



University of Genoa

Politechnic School

Ph.D. School “Science and Technology for Electronic and
Telecommunications Engineering”

Ph.D. Course “Electromagnetism, Electronics,
Telecommunications”

Doctoral Thesis

**Intrusion Detection System
based on time related features
and Machine Learning**

Coordinator:

Prof. MAURIZIO VALLE

Author:

ALESSANDRO FAUSTO

Advisor:

Prof. MARIO MARCHESE

XXXIV Cycle

Questo lavoro è dedicato ai miei Genitori, a mio fratello, a mia moglie, a sua figlia e a tutte le persone che mi hanno supportato e sopportato in questi tre anni applicati alla ricerca scientifica.

Un sentito ringraziamento va al mio tutore scientifico che ha riposto molta fiducia in me e nelle mie capacità tecniche.

Desidero anche ringraziare tutte le persone che mi hanno permesso di usufruire della possibilità di usufruire dell'aspettativa per dottorato con assegno, mantenendo il mio stipendo e il trattamento pensionistico che altrimenti avrei perso per tre anni.

L'amore è la condizione in cui il benessere e la felicità di un'altra persona è essenziale alla tua stessa felicità.

Robert Anson Heinlein

"Cento pagnotte riportate indietro"

... per le nostre povere Armi a piedi oggi è stata una brutta giornata, sia per il tempo brutto, sia perchè dovettero combattere accanitamente ma morendo assai, perchè noi questa mattina abbiamo avuto 100 e più pagnotte che hanno portato indietro dalle prime linee, dei nostri Bersaglieri che erano restati morti.

Artigliere Antonio Grasso - Gorizia, settembre 1916

This work is dedicated to my parents, my brother, my wife, her daughter and all the people who supported and endured me in these three years applied to scientific research.

A special thanks goes to my scientific tutor who trust in me and in my technical skills.

I would also like to thank all the people who allowed me to enjoy the PhD staying on payed leave of absence maintaining also my pension that I would otherwise have lost for three years.

Love is that condition in which the happiness of another person is essential to your own.

Robert Anson Heinlein

"One hundred loaves brought back" ... for our poor armies on foot today was a bad day, both for the bad weather, and because they had to fight hard but dying a lot, because this morning we had 100 or more loaves that they brought back from the front lines, of our Bersaglieri who had remained dead. ...

Gunner Antonio Grasso - Gorizia, september 1916

Abstract

The analysis of the behavior of network communications over time allows the extraction of statistical features capable of characterizing the network traffic flows. These features can be used to create an Intrusion Detection System (IDS) that can automatically classify network traffic. But introducing an IDS into a network changes the latency of its communications. From a different viewpoint it is possible to analyze the latencies of a network to try to identifying the presence or absence of the IDS. The proposed method can be used to extract a set of physical or time related features that characterize the communication behavior of an Internet of Things (IoT) infrastructure. For example the number of packets sent every 5 minutes. Then these features can help identify anomalies or cyber attacks. For example a jamming of the radio channel. This method does not necessarily take into account the content of the network packet and therefore can also be used on encrypted connections where is impossible to carry out a Deep Packet Inspection (DPI) analysis.

Contents

Abstract	iii
List of Figures	vii
List of Tables	xiii
List of listings	xiv
List of acronyms	xv
List of special terms	xxi
1 Introduction	1
2 Analysis of delay added by an IDS	3
2.1 Introduction	3
2.2 Intrusion Detection System	4
2.3 State of the art IDS and SDN	4
2.4 Creation of the prototype SDN-SF-IDS	8
2.5 Description of the implemented SDN-SF-IDS infrastructure	9
2.6 Delays introduced on the network monitored by SDN-SF-IDS	12
2.7 Minimize delays	13
2.8 Auxiliary scripts to support experiments	41

2.9	Final prototypes	49
2.10	Delay measurement ΔT_{TOT}	52
2.10.1	Method of measurement	52
2.10.2	Loading data	52
2.10.3	Measurement graphs	53
2.10.4	System A_2B_5 (hardware SDN switch)	54
2.10.5	Measurements of packet forwarding delay SDN software B_4 switch in bridge mode	55
2.10.6	System delay measurements A_2B_4	55
2.11	Conclusions	62
2.12	Future work	64
3	LoRa gateway IDS	66
3.1	Introduction	66
3.2	IoT infrastructure and LoRaWAN	67
3.3	LoRa packet forwarder analysis	69
3.4	LoRa forward protocol	69
3.4.1	Upstream protocol	73
3.4.2	Downstream protocol	80
3.4.3	Information extracted from network packet analysis	86
3.5	Analysis of captured data	86
3.5.1	Monitoring the connection between LoRa Gateway and LoRa Server	87
3.5.2	Monitoring the connection between end nodes and LoRa Gateway (and LoRa Server)	88
3.6	Software Defined Radio and LoRa physical layer	95
3.7	LoRa signal jamming testbed	96
3.7.1	Software Defined Radio useful tools	114

3.7.2	LoRa temperature sensing IoT demoboard	116
3.8	Analysis of features during jamming	121
3.9	LoRa IDS to detect RF jamming	123
3.10	Possible industrial applications	138
3.11	Related work	142
3.12	Conclusion	145
3.13	Future work	146
4	Conclusions	148
4.1	Future Work	150

List of Figures

2.1	Functional blocks of the architecture making up the SDN-SF-IDS. . .	5
2.2	Logical structure of the prototype	9
2.3	Identification parameters of an UDP or TCP/IP flow	10
2.4	IP packet header	10
2.5	TCP packet header	11
2.6	UDP packet header	11
2.7	Messages exchanges inside SDN-SF-IDS	13
2.8	Connection of SDN-SF-IDS with the Test system used to measure delays	15
2.9	Timing diagram of the Openflow message exchange following the receipt of the 1 st packet (P_1) of a new P flow	16
2.10	Timing diagram of the Openflow message exchange following the receipt of the n th packet (P_n) of the previously detected P flow . . .	17
2.11	Physical structure of the prototype during the test phases	17
2.12	Graph showing peak of new flow (first image) and corresponding avalanche effect (other images) on $\Delta T_{PacketIn}$, ΔT_{TOT} delays and classification time is greater than normal.	23
2.13	Increase in delays following a spike in new flows entering the switch	24
2.14	Packets Delays ΔT_{TOT} of prototype A_1B_1	24

2.15	Number of detected flows and corresponding analysis time (feature extraction, flow classification and SDN rule insertion)	25
2.16	Scheduling of SDN-SF-IDS feature extraction and flow classification procedure	26
2.17	Packets delays ΔT_{TOT} of the A_1B_2 prototype	29
2.18	Packets delay ΔT_{TOT} of the A_1B_3 prototype	29
2.19	Analysis time of a single "PacketIN" (IDS with array structure, prototype A_1B_1)	30
2.20	Analysis time of a single "PacketIN" (IDS with hash structure, prototype A_1B_1)	30
2.21	Simplified diagram of the interaction between ryu and the Threads dedicated to requesting and analyzing statistics	32
2.22	Sequence diagram trigger by SDN switch connection with Ryu Controller	35
2.23	Flowchart of the Ryu procedure <code>_recv_loop()</code>	36
2.24	Flowchart of the Ryu procedure <code>_send_loop()</code>	38
2.25	Flowchart of the Ryu procedure <code>_echo_request_loop()</code>	39
2.26	Result of message handling logic implemented in Ryu "recv_loop" .	40
2.27	active and terminated flows (within statistical calculation window) .	42
2.28	total delay ΔT_{TOT} of network packets	43
2.29	number of detected streams separated by type (IP/TCP/UDP) . .	43
2.30	number of True positive, False positive, True negative, False negative flows	43
2.31	Time used to analyze the statistics	43
2.32	Memory used by the OS and by Ryu	43
2.33	Network throughput of OS ethernet interfaces	44
2.34	Local MAC address format (bit G/L=0)	47

2.35 Unique identifier, minimum and maximum value.	48
2.36 logical structure of the OpenFlow rules as used by the A_1 controller	51
2.37 logical structure of the OpenFlow rules as used by the A_2 controller	51
2.38 Count of packets histogram grouped by its delays. Different color has been used for each protocols.	54
2.39 UDP packet delay histogram (network communications of a month)	57
2.40 TCP packet delay histogram (network communications of a month)	57
2.41 UDP packet delay histogram (network communications of five hours)	59
2.42 UDP packet delay histogram (network communications of two days)	59
2.43 UDP packet delay histogram (network communications of a month)	60
2.44 TCP packet delay histogram (network communications of five hours)	60
2.45 TCP packet delay histogram (network communications of two days)	61
2.46 TCP packet delay histogram (network communications of a month)	61
2.47 Throughput of OpenFlow control channel	62
3.1 Example of a LoRaWAN based IoT network	68
3.2 LoRa packet structure before complete understanding of all data field	70
3.3 LoRa packet common fields	72
3.4 Sequence diagram of upstream protocol: Push Data/Push Ack	73
3.5 Sequence diagram of upstream protocol: Pull Data/Pull Ack	73
3.6 Sequence diagram of downstream protocol: Pull Resp/TX Ack	74
3.7 Upstream protocol: Push Data Message (ID 0x00)	74
3.8 LoRa packet binary format of ["rxpk"][i]["data"] after base64 decoding	79
3.9 Upstream protocol: Push Ack message (0x01)	80
3.10 Downstream protocol: Pull Data message (0x02)	81
3.11 Downstream protocol: Pull Ack message (0x04)	81
3.12 Downstream protocol: Pull Resp message (0x03)	82

3.13 Downstream protocol: TX Ack message (0x05)	84
3.14 Round trip times between Request and Ack (logarithmic scale).	88
3.15 LoRa packets RSSI, indoor antenna.	89
3.16 LoRa packets RSSI, antenna close to a window.	89
3.17 LoRa packets RSSI in the used frequencies.	89
3.18 Utilization of LoRa frequency by device ID.	90
3.19 LoRa end node traffic	90
3.20 Use of Datar by LoRa devices.	91
3.21 LoRa gateway SNR feature for normal traffic.	92
3.22 LoRa gateway received packets.	93
3.23 LoRa gateway inter-arrival time between two consecutive messages in case of fixed periodic data transmissions.	93
3.24 LoRa gateway inter-arrival time between two consecutive messages in case of fixed periodic data transmissions under attack by reactive jammer.	94
3.25 LoRa testbed	97
3.26 Logical structure of LoRa jammer	97
3.27 GNU radio LoRa detector from radio dump file (RX)	98
3.28 Binary format used by GNU Radio to store IQ samples on file	99
3.29 Information obtained by using TShark Lora dissector on LoRa packed decoded by Gr-LoRa	100
3.30 Hexadecimal dump of LoRa packed decoded by Gr-LoRa	100
3.31 Results of VCO Jam on GNU radio LoRa software decoding	102
3.32 GNU radio LoRa jammer simulator by using WBFM or FM modu- lated signals	103
3.33 GNU radio LoRa jammer simulator results by using gr-radar Chirp module	104

3.34	Visual monitoring of trigger value stored in shared memory	105
3.35	reactive jamming in rest mode (lower image) and jam (Upper image)	106
3.36	RF Spectrum WaterFall showing both LoRa and jamming signals	107
3.37	RF Spectrum WaterFall showing the noise power level	107
3.38	RF Spectrum WaterFall showing the jamming and the power peaks of LoRa signals	108
3.39	RF spectrum of a successful or unsuccessful jam of a LoRa signal	109
3.40	Signal presence detection by using GNU radio script	109
3.41	Power of the noise detected by using GNU radio script	110
3.42	Moving average smoothing effect when signal start or end	111
3.43	GNU radio flowgraph for signal presence detection with embedded python block	113
3.44	Jamming signal multiplied by the signal present trigger output	113
3.45	Binary format used by hackrf_transfer to store IQ samples on file	115
3.46	Inspectrum showing packet captured from hackrf_transfer	116
3.47	Arduino MKR WAN 1310 LoRa board	116
3.48	Schematic of testbed board LoRa_A Release 1.00	118
3.49	Board layout of testbed LoRa_A Release 1.00	119
3.50	Fully working prototype of LoRa_A Release 1.00	120
3.51	Graph of the inter-arrival time between two consecutive packets. Notice the "holes" in periodicity due to the action of the jammer	123
3.52	Graph of the SNR feature under the jamming action	124
3.53	Scatterplot of "SNR" and "fiveMinPacketCount" features showing clusters of features in normal and anomalous states	124
3.54	Robust covariance and One Class SVM, changing contamination factor and feature group (A for Odd columns and B for even columns).	129

3.55 Isolation Forest and Local Outlier Factor, changing contamination factor and feature group (*A* for Odd columns and *B* for even columns).130

3.56 Robust covariance on feature group A (above) and group B (below). 132

3.57 Isolation Forest on feature group A (above) and group B (below). . 133

3.58 One class SVM on feature group A (above) and group B (below). . 134

3.59 Local Outlier Factor on feature group A (above) and group B (below).135

3.60 Confusion matrix for Robust covariance algorithm 136

3.61 Confusion matrix for Isolation Forest algorithm 136

3.62 Confusion matrix for One-Class SVM algorithm 136

3.63 Confusion matrix for Local Outlier Factor algorithm 137

3.64 Textual dump of LoRa IDS classification output 137

3.65 Flowchart of the Lora Jam IDS 139

3.66 potential position of services offered for IDS jamming detection . . 141

List of Tables

2.1	Testbeds A_1B_1 , A_1B_2 , A_1B_3	20
2.2	Summary of prototype A_2B_4 and A_2B_5	50
2.3	Delays ΔT_{TOT} of switch B_5 in milliseconds measured during the observation interval of one day	55
2.4	Delays ΔT_{TOT} of B_4	58
3.1	Possible items of "stat" JSON array	76
3.2	Possible items of "rxpk" JSON array	78
3.3	Possible items of "txpk" JSON object	83
3.4	LoRa frequency table	115
3.5	Considered features for the detection of RF jamming	127
3.6	Tested ML algorithms with parameters	131

List of Listings

2.1	TShark command with used parameters	45
3.1	Code extracted from signal detection Trigger block	112
3.2	Commands used to build GNU Radio version 3.8	114
3.3	Dump of Arduino demoboard serial output	122

List of acronyms

5G 5th Generation 63, 95

ANN Artificial Neural Network 7

CAD Channel Activity Detection 144

COTS Commercial off-the-shelf 143, 144

CR Carriage Return 120

CRC Cyclic Redundancy Check 75, 77, 83, 88, 94, 146, 147

CSS Chirp Spread Spectrum 95

CSV Comma-Separated Values 41, 42, 45

DL deep learning 6

DOS Denial of Service 6

DPDK Data Plane Development Kit 7, 27, 28, 62, 63

DPI Deep Packet Inspection iii

DSP Digital Signal Processing 95, 99, 146

FM Frequency Modulation 101, 143

- FPGA** Field Programmable Gate Array 7
- FSK** Frequency Shift Keying 78, 83, 95
- G/L** Global/Local 47
- GIL** Global Interpreter Lock 65
- GOOSE** Generic Object Oriented Substation Events 3, 63
- GPU** Graphics Processing Unit 7
- GRU-RNN** Gated Recurrent Unit Recurrent Neural Network 7
- HTTP** Hypertext Transfer Protocol 7
- I/G** Individual/Global 47
- I/O** Input/Output 33
- I2C** Inter Integrated Circuit 117
- IDS** Intrusion Detection System iii, viii, 1–8, 10, 11, 13, 20, 22, 30, 31, 41, 64, 67, 96, 144, 145, 149
- iForest** Isolation Forest 123, 125, 137
- IoT** Internet of Things iii, 1, 2, 64, 66–68, 72, 145, 149
- IP** Internet Protocol viii, 9, 22, 42, 43, 45, 49, 52, 56, 69, 70, 80, 99, 140
- IPC** Inter-Process Communication 33, 105
- IPS** Intrusion Prevention System 5
- IQ** I and Q Components 99, 114

- ISP** Internet Service Providers 140
- IT** Information Technology 66
- JSON** JavaScript Object Notation 70, 71, 75, 79, 82, 84, 86, 87, 127
- KPI** Key Performance Indicator 63
- LED** Light Emitting Diode 117, 140
- LF** Line Feed 120
- LOF** Local Outlier Factor 123, 126, 131, 145, 149
- LoRa** Long Range protocol 2, 67–76, 78–84, 87, 89–92, 94–99, 101, 107, 108, 114, 116, 117, 120, 121, 126, 127, 138, 140–147, 149
- LoRa-net** Long Range network 70, 71, 74, 82
- LoRaJamIDS** LoRa Jam Intrusion Detection System 123, 127, 128, 137, 138, 140–142
- LoRaWAN** Long Range Wide Area Network 1, 67–70, 95, 96, 121, 144, 145, 149
- LPWAN** Low Power Wide Area Network 67, 68, 149
- MAC** Media Access Control xxiii
- ML** Machine Learning xiii, 4–6, 8, 11, 19, 20, 24, 123, 125, 126, 128, 131, 137, 138, 146
- MoDem** Modulator-Demodulator 69, 70, 72, 73, 77, 87, 96, 120
- NIC** Network Interface Controller xxiii

NIDS Network IDS 6

OCSVM One Class Support Vector Machine 123, 125

OF OpenFlow 4, 8–12, 18, 19, 22, 24, 25, 27, 29–32, 34, 36–39, 42, 46, 49–51, 53, 55, 56, 58, 63–65

OS Operating System 8, 12–14, 17, 18, 21, 27, 42, 48, 98, 149

OTAA Over The Air Authentication 117, 120, 121

OvS Open virtual Switch 8, 9, 13, 14, 19, 21, 28, 31

PNG Portable Network Graphics 41

RBF Radial Basis Function 125

RF Radio Frequency 68, 72, 73, 85, 97, 107, 108, 116, 140, 144

rForest Random Forest 8, 11, 19, 24, 31

RNN Recurrent Neural Network 6

RRD Round-Robin Database 41, 42

RSSI Received Signal Strength Indication 78, 88–90, 144, 145, 149

RTT Round Trip Time 87, 146

RX Receiver 77, 88

SCADA Supervisory Control And Data Acquisition 3, 63

SDN Software Defined Networking 3–15, 17, 19, 21, 22, 25, 27–34, 37, 39, 41, 44, 46–50, 52, 55, 56, 62–65

- SDN-SF-IDS** Intrusion Detection System using Statistical Fingerprint based on Software Defined Networking vii, viii, 3–9, 11–15, 19–22, 25–27, 29, 31, 42, 45–47, 49, 50, 52, 62, 63, 65, 66, 145, 148, 149
- SDR** Software Defined Radio 67, 69, 95–97, 108, 116, 146, 147
- SFD** Start Frame Delimiter 96
- SFTP** Secure File Transfer Program 41
- SNR** Signal-to-Noise Ratio 78, 91, 121, 145, 149
- SPI** Serial Peripheral Interface 117
- SSH** Secure SHell 41, 138
- SVG** Scalable Vector Graphics 41
- SVM** Support Vector Machine 125
- TCP** Transmission Control Protocol viii, 9, 10, 22, 34, 42, 43, 45, 56, 58, 148
- TTL** Transistor-Transistor Logic 117
- TTN** The Things Network 67, 96
- TX** Transmitter 80, 82–85, 88
- UDP** User Datagram Protocol viii, 10, 22, 42, 43, 45, 56, 70, 72, 99, 100, 138, 141, 148
- USB** Universal Serial Bus 95, 117, 120, 146
- UTC** Coordinated Universal Time 75, 77
- VCO** Voltage Controlled Oscillator 101

WAN Local Area Network 19, 52

WAN Wide Area Network 18

WBFM Wide Band Frequency Modulation 101

WPT Wireless Power Transfer 144

List of special terms

airtime When a signal is send from a sender it takes a certain amount of time before a receiver receives this signal. This time is called airtime or Time on Air. 95

ASCIIZ ASCII string terminated using as End Of String a NUL character (ASCII code 0, byte with a value of zero, 0). They are even called C strings. 75, 84

bandpass Filter is a device that passes frequencies within a certain range and rejects (or at least attenuates) frequencies outside that range. 108

base64 is a group of binary-to-text encoding schemes that represent 8-bit binary data in an ASCII string format by translating the data into a radix-64 representation. Base64 is designed to carry data stored in binary formats across channels that only reliably support text content. 71, 79, 86

Bash Bash is a command processor that typically runs in a text window where the user types commands that cause actions. Bash can also read and execute commands from a file, called a shell script. 41

botnet Botnet (a blend of the words “robot” and “network”)A botnet is a number of Internet-connected devices, each of which is running one or more bots. Botnets can be used to perform Distributed Denial-of-Service (DDoS)

attacks, steal data, send spam, and allow the attacker to access the device and its connection. 3, 4, 11, 12, 14, 21, 52

Ethernet family of wired computer networking technologies commonly used in local area networks (LAN). It was commercially introduced in 1980 and first standardized in 1983 as IEEE 802.3. 3, 12–14, 18, 27, 28, 44–49

ethertype EtherType is a two-octet field in an Ethernet frame. It is used to indicate which protocol is encapsulated in the payload of the frame. 45, 52

frame ethernet data link layer unit of IEEE 802.3 standard. 52

greenlet Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed. Coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes. Coroutines provide concurrency but not parallelism. 33, 34

GreenThread The GreenThread class is a type of Greenlet which has the additional property of being able to retrieve the return value of the main function. GreenThread provide concurrency but not parallelism. 33, 34, 36–38

hash MD5 The MD5 message-digest algorithm is used as hash function used to map data of arbitrary size to a 128-bit hash value 44–48

Hugepages Most modern architectures provide memory pages with size larger than the normal 4KBytes. For example, x86 CPUs normally support 4K and 2M (1G if architecturally supported) page sizes, ia64 architecture supports multiple page sizes 4K, 8K, 64K, 256K, 1M, 4M, 16M, 256M and ppc64

supports 4K and 16M. This optimization is more critical now as bigger and bigger physical memories (several GBs) are more readily available. 27

IOMMU In computing, an input–output memory management unit (IOMMU) is a memory management unit (MMU) that connects a direct-memory-access–capable (DMA-capable) I/O bus to the main memory. Like a traditional MMU, which translates CPU-visible virtual addresses to physical addresses, the IOMMU maps device-visible virtual addresses (also called device addresses or I/O addresses in this context) to physical addresses. Some units also provide memory protection from faulty or malicious devices. 27

jamming Jamming is defined as the act of developing and broadcasting another electronic signal to cause interference. The attack is done on the physical layer of a Radio protocol. 1, 2

latency Time delay between the moment the input or signal is sent to the system and the moment its output is available. 6

MAC Address A Media Access Control (MAC) address is a unique identifier assigned to a Network Interface Controller (NIC) for use as a network address in communications within a network segment. This use is common in most IEEE 802 networking technologies, including Ethernet, Wi-Fi, and Bluetooth. 44, 46–49

malware Malware (a blend of the words “malicious” and “software”) is any software intentionally designed to cause damage to a computer, server, client, or computer network. 3, 4, 8, 11, 21

multithread In computer architecture, multithreading is the ability of a central processing unit to provide multiple threads of execution concurrently, sup-

ported by the operating system. This approach differs from multiprocessing. In a multithreaded application, the threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer. 64

nonce is an arbitrary number that can be used just once in a cryptographic communication. It is often a random or pseudo-random number issued in an authentication protocol to ensure that old communications cannot be reused in replay attacks. 144

pcap typical file extension used for dump files created and read by libpcap, WinPcap, and Npcap libraries and containing captured network traffic. 46, 48, 52, 96

preemptive In computer science, preemption is the act of temporarily interrupting an executing task, with the intention of resuming it at a later time. This interrupt is done by an external scheduler with no assistance or cooperation from the task. 64

pure ALOHA the time of transmission is continuous. Whenever a station has an available frame, it sends the frame. If there is collision and the frame is destroyed, the sender waits for a random amount of time before retransmitting it. 69

Python is an interpreted high-level general-purpose programming language. Its design philosophy emphasizes code readability with its use of significant indentation. Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. Python is dynamically-typed and garbage-collected. 8, 29, 41, 65

rack (19-inch) is a standardized frame or enclosure for mounting multiple electronic equipment modules. Each module has a front panel that is 19 inches (482.6 mm) wide. The 19 inch dimension includes the edges or "ears" that protrude from each side of the equipment, allowing the module to be fastened to the rack frame with screws or bolts. Common uses include computer servers, telecommunications equipment and networking hardware, audiovisual production gear, and scientific equipment. 49

scatterplot A is a type of plot using Cartesian coordinates to display values for two variables of a set of data. For each data point the position on the horizontal axis is determined by the value of the first variable and the position on the vertical axis by the other variable. 53, 121

semaphore In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions. 31

thread the smallest sequence of programmed instructions that can be managed independently by a scheduler viii, 25, 31, 32, 65

VT-d Intel's "Virtualization Technology for Directed I/O" (VT-d) isolate and restrict device accesses to the resources owned by the partition managing the device. It provides the following capabilities: • I/O device assignment: for flexibly assigning I/O devices to VMs and extending the protection and isolation properties of VMs for I/O operations. • DMA remapping: for supporting address translations for Direct Memory Accesses (DMA) from

devices. • Interrupt remapping: for supporting isolation and routing of interrupts from devices and external interrupt controllers to appropriate VMs. • Interrupt posting: for supporting direct delivery of virtual interrupts from devices and external interrupt controllers to virtual processors. • Reliability: for recording and reporting of DMA and interrupt errors to system software that may otherwise corrupt memory or impact VM isolation. 27

VT-x VT-x represents Intel's technology for virtualization on the x86 platform.

