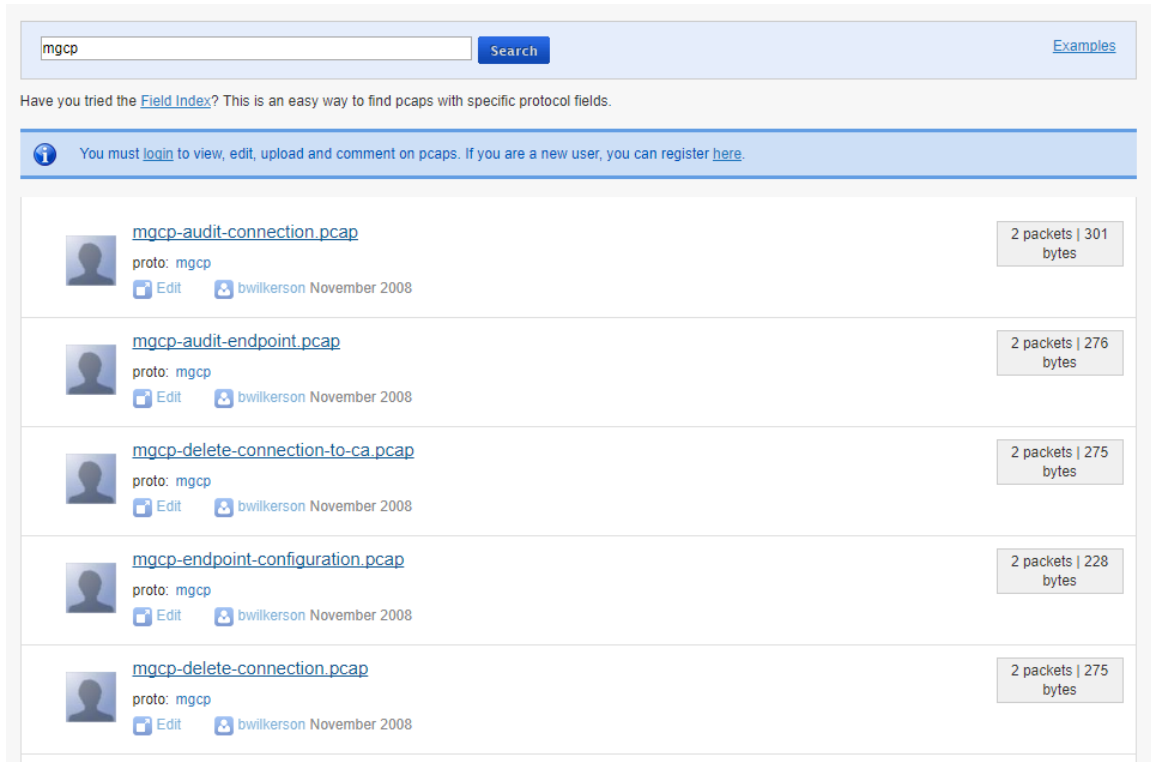


Figure 1.11.. The home page of pcapr

They are useful for all activities about networking, including diagnostics and security [19].

All these platforms became popular and have been used in research activities, but they have to face challenges to host and manage a heavy load of data. Shodan and Censys can perform queries over millions of devices in a fraction of a second, while Wigle and pcaps host data for millions of hotspots and pcaps. In this work, we will see a new distributed platform oriented to radio communications.

1.4.3. Amazon Web Services

As a consequence of the big data revolution, a technology that is becoming more and more popular is cloud computing. Cloud computing is the on-demand availability of computer system resources, which consist mainly of computing power and data storage. The main advantage of cloud computing is that the final user does not manage the resources and the platform directly. A user only needs to configure the services he needs and it is up to the cloud provider to scale accordingly. Platforms on the cloud can host simple applications, but their strength shows up when an application needs to scale for millions of users, requests, and

tons of data to store and search. In this case, it is unlikely that a user has all the resources to deploy such an application on his own, and this is the scenario where cloud computing comes into play [24].

A protagonist long present in the market of cloud computing is **Amazon Web Services (AWS)**, born in 2006 it is the dominant company in this field and offers all kind of services necessary to build an application. It provides a service for data storage, more kinds of databases (both relational and non-relational), computing power for executing code, network configuration, and many other more specific services. In particular, it offers a range of services for dealing with IoT devices, including an MQTT broker, and the possibility to compose these services and make them communicate among each other through triggering events. For this reason, AWS is the preferred platform to build the backend of NetQuack. Here is a brief description of the main services we are going to use in the process of implementing NetQuack:

- **Simple Storage Service (S3)** is the primary service for data storage. Data in S3 are divided into **buckets**, which are containers for all kind of files. An AWS account can create up to 100 buckets and each bucket has no limit in the number of files and size. What we call files, in the AWS platform are referred to as **objects** and they are identified by a **key**, which is something similar to the concept of a directory. Actually, S3 does not have a hierarchical system and it does not store "files". In the point of view of S3, each file is just a bunch of bytes and the key is a simple string to identify it. However, keys are often assigned in a way to reflect a directory structure. Data can be accessed through common HTTP requests such as PUT, GET, LIST, COPY, and DELETE. The cost of this service depends both on the space used for storage and the number of requests.
- **IoT Core** is the set of services and routines to deploy and control remote devices in a network. It is possible to register, create, or delete new devices, which can be monitored in real-time. Each device can be assigned to a group and groups define the messages that a device can send or receive through policy documents. Naturally, this service is the one providing the MQTT broker and a great feature is the possibility to take an action by calling other services upon receiving a specific kind of packet. The cost of this service depends on many aspects: the number of registered devices,

the minutes of connection, the number of exchanged messages, the number of rules triggered and actions executed.

- **AWS Lambda** is one of the most popular services in AWS and it consists of executing a function without hosting it on a server or managing an infrastructure. It is necessary only to define the function with a name and a programming language, then just write the code. It is possible to specify environment variables and limits in memory usage and time (up to 3 GB and 15 minutes of execution). AWS Lambda is paid per number of requests, time of execution, and memory used.
- **Kinesis Firehose** is a service designed for stream applications, where data are not incoming as a unique chunk but as a continuous stream. It acts as a buffer that stores all incoming data until reaching a space or time limit and then it stores all the aggregated data as a unique big object into S3. During this process, Kinesis Firehose can also transform data from one format to another or filter data that we want to discard. It is paid for gigabyte of data ingested and proves to be efficient and cheap.
- **AWS Athena** is a service which makes possible to perform queries on data stored on S3 using standard SQL. This proves to be useful when storing a great amount of data in JSON or CSV format and we need a way to extract information from them. With AWS Athena this becomes simple since data can be retrieved with the classic `SELECT ... FROM ... WHERE` syntax. It is paid for gigabyte of data scanned, but there are ways to optimize this metric, for example through data partition.
- **DynamoDB** is a No-SQL database which demonstrates to be very fast and flexible. However it is more expensive than Athena and S3, so this is not our main choice for storing packets as we will see later. Anyway, it has a role in NetQuack, since DynamoDB is a good solution for storing little quantity of support data that needs to be accessed frequently.
- **API Gateway** is the service which lets the user build a publicly accessible API. These APIs can be created according to various models, including a classic REST API, defining resources and methods over them. Each method can be linked to an AWS service,

generally a Lambda function, making the backend accessible to external users. API Gateway also gives the possibility to export or import the API definition in a YAML document, making it easy to transfer and document.

1.5. Goals and challenges

With the knowledge about the firmware RFQuack and the cloud provider Amazon Web Services, we have the two main blocks to build a distributed system for collecting and storing radio signals. The idea at the base of NetQuack is simple but powerful: an RFQuack dongle is deployed by the user and set to run the automatic frequency and bitrate detection module. Every time the dongle discovers a packet, it is forwarded to the backend which takes care of storing it. Connecting the dongle to the cloud backend is simpler than it seems. The user controls the dongle using a wireless connection over MQTT and generally, the MQTT broker runs on the user machine itself. In this case, instead of using a local MQTT broker, we use the one provided by Amazon Web Services which, upon receiving a packet, triggers a set of events with the purpose of storing its content.

In the next chapter, we will see in detail how this system is implemented: how the firmware has been modified to support data collection, which services are used in the backend and how they communicate. During the process of designing and development, we had to face some issues and decisions. In particular, we had to keep in mind that the goal was building not only a distributed system of dongles that collect radio packets, but also we intended the system to be scalable. This is important to consider when choosing a service from a set of similar ones. Amazon Web Services is very rich and supplies many alternative services for the same objective, but differing in some aspects which can be crucial. Also, the cost factor must be considered: Amazon Web Services has the advantage of being pay-per-use, and this implies that each decision must be translated in terms of the expected cost.

2. Implementation

In this chapter, we will present a detailed implementation of NetQuack. As stated previously, we are using a set of RFQuack nodes as crawlers detecting packets in the air, which are intercepted and then stored by a cloud infrastructure hosted by Amazon Web Services. For each key feature of NetQuack, we will describe how it has been implemented in the backend with the help of a specific cloud service, along with the rationale which brought us to such a choice.

The first half of the chapter will deal mainly with the backend implementation, that is the choice of cloud services and how they cooperate with each other. We will investigate in detail the interception of radio packets, their storage, and the subsequent way of querying over them. This journey will be completed with a description of the API used to interface to this system and its capabilities.

The second half will deal with the interface created for a user-friendly experience. NetQuack has been published with a web interface and we will present the framework chosen for its development along with a demonstration of its usage for central tasks: registering a new device, contributing to the database, and performing queries.