27

watchdog electronic or software timer that is used to detect and recover from computer malfunctions. Watchdog timers are widely used in computers to facilitate automatic correction of temporary hardware faults, and to prevent errant or malevolent software from disrupting system operation. During normal operation, the computer regularly restarts the watchdog timer to prevent it from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to restart the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective actions. The corrective actions typically include placing the computer and associated hardware in a safe state and invoking a computer reboot. 117

Chapter 1

Introduction

The analysis of the behaviour of network communications over time allows the extraction of statistical features capable of characterising the traffic itself. These features can be used for an automatic classification based on the peculiar intrinsic behaviours of the analyzed traffic. Furthermore, the same characteristics can be used to identify anomalies in the network operations.

The thesis is structured in two parts corresponding to Chapter 2 and Chapter 3. Chapter 2 presents the problems and the solution adopted to minimise the delays introduced on network traffic passing through an IDS based on statistical features analysis. From the analysis of the delays introduced by the IDS emerges how it is possible to use these measurements to infer the operations carried out by this system.

It is therefore possible to carry out a statistical analysis of the delays introduced by a system in a normal situation. Then keeping monitored the delays is possible to understand when the system is behaving normally or abnormally. Chapter 3 apply this approach on Long Range Wide Area Network (LoRaWAN) IoT infrastructure under radio jamming attack. Through the analysis of the LoRaWAN protocol in normal state or under attack I have proposed a series of features that characterise

the behaviour of the IoT infrastructure. I have used these proposed features to build an IDS able to detect a radio Jamming attack on the Long Range protocol (LoRa) radio communication channels.

With this thesis we want to underline how intrinsic characteristics of a network protocol sequence exchange can be used a posteriori to carry out an automatic classification of the system status and detect anomalies.

Chapter 2

Analysis of delay added by an IDS

2.1 Introduction

In this chapter I present the implementation of a IDS based on Statistical Fingerprint that exploits the already state-of-the-art Software Defined Networking (SDN) architecture Intrusion Detection System using Statistical Fingerprint based on Software Defined Networking (SDN-SF-IDS) [1]. The IDS collects traffic data and implements the algorithm presented in [1] to detect the presence of Botnet Malware within the network data traffic. The goal of this work is to reduce the delays introduced by the infrastructure itself in a network based on Ethernet in view of an application in industrial environments (for example Supervisory Control And Data Acquisition (SCADA) or Generic Object Oriented Substation Events (GOOSE) networks) in which latencies are not well tolerated. At the same time, we want to verify to what extent it is possible to reduce the delays introduced by the SDN-SF-IDS system.

The first problem to be addressed is the delay introduced by the SDN-SF-IDS system, which can hinder the practical application of the IDS. The performance analysis of the SDN-SF-IDS permit to identify and reduce system bottlenecks.

The next sub-chapters present the improvements applied to the SDN-SF-IDS infrastructure and the results of the measurements on the delay introduced by the SDN-SF-IDS infrastructure on the analyzed network packets. The implemented actions are described in detail.

2.2 Intrusion Detection System

IDS are hardware/software components or groups of devices and components designed to monitor a network or system for malicious activity. The document originally appeared in [2] introduces an IDS based on statistical analysis which, after extracting a fingerprint of network flows, uses a classifier of Machine Learning (ML) to decide whether a network flow is affected by Botnet Malware or not.

Parallel to the evolution of IDS, the need to simplify network management has led to the development of the SDN paradigm, based on the decoupling of the data plane from the control plane. The data forwarding functions are located inside devices (switches, routers, gateways) called switches SDN, while the control functions are concentrated in the SDN controllers. Communication between these two entities is handled through the OpenFlow (OF) signalling protocol. The basic idea in [1] is to implement the IDS detector for Botnet Malware in [2] inside a SDN controller by taking advantage of the packet parsing capabilities provided by a SDN switch. [1] provides the main functional blocks of the proposed architecture also shown in Fig. 2.1.

2.3 State of the art IDS and SDN

The first example of using SDN for IDS can be found in [3] where suspicious traffic was redirect taking advantage of properties of OF in an SDN environment and [4]

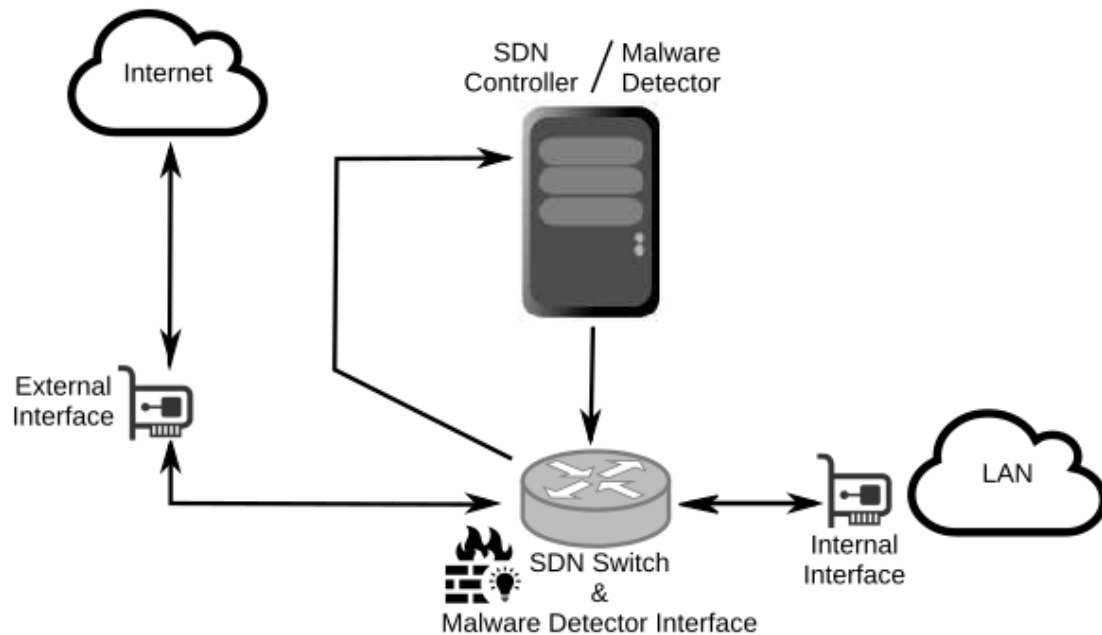


Figure 2.1: Functional blocks of the architecture making up the SDN-SF-IDS.

where the SDN switch was used to collect metadata about forwarded data.

The ML algorithms are used in many technological fields such as image analysis and classification, analysis of large amounts of data, etc. In telecommunications networks, some ML techniques have been used to classify cyber attacks and to detect anomalies. There are multiple state-of-the-art IDS methods applying ML methods for intrusion detection. [5] contains a review of using ML for intrusion detection while [6] presents a detailed overview of the technologies in the field of IDS. In [7] a cloud-based Intrusion Prevention System (IPS) based on SDN is proposed to mitigate the single point of failure problem of a classic IPS architecture. They also show that the CPU utilization factor of a classical IPS is higher than the IPS based on SDN. The author of [8] analyze a combination between SDN and IDS where IDS is deployed at a strategic location in the network and it receives the mir-

ror of the inbound and outbound traffic to and from all the devices on the network to monitor. The focus is put on the high traffic in the link between switches and IDS in this kind of IDS, which can lead to congestion in case of a DOS attack. To address this problem, they propose to use an historical database to keep track of the incident information of the sender. If the sender has no incident registered the traffic mirrored to IDS is defined as given minimum traffic. The simulations they performed shows that in the average, 54.1% of traffic mirrored to IDS is reduced compared to the original schemes. The authors of [9] They compare the Snort IDS (signature based detection) with the Bro IDS (anomaly based detection method) on a SDN network measuring the impact on throughput, delay, packet loss, CPU usage, and memory usage. The test show that Bro IDS outperform Snort IDS for throughput, delay, and packet loss parameters. However, CPU usage and memory usage on bro requires higher resource than Snort IDS.

The author of [10] present a small survey on SDN Network IDS (NIDS) and on possible ML and deep learning (DL) approaches. Another small survey on SDN Intrusion Detection System (IDS) and on possible ML is presented in [11].

A comparison between Ryu and Floodlight SDN controller is presented in [12]. The result show that Floodlight controller has higher bandwidth and lower latency than Ryu controller on a mininet emulated SDN network. A partial review of the available SDN controllers is reported in [13]. The authors measured the Latency times of various SDN controllers but do not consider the SDN controller Ryu used by our SDN-SF-IDS. Latency measurements were also performed in [14] in order to optimize response times of switch-to-controller interactions through an intelligent controller selection mechanism. Each switch adaptively selects its own controller, preferring the one with a shorter response time for flow-routing requests. The Authors of [15] propose to use Recurrent Neural Network (RNN) ML algorithm for anomaly detection in Real-Time IDS based on SDN Environments. They also

conduct a network performance analysis in terms of throughput and latency.

In [16] is proposed a Gated Recurrent Unit Recurrent Neural Network (GRU-RNN) enabled IDS for SDN. The experiment results also show that the proposed GRU-RNN does not deteriorate the network performance.

The author of [17] proposes a hybrid software/hardware IDS on SDN network that uses Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) to implement Artificial Neural Network (ANN). Different performance parameters are analyzed by changing the hardware used (i7 CPU, FPGA, GPU).

Although focused on optimizing software routers alone (not implementing SDN), [18] compares its routing solution to other solutions, addressing similar problems to those addressed in this article using Data Plane Development Kit (DPDK).

The SDN-SF-IDS is based on the statistical analysis of network traffic and in particular on the analysis of simple statistical properties of the most used communication protocols in telecommunication networks. It is assumed that the information carried by packets at the network and transport level, such as the size of the packets and the inter-arrival time between consecutive packets, is sufficient to be able to infer the nature of the application that generated the aforementioned packets and therefore to be able to distinguish between a malicious or a benign application.

This information is closely related to the operations that created it. For example, human interactions related to web browsing generate data exchanges with Hypertext Transfer Protocol (HTTP) characterised by large intervals between two successive requests. This behaviour is related to the time needed by humans to read the information contained in the web page before requesting other web pages. The same operations carried out by automatic web navigation systems (web mirroring tools, etc) while using the same communication protocol HTTP generate a faster data exchange.

The SDN-SF-IDS exploits the SDN paradigm, and in particular the ability of the SDN switches to monitor the network packets that traverse them to compute statistical features used by the IDS to characterize data flows. In this way, part of the statistics calculation is offloaded.

2.4 Creation of the prototype SDN-SF-IDS

The SDN-SF-IDS prototype consists of two parts separated logically and physically (fig. 2.2): the SDN controller and the SDN switch.

For its realization, Open Source software and Operating System (OS) GNU/Linux were used in order to be able to operate, if necessary, on the source code of each component of the system.

The SDN controller was built using Ryu Open Source software specifically modified to contain the SDN-SF-IDS developed in our laboratory. To implement it, we used the Random Forest (rForest) provided by the Python library scikit-learn¹. This algorithm was identified in [1] as one of the three best performing algorithms with respect to the metrics

$$\min(\textit{FalsePositive} + \textit{FalseNegative})$$

which minimizes errors in recognizing a Malware stream or not.

The SDN switch routes packets on the monitored network and works with the SDN controller to perform statistical analysis of the flows. The SDN switch be composed of a hardware switch or a software switch. In both cases, communication between the switch and the SDN controller takes place via the OF protocol. In the first case, a network device (switch, router, etc) is used. In the second case, a computer with OS GNU/Linux and Open virtual Switch (OvS) software is used.

¹<https://scikit-learn.org/stable/> was chosen as the ML algorithm for cataloging the extracted features

The OvS software simulates the operation of a SDN switch by managing the routing of network packets received by the interfaces assigned to it.

2.5 Description of the implemented SDN-SF-IDS infrastructure

The SDN-SF-IDS system (fig. 2.2) is made up of a SDN Switch controlled by a special SDN controller. The SDN switch manages, through special OF flow tables, the network flows received from the #1 and #2 interfaces and is connected via a third network interface dedicated to the SDN Controller who supervises its operations.

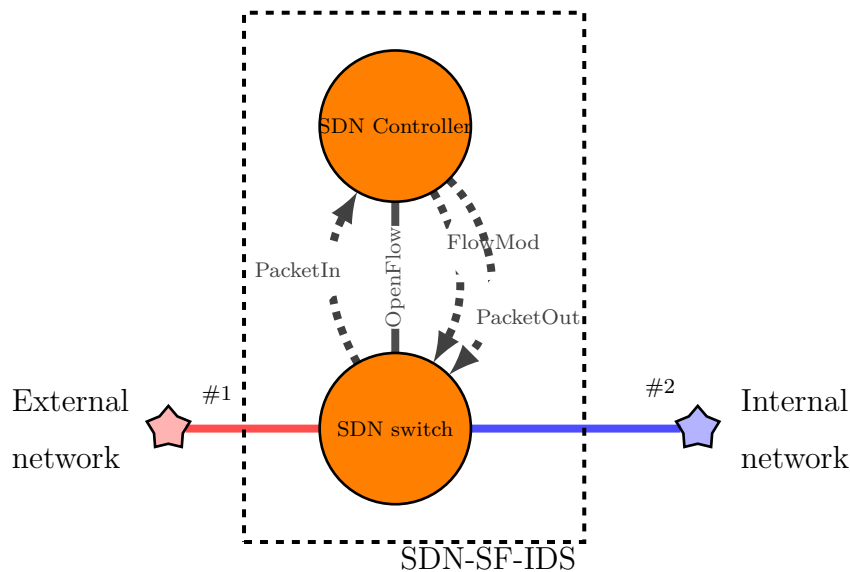


Figure 2.2: Logical structure of the prototype

Network flows are identified through the vector of following five elements (see Fig. 2.3): Internet Protocol (IP) Source and Destination (IP SRC and DST), Source and Destination Ports from Transmission Control Protocol (TCP) (fig.

2.5) or User Datagram Protocol (UDP) (fig. 2.6) headers (SRC and DST Ports), Transported protocol (6 for TCP or 17 for UDP).

Protocol (8bit)	Source IP Address (32 bit)	Destination IP Address (32 bit)	Source port number (16 bit)	Destination port number (16 bit)
--------------------	----------------------------	---------------------------------	--------------------------------	-------------------------------------

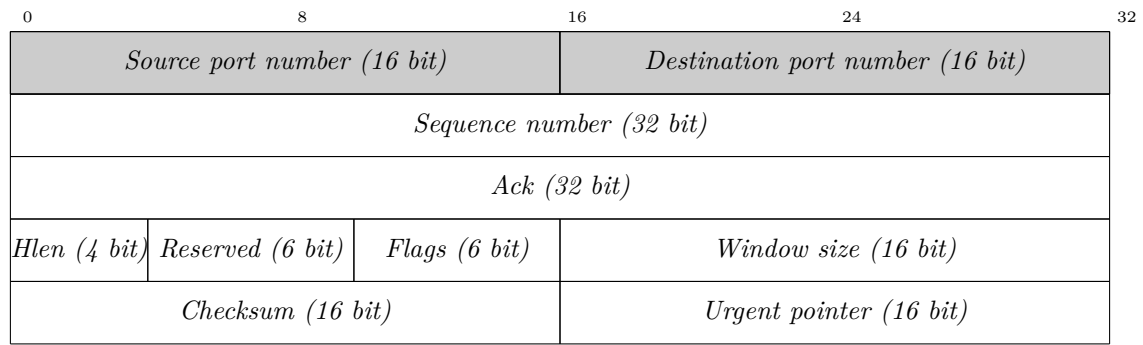
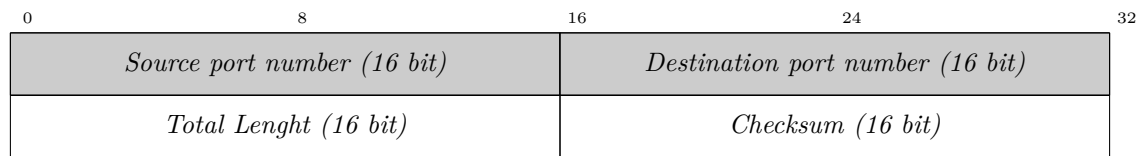
Figure 2.3: Identification parameters of an UDP or TCP/IP flow

0	8	16	24	32
Version (4 bit)	hlen (4 bit)	TOS (8 bit)	Total Length (16 bit)	
Identification (16 bit)			Flags (3 bit)	Fragmentation Offset (13 bit)
TTL (8 bit)	Protocol (8 bit)		Header Checksum (16 bit)	
Source IP Address (32 bit)				
Destination IP Address (32 bit)				
Options (0-40 bytes bit)				

Figure 2.4: IP packet header

For each flow received from the #1 or #2 interfaces, the SDN switch checks if its OF flow table contains a rule to be applied. Instead, for each packet belonging to flows already present in the flow table, the SDN switch applies the rules related to that flow. For example, it can forward or drop packets.

When fully operational, the flow routing OF flow tables contain the management rules for each flow that passes through the switch. Otherwise, for each packet belonging to a stream not present in these tables, the SDN switch sends a OF "PacketIn" message to the SDN Controller asking how to process it. The SDN controller receives this message and passes it to the IDS code which: stores the data related to the new flow within its own data structure, analyses the packet, and then, by using the OF "FlowMod" message, adds the appropriate rules for

**Figure 2.5:** TCP packet header**Figure 2.6:** UDP packet header

monitoring and forward the data flow to the SDN switch. As a last step, it sends the OF "PacketOut" message to the switch to allow the packet to be forwarded.

The training of the ML algorithm is done using a labeled dataset of network flows lasting about three days containing both Botnet Malware traffic and normal network traffic. During this operation, the SDN-SF-IDS system calculates and stores the statistics and, once a threshold (number of analyzed streams) chosen by us has been reached, starts the training phase of the rForest algorithm by using the extracted features.

Once the training is finished, the IDS code saves the trained parameters of the ML algorithm inside a special file that is loaded during the normal execution of SDN-SF-IDS. The SDN-SF-IDS performs at regular intervals the calculation of the statistical features and their classification through the rForest algorithm. Once the class ("malware" or "normal") of each monitored stream is identified, a drop rule is added to the OF flow table of SDN switch for each "malware" class stream. If

necessary, it is possible to change this rule in order to forward the flow catalogued as Botnet to a specific network port of the SDN switch to which further analysis systems can be connected for the analysis of network packets of the flow deemed malicious.

Each of these operations can introduce delays in the forwarding of the analysed packets.

2.6 Delays introduced on the network monitored by SDN-SF-IDS

The SDN-SF-IDS system performs different operations and therefore introduces different delays depending on whether or not the received packet belongs to an already known flow and therefore with rules already present in the OF flow tables (see figure 2.7). If the packet does not belong to an already known flow, the SDN switch forwards the packet to the SDN controller to create the rules necessary to perform the statistical calculations for this new flow.

Particularly significant are the delays related to the exchange of OF messages between the SDN switch and the internal delays of the two systems.

In the first case, the delays are greater as they involve the forwarding of packets to the SDN Controller and their analysis. This time is further extended by the fact that for each new flow the SDN Controller must send two OF messages ("FlowMod" and "PacketOut") to the SDN switch.

Other factors affecting delays depend on the OS side of SDN-SF-IDS infrastructure because, to perform the analysis of a single network packet, many software components must interact by communicating with each other and many lines of code come into play(Ethernet driver reception and transmission queues, context switch between kernel and user space, etc.)

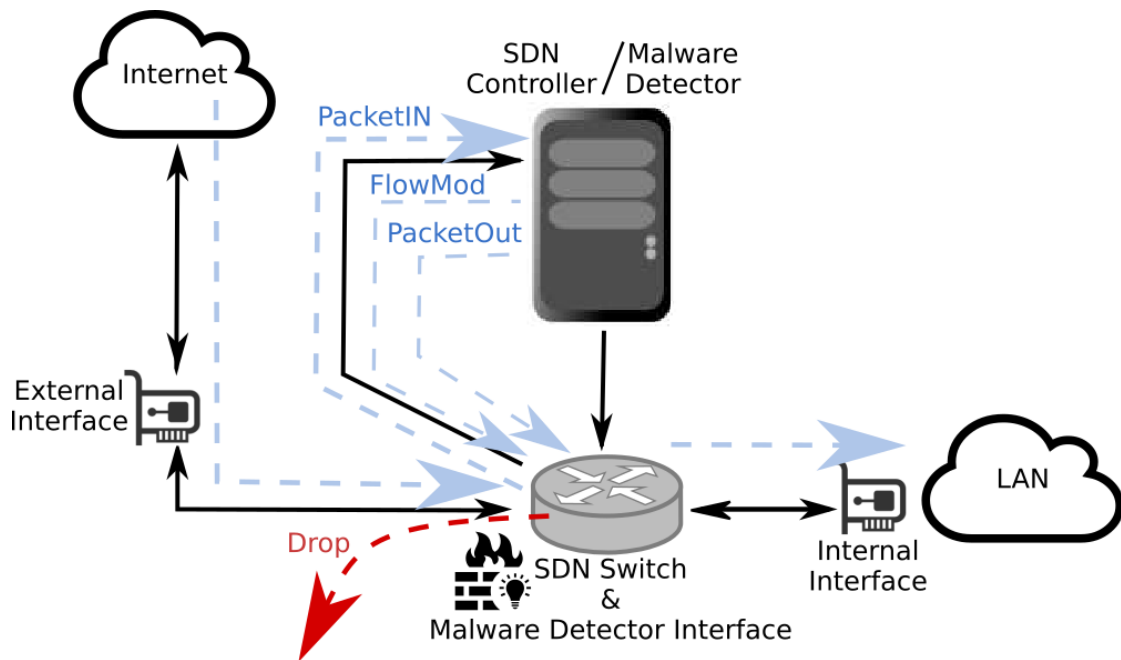


Figure 2.7: Messages exchanges inside SDN-SF-IDS

Each of these interactions adds a delay that needs to be minimized.

2.7 Minimize delays

To minimize the impact of the SDN-SF-IDS infrastructure on the monitored network, it is necessary to reduce the delays introduced by it.

The SDN controller is based on a GNU/Linux OS (Gentoo or Debian) and the SDN software controller Ryu in which the IDS code has been installed. By acting on the IDS code, the time required for the analysis of the new flows has been significantly reduced (see section 2.7 pag. 29).

The SDN switch is based on GNU/Linux OS (Debian or Mint distributions) with (at least) three Ethernet network cards (10/100 or 10/100/1000 Mbit/s) and OvS software implemented SDN switch.

Next subsection includes further information on the used software and hard-

ware. By acting on the software (Linux kernel, OvS, Ryu) and on the hardware (CPU, Ethernet card) of the systems, it was possible to reduce the delays associated with the SDN-SF-IDS.

Laboratory tests

I have created various prototypes of increasing complexity to overcome the bottlenecks gradually highlighted by the tests carried out in order to improve the performance in terms of latency. The analyzes were carried out by sending to the SDN-SF-IDS system the entire set of network flows at my disposal, containing or not containing packets belonging to Botnet. The laboratory tests are aimed at evaluating and reducing the delays introduced by the SDN-SF-IDS. The detected delays have been minimized both by increasing the resources of the used computers, and also through deep modifications of the SDN-SF-IDS code and OS configuration.

The laboratory tests were carried out using the logic schema presented in figure 2.8, connecting the #1 and #2 interfaces of the SDN-SF-IDS to a special system "Tester" capable of generating and receiving a stream of Ethernet packets and calculates the delay suffered by each individual Ethernet packet. The tester sends the network stream to be filtered from the internal Ethernet interface " $ethT_{INT}$ " to port #1 of the switch SDN-SF-IDS and receives on the interface " $ethT_{EXT}$ " the filtered stream outgoing from port #2 of the above system.

Figure 2.9 and figure 2.10 show the sequence diagram of the message exchange within the SDN-SF-IDS infrastructure in case SDN Switch receives a new stream or an already known stream.

By measuring the instant T_1 in which the packet leaves the " $ethT_{INT}$ " interface and the instant T_2 in which it returns from the " $ethT_{EXT}$ " interface, it is possible to estimate the total delay ΔT_{TOT} generated by the SDN-SF-IDS system as $\Delta T_{TOT} =$

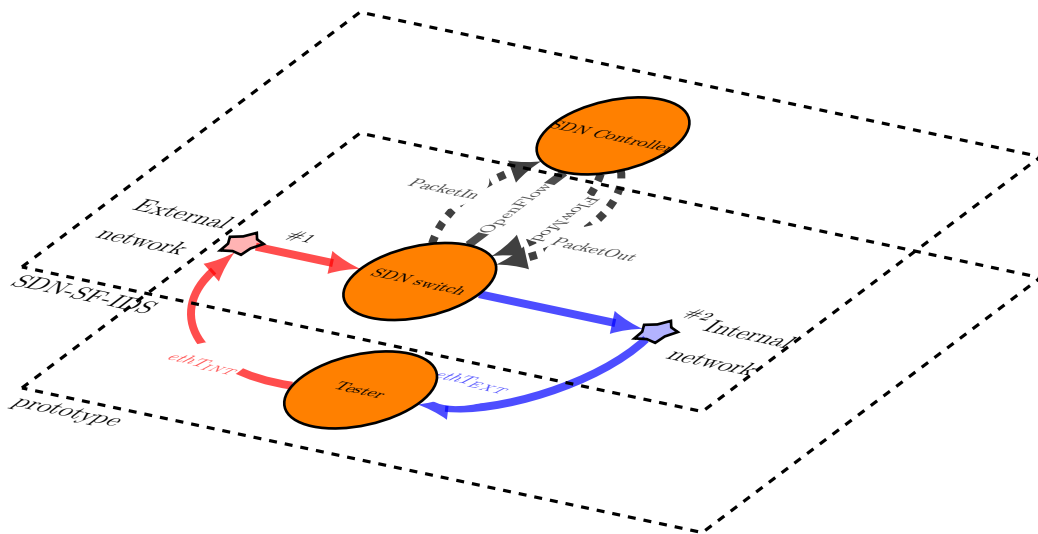


Figure 2.8: Connection of SDN-SF-IDS with the Test system used to measure delays

$T_2 - T_1$. For more information see section 2.8 at page 41.

To simplify the test operations of the SDN-SF-IDS, the SDN Controller itself was used instead of the "Tester", connecting it as in figure 2.11.

In the first laboratory tests, the two parts of the SDN-SF-IDS infrastructure were created using two computers: the first (A) acts as a SDN controller; the second (B) as the SDN switch.

As the test trials progressed, five different versions of the SDN-SF-IDS infrastructure were implemented.

For the first three versions the same Controller A_1 was used by varying only the SDN B_i switch (A_1B_1, A_1B_2, A_1B_3) with the aim of obtaining increasingly reduced latencies at the cost of software complexity ever increasing and ever more binding hardware requirements.

The results of this optimization phase were used to create a final prototype equipped with a new SDN A_2 controller capable of controlling a SDN B_4 switch or a SDN switch hardware B_5 . The latest two versions of the SDN-SF-IDS infras-

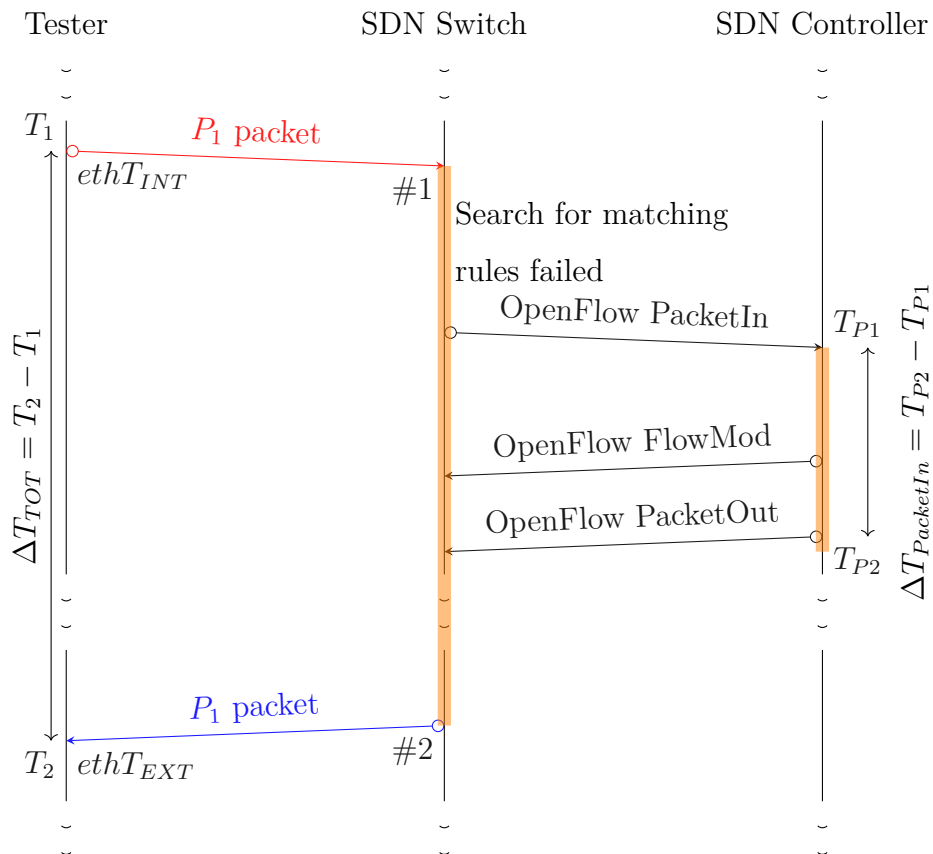


Figure 2.9: Timing diagram of the Openflow message exchange following the receipt of the 1st packet (P_1) of a new P flow

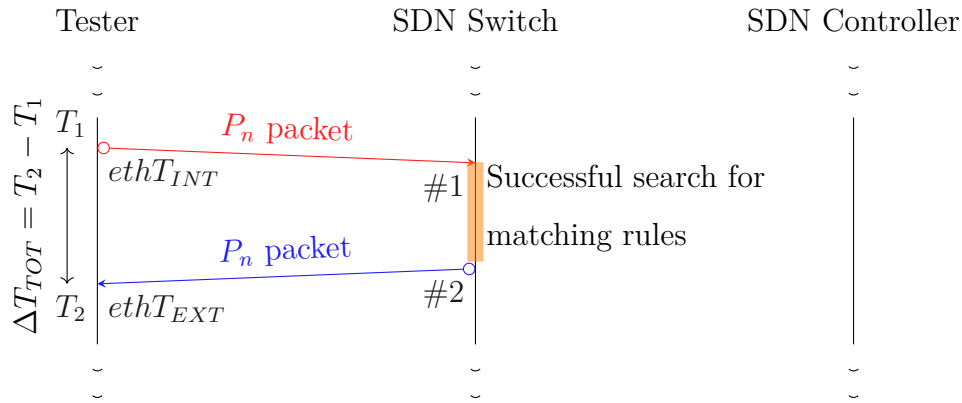


Figure 2.10: Timing diagram of the Openflow message exchange following the receipt of the n^{th} packet (P_n) of the previously detected P flow

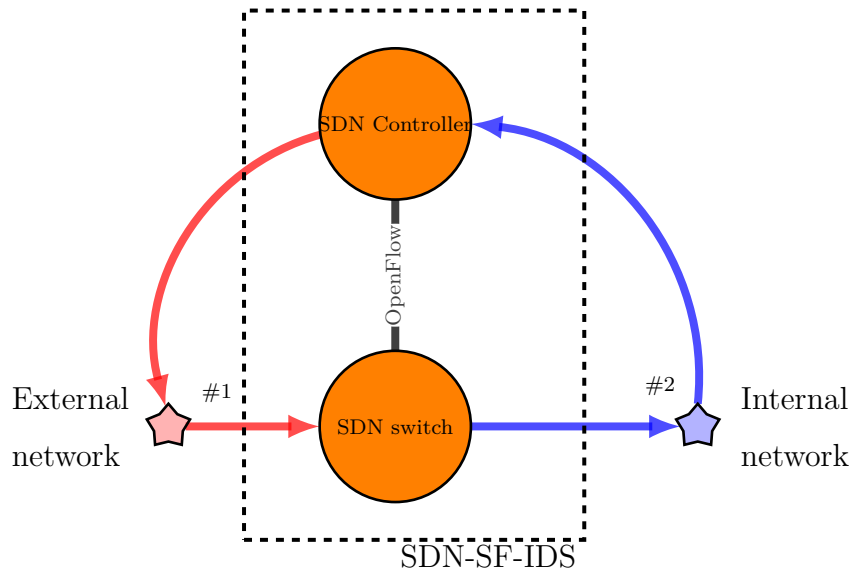


Figure 2.11: Physical structure of the prototype during the test phases

structure (A_2B_4 and A_2B_5) which make up the final prototype of this infrastructure and will be presented in section 2.9 49.

The A_1 processor uses a GNU/Linux OS (Gentoo distribution) and SDN Controller software Ryu (ryu-4.26). The system is equipped with a quad-core Intel® Core™ CPU i5-3340 running at 3.10 GHz, 16 GByte RAM and with four network

cards Gigabit Ethernet:

ethA_{WAN} Wide Area Network (WAN) / Internet;

ethA_{OF} connected to systems B_1, B_2, B_3 dedicated to OF communications;

ethA_{INT} network to be analyzed (internal side);

ethA_{EXT} network to be analyzed (external side).

The B_1, B_2 and B_3 computers use a GNU/Linux OS (Debian or Mint distributions) and are equipped with 3 network cards:

ethB_{OF} Connection dedicated to OF communications;

ethB_{INT} network to be analyzed (internal side);

ethB_{EXT} network to be analyzed (external side).

The B_1 system is a Nokia IP120 device equipped with a 266 MHz AMD Geode CPU, 128 MByte RAM and 3 Ethernet 10/100 Mbit/s network cards. The B_2 system is a PC equipped with a 2.4 Ghz quad core Intel Q6600 CPU, 4 GByte of RAM and 3 Ethernet 10/100/1000 Mbit/s network cards. The B_3 system is a PC equipped with an Intel[®] Core[™] i5-7400 CPU at 3.00 GHz, 16 GByte of RAM and 3 network cards Ethernet 10/100/1000 Mbit/s.

The systems A_1B_1, A_2, B_1, A_1B_3 are connected as in figure 2.11 in order to create the following networks:

WAN port *ethA_{WAN}* directly connected with WAN and Internet;

CTRL direct connection between *ethA_{OF}* and *ethB_{OF}* dedicated to OF connection;

INT direct connection between $ethA_{INT}$ and $ethB_{INT}$ used to simulate an internal Local Area Network (LAN). The traffic to be analyzed and filtered is sent by $ethA_{INT}$ port and received by $ethB_{INT}$ port;

EXT direct connection between $ethA_{EXT}$ and $ethB_{EXT}$ used to simulate the external. The $ethA_{EXT}$ port receive the WAN traffic filtered by the SDN-SF-IDS and sent from $ethB_{EXT}$ port;

A virtual SDN Switch composed of two logic ports (#1 and #2) has been created inside each B_i computer using the OvS software. Port #1 is assigned to the $ethB_{EXT}$ interface and consequently is dedicated to the input traffic to be analyzed and filtered. Port #2 is assigned the $ethB_{INT}$ interface and is therefore dedicated to output (filtered) traffic. The OF communication of the control plane (Ryu - OvS) takes place via the dedicated connection $ethA_{OF} \iff ethB_{OF}$.

The B_i switch receives the traffic to be filtered on the $ethB_{EXT}$ interface, forwards it to the virtual port #1 of OvS which checks if in the OF flow table there are rules applicable to the received packet. If so, it applies the corresponding rule which can contain a drop command or a forward rule to the exit port #2. In this case, the packet is passed to the virtualSDNport #2 and then sent in output without modification by the associated $ethB_{INT}$ card. If no rule matches, the packet is forwarded to the SDN A_j Controller through the dedicated CTRL connection. The SDN B_i switch works as a pass through filter with the sole purpose of monitoring and acting on data flows. The SDN A_j Controller receives the first packet of each new stream detected by B_i over the CTRL connection. Then, it creates and sends the monitoring and packet forwarding rules to B_i which, at each predetermined interval, sends the statistics of all the monitored flows to the A_i controller which, through a supervised ML algorithm rForest, catalogs the flows and adds rules for managing flows based on their assignment ("normal" or "malware").

test-bed	Controller A_1 : CPU Intel® Core™ i5-3340 running at 3.1 GHz, 16 GByte RAM, 4 Eth 10/100/1000 Mbit/s			
	description of SDN switch B_n	$\Delta T_{PacketIn}$	ΔT_{TOT}	notes
A_1B_1	switch software CPU AMD Geode running at 266 Mhz, RAM 128 MByte, 3 Eth 10/100 Mbit/s	average 2 ms, peaks < 14ms	-	IDS use array data structure
A_1B_1	switch software CPU AMD Geode running at 266 Mhz, RAM 128 MByte, 3 Eth 10/100 Mbit/s	average 1 ms, peaks < 3 ms	average 12.01 ms, peaks < 189.61 ms	IDS use hash data structure
A_1B_2	switch software CPU Intel® Q6600 running at 2.4 Ghz, 4 GByte RAM, 3 Eth 10/100/1000 Mbit/s	-	average 1.09 ms, peaks < 54.29 ms	IDS use hash data structure B_2 has more computational power

Table 2.1: Testbeds A_1B_1 , A_1B_2 , A_1B_3

For more information on the ML mechanism used, see the aforementioned [1] and [2].

The entire prototype has been designed to work in real-time with constant performance monitoring (further information can be found in the section 2.8 at page 41).

Table 2.1 shows a summary of the A_1B_1 , A_2B_1 , A_1B_3 prototypes with a technical description of the prototypes hardware, the optimization done and obtained result.

The A_j controller can analyze in detail the results of the filter action performed by SDN-SF-IDS as it creates the data streams to be filtered by sending them to the EXT network which is analyzed in real-time by B_j . At the same time the same

A_j receives the outgoing network traffic from the SDN switch through the INT network (free of infected streams). Through the analysis of the flows sent on EXT and received by INT, it is possible to verify the correct elimination of the infected flows. With the same method, by temporarily disabling the rules for dropping the Botnet Malware streams, you can measure the delay that each individual packet undergoes, from the instant when it is sent from the $ethA_{EXT}$ interface to the instant when is received in the $ethA_{INT}$ interface. The INT network data flow is created through port mirroring of the $ethW_{AN}$ connection with the ability to add any network sample ("normal" or "malware") to the traffic.

The debug scripts running on A_i measure the OS performance of both computers and individual software (OvS and Ryu) every 10 seconds. For more information refer to the section 2.8.

Laboratory tests: prototype A_1B_1

The first tests carried out were aimed at determining the influence of the computational power of the SDN Switch software and on the delays suffered by the analyzed packets. The CPU of B_1 system was chosen among those with the least computational power at my disposal. The A_1B_1 system did not experience particular overloads during the entire test period and the OS, even with limited resources, always remained responsive. This prototype was fundamental to bring out two SDN-SF-IDS problems related to each other and related to the SDN controller.

The implementation, created based on the Ryu open source software, introduces delays both in the management phase of the new flows and in the phase of acquiring the statistics (carried out periodically) aimed at the classification and filtering of the flows.

With reference to the management of new flows, delays are introduced only for the first packet of each new flow. An important part of this delay is linked

to the fact that the first packets of the new flows do not find a rule dedicated to them within the OF flow tables of the SDN switch. Then, they require the OF "PacketIn" message to be sent to the SDN Controller. When the SDN Controller receive the OF "PacketIn" message analyse it and, in response, the OF "FlowMod" and OF "PacketOut" messages has to be sent back to the SDN switch. If a rule is already present, the packet only passed trough the SDN switch with timing below one millisecond. In the following, $\Delta T_{PacketIn}$ will indicate the time necessary for the IDS module to complete the handling of the OF "PacketIn" message. The time taken for the packet to go through the entire SDN-SF-IDS will be indicated as ΔT_{TOT} .

From the analysis emerged that the frequency of new flows detected by the SDN switch (and therefore sent to the controller for analysis) is not constant over time but there are peaks of new flows and consequently peaks of OF messages "PacketIn" sent to SDN Controller (see Fig. 2.12).

The OF "PacketIn" messages are processed in sequence so the delays add up with a consequent rapid extension of the response times (Fig. 2.13). Most of the burst of new flows are caused by traffic carried over the UDP protocol. The low percentage of traffic peaks generated by TCP is due to the fact that in the analyzed traffic (mirror of our local network) there are stable TCP connections of long duration which result in a low rate of new flows IP per second. For example during a test 2 500 111 network packets were analysed and only 52 242 "PacketIn" messages has generated.

In the prototype A_1B_1 , the time $\Delta T_{PacketIn}$ (figure 2.19) is about 2 milliseconds but in correspondence to the arrival of peaks it can increase reaching 14 ms.

The SDN-SF-IDS demo code used in [1] has been revised, extended and optimized. The data structure used to trace new network flows has been re-implemented using a hash table optimized for accessing data via the flow identifier. This action,

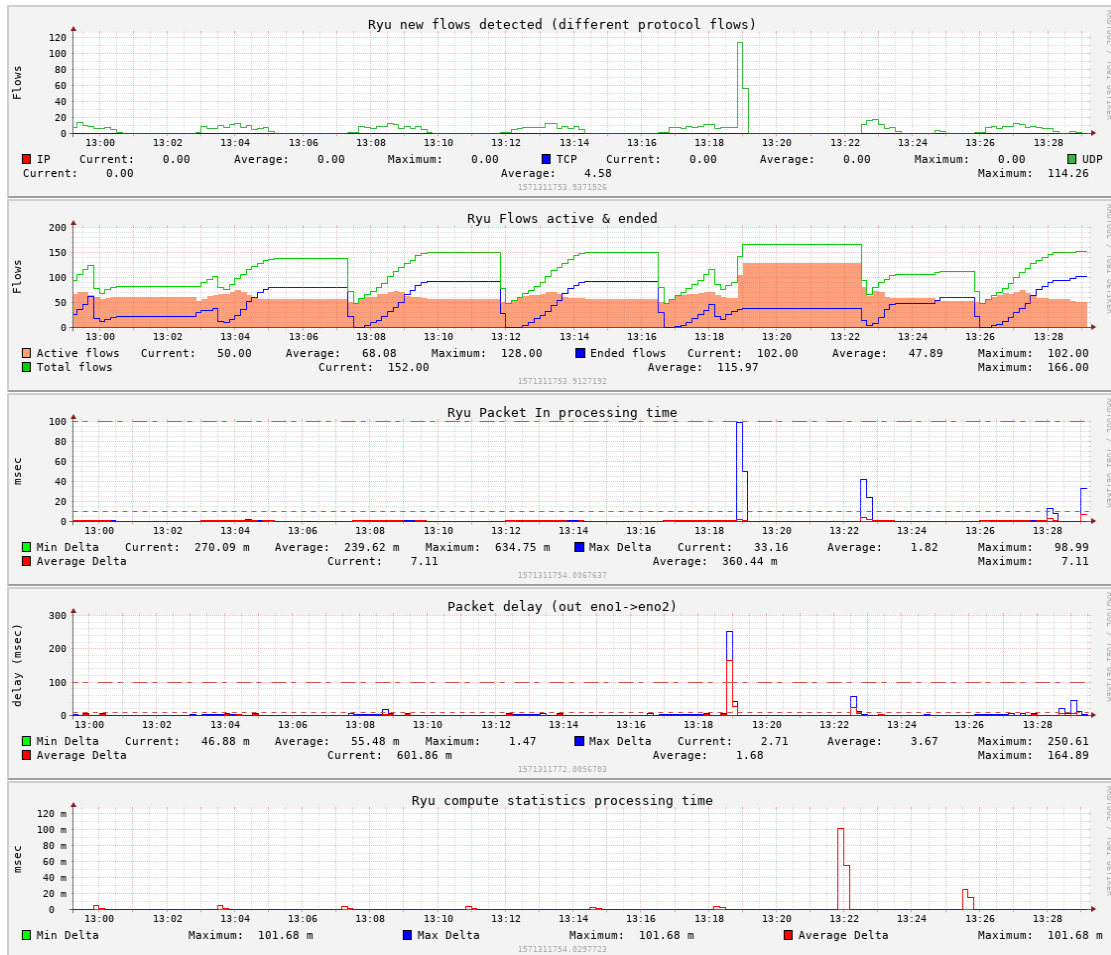


Figure 2.12: Graph showing peak of new flow (first image) and corresponding avalanche effect (other images) on $\Delta T_{PacketIn}$, ΔT_{TOT} delays and classification time is greater than normal.

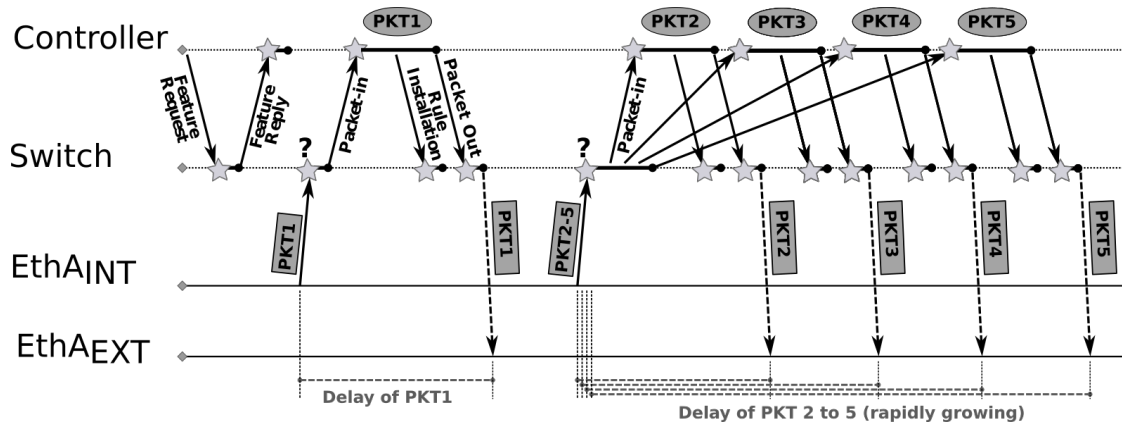


Figure 2.13: Increase in delays following a spike in new flows entering the switch

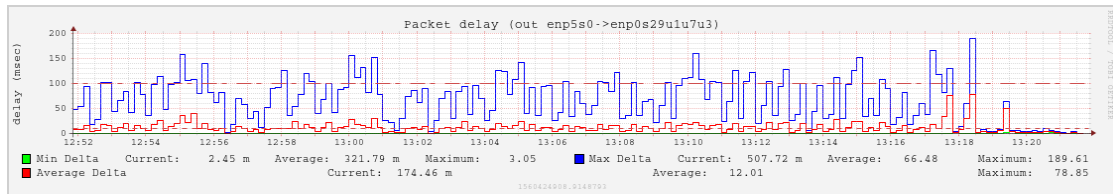


Figure 2.14: Packets Delays ΔT_{TOT} of prototype A_1B_1

combined with other minor changes, resulted in a reduction of the $\Delta T_{PacketIn}$ time to 1 ms average time with 3 ms peaks related to OF request peaks (see figure 2.20).

With reference to the statistics acquisition phase, the delay is linked to the fact that the calculation of the statistics and the cataloging action performed by the ML algorithm are carried out within the main Ryu process. For this reason, Ryu is no longer able to process other requests, including the receipt of new flows, until the end of the cataloging phase (see figure 2.15 and figure 2.16).