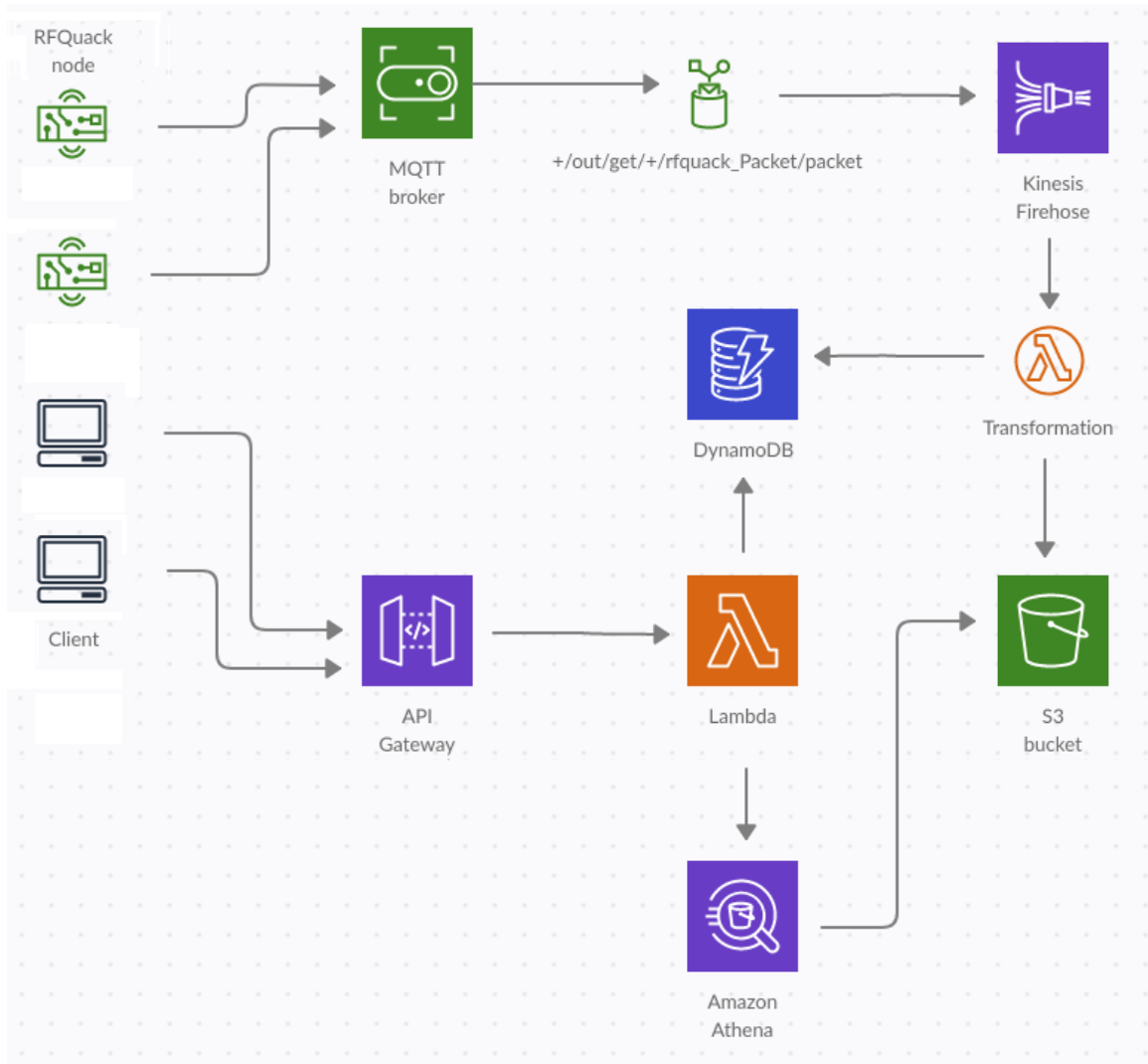


Figure 2.1.. A summary of the architecture of NetQuack

2.1. Data collection

The first step for the implementation of NetQuack is building a component capable of recognizing packets from the air and forwarding them to another component that will handle their storage. For this aim we remind the architecture of an RFQuack node: it is composed not only by a hardware dongle with an MCU and a radio chip, but also by an interactive IPython shell controlled by the user to configure the radio. The shell and the dongle communicate through the MQTT protocol. Thanks to this fact, transposing an RFQuack node from a local environment to a distributed one is straightforward: the MQTT broker, instead of running locally on the user machine, is hosted on the cloud. This is done through the IoT Core platform on Amazon Web Services, which provides a handy environment for deploying and managing a distributed network of connected devices.

However, this brings some issues we have to deal with, in particular about security. Since the MQTT broker is shared on the whole network and the command-line interface of RFQuack can potentially control every connected dongle, we need to find a way such that a user can control and configure only the dongles he owns. IoT Core can help us with two features: certificate-based authentication and policy documents. Certificate-based authentication enforces that only registered devices can connect to the network and each device must prove its identity through a certificate which is issued by Amazon root CA upon registration. Policy documents are JSON-formatted documents that describe which actions a device can perform. In particular, it defines which client ID a device is allowed to use when connecting and the messages it can send, receive, and subscribe to according to the topic. This is exactly what we need to reach the objective that a user can control only his own dongle using the shell.

Before showing the policy document we used in NetQuack, it is useful to remind how the communication happens between the dongle and the shell. Normally all messages flowing from the shell to the dongle match the topic `rfquack/in/#`, while all messages coming out from the dongle match the topic `rfquack/out/#`. So we have a way to distinguish the source of a message, but this works in a scenario with only one dongle. In a distributed environment with multiple dongles, how can we know which device sent a specific message? This problem can be solved by exploiting some features of IoT Core: this service not only forces each device to have its own certificate, but all devices must be registered

with a unique name, and it is possible to use this name as a prefix in the topic. So if we have two dongles, we can register the first one as, for example, **TheBigEar** and the second one as **TheSmallEar**, then configure them to send messages matching the topics `TheBigEar/out/#` and `TheSmallEar/out/#`, letting us distinguish which one has sent a message, instead of a generic `rfquack/out/#`. Implementing this feature is very simple and required a small change in the RFQuack firmware: by default, all messages were sent with `rfquack` as prefix topic. We changed this so that the topic of each message starts with the client ID of the dongle, that is the identifier the device used to register itself on IoT Core.

However, this solution is not complete yet, since we had to face another tricky problem: both the shell and the dongle use the same prefix topic, which coincides with the client ID, but since the shell and the dongle act as two different devices, they cannot connect at the same time using the same client ID. This has been solved in a hacky but effective way: in IoT Core, upon registering a new object, it is possible to assign attributes to it, that is a set of key-value pairs, and reference them inside the policy document. This way, when registering a new dongle in the system, let us call it `SampleName`, two objects are created inside IoT Core: one has the real name `SampleName`, while a second object is created with the name `SampleNameShell`, that is the name we have chosen plus the string "Shell". This fictitious object is also assigned an attribute `dongle=SampleName`, which is an attribute with the name of the actual dongle. With this trick, the shell controlling the dongle `SampleName` will need to use `SampleNameShell` as client ID, but it will be able to send messages with a topic matching `SampleName/out/#`, thus controlling its legit dongle. Here is the policy document assigned to both the dongle object and the shell object in IoT Core.

```
1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Effect": "Allow",
6       "Action": "iot:Connect",
7       "Resource": "arn:aws:iot:eu-central-1:123456789:client/${iot:
Connection.Thing.ThingName}"
8     },
9     {
10      "Effect": "Allow",
11      "Action": "iot:Subscribe",
12      "Resource": [
13        "arn:aws:iot:eu-central-1:123456789:topicfilter/any/in/*",
14        "arn:aws:iot:eu-central-1:123456789:topicfilter/${iot:"
```

```

15     Connection.Thing.ThingName}/in/*",
16         "arn:aws:iot:eu-central-1:123456789:topicfilter/*/out/*"
17     ],
18     {
19         "Effect": "Allow",
20         "Action": "iot:Receive",
21         "Resource": [
22             "arn:aws:iot:eu-central-1:123456789:topic/any/in/*",
23             "arn:aws:iot:eu-central-1:123456789:topic/${iot:Connection.
24             Thing.ThingName}/in/*",
25             "arn:aws:iot:eu-central-1:123456789:topic/${iot:Connection.
26             Thing.Attributes[dongle]}/out/*"
27         ]
28     },
29     {
30         "Effect": "Allow",
31         "Action": "iot:Publish",
32         "Resource": [
33             "arn:aws:iot:eu-central-1:123456789:topic/${iot:Connection.
34             Thing.ThingName}/out/*",
35             "arn:aws:iot:eu-central-1:123456789:topic/${iot:Connection.
36             Thing.Attributes[dongle]}/in/*",
37             "arn:aws:iot:eu-central-1:123456789:topic/any/in/set/ping/
38             rfquack_VoidValue/ping"
39         ]
40     }
41 ]
42 }

```

Listing 2.1. Policy document for devices connecting to MQTT on Amazon Web Services

By configuring the policy document in Listing ?? and enabling MQTT over SSL on RFQuack we were able to bring the MQTT broker on the cloud and make it usable by different nodes without interfering with each other. The next step is to find a way to sniff the radio packets detected by the dongle before storing them. Also in this case, IoT Core comes with a useful feature: we can set rules that are triggered upon receiving messages with a certain topic and as a result, they initiate some actions. In the RFQuack firmware, when the radio detects a packet, it is transported to the shell on a message with a topic matching `+/out/get/+/rfquack_Packet/packet`. We can set a rule on this topic so that the backend sniffs it and stores its content in a database. The syntax used in the packet filtering feature of IoT Core is based on SQL and it allows to add further information besides the payload. In particular, we use this rule inside NetQuack: `SELECT topic() AS topic, encode(*,'base64') AS payload, timestamp() AS timestamp FROM '+/out/get/+/rfquack_Packet/packet'`

Hence every message bringing the content of a radio packet is marked

with the topic, the payload in Base64, and the timestamp: this information is aggregated into a JSON object and forwarded to a service of our choice. In this case, the object is put in a buffer handled by Kinesis Firehose.