The time required to compute statistical features, classify the flows using rForest ML algorithm on them and sending corresponding OF "FlowMod" messages is not high but not negligible. It takes 101.68 ms to process 166 streams (see figure 2.15).

In figure 2.15 the Ryu main process stay blocked for 100ms, this causes a rapid

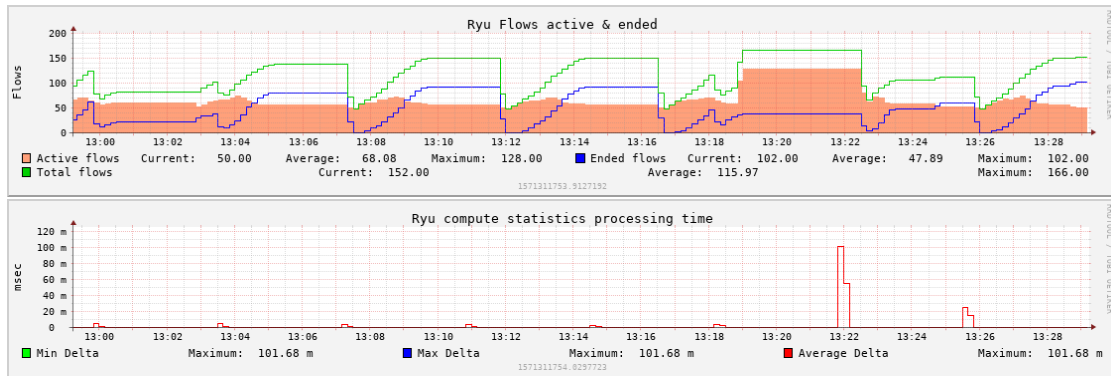


Figure 2.15: Number of detected flows and corresponding analysis time (feature extraction, flow classification and SDN rule insertion)

increase in the length of the queue of OF messages awaiting response (most are "PacketIn") with the associated increase in the response time of the SDN Controller to OF requests "PacketIn" and consequently the entire SDN-SF-IDS infrastructure.

To solve this problem, the analysis of the flow statistics and their classification have been moved to a special Thread which processes them in parallel. This way Ryu main process is free to immediately proceed with handling other OF messages sent to it by the SDN switch. For more detailed information, refer to section 2.7 page 29.

Once the time required to process OF "PacketIn" requests was minimized, I moved on to measuring the ΔT_{TOT} delay times introduced by the overall SDN-SF-IDS system. To do this, a specific script described in section 2.8, was used.

The total delays ΔT_{TOT} measured in prototype A_1B_1 (see Fig. 2.14) show an average value of 12.01 ms with peaks of 189.61 ms of maximum delay in correspondence of new flow peaks.

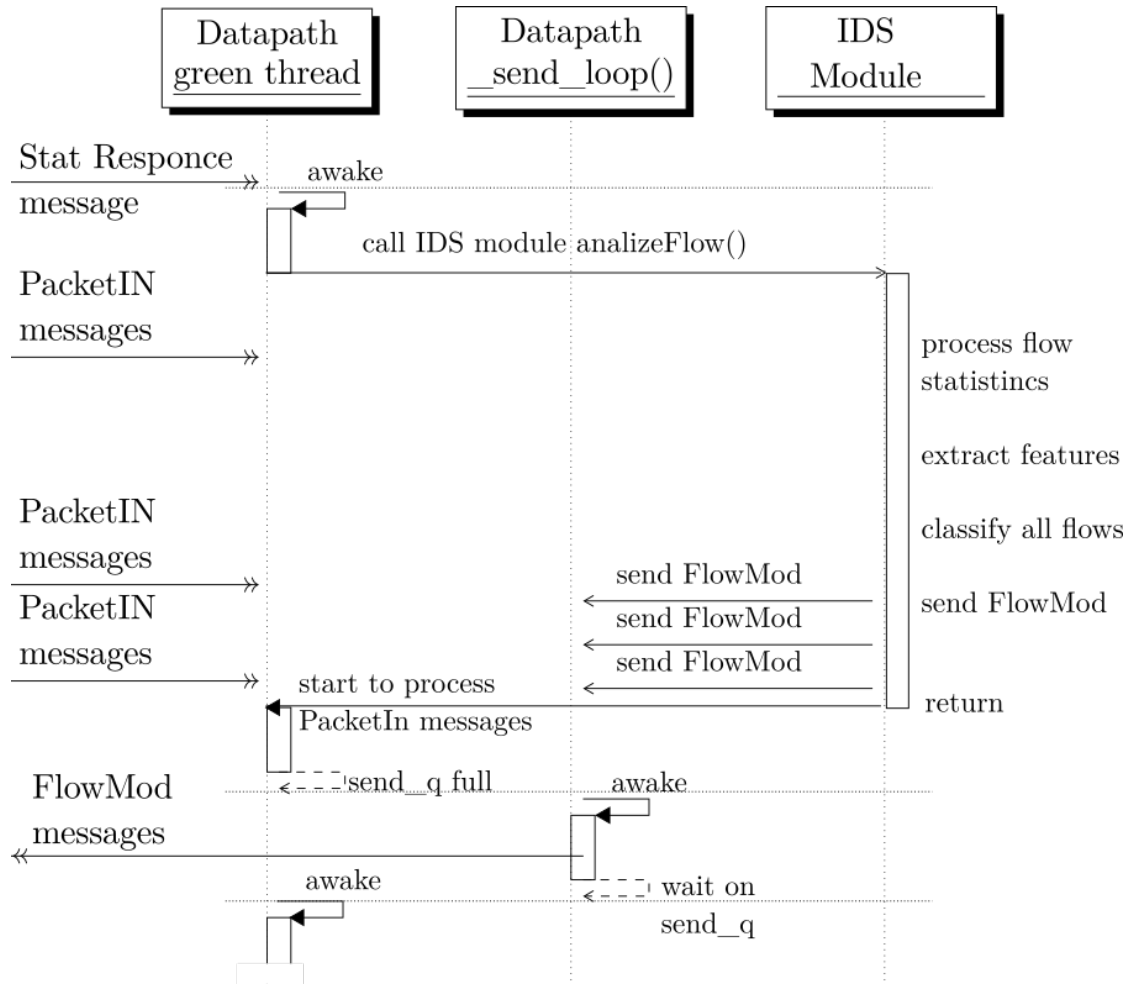


Figure 2.16: Scheduling of SDN-SF-IDS feature extraction and flow classification procedure

Laboratory tests: prototype A_1B_2

Once the optimization of the SDN A_1 controller code is complete, We assessed the impact of higher available system resources on system performance by replacing the SDN B_1 switch with the better performing one B_2 system.

As expected, the delays introduced by the SDN-SF-IDS ΔT_{TOT} infrastructure (see Fig. 2.17) decreased to 1.09 ms average, with peaks of 54.29 ms, thanks to a faster management of the OF flow tables due to a more powerful CPU.

The $\Delta T_{PacketIn}$ time required to process the OF "PacketIns" message has remained comparable to that obtained with the A_1B_1 prototype as it is linked to operations performed on the SDN A_1 controller.

Laboratory tests: DPDK libraries

At the end of these measurements, a performance limit emerged due to the software nature of the SDN switch running inside a general purpose OS, such as GNU/Linux. An analysis of the state of the art² shows that the presence of a scheduler that manages the execution of parallel processes and other strictly technical elements introduces waiting times that result in delays that cannot be optimized.

DPDK can be used to overcome this problem. This development kit has been designed to ensure faster handling of packets received by Ethernet cards at the price of a significant increase in complexity. The DPDK libraries work in close synergy with the Linux kernel and require the presence of the most advanced architectural solutions offered by today's CPUs (Hugepages, IOMMU, VT-x, VT-d, etc.). First of all, it is necessary to act on the boot parameters of the Linux kernel to enable the use of the Hugepages (pages larger than 4KBytes) and configure the DPDK

²R. Giller. Open vswitch with DPDK overview. <https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview>

libraries so that they take advantage of them benefit. Then, a part of the system CPUs has to be reserved for running the DPDK libraries and the software that uses them, preventing their use from the Linux scheduler.

In order to use a network card with these libraries, special optimized drivers^{3,4} which are only available for a small set of network cards are needed.

Laboratory tests: prototype A_1B_3

The B_2 system does not support the minimum hardware requirements to use DPDK⁵, so we used the B_3 system equipped with an Intel 82571EB/82571GB Gigabit Ethernet D0/D1 network card rev 06 (double RJ-45 port). The DPDK library version 19.05.0⁶ was compiled and installed on system B_3 . The configuration was done in a manner consistent with the DPDK guidelines⁷. The OvS software version 2.11.1⁸ was compiled to take advantage of the DPDK libraries to further improve the performance of the SDN controller. The results of the tests (Fig. 2.18) show a reduction of the maximum delays ΔT_{TOT} to 11.76 ms. The minimum delays are approximately 141 μ s.

To obtain these results was used one of the least performing models supported by the DPDK.

³DPDK Linux drivers are required. <http://doc.dpdk.org/guides/linuxgsg/linuxdrivers.htm>

⁴DPDK overview of networking drivers. <http://doc.dpdk.org/guides/nics/overview.htm>

⁵DPDK system requirements. <http://doc.dpdk.org/guides/linuxgsg/sysreqs.htm>

⁶<http://git.dpdk.org/dpdk/snapshot/dpdk-19.05.tar.gz>

⁷Configure open vswitch with DPDK on Ubuntu server 17.04. <https://software.intel.com/en-us/articles/set-up-open-vswitch-with-dpdk-on-ubuntu-server>

⁸<https://www.openvswitch.org/releases/openvswitch-2.11.1.tar.gz>

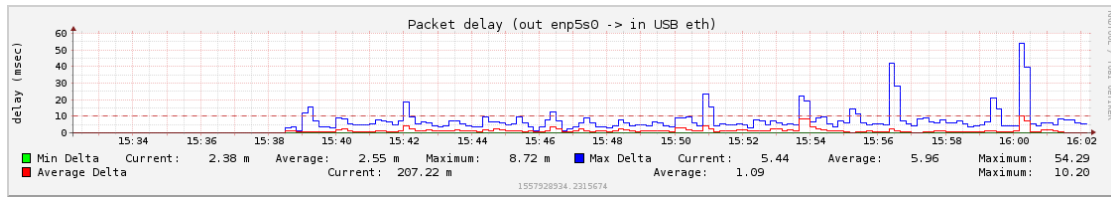


Figure 2.17: Packets delays ΔT_{TOT} of the A_1B_2 prototype

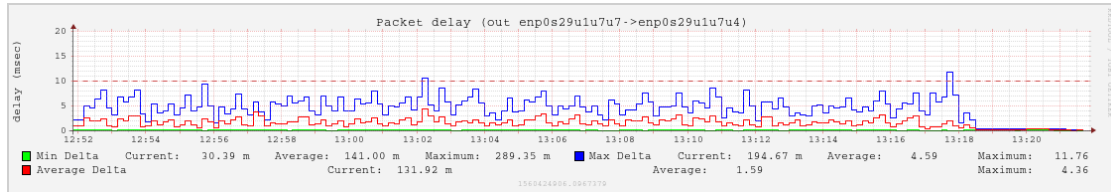


Figure 2.18: Packets delay ΔT_{TOT} of the A_1B_3 prototype

Optimization of SDN Controller

The optimization of the SDN Controller code went in parallel with the one of the SDN switch software.

As mentioned in section 2.1, the Open Source Ryu software, running on the SDN A_1 and A_2 controllers, was written in Python. The use of this language, on the one hand, allows a rapid modification of the code with simple debugging procedures, but, on the other hand, it does not guarantee to obtain the maximum possible performance. Other languages, such as C++, could achieve higher performance, but the debugging would be more complex. For this reason, we have chosen to continue to use the Python language during the optimization and improvement phases of the SDN-SF-IDS.

The first tests (second row of Table 2.1) based only on the measurement of the time needed to process a single OF "PacketIn" message had highlighted the slowness of the insertion and search phases of the flows within the data structure used for their storage. The first optimization carried out focused on minimizing the time required for interactions with the data structure containing the flows.

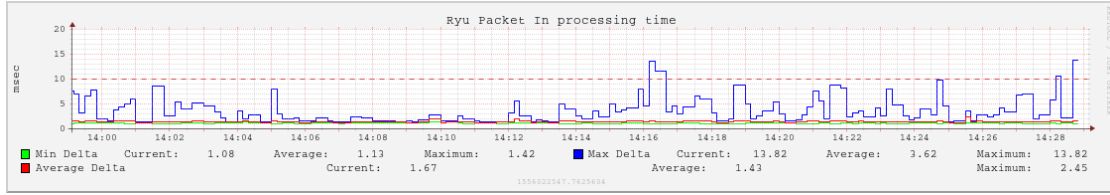


Figure 2.19: Analysis time of a single "PacketIN" (IDS with array structure, prototype A_1B_1)

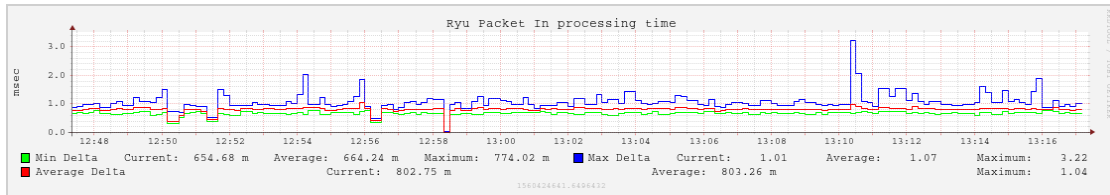


Figure 2.20: Analysis time of a single "PacketIN" (IDS with hash structure, prototype A_1B_1)

We have changed a data structure based on arrays (list) into a structure based on hash tables (dictionary). These changes have resulted in an increase in system performance (figure 2.19 and figure 2.20) reducing the peaks of $\Delta T_{PacketIn}$ from 13.82 ms to 3.22 ms.

However, the tests revealed a problem strictly related to the structure of the Ryu software which is based on a single process programming model. The handling of each received OF packet is done sequentially, so parsing the one OF "STAT_REQ" message blocks Ryu execution until it is finished. Meanwhile other OF messages sent to the SDN Controller A_i remain stuck in the system queue of the socket associated with OF port 6633 without being processed. For further information see section 2.7. This problem occurs for each OF message received but it is particularly critical during the receipt of the OF "STAT_RESP" message as, upon receipt, the IDS module performs the extraction of the statistical features and the cataloging of the flows. These operations take much more time to complete

than the time required to manage other OF messages. Furthermore, the statistics calculation procedure is repeated at regular time intervals, blocking Ryu for a variable time depending on the number of packets to be cataloged. This causes further congestion in the handling of OF "PacketIn" messages and the consequent slowdown in the management of new flows.

The IDS module of the SDN-SF-IDS has been enhanced by adding two parallel running Threads to the main Ryu process.

This solution allowed us to perform slower operations within a separate Thread leaving the main process free to handle incoming OF messages. The Thread "*MONITOR*" cyclically wakes up to request the SDN switch to send the statistics of the flows it monitors and then returns to sleep. The Thread for calculating the statistics "*STAT_THR*" is awakened by the main process when the statistics are ready to be calculated. The Thread moves the necessary data within a data structure reserved for it. In this way, the execution of the Thread and the main process continues in parallel without concurrent access to the data. The main process signals through a special Semaphore the presence of new data to the Thread "*STAT_THR*" and immediately starts managing the OF messages again, finding the data structure empty and ready to store the data relating to the new flows.

The Thread "*STAT_THR*" calculates the statistics of the flow and performs the cataloging using the rForest algorithm, then proceeds to insert any drop rules inside the switch and finally goes back to rest. The execution time of these Threads is limited to a few milliseconds at each statistics request interval (configurable by the user and set to 30 or 220 seconds in tests). Figure 2.21 shows a summary of the possible interactions between all agents in action: OvS on the SDN switch; Ryu and the Thread dedicated to requesting and analyzing the statistics on the SDN Controller. In the diagram, it has been shown that having separated the flow cataloging phase in a special Thread allows Ryu to immediately manage the OF

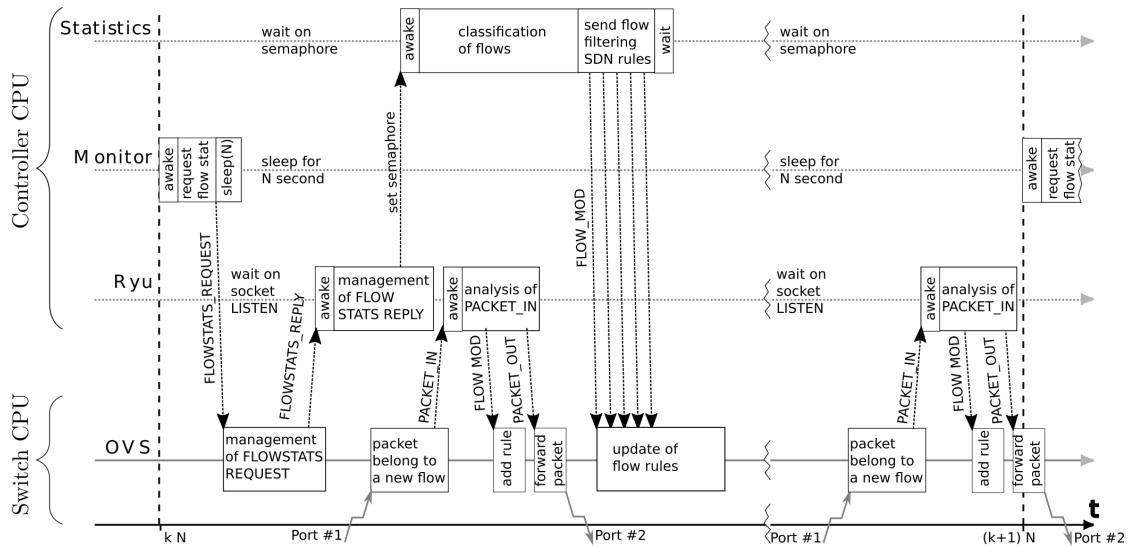


Figure 2.21: Simplified diagram of the interaction between ryu and the Threads dedicated to requesting and analyzing statistics

"PacketIn" messages received by the SDN switch.

The second criticality that emerged from the tests is a relation between peaks in the number of OF messages received and increasing slowdowns in their analysis.

A slowdown in the parsing phase causes a slowdown in the handling of all other OF requests sent to the SDN Controller. This behaviour depends on a sequential and non-parallel management of the phases of reception and analysis of OF messages.

In order to better manage the problem and understand in depth the interdependencies related to Ryu's internal functioning, we have analyzed the source code of Ryu version 4.31⁹).

⁹<https://github.com/faucetsdn/ryu/commit/050bfb711ed9f3bcab458a904eab74aad90b076>

Ryu software analysis

Ryu's source code uses the Greenlet¹⁰ library that allows you to manage calls to Input/Output (I/O) functions in a clean and simple way blocking and provides through the GreenThread¹¹ a cooperative multitasking execution logic¹². The GreenThread alternate the execution on single CPU¹³, in a given instant of time only one GreenThread is running on the system. Specifically, scheduling can take place in the following ways: voluntarily with cooperative logic (via `hub().sleep(0)`); when the GreenThread sleep waiting for data (`socket.read()`, `queue.get()`); when Inter-Process Communication (IPC) primitives (queues, semaphores, etc) are used. Each scheduling event corresponds to a CPU contention by all GreenThreads awaiting execution. The scheduling logic is implemented in the hub class¹⁴.

The analysis of the main classes used in the Ryu software the majority of which are implemented in the code file `controller.py`¹⁵ showed:

- the main process of Ryu waits, with the Controller class, for connection requests sent by the switches (line 205);
- For each connection request received by a new SDN switch, a specific Datapath class is instantiated;
- After being instantiated, the Datapath class is put into execution on a special GreenThread;
- The Datapath class creates two further GreenThread, the first GreenThread

¹⁰<https://eventlet.net/#eventlet>

¹¹<http://eventlet.net/doc/modules/greenthread.html>

¹²<https://eventlet.net/doc/hubs.html#how-the-hubs-work>

¹³https://eventlet.net/doc/basic_usage.html

¹⁴<https://github.com/eventlet/eventlet/blob/master/eventlet/hubs/hub.py>

¹⁵<https://github.com/osrg/ryu/blob/master/ryu/controller/controller.py>

will manage the transmission of messages (line 460), the second is dedicated to verify the connection (OF "Echo request") with the SDN Switch (line 466) and finally starts an infinite loop for receiving messages (line 469);

To manage a new SDN switch, Ryu generates three new GreenThread but only one of them will be running on the CPU. Therefore, an effective parallelism can never take place between receiving and sending OF messages.

The sequence diagram resulting from the connection of a single SDN switch is shown in figure 2.22. In this case, the Greenlets in scheduling are:

1. The main Greenlet waiting, with the OpenFlowController, class for new SDN switches to manage (lines 124-218);
2. The Greenlet in charge of receiving the OF messages through the `def _recv_loop()` member function (line 320) of the Datapath class;
3. The Greenlet in charge of sending OF messages to the switch through the `def _send_loop()` memberfunction (line 383) of the Datapath class;
4. The Greenlet in charge of sending "echo requests" OF message to the switch at regular intervals through the `def _echo_request_loop()` member function (line 442) of the Datapath class.

The OpenFlowController class waits for most of the time (inside the function `server.serve_forever()`) for new connections (TCP listen) on OF port TCP 6633. Its GreenThread is scheduled to run only when it receives a connection request¹⁶. Then, through the function `def datapath_connection_factory()` (passed as parameter "handle" in the creation phase of the class StreamServer file controller.py lines 185-202) it creates the Datapath class and executes it on a special GreenThread created for this purpose through the `hub.spawn ()` function.