2.2. Data storage

So far we are able to detect packets arriving from a network of RFQuack nodes. Our next step is to collect these data and store them in an efficient way. In the previous section, we have cited Kinesis Firehose, a fully managed service for delivering real-time streaming data to other destinations. It is suitable for our use case, since radio packets are a small piece of data which arrive continuously, much as it happens with a video streaming transmission. To better understand why we need this service, let's imagine for a moment what we would do if it were not used in NetQuack.

As we already know, radio packets are intercepted by the MQTT broker of IoT Core and they are supposed to be stored in an S3 bucket, the main service used in AWS for data storage. In the absence of an intermediary service, such as Kinesis Firehose, every time a packet is detected, a new object would be created on S3 containing only the newly received packet. This behavior would not scale at all in a scenario of thousands of packets per second and its cost would increase dramatically: PUT requests on S3 are paid 0,0054 \$ every 1000 requests and receiving 1000 packets per second will result in a cost of 19,44 \$ after only one hour. Furthermore, a huge quantity of small files is enormously less efficient to query than a unique giant file. For these reasons, it is rational to insert an intermediary service between IoT Core and S3 with the objective of buffering small data and aggregating them in a unique file before delivering it to S3.

Kinesis Firehose can be configured so that it flushes its content after reaching a specific size or a timeout expires. We have set these parameters to their maximum allowed values, that is 128 megabytes for the buffer size and 900 seconds for the timeout. This choice minimizes the cost at the expense of a delay between the reception of a packet and its storage.

Another useful feature in Kinesis Firehose is the automatic conversion of data from standard JSON format to Apache Parquet. Apache Parquet is a free and open-source column-oriented data storage format of the

Hadoop ecosystem. This format claims to be highly efficient in terms of compression and query, since data are organized in a columnar format and they do not need to be wholly scanned when executing a query. However, the conversion to this format does not come for free and has a cost in terms of gigabytes of data converted, moreover it is a binary format working only in the Apache environment. After some experiments we stated that Apache Parquet is truly efficient and can break down the storage cost of various orders of magnitude, hence it is a reasonable choice to store packets in this format.

```

1 def transformation(event, context):
2     locations = dict() # cache for dongle locations
3     output = []
4
5     for record in event['records']:
6         message = json.loads(base64.b64decode(record['data']))
7
8         # retrieve topic, payload and timestamp
9         topic = message['topic']
10        payload = base64.b64decode(message['payload'])
11        timestamp = message['timestamp']
12        # deserialize the packet
13        pb_packet = rfquack_pb2.__dict__.get("Packet")()
14        pb_packet.ParseFromString(payload)
15        # extract dongle and location
16        dongle = topic.split('/')[0]
17        if dongle in locations:
18            latitude = locations[dongle][0]
19            longitude = locations[dongle][1]
20        else:
21            dynamodb = boto3.resource('dynamodb')
22            table = dynamodb.Table('dongle')
23            result = table.query(KeyConditionExpression=Key("name").eq(
24                dongle),
25                                FilterExpression="attribute_not_exists(
26                to_time)")
27            # should never happen
28            if result["Count"] == 0:
29                output.append({
30                    'recordId': record['recordId'],
31                    'result': 'ProcessingFailed',
32                    'data': record['data']
33                })
34            continue
35        # extract location
36        latitude = float(result['Items'][0]['latitude'])
37        longitude = float(result['Items'][0]['longitude'])
38        # save in cache
39        locations[dongle] = (latitude, longitude)
40        # extract fields
41        packet = dict()
42        packet['data'] = pb_packet.data.hex()
43        packet['timestamp'] = timestamp

```

```

42     packet['latitude'] = latitude
43     packet['longitude'] = longitude
44     packet['carrierFreq'] = pb_packet.carrierFreq
45     packet['bitRate'] = pb_packet.bitRate
46     packet['modulation'] = pb_packet.modulation
47     packet['syncWords'] = pb_packet.syncWords.hex()
48     packet['frequencyDeviation'] = pb_packet.frequencyDeviation
49     packet['RSSI'] = pb_packet.RSSI
50     packet['model'] = pb_packet.model
51     packet['dongle'] = dongle
52
53     output.append({
54         'recordId': record['recordId'],
55         'result': 'Ok',
56         'data': base64.b64encode( (json.dumps(packet) + '\n').encode()
57     })
58
59     return {
60         'records': output
61     }

```

Listing 2.2. The transformation routine which processes data in a format compliant for Apache Parquet

In Kinesis Firehose it is possible to alter data after it has been received so that it can be transformed, filtered, or uncompressed before being stored in S3 or, in our case, being converted into Apache Parquet. This process of data transformation is even mandatory in the case of converting data into Apache Parquet, since the conversion routine expects each record to be in JSON format with specific fields. However, our packets must be converted in any case, since the payload of a packet is firstly generated as a binary Protobuf string and then converted into Base64 by IoT Core. For each packet, our custom transformation routine (which is nothing else than a Lambda function) needs to decode the Base64 string, deserialize its content into a Protobuf object and finally convert this thing into a JSON object following the specifications of Apache Parquet. In this process, we also add the geographic coordinates of the dongle which received the current packet. Due to limitations in the input and output size of Lambda functions imposed by AWS, the transformation routine is called every time Kinesis' buffer ingests 3 megabytes of data. Here is the code of the transformation routine.

Finally, we can set a prefix for the key of the S3 object that will be stored. We have chosen a timestamp-based prefix, that is `date=!timestamp:yyyy-!timestamp:MM-!timestamp:dd/`. The reason for this choice will be more clear in the next section, where we will discuss how to query these data.

2.3. Query

At this point, we reached the goal of storing packets coming from a network of dongles. Our next objective is to perform queries, which is the most interesting part of our application. We anticipated that we were going to use AWS Athena for this task, an interactive query service that makes it easy to analyze data stored in S3 using standard SQL. Through this service, we can make queries with classic SQL without worrying about the format of our data, which can be JSON, CSV, or more exotic ones such as Apache Parquet.

2.3.1. Partitioning

This service is paid for gigabyte of data scanned, and if data are imported without any structure or optimization, the cost and average execution time for a query could be very high, because Athena would be obliged to scan the entire database. Fortunately, there are many tricks to improve the performance and reduce the impact on the bill, one of the most common is partitioning. If data are partitioned according to a certain field, every query involving a filter over this field is boosted significantly, since Athena can immediately skip all data not matching the filter on the partition field. The problem is deciding the field to use as a partition.

Initially, we thought to use a multilevel partition, with the first field being the date of detection of the packet, then the geographic location, and finally the frequency. This seemed to be a good choice because most queries on radio packets would contain a date range, a location, and a frequency band. Despite this is technically possible, it showed not to be as optimal as we thought. Having too many partitions would cause an overhead that nullifies the expected benefit, while partitioning makes sense only over fields with discrete values: this is true in the case of dates, but certainly it is not in the case of locations and frequencies (which are described by floating-point numbers). Moreover, we found on AWS documentation that Athena supports a limited number of 20000 partitions. With such a fine-grained partitioning, this limit would be exceeded rapidly. So finally we opted for a simpler solution where partitioning is based only on date: it is a discrete value and creates a reasonable number of partitions. This choice explains why Kinesis Firehose stores data on S3 using `date=!timestamp:yyyy-!timestamp:MM-!timestamp:dd/` as a prefix

as stated in the previous section: the prefix is used to assign an object to a specific partition.

Another way to optimize queries in Athena is the format used to store data. In our case, data are stored in Apache Parquet and this format is more efficient to query with respect to JSON for a number of reasons. The first one is that data in Apache Parquet are heavily compressed, and since Athena is paid per gigabyte of data scanned, if data are compressed in the first place every query will be less expensive even in the worst case where all data are scanned. Moreover, Apache Parquet is a columnar format, so that Athena can directly skip all fields not used as filters resulting in faster and further cheaper queries. This format even keeps track of minimum and maximum values for each numeric field, resulting in another source of optimization for Athena.

```

1 CREATE EXTERNAL TABLE `${var.DATABASE_PACKETS}`.`${var.TABLE_PACKETS}
   `(
2   `timestamp` string,
3   `latitude` float,
4   `longitude` float,
5   `carrierFreq` float,
6   `bitRate` float,
7   `modulation` string,
8   `syncWords` string,
9   `frequencyDeviation` float,
10  `RSSI` float,
11  `model` string,
12  `dongle` string,
13  `data` string)
14 PARTITIONED BY (`date` date)
15 STORED AS PARQUET
16 LOCATION 's3://${aws_s3_bucket.bucket_packets}/'
17 TBLPROPERTIES ("parquet.compression"="SNAPPY");

```

Listing 2.3. Definition of the table used in S3 to store packets

With all this information in our hands, we can see and understand the SQL command in Listing 2.3 used to generate the table along with all fields stored for each packet.

2.3.2. Caching

Finally, there is a further optimization we can do to avoid scanning in vain, which is caching. Athena does not implement caching natively, because it is decoupled from the actual storage, so it has no way to understand if cached data are still valid or out-of-date. However, Athena still saves the results of each query on a separate S3 bucket and we can exploit this to implement caching manually. Since our database is a

set of radio packets which accumulate over time and are partitioned by date, we know for sure that if a query has a filter on the date and this date is set in the past (whereby "past" we mean any date before today) then it is guaranteed that the query will always provide the same result. Moreover, the function which requests a query to Athena in AWS SDK does not reply directly with the query result, but with an identifier named "query execution id". The results are retrieved through a different function which takes the query execution id as a parameter.

This way caching becomes easy to implement: first, we force each query made by users to contain a filter on the date, then before performing the query, we check if that query has been already executed (for this purpose we use a support table on DynamoDB containing for each query the corresponding SQL code and the query execution id). If so, we do not ask Athena to compute it again, but we reply directly with the query execution id. If not, we execute the query, but before returning the query execution id, we store it in the support table. Of course, this process is done only if the query happens on a date range in the past. Unfortunately, query results can not be stored indefinitely, since Athena automatically deletes past queries after 60 days. For this reason, each query is cached along with a timestamp and a periodic routine deletes old queries from the support table.

```

1 def make_query(query, cache):
2     athena_client = boto3.client('athena')
3
4     if cache:
5         dynamo_client = boto3.resource('dynamodb')
6
7         query_hash = hashlib.sha256(query.encode()).hexdigest()
8         table = dynamo_client.Table(os.environ["QUERY_TABLE_DYNAMO"])
9         result = table.get_item(Key={'hash': query_hash})
10
11        # cache hit
12        if 'Item' in result:
13            return {
14                "query_execution_id": result["Item"]["query_execution_id"]
15            }
16
17        # There is no Lambda trigger, when the query terminates
18        def poll_status(qei):
19            while True:
20                result = athena_client.get_query_execution(QueryExecutionId =
21                    qei)
22                state = result['QueryExecution']['Status']['State']
23
24                if state == 'SUCCEEDED':
25                    return result
26                elif state == 'FAILED':

```

```

26     return result
27
28     time.sleep(1)
29
30 response = athena_client.start_query_execution(
31     QueryString=query,
32     QueryExecutionContext={
33         'Database': os.environ["DATABASE_PACKETS"]
34     },
35     ResultConfiguration={
36         'OutputLocation': 's3://' + os.environ["BUCKET_QUERY"] + '/'
37     }
38 )
39
40 query_execution_id = response["QueryExecutionId"]
41 result = poll_status(query_execution_id)
42
43 if result['QueryExecution']['Status']['State'] == 'SUCCEEDED':
44     if cache:
45         table = dynamo_client.Table(os.environ["QUERY_TABLE_DYNAMO"])
46         table.put_item(
47             Item={
48                 'hash': query_hash,
49                 'query_execution_id': query_execution_id,
50                 'timestamp': Decimal(str(time.time()))
51             }
52         )
53     return {
54         "query_execution_id": query_execution_id
55     }
56
57 return {
58     'message': 'database error'
59 }

```

Listing 2.4. The query routine checks if the results already exist in cache

With this mechanism, we finally reached the milestone of a system that collects radio packets from a network of dongles, stores them in the cloud, and allows the execution of queries. So far we have seen a detailed implementation of the backend, but there is no way for a user to effectively interface to it, since we have not presented yet how to register a new device in the network and how to execute queries. In the next section, we will show the external API offered by NetQuack and an interface built around it.

2.4. API description

The next objective of our application is to make it accessible to external users, so that they can make use of its services and features. With

this goal, Amazon Web Services presents API Gateway, a cloud service for creating, publishing, maintaining, monitoring, and securing both REST and HTTP APIs at any scale. Using this service is pretty straightforward: with the intuitive web interface, it is possible to create a new REST API ready to be filled with resources and methods. Each method can then be linked to other services to execute its business logic (generally a Lambda function) and decorated with input parameters and output schemas.

Since the main features we need to offer are the registration of a new dongle and the possibility to perform queries, we created two REST resources: one called "dongle" for device management and another called "query". At the moment, the operations available in the API are the following:

- **GET /dongle** takes no parameters and returns a list of all deployed nodes along with their name and geographic location.
- **POST /dongle** is used to register a new dongle: it takes as parameters the name of the device and the geographic location. If the requested name is available and the coordinates are valid, NetQuack registers the device by creating two objects in IoT Core (one for the dongle and another for the shell, as explained previously) and issuing a certificate necessary to the device for connecting to the MQTT broker. The method replies with the certificate plus the public and private key which must be flashed into the RFQuack firmware.
- **PUT /dongle** is a method for updating an existing dongle. If a user wants to change the location of a device, he needs to call this method with the name of the device, the new geographic location, and a token which is the result of running `sha256sum` on the private key file. This is a way to limit this operation to the legit owner of the dongle without the need of registering users and managing passwords.
- **POST /query** performs a request for a query. Its parameters are the fields of the query, in particular a date range, a location area, frequency, bitrate, modulation, and packet content. The method validates the parameters (notable is the limit on the date range which can not be longer than seven days) and if they are valid, it replies with a query execution id. It is not guaranteed that the

query is actually executed, since the method will look for eventual cached results.

- **GET /query** is used to fetch the results of a query given the query execution id as parameter. The method replies with at most 100 results in CSV format along with a string called "next token". This token is an additional parameter to send alongside the query execution id for fetching the following 100 results. It is a convenient way to achieve pagination and it is implemented natively by Athena.

Both resources also exhibit an **OPTIONS** method which is necessary to implement Cross-Origin Resource Sharing (CORS), a mechanism that allows those resources to be requested from a web page hosted on a different domain. Finally, API Gateway allows to insert descriptions for each resource, method, and parameter serving as documentation and the whole API can be exported in an OpenAPI compliant format. This way the API can be easily documented as it gets updated in the future.

2.5. Web interface

Once we have defined a public API to access the backend of NetQuack, we can build some user-friendly interfaces which interact with this API. We wanted to create a web-interface, since by this time the browser is the most common way to interact with any application. All modern web applications are highly dynamic and built with the support of frontend frameworks, hence the first decision to make consists of the choice of the framework to use. With some research, we obtained a list of the most popular ones and we restricted our choice to one of these three: Angular, React and Vue. All of them are currently very popular and constantly updated, so we needed to evaluate many aspects to make a fair decision. We finally came up choosing React [9].

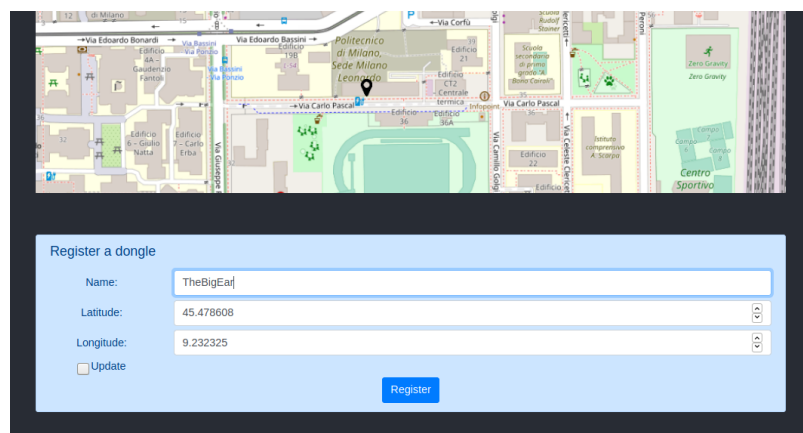
The reason for this choice is that we wanted to create a minimalistic web interface and React seems the most suitable for this kind of task: it is a lightweight framework that is easy to learn and develop with, as well as widely supported and documented for the most common use cases. Also, its structure based on component hierarchy makes applications easy to maintain, document, and further develop. Along with it, we have used two other libraries for web development: **React Bootstrap**

for an easy definition of the CSS [3] and **OpenLayers**, a library for the insertion and manipulation of maps based on OpenStreetMap data [18].

2.5.1. Structure

Since the most important features to implement are the registration of a new dongle and the execution of queries, we thought about organizing the web interface with a tab structure. The user can choose the operation to perform by selecting it on a navigation bar which is situated on the top of the web page. Upon clicking on the navigation bar, an input form related to the selected operation appears on the screen and the user just needs to fill the form and submit the request. In both cases of registering a dongle and executing a query, the user could need to insert some geographic coordinates, which are pairs of latitude and longitude (for example when querying for packets received in a certain area).

Since raw geographic coordinates are unfriendly and counter-intuitive, we decided to accompany the forms with a map to simplify the process of inserting these coordinates. If a user needs to register a new dongle, he just clicks the desired location on the map: the application will show a marker on the selected point and will automatically fill the form with the correct coordinates. While if the user wants to perform a query in a specific area, he just zooms to that area on the map, and also in this case the form will be filled automatically with the right coordinates. Moreover, upon loading the page, the map is filled with markers that indicate the position of deployed RFQuack dongles. This is useful to see visually where nodes are located and to know in advance where it is possible to find some radio packets.



The screenshot displays a web interface for registering a new device. At the top, there is a map of a city area, likely Milan, showing streets and buildings. A red marker is placed on the map, indicating the location of the device. Below the map, there is a form titled "Register a dongle". The form contains the following fields:

- Name: TheBigEar
- Latitude: 45.478608
- Longitude: 9.232325
- An "Update" checkbox, which is currently unchecked.
- A "Register" button.