¹⁶<https://github.com/osrg/ryu/blob/master/ryu/lib/hub.py> line 150

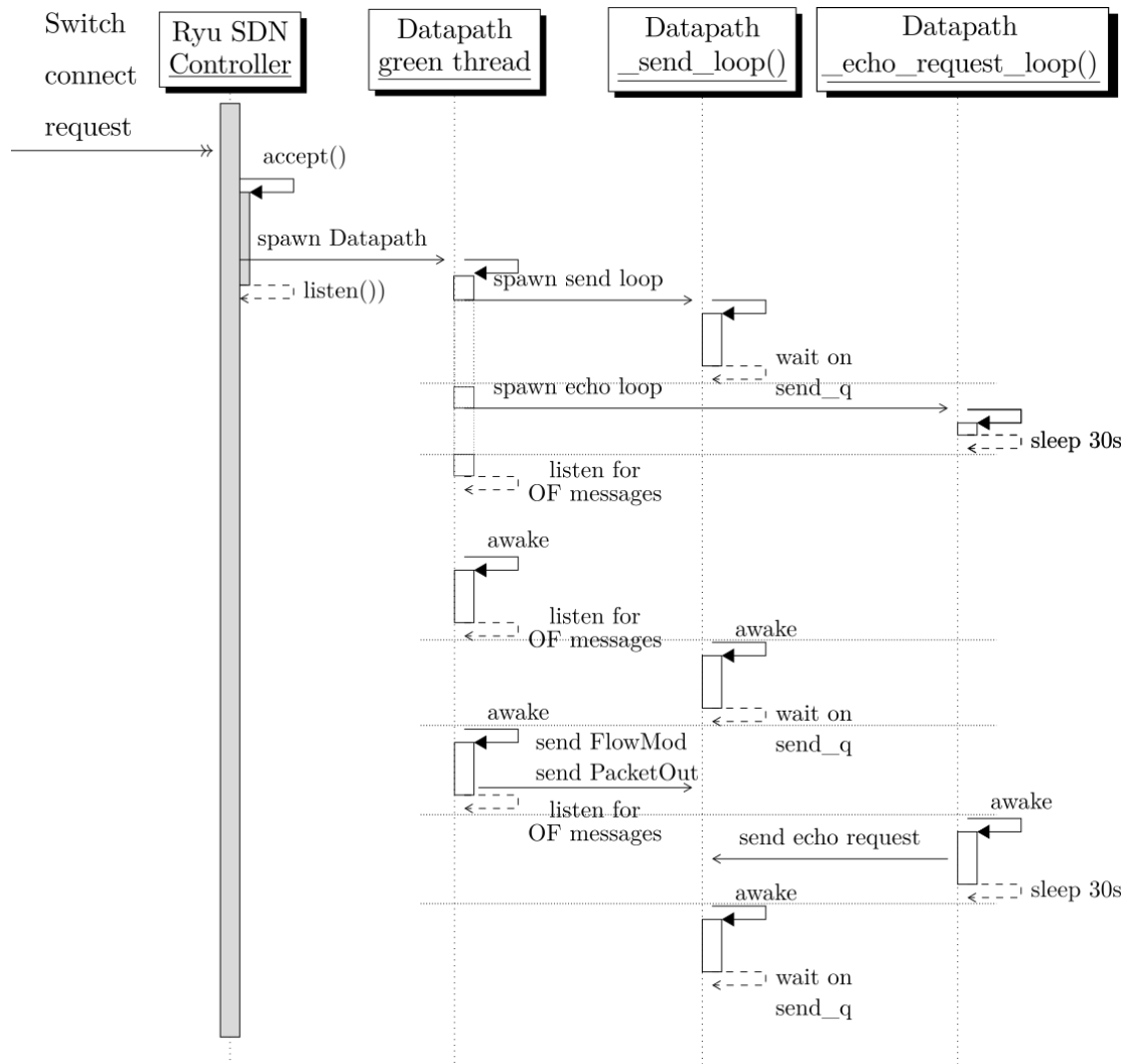


Figure 2.22: Sequence diagram trigger by SDN switch connection with Ryu Controller

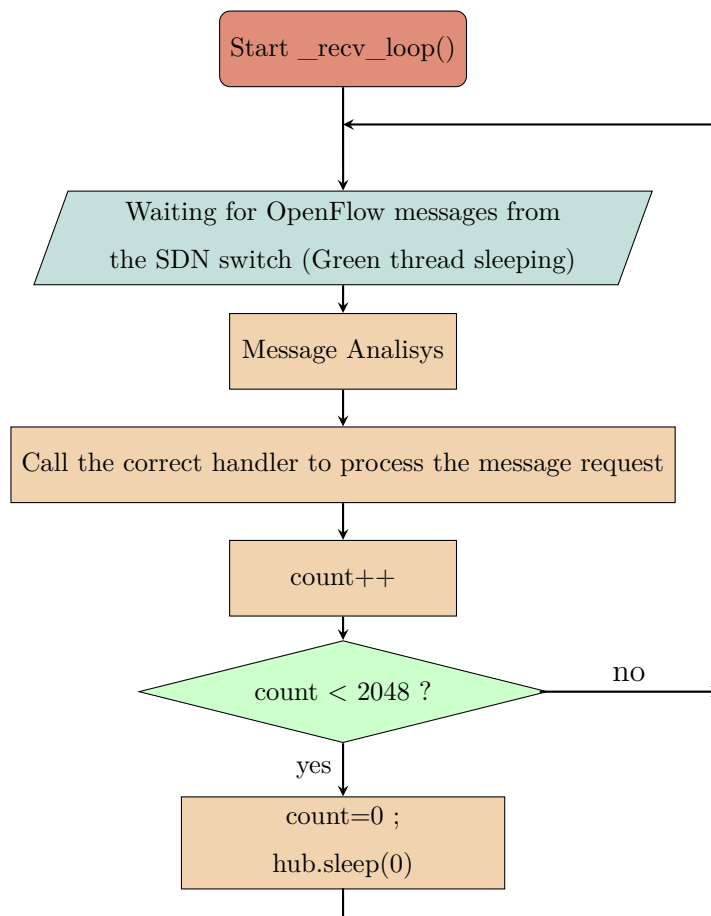


Figure 2.23: Flowchart of the Ryu procedure `_recv_loop()`

The `_recv_loop()` member function (fig. 2.23) consists of an infinite loop in which the `GreenThread` waits for new messages OF from the switch of its competence (`socket.recv()`) and analyzes it by calling the appropriate management functions (handler) after which it returns waiting for new messages.

If there are no new OF requests inside the socket's reception queue, the `GreenThread` goes into sleep releasing the CPU otherwise the execution of the infinite loop resumes immediately without waiting. In the infinite loop, the received messages are tracked and every 2048 processed OF messages the CPU is released voluntarily (`hub.sleep(0)` *#line 381*). During the analysis and management

phase of the individual OF messages (through the handler functions), it is possible that they are created, using the `send_msg()` and `send()`, new OF messages to send. These functions do not send the message directly but put it in a special “send_q” data queue capable of managing up to a maximum of 16 elements. These messages to be sent to the SDN switch remain blocked within this queue until the `GreenThread _send_loop()` ends. In the `_recv_loop()` function, there are three different situations in which the `GreenThread` can go into sleep freeing the CPU to the other `GreenThread`:

1. socket queue is empty;
2. queue for sending data send_q full (16 elements);
3. the loop processed 2048 received OF messages.

The Ryu developers left a comment (line 376-383) regarding the need to improve the logic regarding the voluntary suspension of 2048th message.

We need to schedule other greenlets. Otherwise, Ryu cannot accept new switches or handle the existing switches. The limit is arbitrary. We need the better approach in the future.

This logic of managing messages in groups, until the “send_q” queue is saturated or to process 2048 messages, justifies the increase in the time required to process the OF "PacketIn" messages that emerged in the presence of high message peaks of OF messages to manage. Furthermore, analyzing the traffic of the OF connection, it can be seen that during the peaks of OF messages, the messages are exchanged between the two sides of the connection according to a series of messages of the same type grouped together. This is a direct consequence of the request handling method in which a series of N OF "PacketIn" messages are accepted and handled until the “send_q” queue becomes full, then the `GreenThread`

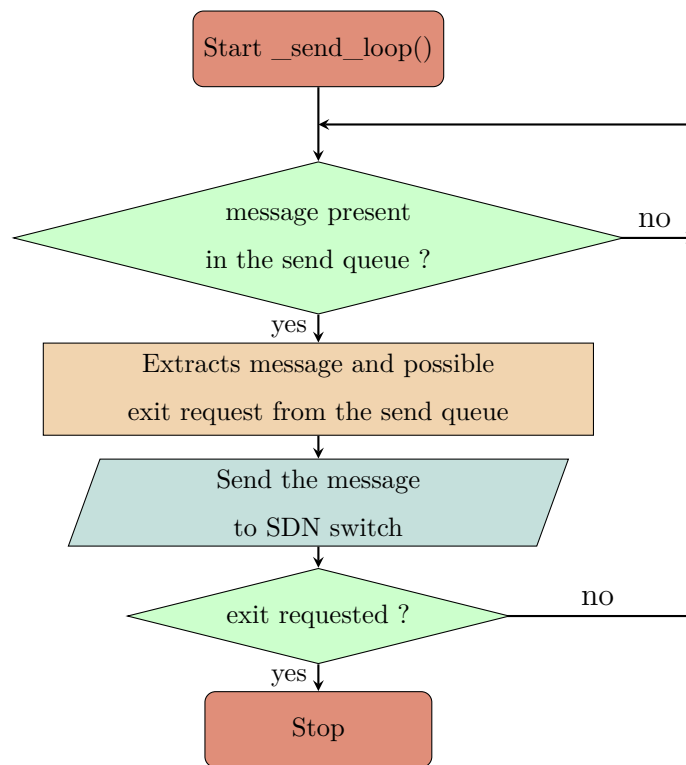


Figure 2.24: Flowchart of the Ryu procedure `_send_loop()`

"send_loop" starts and empties the queue by sending a series of OF 'FlowMod' and "PacketOut" messages after which the GreenThread starts running again. This data sending scheme restarts ending only at the end of the peak of OF "PacketIn" messages.

If the number of OF messages per second arriving at the Controller socket is greater than the number of messages extracted from the socket itself and processed by Ryu in one second, the socket queue would never be empty. Consequently, the GreenThread running `_recv_loop()` would keep control of the CPU preventing the scheduling of the other GreenThread for a time equal to that necessary to process 2048 messages after which it would release the CPU allowing to the GreenThread running `_send_loop()` to send all the OF requests in the send queue "send_q" and to the GreenThread running `_echo_request_loop()` to send any OF "echo

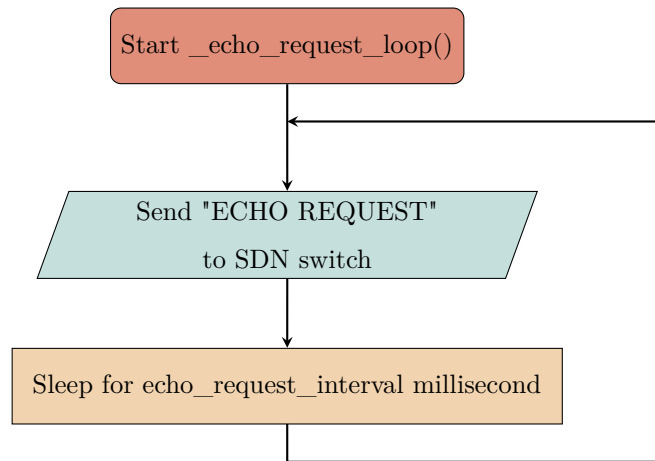


Figure 2.25: Flowchart of the Ryu procedure `_echo_request_loop()`

request". It should be noted that the time `_recv_loop()` remains running may be less than this because when the message queue reaches the limit of 16 elements it automatically blocks the `_recv_loop()` (which inserts the data into it). Not all OF messages require a response to be sent to the SDN switch, so the queue of messages to be sent can grow at a rate lower than that of OF packets received.

These are the reasons why at a peak of "PacketIn" OF message requests there is a rapid increase in the time required to complete the request management with consequent significant delays in sending the OF "FlowMod" and "PacketOut" messages. The "PacketOut" OF message linked to a "Packet_In" OF message can remain stuck in the "send_q" queue until the "recv_loop" processes up to a maximum of 8 OF "PacketIn" messages after which the queue becomes saturated, the "recv_loop" goes into sleep and the "send_loop" goes into execution disposing of the packets to be sent. This as presented in Fig. 2.26 causes the increasing delays which emerged during all the tests.

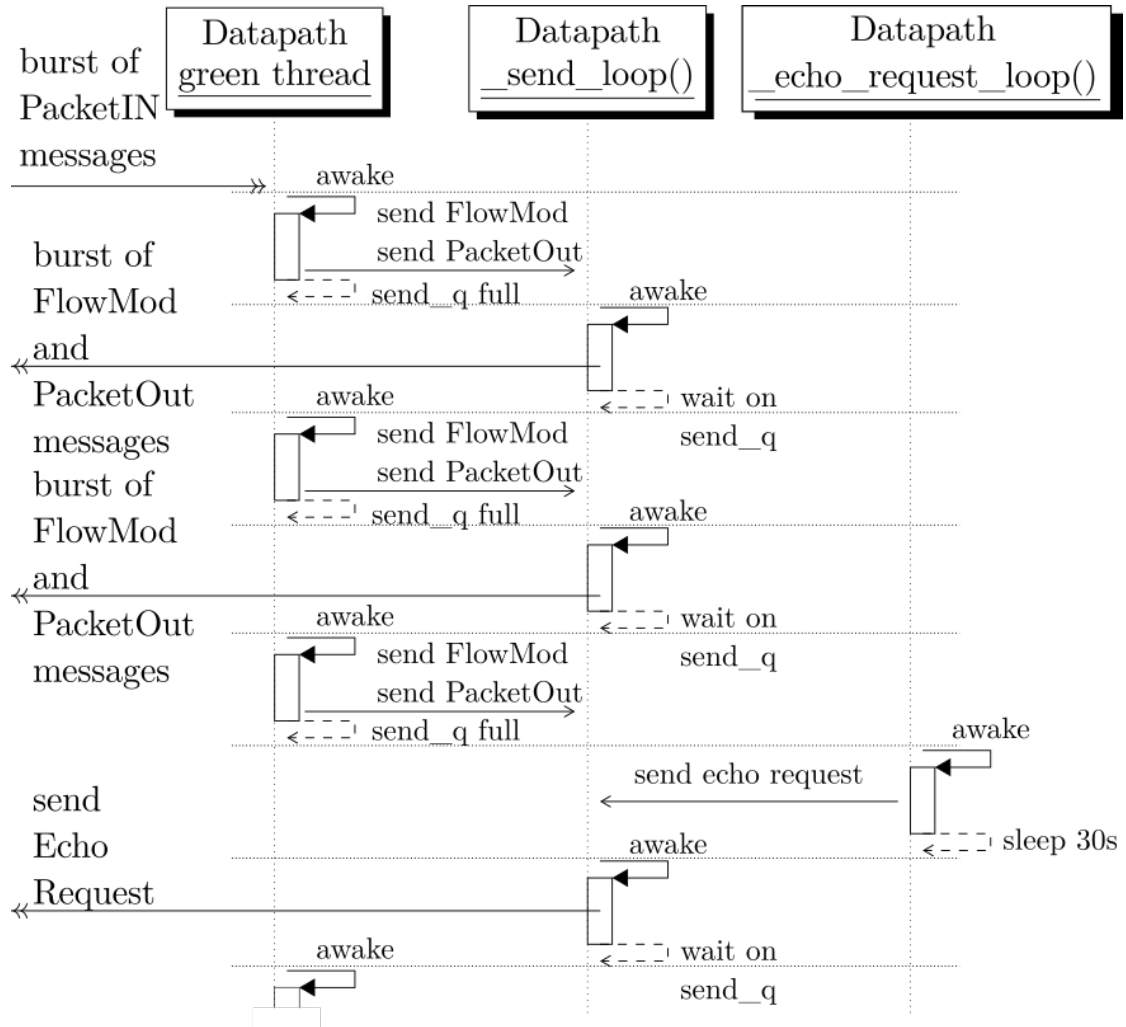


Figure 2.26: Result of message handling logic implemented in Ryu "recv_loop"

2.8 Auxiliary scripts to support experiments

Bash scripts or Python programs have been created to facilitate and automate the execution of the tests by monitoring the results in real-time through special graphs and saving all the metrics of our interest in special files. These scripts have been gradually simplified and integrated with each other, in order to reduce the operations to be performed and ensure better replicability of the tests themselves. Most of these scripts save the acquired data both in a text file in Comma-Separated Values (CSV) format, and in a database of type Round-Robin Database (RRD)¹⁷. This database is used to automatically create graphics in Portable Network Graphics (PNG) and Scalable Vector Graphics (SVG) format every minute. These results and graphs can be viewed in real-time even remotely through a web page or an Secure File Transfer Program (SFTP) connection. The CSV file can be imported into many software (MatLab, Excel, OpenOffice, etc) for further analysis.

All the operations necessary to start and monitor the tests can be managed both from the local console and from the remote console (Secure SHell (SSH) protocol). To ensure the operation of the commands even in case of interruption of the SSH session from which you operate, the software Tmux¹⁸. The tests are carried out through the use of four programs executed in parallel within the SDN controller:

1. the Ryu software containing the IDS module;
2. the debug script;
3. the script for calculating network packet delays;
4. the script that sends the packets to be scanned.

¹⁷High performance data logging and graphing system for time series data <https://oss.oetiker.ch/rrdtool/>

¹⁸tmux: terminal multiplexer <https://github.com/tmux/tmux/wiki>

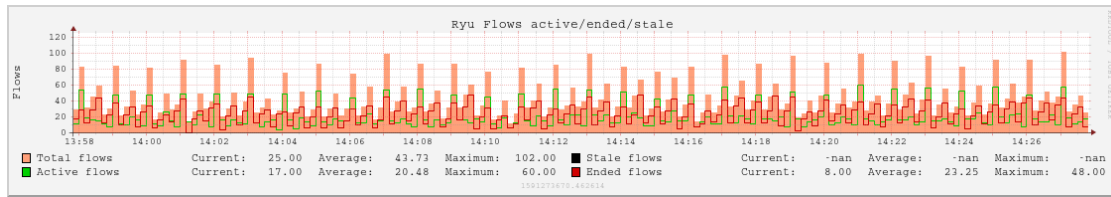


Figure 2.27: active and terminated flows (within statistical calculation window)

Since all these scripts are run within the same machine, the used clock is common. Therefore, problems related to computers not perfectly synchronized with each other are avoided and the times detected by them can be directly compared

The first console runs Ryu-based SDN-SF-IDS software using the command
`ryu-manager switch_IP.py 2>\&1 | tee -a log.txt`

The output, which contains a large amount of information (received OF messages, cataloging of flows, times used, etc), is displayed on the screen and simultaneously saved in the "log.txt" file.

In the second console, the debug script analyzes both the status of the OS (used memory, etc) and the "log.txt" file, extracting various information (including the time $\Delta T_{PacketIn}$). The extracted data is stored in CSV and RRD format. The latter is used to create the following graphs every minute: 1. active and terminated flows (within the statistical calculation window) (figure 2.27); 2. total delay ΔT_{TOT} of network packets (figure 2.28); 3. number of detected streams separated by type (IP/TCP/UDP)(figure 2.29); 4. number of True positive, False positive, True negative, False negative flows (figure 2.30); 5. Time used to analyze the statistics (figure 2.31); 6. time $\Delta T_{PacketIn}$ spent processing the OF "PacketIn" message (figure 2.19 and figure 2.20); 7. Memory used by the OS and by Ryu (figure 2.32); 8. Network throughput (figure 2.33);

In the third console, the script that calculates the network packet delays is executed. The fourth console sends the packets to be analyzed.

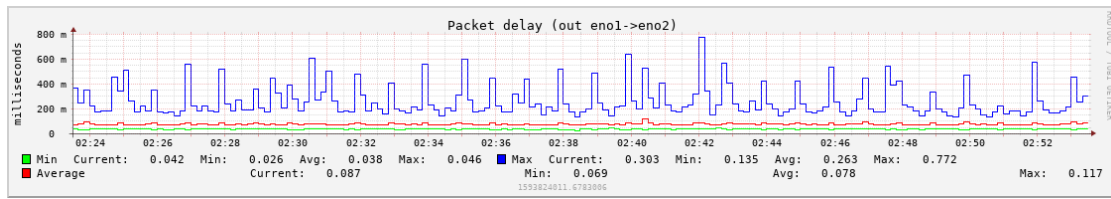


Figure 2.28: total delay ΔT_{TOT} of network packets

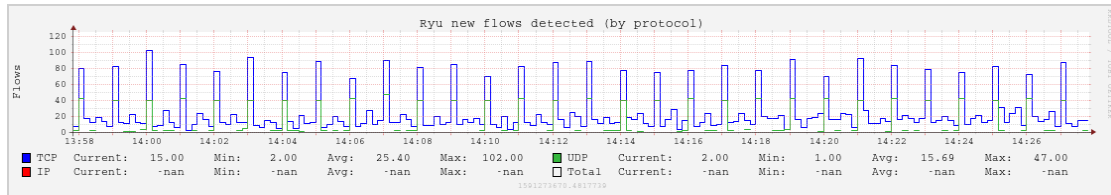


Figure 2.29: number of detected streams separated by type (IP/TCP/UDP)

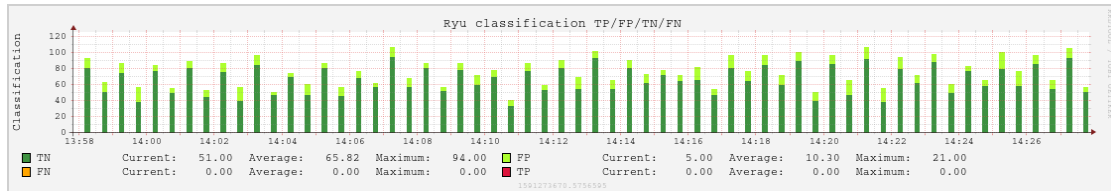


Figure 2.30: number of True positive, False positive, True negative, False negative flows

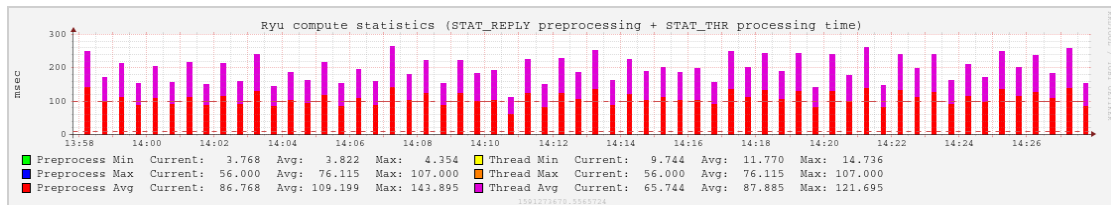


Figure 2.31: Time used to analyze the statistics

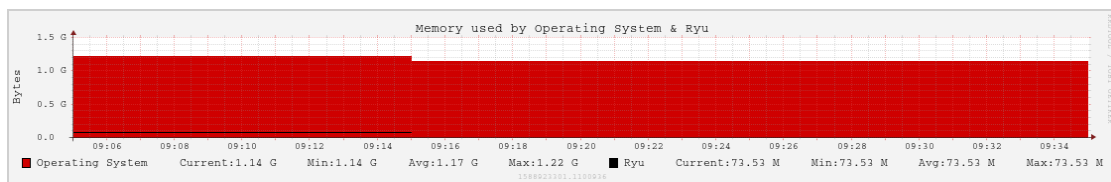


Figure 2.32: Memory used by the OS and by Ryu

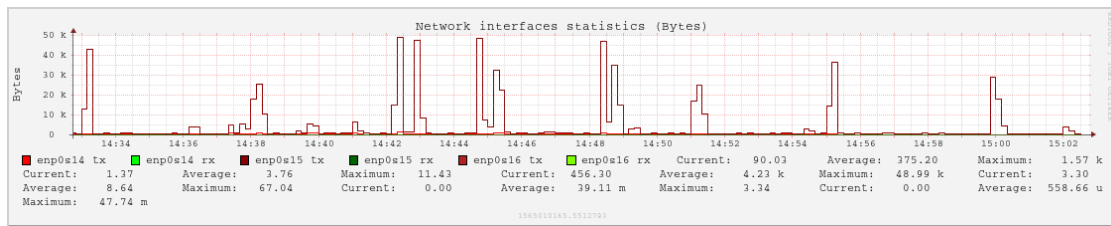


Figure 2.33: Network throughput of OS ethernet interfaces

The delay calculation method depends on the correct and univocal identification of the network packets. The two previous scripts are closely related to each other by the unique identification method of network packets.

Two methods have been developed to associate a unique identifier to a Ethernet packet that allows a certain identification. The first method uses a system based on the comparison of the Hash MD5 of the Ethernet packet, while the second method is based on the direct use of the source MAC Address field of Ethernet packet.

In case we rely on Hash MD5, the two scripts behave as described below.

To uniquely identify a single network packet, the Hash MD5 value calculated on the entire Ethernet packet is used. It is assumed that hash collisions do not occur, that is if there are no different packets that give the same hash value. Unfortunately, this method does not allow to distinguish two identical packets sent at different instants. This problem is detected by the script when it receives more than one pair of identical Hash MD5. In this case, to avoid errors, the script ignores all Ethernet packets whose Hash MD5 occur more than twice.

The script uses the "TShark" network packet capture software to monitor all packets that are sent or received by the "*ethA_{EXT}*" and "*ethA_{INT}*" interfaces of the SDN Controller. These interfaces, as seen in Fig. 2.11, are connected respectively to port #1 (input) and port #2 (output) of the SDN Switch B_4 or B_5 . The time calculation script executes the "TShark" capture software with appropriate param-

```
tshark -te -o frame.generate_md5_hash: TRUE -i ethAEXT -i ethAINT  
→ -Tfields -e frame.md5_hash -e frame.time_epoch -e frame.number  
→ -e ip.src -e ip.dst -e tcp.srcport -e tcp.dstport -e  
→ udp.srcport -e udp.dstport -Y "not stp and not arp and not  
→ loop"
```

Listing 2.1: TShark command with used parameters

eters (listing 2.1) in order to obtain for each packet received from the "*ethA_{EXT}*" and "*ethA_{INT}*" the following data:

1. the time instant when the network interface receives or sends the packet;
2. the value of the Hash MD5 of the entire Ethernet packet;
3. the fields of interest of the Ethernet and IP protocols (Ethertype, source and destination IP addresses, IP protocol conveyed (TCP/UDP), TCP/UDP source and destination port, etc.).

The script receives all this data in real-time through a pipe connected to the "Tshark" command executed in the background and stores them temporarily in a support structure.