Figure 2.2.. Registering a new device into NetQuack on the Web interface

RFOQuack

The search engine for Radio Signals.

Dongle **Query**

OpenStreetMap contributors.

Query for a packet

Type a hexadecimal string, use % as wildcard.

Query:

Date (from ... to):

Latitude (from ... to):

Longitude (from ... to):

Frequency (MHz):

Bitrate (kbps):

Modulation:

Search

Figure 2.3.. Home page of NetQuack

2.5.2. Design

React is a framework primarily designed for building user interfaces. Its basic blocks are components: a component is a JavaScript class with a `render()` method which returns an HTML representation of the component. This representation is not static, but it is generated through some internal logic, so that a component can change the visual aspect during its lifetime. Each component has some properties, which can be classified either as **props** or **state**: props are read-only values fixed during the initialization of the component, while state is a writable object. Every time the state object changes, the `render()` method is automatically called, thus the graphical aspect on the web page is updated. Moreover, a component can contain other components in their HTML representation, creating a hierarchy tree.

This pattern in the strength of React and it allows the creation of dynamic interfaces with little effort. The limitation of this workflow is that state management is not easy to handle: state can naturally flow from parent components to children ones, while the opposite direction is not immediate and can be done using callbacks. For this reason, React is generally coupled with other libraries (in particular Redux) for state management. However in our case, we have only a few variables to keep track of, and therefore we decided to rely only on React without any support for state management. Of course, this scenario can change in the future if the web interface will need to host new features.

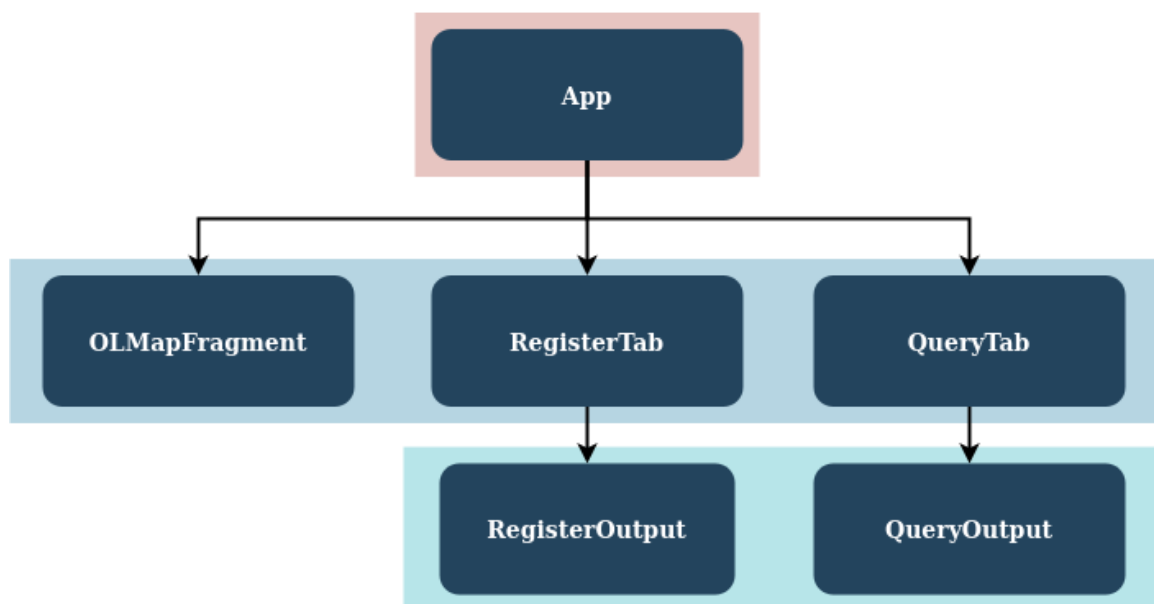


Figure 2.5.. The hierarchy of React components used in the Web interface of NetQuack

The web interface of NetQuack has six components and they are arranged as in Figure 2.5.

- **App**: the root component. Every React application has a root component which is commonly called App.
- **OLMapFragment**: it contains the map and all variables necessary to describe it, such as the list of markers.
- **RegisterTab**: the form for registering a new dongle.
- **RegisterOutput**: the component which hosts the response after submitting the form in RegisterTab.
- **QueryTab**: the form for querying.
- **QueryOutput**: the component which contains the query results.

The application can be hosted on any web server, which ideally just needs to provide the web page, while all requests to the API are done by calling directly the API itself without using the webserver as an intermediary. Unfortunately, this is true only for query requests, while the registration of a new dongle is actually performed using the web server as a proxy. This happens so that a user can obtain a zip file with all the needed certificates as a response. However, this is not an obstacle for scalability: registering a dongle is an operation performed sporadically and it is hardly executed by many users concurrently. On the other hand, querying is a more intensive operation and it is likely to be performed concurrently, but this operation is requested directly to the API without the participation of the web server, which then does not act as a bottleneck.

2.6. Summary

In this chapter, we have discussed in detail the implementation of NetQuack and its internal workings, with a major interest in the overall architecture. Now we are aware of the path undertaken by a radio packet, from being captured in the air until being stored in a cloud database. The backend system is publicly available and can be replicated by anyone owing an AWS account through a Terraform declaration file. After the creation of a working system, we want to know how it behaves in a realistic scenario and this is achieved by some experimental analysis that we will present in the next chapter.

3. Experimental analysis

During the development of NetQuack, we had to keep in mind that our goal was not only the creation of a working system, but also the creation of a system able to scale for thousands of nodes, each one receiving thousands of packets every second. To demonstrate this achievement, we performed a scalability analysis where we stressed the system by sending to it huge quantities of fake packets. The results are encouraging and by repeating the stress test with different parameters we have been able to describe parametrically the usage of each cloud service.

The other aspect to consider when developing a cloud infrastructure is the monetary cost and its effective workload. Also in this case, we have set an experiment with a real dongle receiving packets for a specific period of time and then calculated the effective usage of cloud services along with their cost. The results are not precise, since the crowding of a frequency band varies depending on the location, but it is still a reasonable way to know the order of magnitude of the real usage of each resource, making it possible to predict the cost of the system.

3.1. Cost analysis

As the first experiment we wanted to perform on the system, it concerns the estimation of the workload for a dongle looking for real packets and the expected cost of storing these data. Before describing the experiment and its results, it is necessary to know how much each service in AWS is paid and how to calculate the expected cost. AWS services are paid on demand and the cost of each service can depend on many parameters. The detailed description of these costs is listed in Table 3.1. These data are valid at the time of writing for the region Europe (Frankfurt), which is the one we worked on.

Table 3.1.. Cost overview of AWS services

Service	Cost
S3	
Storage	\$ 0.023 for each GB
PUT requests	\$ 0.0054 for 1000 requests
GET requests	\$ 0.0043 for 10000 requests
Lambda	
Requests	\$ 0.20 for 1 million requests
Execution	\$ 0.0000166667 for second and GB
Athena	
Query	\$ 5 for GB of scanned data
DynamoDB	
Storage	\$ 0.306 for each GB
Read Units	\$ 1.525 for 1 million read units
Write Units	\$ 1.525 for 1 million write units
IoT Core	
Deploy devices	\$ 0.12 every 1000 registered devices
Connectivity	\$ 0.096 for 1 million minutes of connection
Messages	\$ 1.20 for 1 million messages
Rules	\$ 0.18 for 1 million activated rules
Actions	\$ 0.18 for 1 million actions executed
Kinesis	
Data ingestion	\$ 0.033 for each GB of data ingested
Conversion	\$ 0.02 for each GB of data converted

The majority of the services listed above have a cost directly dependent on the quantity of ingested packets, while only a few are related to registering devices and querying data, in particular Athena, storage and write units in DynamoDB, and the deployment of services in IoT Core. For this reason, those services are not considered by our experiment. For estimating the system cost, we have deployed an RFQuack node powered by an ESP32 and equipped with a CC1101. We then run the Automatic Frequency and Bitrate detection module eight hours a day for seven days and we registered the consumption of each service. At

the end of the week, we translated these consumptions into a total cost. During this week, we have stored 158444 packets into 221 files in S3 and a total of 36 MB, while the cost of this operation amounts to \$ 0.31. The detailed results can be read in Table 3.2.

Table 3.2.. Cost analysis for one dongle running eight hours a day for seven days

Service	Consumption	Cost
S3		
Storage	36 MB	\$ 0.000828
PUT requests	221 requests	\$ 0.001193
Lambda		
Requests	427 requests	\$ 0.000085
Execution	326 GB/seconds	\$ 0.00543
DynamoDB		
Read Units	110.5 read units	\$ 0.000169
IoT Core		
Connectivity	3360 minutes	\$ 0.000323
Messages	158,444 messages	\$ 0.19
Rules	158,444 rules	\$ 0.02825
Actions	158,444 actions	\$ 0.02825
Kinesis		
Data ingestion	1.143 GB	\$ 0.03772
Conversion	1.143 GB	\$ 0.02286
Total	—	\$ 0.31

From these results, it is visible that the major contribution to cost is given primarily by the number of messages exchanged with the MQTT broker. Another notable portion of the cost is given by Kinesis Firehose and the other parameters of IoT Core. All these metrics are directly proportional to the number of received packets, so they vary based on the crowding of the radio spectrum. Conversely, the contribution given by the storage on S3 and the execution of Lambda functions is negligible. In the case of storage, this is the outcome of using a highly compressed format such as Apache Parquet. In fact, this same experiment has been done previously also with a different configuration of AWS services where packets were stored in plain JSON without any transformation. The results can be seen in Table 3.3.

In this case, the total cost is \$ 1.11. In this second experiment it is worth noting that, in front of a quadrupled number of packets, the cost of storage in S3 is more than ten times higher due to the lack of compression. This can seem a little achievement in terms of absolute value, however there are two facts to consider: storage cost is the only metric which does not depend on the crowding of the radio spectrum, conversely it is a fixed cost which becomes bigger and bigger with time

Table 3.3.. Cost analysis of a previous version of NetQuack

Service	Consumption	Cost
S3		
Storage	466 MB	\$ 0.010718
PUT requests	662 requests	\$ 0.003575
GET requests	331 requests	\$ 0.000142
Lambda		
Requests	338 requests	\$ 0.000068
Usage	428 GB/seconds	\$ 0.007133
DynamoDB		
Read Units	165.5 read units	\$ 0.000252
IoT Core		
Connectivity	5040 minutes	\$ 0.000484
Messages	633,305 messages	\$ 0.76
Rules	633,305 rules	\$ 0.114
Actions	633,305 actions	\$ 0.114
Kinesis		
Data ingestion	3 GB	\$ 0.099
Total	—	\$ 1.11

since the database accumulates data stored indefinitely. Hence lowering the cost of storage is an important goal in the long term standpoint. Secondly, the Athena query service is paid for gigabytes of data scanned: if data are compressed, a query would cost less than the same query performed on uncompressed data. This analysis explains quantitatively why Apache Parquet has been a reasonable choice when configuring Kinesis Firehose.

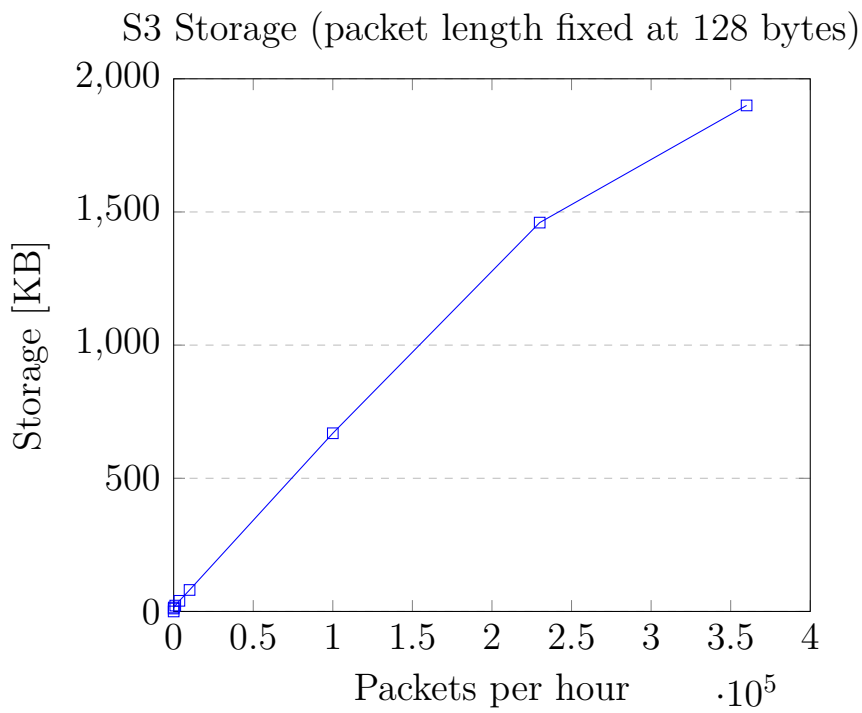
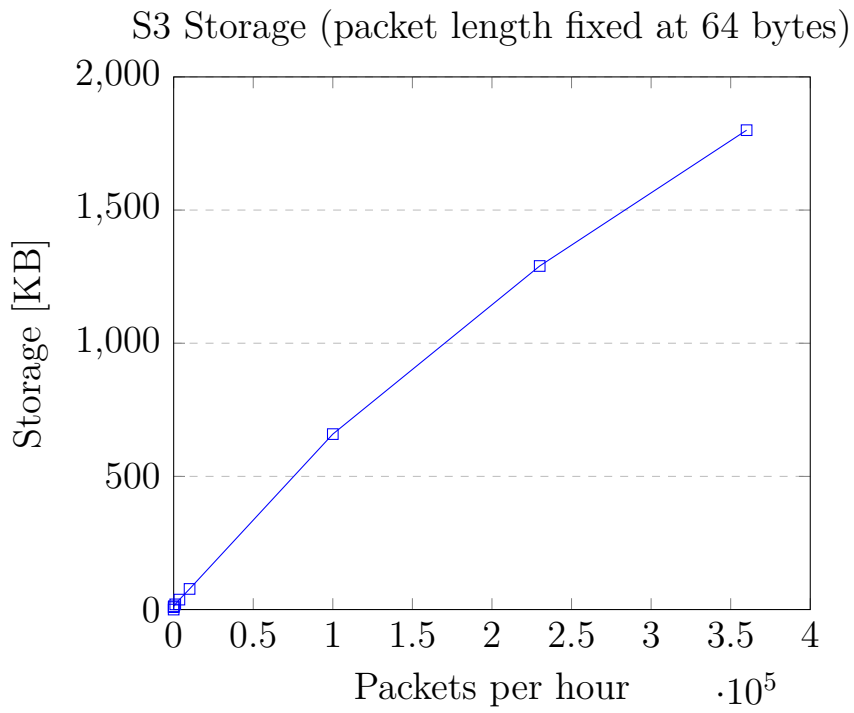
3.2. Scalability analysis

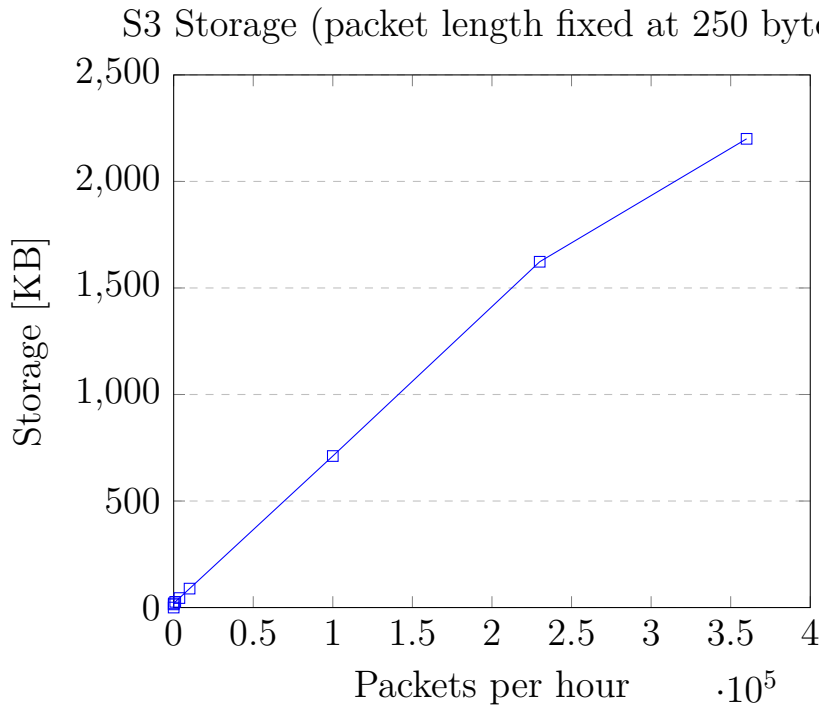
In order to measure how the system behaves in front of a variable workload, we have performed a scalability analysis which consists of sending an intended quantity of radio packets and measuring the use of each service as a response. The purpose is to understand how the utilization of each service grows as a function of the number of received packets until its upper limit. The parameters which determine the workload are mainly the rate of received packets for an hour, but also the average length of the packets and the number of deployed dongles. In our experiment, we deployed a dongle and set it up to send a specific quantity of packets in the span of one hour. We repeated this experiment multiple times changing the length of the packets and their total quantity by different orders of magnitude. Hence the results can be interpolated to extract a function of packet length and packet rate. On the other

hand, measuring the impact of the number of dongles is harder, since we do not have physically the possibility to deploy hundreds or thousands of dongles to repeat the experiment. However, we have good reasons to state that the number of dongles has a limited impact on the resources, as will be explained later.

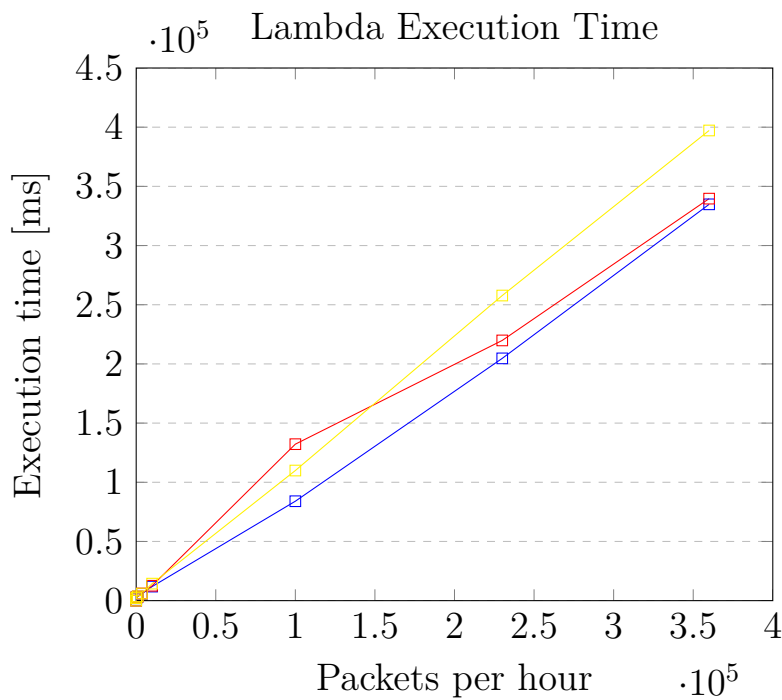
The experiment has been performed with a wide variety of packet rates, in particular 10, 100, 1000, 3600, 10000, 100000, 230000, and 360000 packets per hour. Each one of these rates has been tested with three different packet sizes, which are 64 bytes, 128 bytes, and 250 bytes. The raw numerical results representing the activity of cloud resources for each experiment are available in Appendix A.