This data is associated with each other through the same Hash MD5 values every minute. The analysis result is saved in appropriate files in CSV format.

These data are linked to the output of the same packet from the "*ethA_{EXT}*" interface and to its reception from the "*ethA_{INT}*" interface after undergoing the delays introduced by the passage inside the SDN-SF-IDS. So, if the interface "*ethA_{EXT}*" sends a packet P with value $MD5_P$ of Hash MD5 at time T_1 and a packet with the same value $MD5_P$ reappears on the interface "*ethA_{INT}*" at time T_2 the time ΔT_{TOT} taken by the P packet to traverse the system SDN-SF-IDS is

equal to $T_2 - T_1$. Since the associations are made every minute, it is not possible to measure delays longer than this time interval.

The data submission script uses the "tcprewrite"¹⁹ program to send to the SDN-SF-IDS, through the " *ethA_{EXT}* "connected to the network to be filtered, the network packets previously captured and stored in a Pcap file. The packet transmission takes place in compliance with the original timing sequence of the captured flow, otherwise the statistics would be distorted. Furthermore, through specific commands, it is also possible to carry out the port mirroring of the main interface of the system so that the packets sent or received by it are automatically replicated and sent from the "*ethA_{EXT}*" interface to the SDN switch.

Using the packet identification method based on the Hash MD5 of the Ethernet packets it is complex to correlate the delay times ΔT_{TOT} of a specific packet with the $\Delta T_{PacketIn}$ parsing the associated OF "PacketIn" message. In order to associate a OF "PacketIn" message to the delay suffered by the first packet of the flow that generated it, it would have been necessary to modify the SDN-SF-IDS code in order to calculate and print the Hash MD5 of the packet Ethernet linked to OF "PacketIn" messages. But this is not possible as the OF "PacketIn" message contains only a limited part of the Ethernet packet to be parsed. Moreover, the Hash MD5 calculation operation would have added unjustifiable delays to the management of the OF "PacketIn" message itself.

This problem, combined with the fact that tracing based on Hash MD5 is not able to manage the sending of multiple copies of the same Ethernet packet, has led to the creation and use of the second method of packet identification which does not present these problems. But it is more invasive.

The second method used to uniquely trace a Ethernet packet is based on changing the Ethernet "source MAC Address" field of each packet to be monitored with

¹⁹<https://tcpreplay.appneta.com/wiki/tcprewrite>

that it is possible to follow the temporal evolution of the events related to the reception of individual packets, including any OF "PacketIn" messages (generated by system only upon receipt of the first packet of a new stream).

The delay calculation script detects the instant T_1 when each packet is sent through the "*ethA_{EXT}*" interface and the instant T_2 when it is received by the "*ethA_{INT}*". At the same time, the monitoring script extracts the information regarding the instant in which the handling procedure of the OF "PacketIN" message associated with the new flows arriving at the SDN controller begins and ends. Given that the Ethernet and IP headers of the packet to be analyzed (for which the OF "packetIN" message was sent) are contained within the OF "PacketIn" message, it is possible to use the "source MAC Address" to uniquely identify it. The data is saved on two separate files, but in this case both contain the identifier so it is easy to merge all the information together.

2.9 Final prototypes

Once this phase of optimising the software switch and the SDN-SF-IDS code inside the SDN Controller A_1 was completed, the knowledge acquired was used to create the final prototype which can use both types of SDN switches (software or hardware) in order to compare the delays.

This prototype, housed inside a Rack (19-inch), consists of the Controller A_2 , a software SDN switch B_4 and a hardware SDN switch B_5 . The A_2 controller is equipped with an Intel[®] Xeon[™] E3110 CPU running at 3.00GHz, 2 GByte of RAM and 3 Ethernet 10/100/1000 Mbit/s cards. The B_4 software switch is equipped with an Intel[®] Core[™] i5-3450S CPU running at 2.80GHz, 8 GB RAM and 3 Ethernet 10/100/1000 Mbit/s cards. The B_5 hardware switch consists of an HPE Aruba M2940-48G switch with firmware WC.16.07.0003.

proto- type	Controller A_2 : CPU Intel® Xeon™ E3110 running at 3.00GHz, 2 GByte of RAM, 3 Eth 1000 Mbit/s description of SDN switch B_n	ΔT_{TOT}
A_2B_4	switch software CPU Intel® Core™ i5-3450S running at 2.80GHz, 8 GByte RAM, 3 Eth 10/100/1000 Mbit/s	96% of packet \leq 10ms, peak of 150ms
A_2B_5	switch hardware HPE Aruba M2940-48G WC.16.07.0003	98% of packet \leq 10ms, peak of 150ms

Table 2.2: Summary of prototype A_2B_4 and A_2B_5

Table 2.2 shows a summary of the ΔT_{TOT} obtained with the two configurations of the final prototype.

In A_2B_4 and A_2B_5 systems the CTRL dedicated connection is not present and the OF communications pass through the local network.

Following the introduction of the hardware B_5 SDN switch in the SDN-SF-IDS system, slight changes to the logical structure of the OF rules inserted by the SDN controller A_2 in the OF flow table of SDN switches was applied in order to use OF rules compatible with the hardware tables present in switch B_5 .

Figure 2.36 presents the logical structure of the OF rules as used by the A_1 controller. In figure 2.37, the new logical structure of the rules compatible with the OF hardware accelerated flow table as used by the controller A_2 is presented.

Therefore, the SDN A_2 controller is able to operate both with the SDN B_4 switch and with the SDN B_5 switch using the same rules for monitoring, forwarding or dropping streams (see figure 2.37).

The ΔT_{TOT} delay measurements introduced by the SDN-SF-IDS A_2B_4 system (with SDN software switch) were repeated before and after the changes to the logic used in the OF flow table verifying that the changes did not cause substantial differences in the delays suffered by the packets due to the passage in the SDN-

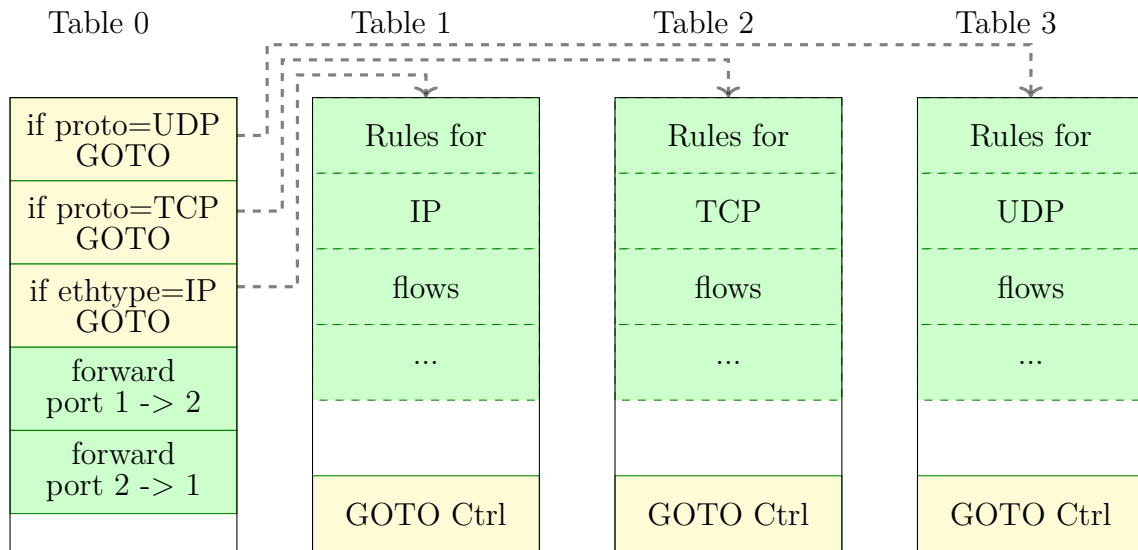


Figure 2.36: logical structure of the OF rules as used by the A_1 controller. Incompatible with SDN hardware switch.

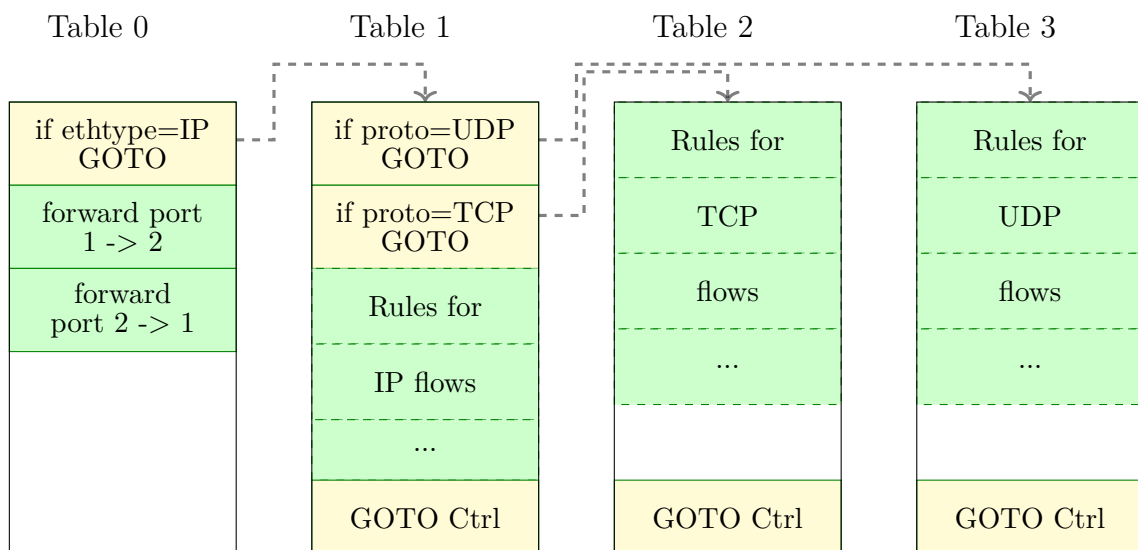


Figure 2.37: logical structure of the OF rules as used by the A_2 controller. Compatible with both SDN hardware switch and SDN software switch.

SF-IDS system.

The total delay measurements ΔT_{TOT} introduced by the SDN-SF-IDS A_2B_5 and A_2B_4 have been taken. The results obtained gave comparable results.

Therefore it is believed that the software B_4 SDN switch has been satisfactorily optimized and can be used indistinctly in place of the hardware B_5 SDN switch. Both solutions are viable and can be chosen depending on the cost or needs.

2.10 Delay measurement ΔT_{TOT}

The results presented in this section were obtained using the final prototypes A_2B_4 and A_2B_5 . The traffic analyzed consists of the entire traffic of the laboratory WAN network and all the Pcaps at my disposal containing Botnet.

2.10.1 Method of measurement

For each packet sent in input to SDN-SF-IDS and received in output from SDN-SF-IDS the sending and receiving times, the delay, the IP or the Ethertype of the packet, the source/destination IP address and port, the length of the received Frame ethernet.

The data is stored for each individual packet without making any time average.

2.10.2 Loading data

The saved data is of considerable size as it contains millions of samples.

During data loading, the data throughput (equation 2.1) is calculated over an interval of 1 second,

$$Throughput_i = \sum_{j=T_i}^{T_{i+1s}} packetSize_j \quad (2.1)$$

the average delay (equation 2.2) over an interval of 1 second,

$$AverageDT_i = \frac{1}{n} \sum_{j=1}^n \Delta T_j : T_j \in [T_i, T_i + 1s] \quad (2.2)$$

the maximum delay (equation 2.2) over an interval of 1 second,

$$MaxDT_i = \max_{j \in [1, n]} \Delta T_j : T_j \in [T_i, T_i + 1s] \quad (2.3)$$

Due to the large number of samples, analyzing the data and creating graphs requires many quarter of hour and a very large amount of memory.

2.10.3 Measurement graphs

For each test carried out, the following graphs are generated:

- Throughput of OF control socket (figure 2.47);
- total throughput of packets forwarded to the system;
- delays suffered by individual network packets (figure 2.38);
- Histogram of count of packet received with delays included in the time interval from 0 to 200ms with subdivision every 10ms (figure 2.38);
- pie chart of the percentage of packets received as the delay varies, grouped into 10ms intervals (figure 2.38);
- Histogram of count of packet received with delays included in the time interval from 0 to 20ms with subdivisions of 1ms; (figure 2.38).

The plot of delays suffered by individual network packets is made up of a Scatterplot graph in which for each received packet a circle of 2 pixels is drawn at the coordinates with abscissa equal to the sending time and ordinate equal to the

ΔT_{TOT} delay in milliseconds. The density of the points is therefore proportional to the number of packets received in that specific time and delay interval.

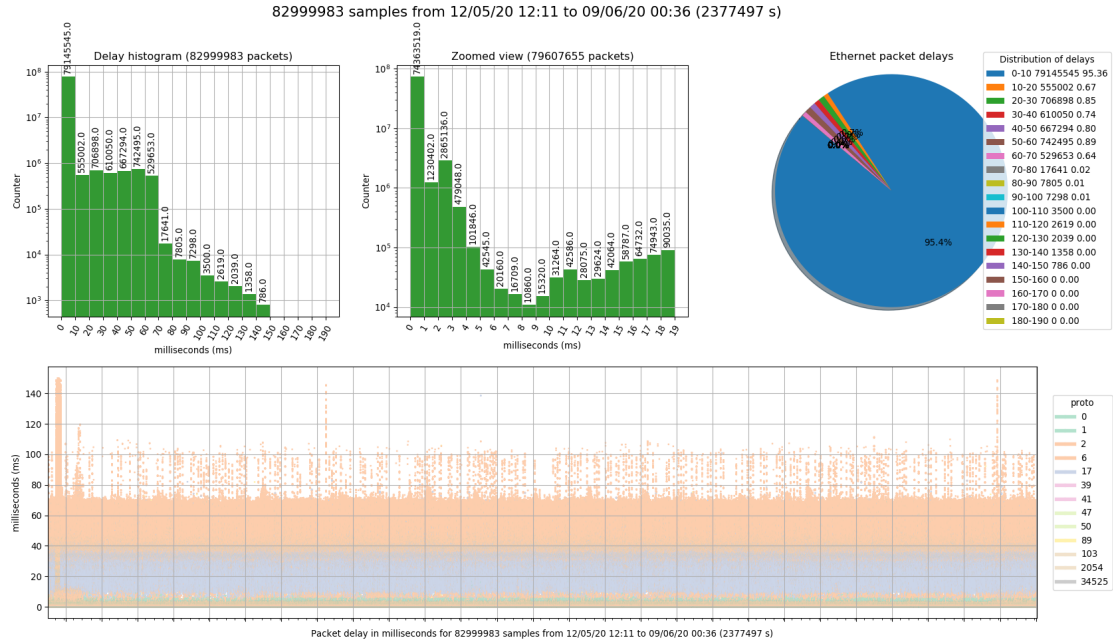


Figure 2.38: Count of packets histogram grouped by its delays. Different color has been used for each protocols.

2.10.4 System A_2B_5 (hardware SDN switch)

The results were obtained with previous versions of the analysis script which presented only the total graphs without creating the graphs divided by protocol.

The 93.9 % of packets are delayed between 0-5ms, 4.20 % between 5-10ms (see table 2.3).

samples	ethertype	Protocol	max delay	delay range	% of packet
10 000 001	all	all	150	0-10	98.1

Table 2.3: Delays ΔT_{TOT} of switch B_5 in milliseconds measured during the observation interval of one day

2.10.5 Measurements of packet forwarding delay SDN software B_4 switch in bridge mode

The bridge mode represents the absolute minimum obtainable delay and corresponds to having all necessary OF rule installed in the OF flow table of SDN switch for all possible flows without having to start the OF "Packet In"/"Flow Mod"/"Packet Out" message exchange.

All packets cross the bridge without triggering any interaction with the SDN controller. Almost all packets passed through the switch with delays ranging from 0 to 1 ms. Only 16 packets on 58 749 (0.027 %) took between 1 and 5 ms.

2.10.6 System delay measurements A_2B_4

Measurements have been made with increasing number of samples:

1. 1 000 001 samples, about 6 hours in the afternoon: the data stream contains the network traffic downloads with episodes of web browsing traffic.
2. 10 000 001 samples, about one day: the data stream contains the downloaded network traffic with episodes of web browsing traffic.
3. 82 999 983 samples, about one month: the data stream contains the network traffic downloads with episodes of web browsing traffic.

As expected, from the graphs (2.41 - 2.46) there is a clear dependence between the delays suffered and the protocol used. This dependence is linked to the number of OF rules checked (see figure 2.37) before being forwarded by any interaction with the SDN Controller.

- Non IP protocols are forwarded after a couple of interactions with the rules of the first OF flow table with minimal delays;
- IP protocols other than UDP and TCP are forwarded once the second OF flow table is reached and suffer longer delays;
- The protocols IP TCP and UDP are forwarded once the second or third OF flow table is reached respectively, assuming they have already been analyzed (slightly higher delay than the previous). If not, OF messages are exchanged with the Controller causing longer delays.

As the number of samples contained in the pcap file used varies, the trends remain practically similar except for small variations linked to the times of the day and the type of flows captured.

Table 2.4 shows the delays in milliseconds measured during the observation interval of one month (82 999 983 samples).

As you can see in the figure 2.39, 2.41, 2.42 and 2.43 the histogram of the delays of the UDP protocol shows a stable plateau between 0-40ms which rapidly decreases between 40 and 90, reaching zero on 110ms. So there is no packet with delay higher than 110 ms with the exclusion of 2 packet with delay between 130 and 140ms. The connectionless nature of UDP linked to the fact that often the source and destination ports change for each packet causes a high interaction with the controller that has to catalog the new flows and insert the appropriate rules. This leads to noticeable increases in delay, a considerable number of packets reaching 40ms delays.

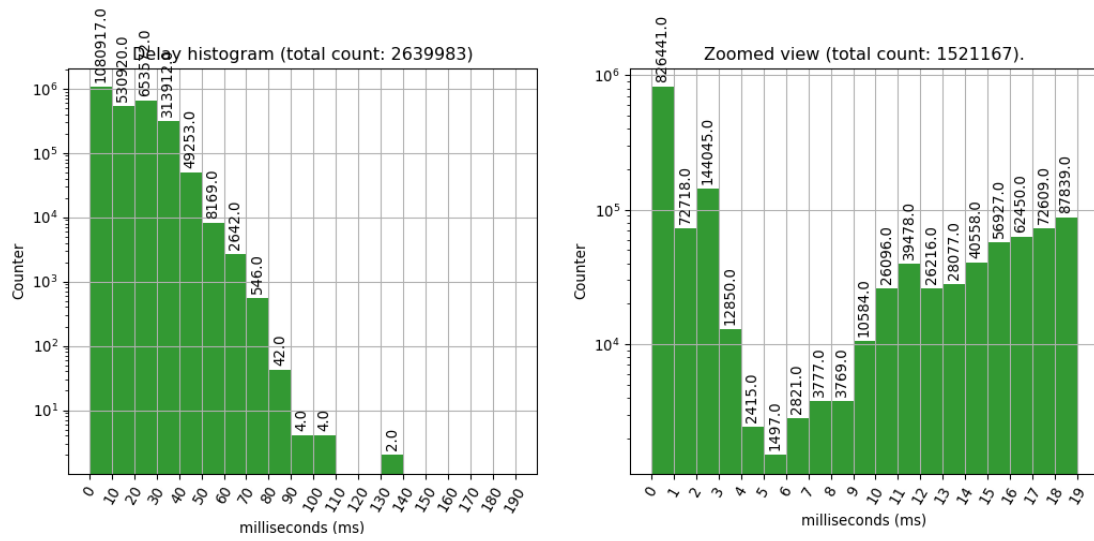


Figure 2.39: UDP packet delay histogram (network communications of a month)

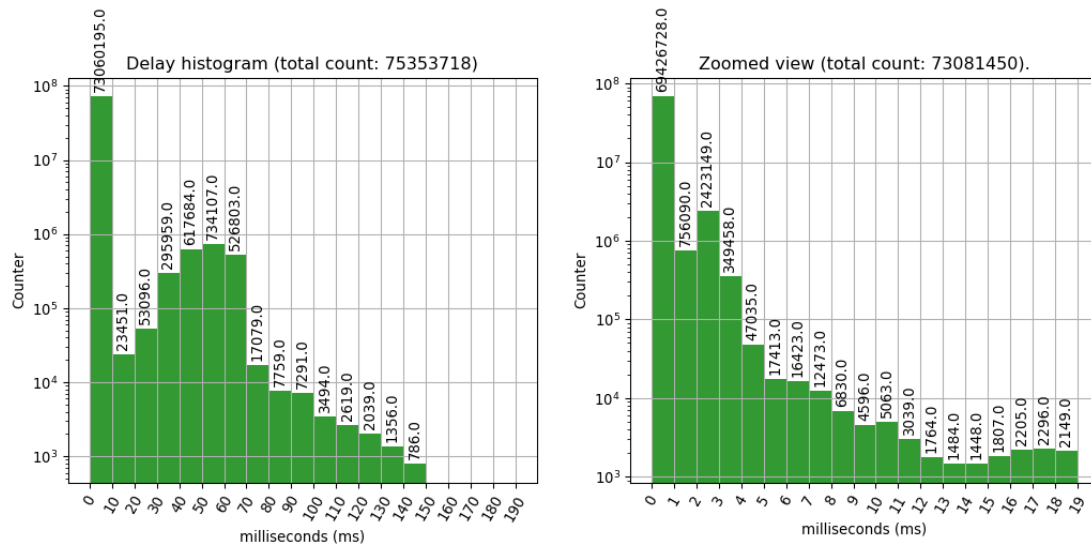


Figure 2.40: TCP packet delay histogram (network communications of a month)

samples	ethertype	Protocol	max delay	delay range	% of packet
146	0x0800	0x00 IPv6 Hop by Hop	1	0-1	100.00
411 485	0x0800	0x01 ICMP	80	0-6	99.38
195 775	0x0800	0x02 IGMP	110	0-10	99.82
75 353 718	0x0800	0x06 TCP	150	0-10	96.96
2 639 983	0x0800	0x11 UDP	140	0-40	97,70
1 188 637	0x0800	0x27 TP++	8	0-1	99.99
3	0x0800	0x29 IPv6 encapsulation	3	2-3	100.00
2	0x0800	0x2F GRE	18	2-18	100.00
1 183 910	0x0800	0x32 ESP	7	0-1	99.99
237 477	0x0800	0x59 OSPF	110	1-3	99.47
80 419	0x0800	0x67 PIM	100	1-3	99.52
1 680 234	0x0806	ARP	9	0-1	99.99
28 194	0x86dd	IPv6	2	0-1	99.98
82 999 983	all	all	150	0-10	95.36

Table 2.4: Delays ΔT_{TOT} in milliseconds of B_4 measured during the observation interval of one month (82 999 983 samples)

As you can see in the figure 2.40, 2.44, 2.45 and 2.46 the histogram of the delays of the TCP protocol shows a peak between 0-4ms, a few samples (0.1 %) between 4-30ms, a new sample increment between 30-70 (2.88 %) which slowly decreases until it reaches 150ms. The peak between 0 and 4ms is linked to the packets belonging to a flow already present in the OF flow table while the second peak between 30-70ms is linked to the interaction with the controller for the insertion of the relative rule in the OF flow table.

The OF control channel flow throughput plot shown in figure 2.47 together with

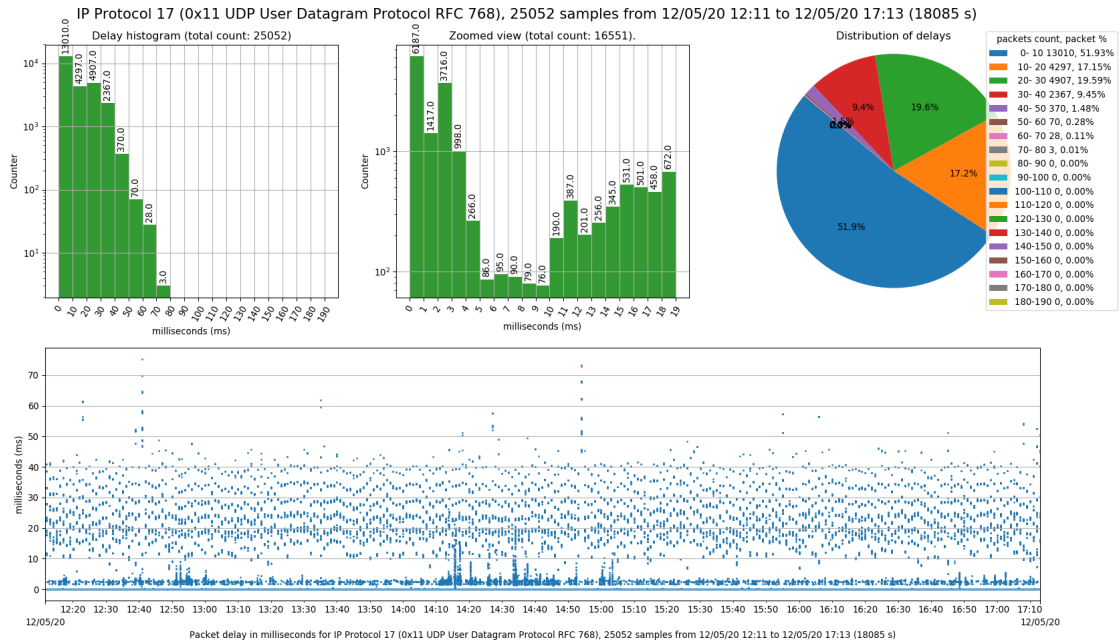


Figure 2.41: UDP packet delay histogram (network communications of five hours)

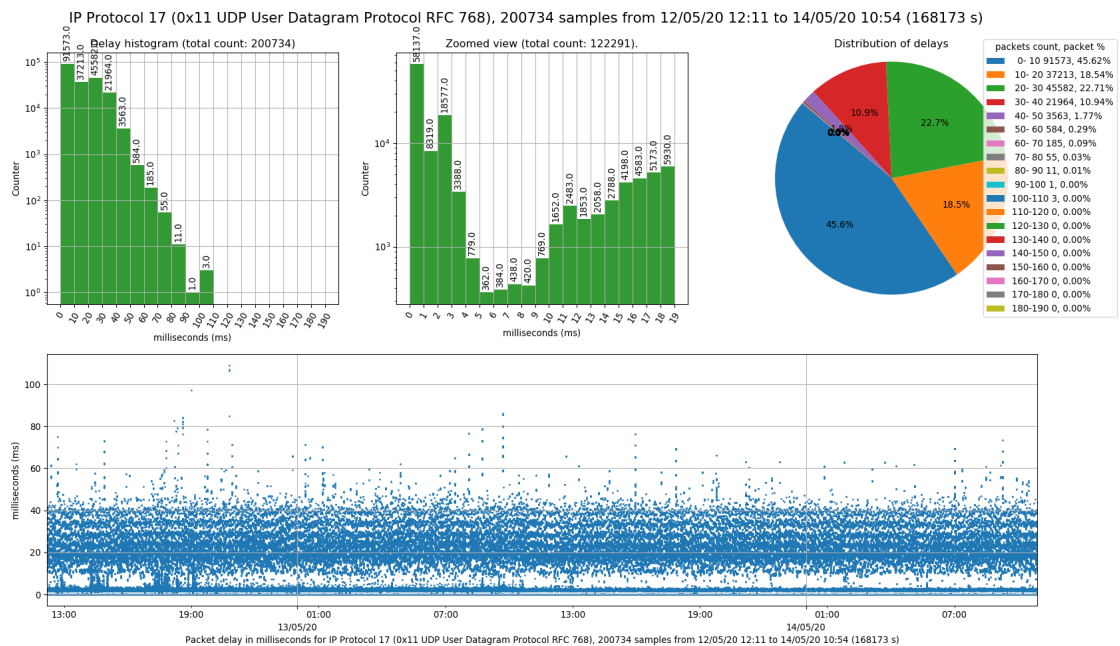


Figure 2.42: UDP packet delay histogram (network communications of two days)

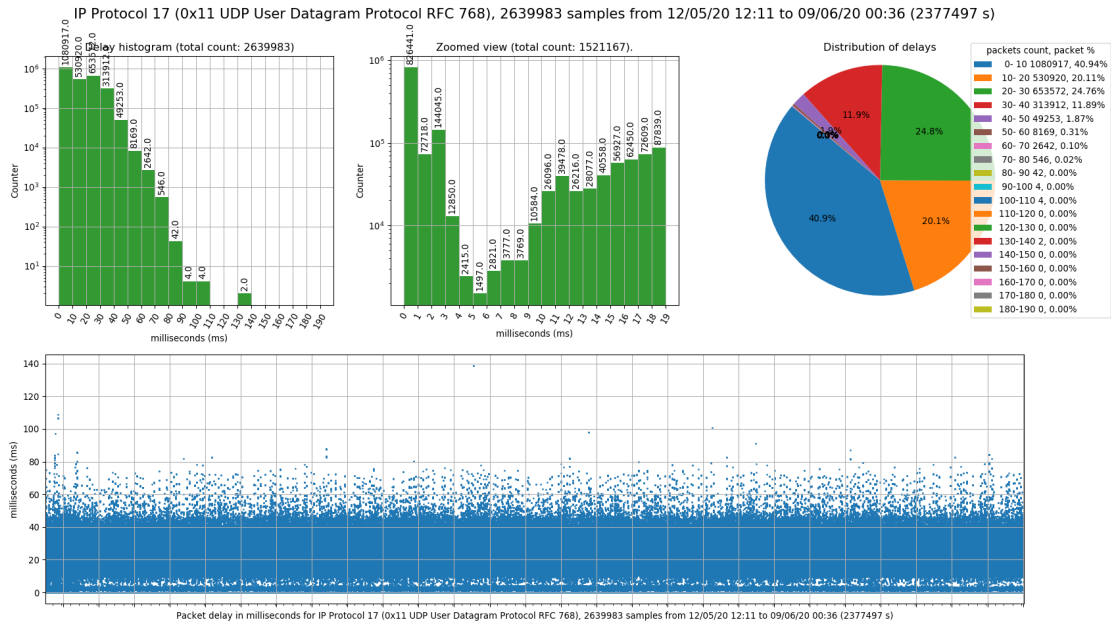


Figure 2.43: UDP packet delay histogram (network communications of a month)

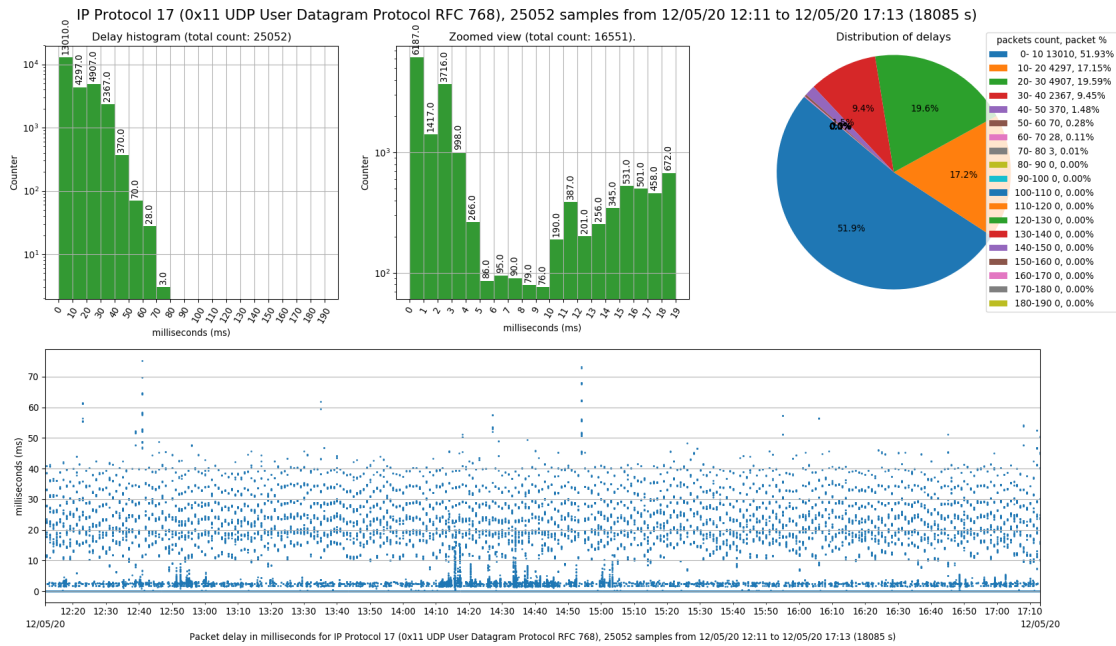


Figure 2.44: TCP packet delay histogram (network communications of five hours)

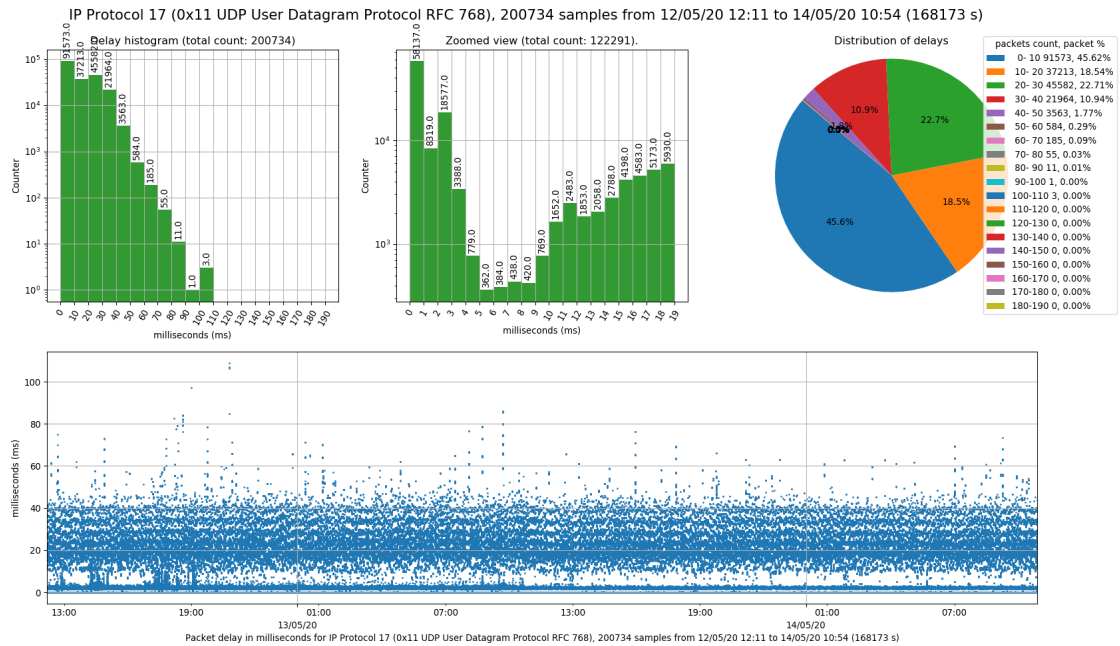


Figure 2.45: TCP packet delay histogram (network communications of two days)

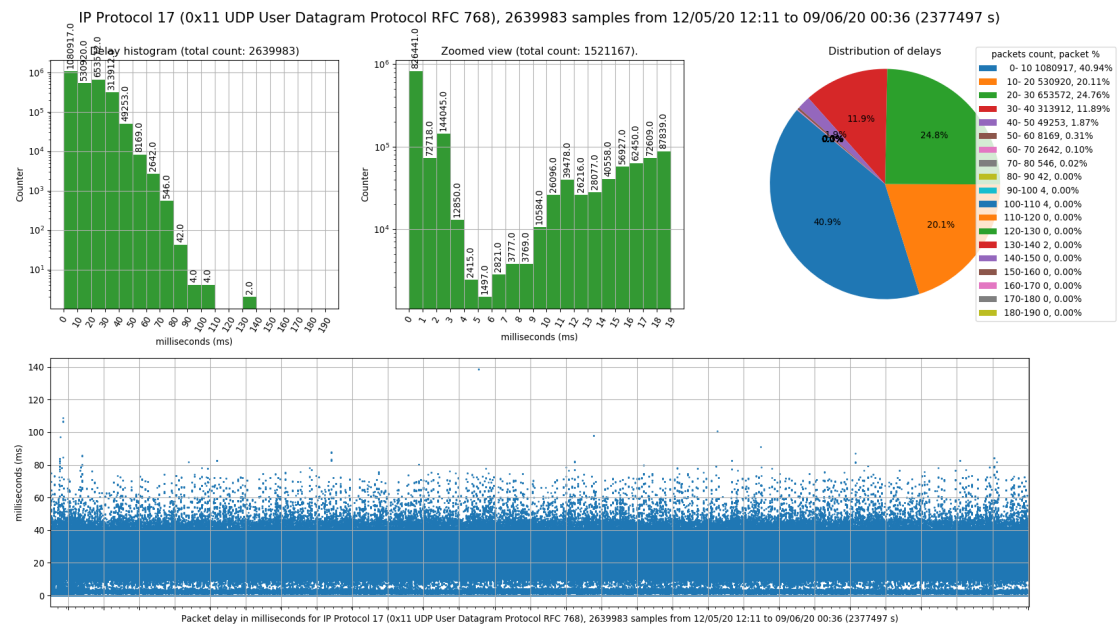


Figure 2.46: TCP packet delay histogram (network communications of a month)

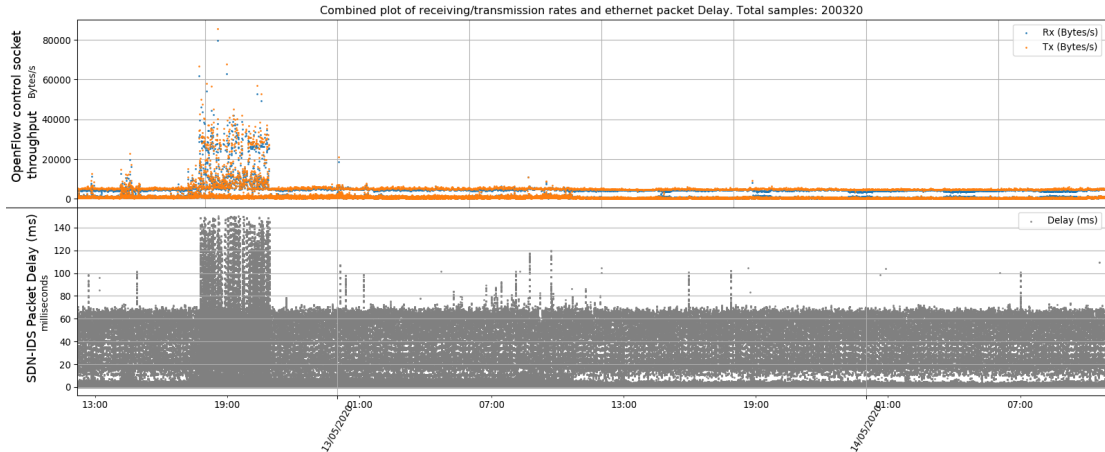


Figure 2.47: Throughput of OpenFlow control channel

δT_{TOT} of SDN-SF-IDS has remained fairly constant and does not consist of high data throughput. However, it should be noted a peculiar episode in which there is a marked increase in traffic in correspondence with a peak of ΔT_{TOT} delays. This event appears to be related to a system overload that caused a large part of the memory to swap into disk swap partition resulting in a large decrease in computational performance.

In order to guarantee the reliability of a SDN software switch, it is necessary to balance the computational resources of the system in the best possible way and avoid carrying out operations that could lead, even only momentary, to shortage or lack of resources (RAM, CPU, disk space, etc).

2.11 Conclusions

The delays introduced by the SDN-SF-IDS system have been significantly reduced.

The B_5 hardware switch has slightly better performance than the B_4 software switch (optimized with DPDK) but is still comparable. A software SDN switch to have a high level performance, comparable to an hardware SDN switch, must use

the DPDK libraries which require professional network cards, thus increasing the price of this solution. To date, the cost of a B_4 computer configured to operate as a high performance SDN switch is less than the cost of a hardware SDN switch B_5 . However, a non-perfectly balanced software SDN switch can get blocked due to unexpected software problems such as an unexpected software upgrade, an overload of the CPU or an exhaustion of available memory leading to swap memory on disk can ruin the performance of the system.

The SCADA and GOOSE industrial networks have a very stringent Key Performance Indicator (KPI) for maximum delays [19] [20] [21]. These KPIs imply device-to-device transfer times of less than 20 ms for non-tripping and P2/P3 class messages and less than 100 ms for non-tripping and P1 class messages. The KPI of tripping messages and intervention class P1 messages require transfer times < 10 ms and the transfer times of the KPI of tripping and P2/P3 intervention class messages < 3 ms. For these messages, it is necessary to further reduce the latencies of the SDN-SF-IDS system.

The SDN-SF-IDS code that analyzes the statistics collected by the SDN switch can be separated from the system by moving it to the 5th Generation (5G) edge as it has no particular constraints. This cannot be done with the part of the code that manages the start of the statistical analysis of each individual flow (OF "PacketIn" messages) as the response times to these messages are critical for the global delays of the SDN-SF-IDS system. The traffic generated by the OF connection during the prototype tests was found to be limited (fig. 2.47) but the system is very sensitive to latencies introduced during the management of new flows (OF "PacketIn" messages) The SDN Controller must analyze and respond to these messages as quickly as possible to avoid a rapid increase in delays following the arrival of OF "packetIn" message peaks (section 2.7 and figure 2.13).

The measurement of the delays introduced by the SDN-SF-IDS to the analyzed

network traffic has revealed how a statistical analysis of the distributions of these delays can be used to infer the operations that the system has performed on the analyzed packets. The point of view can be reversed by exploiting the distribution of delays suffered by a packet within an unknown system to characterise a normal operating model and use anomaly detection techniques to identify anomalous behaviours.

For example, it can be assumed that the inter-arrival time between consecutive packets belonging to a periodic exchange of information carried out at predetermined intervals is sufficient to be able to infer any unexpected and potentially anomalous delays. This technique was used to build the IDS for an IoT infrastructure discussed in the next chapter.

2.12 Future work

The SDN Controller is based on server machines with limited resources when compared to today's standards. Currently, the amount of memory required to run the system is less than 2 GByte and an old bi-processor system can be used without problems (PowerEdge R200 from 2010). In the next months i can have at my disposal a more powerful system, and I can check if there is still room for performance improvement.

I have purchased a network card with higher performance to verify if the performance of software SDN switch can be further improved or if lead only to a sharp increase in system costs.

The Ryu message reception and analysis loop could be modified using an architecture that uses Multithread programming with Preemptive scheduling could process these events in parallel taking the best out of current multiprocessor CPU and reducing delays in handling OF messages. But the complexity of the code can

grow considerably.

Enabling the use of multiple parallel Thread the controller can process received OF messages in parallel. A slowdown in parsing a packet would only slow down one Thread while the others would continue to work independently. This could lead to an asynchronous handling of the OF requests and to the violation of the temporal consequentiality of the OF operations as requested by serving in the wrong order the requests. In the case of the receipt of two OF requests, if the second request took less time than the first to be completed, any replies would arrive at their destination in reverse order than expected.

It should be noted that using the Python language could come into play the Global Interpreter Lock (GIL) which manages access to the interpreter to permit the use to only one Thread at a time. The presence of GIL is controversial²⁰ as it partially limits the performance of a multithreaded Python²¹. Many developers would like to get rid of it, and there are Python interpreters available that don't use this system²².

The next improvements of the SDN-SF-IDS in order to minimize the problem with the management of bursts of OF requests are to be found in the optimization of the Ryu SDN controller source code or in its replacement with other software able to avoid the aforementioned congestion in receiving OF messages. In the second case it would also be necessary re-implement of the IDS module to allow its integration in a different SDN Controller.

²⁰GIL <https://wiki.python.org/moin/GlobalInterpreterLock>

²¹<http://dabeaz.blogspot.com/2010/01/python-gil-visualized.html> and <http://dabeaz.com/python/UnderstandingGIL.pdf>

²²<https://github.com/larryhastings/gilectomy>

Chapter 3

LoRa gateway IDS

3.1 Introduction

From the distribution of SDN-SF-IDS delays emerged that the timing to process network packets of a specific protocol strictly depends on the operations carried out on them by the system under analysis.

An Information Technology (IT) infrastructure under normal conditions should perform the same operations on the same type of packets to be forwarded and the introduced delays should be similar in case of no anomalies. Observing a whole system as a black box, it is possible to observe an almost constant trend in this timings. This timings can be affected and modified by problems or changes in the software execution.

Furthermore, the measurements carried out on the delays added to packets can be done for any protocols and for packets sent and received through any standard communication media that can be monitored. It is not important to understand the contents of the analysed packets but only the delay they suffer. Therefore, the method can also be applied to proprietary or encrypted protocols.

Devices belonging to a monitoring IoT infrastructure often carry out repetitive

operations to send temperature, pressure and other physical measurements to a central database. These infrastructures are characterised by messages sent with regular intervals with a consequent constant throughput. So, I verified if this technique could be applied to a LoRaWAN infrastructure which consists of a radio link part and a wired link network part. I created an IoT infrastructure consisting of temperature monitoring devices connected to a LoRaWAN network with an IDS able to analyze the packets sent to and from the LoRa gateway to a cloud database platform (by example The Things Network (TTN) server) in order to detect anomalies.

This chapter is structured as follow: introduction of IoT and LoRaWAN infrastructure, presentation of LoRa Forward protocol used for communications between LoRa gateway and the LoRa server, analysis of all information that can be extracted from the intercepted communication and used to characterise the IoT application, presentation of Software Defined Radio (SDR) and LoRa physical layer, description of LoRa jamming testbed and IDS used to detect the presence of the jammer, a presentation of related work on jamming and IDS for LoRa, conclusion and future work.

3.2 IoT infrastructure and LoRaWAN

According to Ericsson mobility report¹, there will be 18 billion IoT devices by 2022. Low Power Wide Area Network (LPWAN) has been introduced to correctly handling the characteristics of IoT devices. IoT refers to the fastest growing network of physical things that can be accessible through the Internet. These variety of IoT applications and devices originate different solutions. When the application needs to transmit a large amount of data over a small area, devices with high

¹<https://www.ericsson.com/en/reports-and-papers/mobility-report>

capacity battery equipped with ZigBee, Bluetooth or WiFi protocols are typically used. Instead, when the application needs to transmit a small amount of data with low bitrate over a large area, devices with limited capacity battery but equipped with LPWAN protocols, such as Sigfox, LoRa, or other LPWAN solutions, are typically used.

LoRaWAN is a specification developed by the LoRa Alliance and got my attention due to the growing number of IoT solutions used in industrial systems, such as smart electricity grids, smart cities, smart agriculture, and healthcare management, that require strong security support.

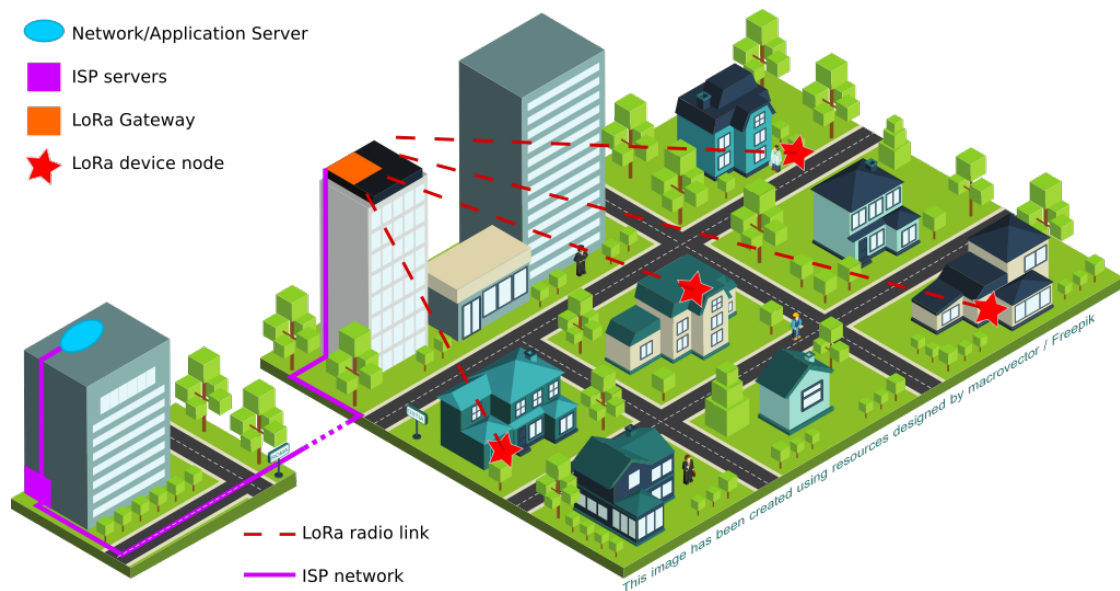


Figure 3.1: Example of a LoRaWAN based IoT network

A LoRaWAN network is composed by LoRa end nodes, gateways, network server(s), and application server(s) as illustrated in Figure 3.1. The LoRa end node transmits packets on Radio Frequency (RF) uplink channels (see table 3.4) and one or more LoRa gateways can receive the same packet. Each LoRa gateway forwards the received packets to the LoRa network server. The LoRa network server can receive one or more copies of the same packet from different gateways but

it forwards just one copy to the LoRa application server discarding the duplicated ones. The network server is also responsible to select the proper LoRa gateway for the downlink transmission.