If we plot the results into some graphs, we can easily see the relationship between our fixed parameters (packet rate and packet size) and the consumption of each service. In particular, the graphs below show the storage space in S3 in kilobytes as a function of packet rate and size after an hour of execution. Of course, the packet size increases storage by a little factor, but the interesting fact is that storage space has a slightly sublinear increment with regard to packet rate. The number of PUT requests is harder to determine: according to the architecture of NetQuack, a PUT operation on S3 is performed every time the buffer in Kinesis Firehose reaches a timeout of 15 minutes or it is filled with 128 MB of data. In our experiments we never reached the point of filling the whole buffer with 128 MB in less than 15 minutes, so ideally we should execute only 4 PUT requests in an hour. As we can see from the results, however, packets are often split into 5 or 6 files instead of 4. This happened for two reasons: firstly the experiments did not have an exact duration of one hour, but they likely took one hour plus a few seconds or minutes. This is enough for having one more PUT request. Secondly, it looks like Kinesis Firehose has some internal logic which causes some packets to be stored into S3 in the next batch instead of the current one, probably for some optimization reasons. This explains the presence of 6 PUT requests in one hour. The only way to have more than 6 PUTs in one hour is to fill the buffer with more than 128 MB: it can be done roughly with at least one million packets per hour. Some limitations imposed by AWS (which will be explained later) did not allow us to test this possibility.

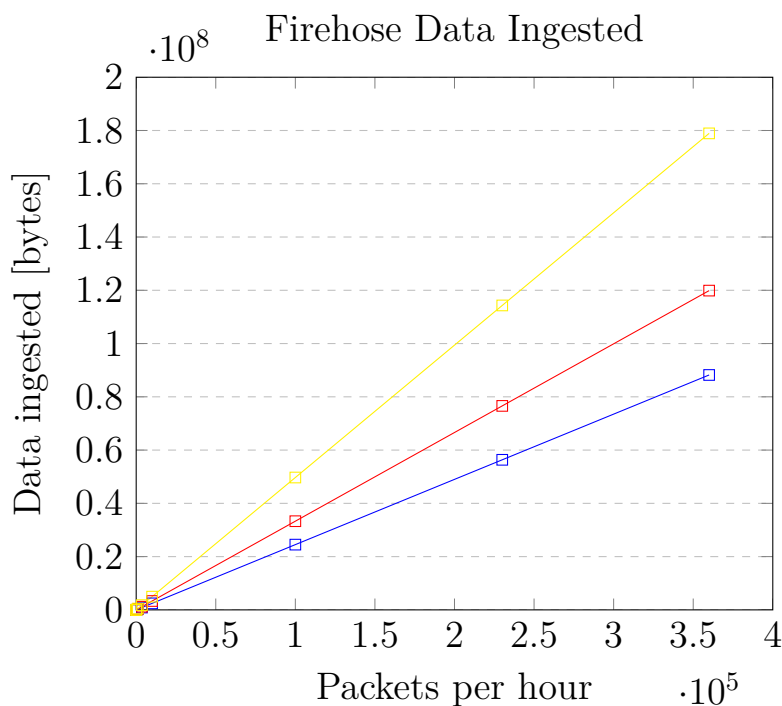




The next graph plots the execution time of a Lambda function and the number of requests to it. The function which is executed during the data collection phase is the one employed to transform packets from the Protobuf format into JSON. Differently from the PUT operation, this Lambda function is called every time the buffer of Kinesis Firehose is filled with only 3 MB of data. This explains why the function is called only five times when the rate is below 100000 packets per hour: it never fulfills 3 MB in less than 15 minutes. Then it starts being executed more often, with the number of invocations increasing linearly with the packet rate. It is worth noting that the read units of DynamoDB are exactly half the number of executions of the Lambda function. This is explained by the fact that the transformation routine needs calling DynamoDB to tag each packet with the geographic location of the dongle which detected it. Since in our experiment, we used only one dongle, and each call to DynamoDB requires a half-read unit, this explains why the read units of DynamoDB are half the number of Lambda invocations. This is the only scenario where the consumption of a service depends directly on the number of deployed dongles. Despite we could not measure it experimentally, we can infer that the consumed read units of DynamoDB are half the number of Lambda invocations multiplied by the number of deployed dongles.



Finally, the last graph is related to data ingested by Kinesis Firehose. In this case, there are no particular surprises. The quantity of data processed by Kinesis Firehose is calculated deterministically when the number of received packets and their size is known and fixed. The same is valid for the quantity of data converted into Apache Parquet: they coincide with the totality of received data.



Another service that is consumed when collecting data is IoT Core. However, we have not listed it in the previous tables because it is completely deterministic. The consumption of IoT Core is based on four parameters. Three of these (namely the number of messages, the number of triggered rules, and the number of executed actions) coincides with the number of packets received. The fourth parameter is the connection time to the broker: also in this case, the value is calculated analytically. If the experiment lasts one hour and we have N dongles deployed, the connection time consumed is just N hours.

The results we have seen so far rely on only one measurement, which can make them look not precise. However, there are some aspects to consider. First, each measurement requires one hour of sending data: repeating the same experiment more times (for example ten times) would require a whole day just for filling one column of those tables. Secondly, we know that the majority of those metrics are deterministic or they have a little variance, hence repeating the same experiment multiple times would bring almost identical results. The only metrics which can change in front of identical conditions are the execution time of the Lambda routine and, to a lesser extent, the storage space, since data are compressed and the compression rate depends on the internal structure of the received packets. To demonstrate this and make the experiment more robust, we repeated the experiment ten times only in the worst case, that is a packet rate of 360000 packets per hour and a size of 250 bytes. In Table 3.4 we can see a comparison between the real results and the expectation based on the previous measurement.

Table 3.4.. Comparison between expected and real results

Packets per hour	Expected	Real
S3		
Storage [KB]	7112	6957
PUT requests	45	44
DynamoDB		
Read Units	130	129,5
Lambda		
Requests	260	259
Execution time [ms]	1,099,280	1,113,700

Finally, we want to explain the reason why the maximum rate used in the experiments is 360000 packets per hour. This is a limit imposed by AWS and stated in their documentation. If we explore these limits further, we discover that:

- The MQTT broker handles up to 20000 messages per second by all connected dongles.
- A single device can transmit up to 100 messages per second.
- IoT Core can evaluate up to 20000 rules per second.
- The MQTT broker can manage 500000 simultaneous connections.
- Kinesis Firehose can store in its buffer no more than 1000 messages per second.

If each message represents a packet, we can see that a single node can transmit no more than 360000 packets in one hour, while the whole network of nodes is bounded by 72 million packets per hour. However, Kinesis Firehose can not handle this traffic and it is limited by 3,6 million messages per hour. Messages exceeding this limit are just discarded. Nevertheless, there are ways to increase this limit: if the services are placed in a region among US East (Virginia), US West (Oregon), and Europe (Ireland), the limit for incoming packets in Kinesis Firehose is by default 5000 messages per second. This translates to 18 million packets per hour. A further increase of this limit is possible, but it must be requested manually to the Support Center of Amazon Web Services.

4. Conclusions and future works

In this conclusive chapter, we evaluate the results achieved by this project and compare them with the goals we set at the beginning of the thesis. We also discuss its limitations and propose some future work to improve the effectiveness of NetQuack and its environment.

4.1. Future works

In this section, we highlight some limitations of our approach and propose some solutions that could be further investigated.

- **Variable number of fields for a packet:** so far packets are stored in an S3 bucket with a tabular format which resembles the tables of relational databases. In other words, each packet has a fixed set of attributes along with their type, as it happens in a SQL-like database. This is the drawback of a service like AWS Athena, which makes it possible to query data stored on files, but it expects these data to have a fixed structure. If in the future we want to add new fields for newly stored packets, we need a new bucket and a new table to query them. An improvement for this scenario is the automatic handling of different "versions" of packet structure to lighten the impact of this limitation.
- **Noise filtering:** upon testing the correct functioning of NetQuack with a real dongle, we often received many packets which are actually just noise. While the majority of these packets can be removed simply by raising the RSSI threshold level on the dongle, this is not enough in all situations, since it also happened that the fake packets were the result of a weak but existing signal which resulted in packets composed only by `0xFF` or `0x00` bytes. A way to detect possible fake packets would be great to avoid polluting the database with useless data, especially if performed directly on the RFQuack firmware, since it would avoid sending a message which uselessly consumes bandwidth and IoT Core resources. A general approach could be based on payload entropy, however, a more specific solution would require in-depth analysis and further experimentation.
- **Cost and resource optimization:** from the analysis we performed about cost and scalability, we discovered that the number of sent messages constitute the majority of the system cost and they represent the main limit to the overall scalability. Since at the current state each MQTT message is used to send a single packet, a possible improvement in the RFQuack firmware would be grouping more packets in a single message if they are received in a short amount of time. This optimization would reduce the cost and further improve the scalability of the system, however

it should not limit those scenarios where packets must be received immediately, something usually required in cybersecurity use cases.

- **Integration of other modulations:** currently the automatic signal detection routine of RFQuack, which takes care of discovering and sending packets to the NetQuack infrastructure, works only with signals modulated in OOK. An improvement that would benefit the usefulness of NetQuack for example is the recognition of signals in other modulations, such as the common 2-FSK. Another notably useful improvement would be the compatibility with the LoRa modulation, which would allow NetQuack to receive packets generated by the LoRaWAN network, attracting considerably the interest of researchers.

4.2. Conclusions

We began this thesis by discussing the main tools used by cybersecurity researchers in the field of radiofrequency analysis and presenting some examples of applications based on distributed data collection for cybersecurity purposes. We wanted to create a large-scale catalog of digital radio signals fed by a network of radios and as a first step, we were looking for the best choice to implement this network. We have chosen RFQuack due to its automatic signal recognition routine, which transforms every dongle in a crawler of RF signals. After this important step, we have designed and built the backend architecture intended for storing and querying these data coming from the network. Then we made this backend accessible through a public API and we built a handy web interface that makes use of its services. Once the whole infrastructure has been created, we measured experimentally the cost of its maintenance and its upper limits in terms of scalability. The results are encouraging and demonstrated that the system can already handle a large quantity of data. The experiments also suggested some aspects that could be improved to reach even higher efficiency and scalability. We believe this system can evolve into a vast network of nodes scanning the spectrum and contributing to a database that will be useful for researchers, while many enthusiasts will join the project and give their contribution.

A. Appendix

Table A.1.. Scalability analysis results for 64 bytes long packets (Part 1)

Packets per hour	10	100	1000	3600
S3				
Storage [KB]	9	12.3	19.5	37.4
PUT requests	4	5	5	5
DynamoDB				
Read Units	2	2.5	2.5	2.5
Kinesis				
Data ingestion [byte]	2450	24,500	245,000	882,000
Conversion [byte]	2450	24,500	245,000	882,000
Lambda				
Requests	4	5	5	5
Execution time [ms/req]	500	543	685	1157
Execution time [ms]	2000	2715	3425	5785

Table A.2.. Scalability analysis results for 64 bytes long packets (Part 2)

Packets per hour	10000	100,000	230,000	360,000
S3				
Storage [KB]	77.4	658.8	1290	1800
PUT requests	5	6	6	6
DynamoDB				
Read Units	2.5	7	17	28
Kinesis				
Data ingestion [byte]	2,450,000	24,500,000	56,350,000	88,200,000
Conversion [byte]	2,450,000	24,500,000	56,350,000	88,200,000
Lambda				
Requests	5	14	34	56
Execution time [ms/req]	2376	6000	6020	5979
Execution time [ms]	11880	84000	204680	334824

Table A.3.. Scalability analysis results for 128 bytes long packets (Part 1)

Packets per hour	10	100	1000	3600
S3				
Storage [KB]	11	14.8	22	39.9
PUT requests	4	5	5	5
DynamoDB				
Read Units	2	2.5	2.5	2.5
Kinesis				
Data ingestion [byte]	3300	33,300	333,000	1,198,800
Conversion [byte]	3300	33,300	333,000	1,198,800
Lambda				
Requests	4	5	5	5
Execution time [ms/req]	5040	554	702	1213
Execution time [ms]	2160	2770	3510	6065

Table A.4.. Scalability analysis results for 128 bytes long packets (Part 2)

Packets per hour	10000	100,000	230,000	360,000
S3				
Storage [KB]	80.6	669.1	1460	1900
PUT requests	5	5	6	6
DynamoDB				
Read Units	2.5	12.5	20.5	33
Kinesis				
Data ingestion [byte]	3,330,000	33,300,000	76,590,000	119,880,000
Conversion [byte]	3,330,000	33,300,000	76,590,000	119,880,000
Lambda				
Requests	5	25	41	66
Execution time [ms/req]	2517	5288	5362	5146
Execution time [ms]	12585	132,200	219,842	339,636

Table A.5.. Scalability analysis results for 250 bytes long packets (Part 1)

Packets per hour	10	100	1000	3600
S3				
Storage [KB]	14.9	19.6	26.8	44.6
PUT requests	4	5	5	5
DynamoDB				
Read Units	2	2.5	2.5	2.5
Kinesis				
Data ingestion [byte]	4970	49,700	497,000	1,789,200
Conversion [byte]	4970	49,700	497,000	1,789,200
Lambda				
Requests	4	5	5	5
Execution time [ms/req]	510	539	728	1200
Execution time [ms]	2040	2665	3640	6000

Table A.6.. Scalability analysis results for 250 bytes long packets (Part 2)

Packets per hour	10000	100,000	230,000	360,000
S3				
Storage [KB]	88.9	711.2	1623	2200
PUT requests	5	6	6	6
DynamoDB				
Read Units	2.5	13	29,5	45,5
Kinesis				
Data ingestion [byte]	4,970,000	49,700,000	114,310,000	178,920,000
Conversion [byte]	4,970,000	49,700,000	114,310,000	178,920,000
Lambda				
Requests	5	25	41	66
Execution time [ms/req]	2876	4228	4369	4365
Execution time [ms]	14,380	109,928	257,771	397,215

Acronyms

ASK	Amplitude Shift Keying
FSK	Frequency Shift Keying
PSK	Phase Shift Keying
OOK	On-Off Keying
MQTT	Message Queuing Telemetry Transport
QoS	Quality of Service
SDR	Software Defined Radio
URH	Universal Radio Hacker
IQ	In-phase and quadrature components
MCU	Micro Controller Unit
AWS	Amazon Web Services
S3	Simple Storage Service
CORS	Cross-Origin Resource Sharing

Bibliography

- [1] Degeler Andrii. 13 million mackeeper users exposed after mongodb door was left open. <https://arstechnica.com/security/2015/12/13-million-mackeeper-users-exposed-after-mongodb-door-was-left-open/>
- [2] Eric Blossom. *GNU radio: tools for exploring the radio frequency spectrum*, page 4. Linux journal 2004.122, 2004.
- [3] Bootstrap. Bootstrap 4 components built with react. <https://github.com/react-bootstrap/react-bootstrap>.
- [4] Censys. Censys: Your attack surface is evolving... and growing. <https://www.censys.io/>.
- [5] Comthings. Pandwarf: one rf tool to (almost) rule them all. <https://pandwarf.com/>.
- [6] Joachim Tapparel et al. An open-source lora physical layer prototype on gnu radio. *12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2020.
- [7] LeeHur Shing et al. Vulnerabilities of radio frequencies. *12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2015.
- [8] Simon A. Eugster. Exchanged packets of an mqtt connection with qos = 0. https://commons.wikimedia.org/wiki/File:MQTT_protocol_example_without_QoS.svg.
- [9] Facebook. React: Javascript library for building user interfaces. <https://github.com/facebook/react>.
- [10] Great Scott Gadgets. Hackrf: open source hardware for software-defined radio. <https://greatscottgadgets.com/hackrf/>.

- [11] Great Scott Gadgets. Yard stick one: a sub-1 ghz wireless test tool controlled by your computer. <https://greatscottgadgets.com/yardstickone/>.
- [12] Travis Goodspeed. Promiscuity is the nrf24l01+'s duty. <http://travisgoodspeed.blogspot.com/2011/02/promiscuity-is-nrf24l01s-duty.html>.
- [13] Andrea Guglielmini. Rfquack: A research platform for radio frequency security analysis. 2020.
- [14] Texas Instruments. Cc1101 - low-power sub-1 ghz rf transceiver. <https://www.ti.com/lit/ds/symlink/cc1101.pdf>.
- [15] Samy Kamkar. Drive it like you hacked it: New attacks and tools to wirelessly steal cars. <https://samy.pl/defcon2015/2015-defcon.pdf>, 2015.
- [16] Hill Kashmir. Camera company that let hackers spy on naked customers ordered by ftc to get its security act together. <https://www.forbes.com/sites/kashmirhill/2013/09/04/camera-company-that-let-hackers-spy-on-naked-customers-ordered-by-ftc-to-get-its-security-act-together/>.
- [17] Marc Newlin. Injecting keystrokes into wireless mice. 2016.
- [18] OpenLayers. Openlayers: a high-performance, feature-packed library for creating interactive maps on the web. <https://github.com/openlayers/openlayers>.
- [19] pcapr. pcapr - a collection of pcaps. <https://www.pcapr.net/>.
- [20] Johannes Pohl and Andreas Noack. *Universal Radio Hacker: A Suite for Analyzing and Attacking Stateful Wireless Protocols*, page 6. Proceedings of the 12th USENIX Conference on Offensive Technologies, 2018.
- [21] RfCat Project. Rfcats - swiss-army knife of ism band radio. <https://github.com/atlas0fd00m/rfcats>.
- [22] RFQuack Project. Rfquack - the versatile rf-analysis tool that quacks! <http://rfquack.it>.
- [23] RTL-SDR. Rtl-sdr (rtl2832u) and software defined radio news and projects. <https://www.rtl-sdr.com/>.

- [24] Amazon Web Services. Aws: a scalable cloud infrastructure. <https://aws.amazon.com/>.
- [25] Shodan. Shodan: The search engine of the internet of things. <https://www.shodan.io/>.
- [26] WiGLE. Wigle: All the networks, found by everyone. <https://www.wigle.net/>.