The LoRaWAN specifications define three different types of end nodes: class A, class B and class C. In our testbed, we use only the Class A end node which uses Pure ALOHA protocol to transmit data on the uplink channel and has two small downlink receiving window after the end of each uplink transmission.

3.3 LoRa packet forwarder analysis

The LoRa gateway we used is based on a Raspberry PI 2 board equipped with a LoRa Modulator-Demodulator (MoDem) Shield and specific software.

From the LoRa gateway, two types of analysis has been conducted on the network layer:

- Analyses of communications between LoRa Gateway and LoRa Server;
- Analyses of communications between LoRa end nodes and LoRa Gateway.

Another type of analyses can be done by using an SDR card to monitor the radio link or act on the radio spectrum. See section 3.6 for more information on SDR card.

3.4 LoRa forward protocol

The LoRa Gateway communicate with the LoRa end nodes by using a LoRa MoDem and with the LoRa Server through an IP network.

The LoRa Gateway packet forwarding application runs on the Raspberry Pi and does the following operations:

1. Read the received LoRa packets from the LoRa MoDem and forward them to the LoRa Server through an IP network;
2. Receive LoRa packets from the LoRa Server and send them on the air by using the LoRa MoDem;
3. At predefined time intervals, this application sends internal state information and statistics to the LoRa Server.

This application is open source and is available on github repositories² in different version depending on the used hardware.

To monitor the communication between the LoRa end nodes and the LoRa server, it is sufficient to acquire all network UDP packets that use 1700 as source or destination port.

At a first analysis, the protocol is composed by a binary encoded header sometimes followed by a JavaScript Object Notation (JSON) Object written in textual format with only the LoRa message item scrambled and not understandable. So at least part of the information transmitted can be inferred from network capture without knowing the protocol.

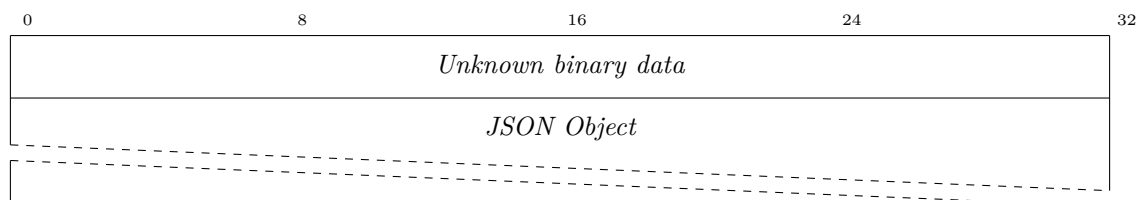


Figure 3.2: LoRa packet structure before complete understanding of all data field

Using the public information contained in the github repositories, LoRaWAN communications can be identified knowing how the Long Range network (LoRa-

²<https://github.com/LoRa-net>

net) packet forwarder protocol is structured. I found the documentation for the revision 1.4³ and 1.5⁴ of LoRa packet forwarder specification. Similar information can be found in a public document written by Semtech (Revision 1.0 – July 2015)
5

The first two documentations seem different version of the same protocol. The "ANNWS.01.2.1.W.SYS" application note seems strictly related to the LoRa-net revision 1.4 but differs substantially in Push Ack binary format. Also in JSON Object some names and structures are different. substantially In the following section the LoRa-net documentation (revision 1.4 and 1.5) has been used but any differences with "ANNWS.01.2.1.W.SYS" application note has been taken in account.

However, these complete protocol information have been useful to decode the binary fields and understand unclear items present inside the JSON Object. The documentation shows that the LoRa message item inside the JSON Object has been encoded by using Base64 to avoid problems related with non printable characters present in the encrypted message. After decoding this entry, the complete LoRa payload cannot be analysed because it is encrypted.

A simple analysis of the captured packets allows to extract statistical features related to the use of the radio channel and the behaviour of the gateway or the LoRa server. But to be able to generate features related to the behaviour of a single end node, it is necessary to find and decode the device unique identifier. This is indispensable because it is necessary to group together all the packet related to a

³LoRa-net packet forwarder (rev 1.4) https://github.com/LoRa-net/packet_forwarder/blob/master/PROTOCOL.TXT

⁴LoRa-net sx1302_hal packet forwarder (rev 1.5) https://github.com/LoRa-net/sx1302_hal/blob/master/packet_forwarder/PROTOCOL.md

⁵semtech "ANNWS.01.2.1.W.SYS" application note https://things4u.github.io/Projects/SingleChannelGateway/DeveloperGuide/5_LoRa_TTN_Reading/LoRa%20gateway%20to%20network%20server%20interface%20definition.pdf

specific end nodes.

The protocol does not authenticate the identity of LoRa Gateway or LoRa Server. Acknowledge messages are present but unused, the protocol does not perform any retransmission when an acknowledge message is lost.

The communications between LoRa gateway and LoRa Server use UDP protocol with an ephemeral source port and 1700 as destination port. The communications between LoRa Server and LoRa Gateway use UDP protocol with source port set to 1700 and an ephemeral port as the destination port.

Thanks to the documentation, it was possible to fully decode the captured packets. The format of the packet in Figure 3.3 is different depending on the type of message (operation ID). The first three fields are always present: 1. protocol version (1 or 2); 2. random token; 3. operation ID (0 ... 5).

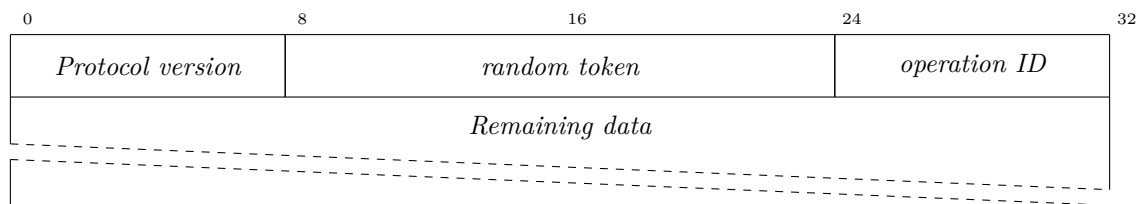


Figure 3.3: LoRa packet common fields

Depending on the operation ID field, the format of remaining fields changes.

The packets analysed in this chapter belong to the version 2.

The protocol can be distinguished between two categories (Upstream and Downstream) and define three different packet exchange sequences:

1. **Upstream protocol:** this sequence is used to send to the IoT Server the data received from the LoRa MoDem (LoRa RF packets).
2. **Downstream protocol:** this sequence (repeated every N seconds) is used to open (and keep open) a bidirectional route between LoRa Gateway and

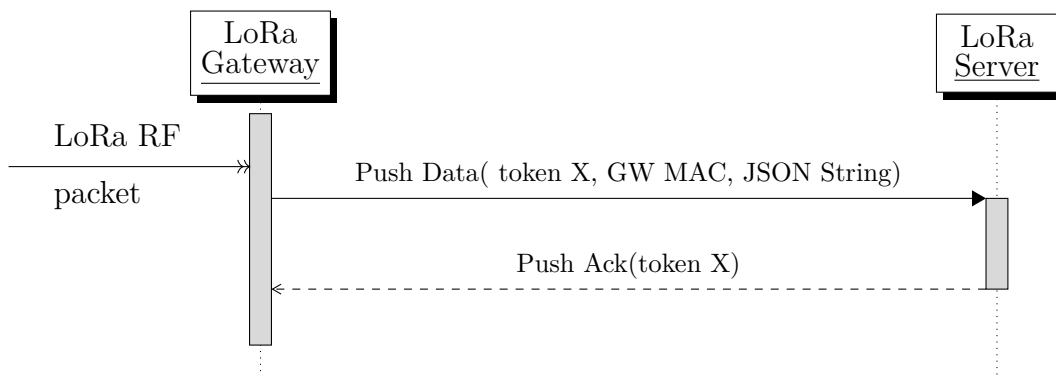


Figure 3.4: Sequence diagram of upstream protocol: Push Data/Push Ack

LoRa Server.

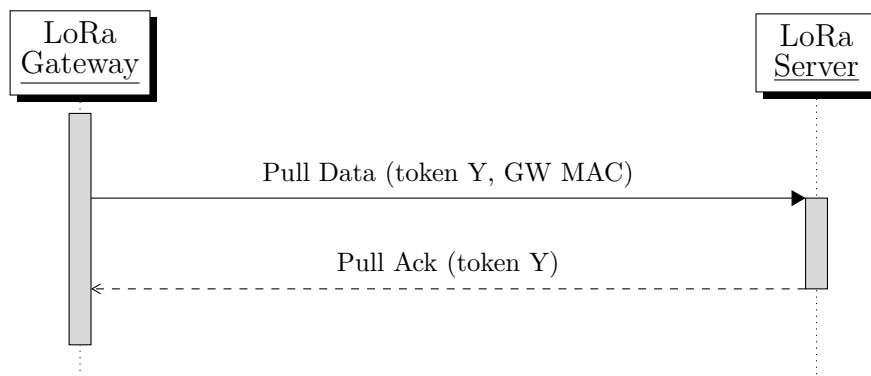


Figure 3.5: Sequence diagram of upstream protocol: Pull Data/Pull Ack

- Downstream protocol:** This sequence is used to sent to the LoRa Gateway the LoRa Packets to be transmitted over the air (LoRa RF packets) by the LoRa MoDem.

The RF communication channel between LoRa Gateway and LoRa end nodes is the "downlink".

3.4.1 Upstream protocol

The upstream protocol uses two operation ID (0x00 Push Data/0x01 Push Ack).

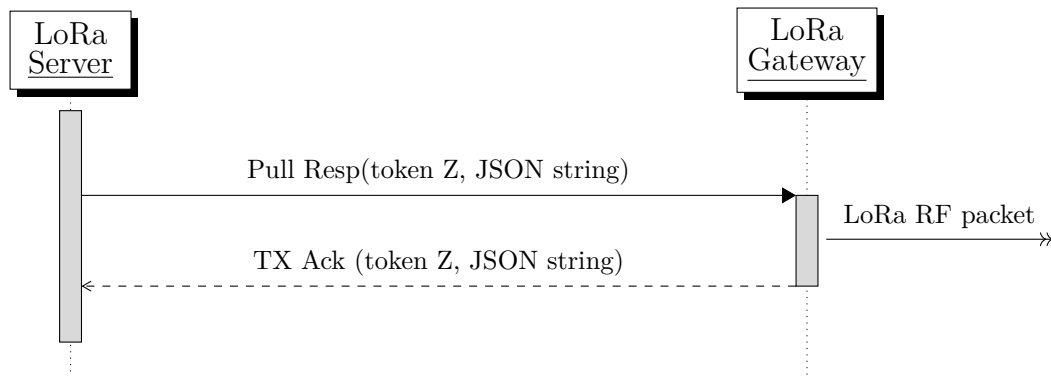


Figure 3.6: Sequence diagram of downstream protocol: Pull Resp/TX Ack

Operation ID 0x00 Push Data

The Push Data (0x00) message is used to send data to the LoRa Server. This message can be used to send the data received from one LoRa end node or the status of the LoRa Gateway. The LoRa-net documentation does not set any length limit but the "ANNWS.01.2.1.W.SYS" application note sets a maximum length of 2408 bytes.

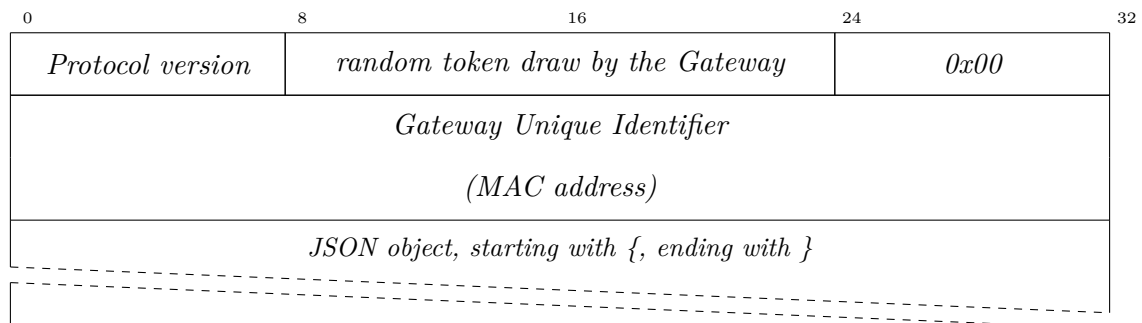


Figure 3.7: Upstream protocol: Push Data Message (ID 0x00)

As shown in Figure 3.7, in addition to the common fields two new fields, are present:

1. Gateway Unique Identifier: identify the gateway;

2. JSON Object: an ASCIIZ in JSON format;

The JSON Object can contain:

1. The status of the LoRa Gateway inside a "stat" item. See Table 3.1 for a complete list of possible items;

```
{  "stat":{ "time":"2020-03-20 08:51:11 GMT",
           "lati":36.55989, "long":136.65347, "alti":69,
           "rxnb":1, "rxok":0, "rxfw":0,
           "ackr":100.0, "dwnb":0, "txnb":0
        }
}
```

Name	Type	Rev.	Seen	Description
time	string	1.4	Yes	Coordinated Universal Time (UTC) 'system' time of the gateway, ISO 8601 'expanded' format
lati	number	1.4	Yes	UTC latitude of the gateway in degree (float, N is +)
long	number	1.4	Yes	UTC longitude of the gateway in degree (float, E is +)
alti	number	1.4	Yes	UTC altitude of the gateway in meter (integer)
rxnb	number	1.4	Yes	Number of radio packets received (unsigned integer)
rxok	number	1.4	Yes	Number of radio packets received with a valid Physical Cyclic Redundancy Check (CRC)
rxfw	number	1.4	Yes	Number of radio packets forwarded (unsigned integer)

Table 3.1 – *Continued on next page*

Continued from previous page

Name	Type	Rev.	Seen	Description
ackr	number	1.4	Yes	Percentage of upstream datagrams that were acknowledged
dwnb	number	1.4	Yes	Number of downlink datagrams received (unsigned integer)
txnb	number	1.4	Yes	Number of packets emitted (unsigned integer)
temp	number	1.5	No	Current temperature in celsius degree (float). Warning: not present in "ANNWS.01.2.1.W.SYS" application note

Table 3.1: Possible items of "stat" JSON array with explicit dependence on revision. Information taken from github LoRa-net.

- The LoRa packets received by the LoRa Gateway inside a "rxpk" item array. Each array entry contains (inside specific fields) the payload of received LoRa packet and all related information. See Table 3.2 for a complete list of possible fields;

```
{  "rxpk" : [ { "tmst":4217091219,
                "time":"2020-03-20T08:52:24.052551Z",
                "tmms":1268729563052,
                "chan":2, "rfch":1, "freq":868.500000,
                "stat":1, "modu":"LoRa", "datr":"SF7BW125",
                "codr":"4/5", "lsnr":8.2,
                "rssi":-92, "size":51,
                "data":"UUFZb0FmOGRkUUFcbjRXWk9NU3Z6eUp5bWZTM
                    OVCaUxFR012QXgwTFpEbHZvOHZFvEhRTmVBN1
```

```
NrTE9mUnFQK09ySngK"
```

```
} ]
```

```
}
```

The "ANNWS.01.2.1.W.SYS" application note says that the "rxpk" item may be a single object and not an array.

```
{  "rxpk": {...}
}
```

Name	Type	Rev.	Seen	Description
time	string	1.4	Yes	UTC time of pkt RX, us precision, ISO 8601 'compact' format
tmms	number	1.4	Yes	UTC time of pkt RX, number of milliseconds since 06.Jan.1980
tmst	number	1.4	No	Internal timestamp of "RX finished" event (32bit unsigned). Warning: not present in "ANNWS.01.2.1.W.SYS" application note
freq	number	1.4	Yes	RX central frequency in MHz (unsigned float, Hz precision)
chan	number	1.4	Yes	Concentrator "IF" channel used for RX (unsigned integer)
rfch	number	1.4	Yes	Concentrator "RF chain" used for RX (unsigned integer)
mid	number	1.5	No	Concentrator MoDem ID on which pkt has been received. Warning: not present in "ANNWS.01.2.1.W.SYS" application note
stat	number	1.4	Yes	CRC status: 1 = OK, -1 = fail, 0 = no CRC

Table 3.2 – Continued on next page

Continued from previous page

Name	Type	Rev.	Seen	Description
modu	string	1.4	Yes	Modulation identifier "LORA" or "FSK"
datr	string	1.4	Yes	LoRa datarate identifier (eg. SF12BW500)
	number	1.4	No	Frequency Shift Keying (FSK) datarate (unsigned, in bits per second)
codr	string	1.4	Yes	LoRa ECC coding rate identifier
rssi	number	1.4	Yes	Received Signal Strength Indication (RSSI) of the channel in dBm (signed integer, 1 dB precision).
rssis	number	1.5	No	RSSI of the signal in dBm (signed integer, 1 dB precision).Warning: not present in "ANNWS.01.2.1.W.SYS" application note
lsnr	number	1.4	Yes	LoRa Signal-to-Noise Ratio (SNR) ratio in dB (signed float, 0.1 dB precision)
foff	number	1.5	No	LoRa frequency offset in Hz (signed integer).Warning: not present in "ANNWS.01.2.1.W.SYS" application note
size	number	1.4	Yes	RF packet payload size in bytes (unsigned integer)
data	string	1.4	Yes	Base64 encoded RF packet payload, padded

Table 3.2: Possible items of "rxpk" JSON array with explicit dependence on revision. Information taken from LoRa-net github. Column Seen reports whether the field was seen on the wire

3. Both "stat" and "rxpk" information.

```
{  "rxpk": [ {...}, ...],
   "stat": {...}
```

```
}

```

The "ANNWS.01.2.1.W.SYS" application note says that the "rxpk" item may not be an array but a single object.

```
{  "rxpk":{...},
   "stat":{...}
}
```

The "data" item of "rxpk" array contains the Base64 encoded string of LoRa data entries in binary format, as shows in the Table 3.2. The structure of the binary format of LoRa data entry shown in Figure 3.8 has been extracted from the source code of the LoRa forwarder.

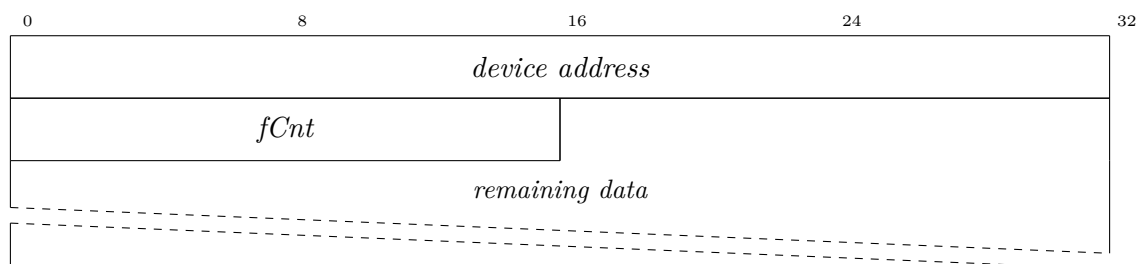


Figure 3.8: LoRa packet binary format of ["rxpk"][i]["data"] after base64 decoding

After the decoding of the "data" item, the fields "device address" and fCnt can be read, but the rest of the binary data is encrypted.

The "ANNWS.01.2.1.W.SYS" application note describes another format for this JSON Object: 1. The "rxpk" item can be an array or a single value. 2. Unknown item can be present inside the JSON object (example: "other", etc).

The Push Data message is acknowledged from the LoRa Server through the Push Ack (0x01) message.

Operation ID 0x01 Push Ack

The Push Ack (0x01) message is sent from LoRa Server to LoRa Gateway immediately after the reception of one Push Data message to confirm its correct reception. For our analysis, it is not a problem if the LoRa Gateway does not receive an acknowledgement message.

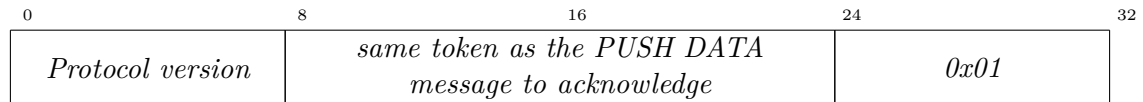


Figure 3.9: Upstream protocol: Push Ack message (0x01)

As shown in Figure 3.9, this message uses only the common header fields.

3.4.2 Downstream protocol

The downstream protocol defines four possible operation ID (0x02 Pull Data/0x03 Pull Ack/0x04 Pull Resp/0x05 TX Ack).

Operation ID 0x02 Pull Data

The Pull Data (0x02) message is sent to LoRa Server at regular intervals (configured in gateway) to establish and keep open the connection channel between LoRa Gateway and LoRa Server. This is needed when the LoRa Gateway is inside a NAT or a firewall. When the NAT assigns a public IP/port to the LoRa Gateway, the route is open and the message originated from the server can reach the LoRa Gateway. When a firewall is present, it can block requests incoming from the external interface unless a matching outgoing traffic is present. Moreover, a firewall can be configured to kill idle sessions, so, a keep alive technique must be used.

As shown in Figure 3.10, there is the Gateway Unique Identifier in addition to the common fields.

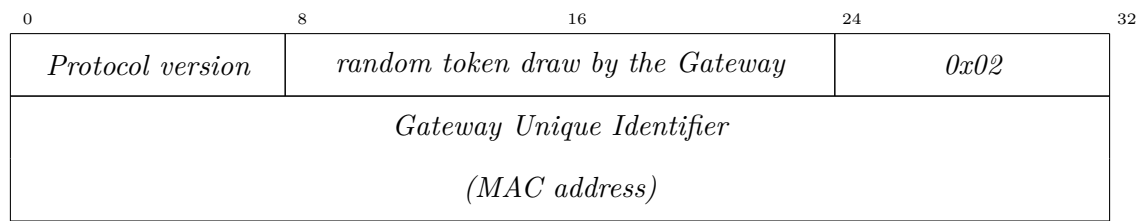


Figure 3.10: Downstream protocol: Pull Data message (0x02)

This message is acknowledged from the LoRa Server by using the Pull Ack (0x04) message. The reception of the Pull Ack indicates the correct opening of a bidirectional route between LoRa Gateway and LoRa Server and the LoRa Gateway must be ready to receive requests from the LoRa Server.

Operation ID 0x04 Pull Ack

The Pull Ack (0x04) message is sent from the LoRa Server to the LoRa Gateway. When this packet reaches the LoRa Gateway, it is an implicit confirmation of the establishment of one bidirection route. Now, the LoRa Server can send Pull Resp (0x03) messages when it is needed.

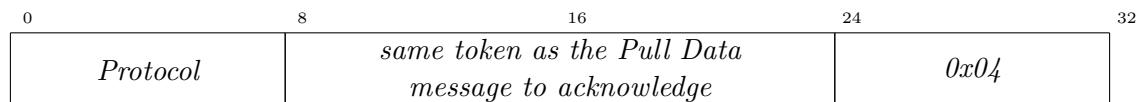


Figure 3.11: Downstream protocol: Pull Ack message (0x04)

As shown in Figure 3.11 this message uses only the common header fields.

The "ANNWS.01.2.1.W.SYS" application append the Gateway Unique Identifier to the common header.

Operation ID 0x03 Pull Resp

The Pull Resp (0x03) message is used to send from LoRa Server to the LoRa Gateway the LoRa Packet to be emitted on Air.

The LoRa-net documentation does not set any length limit, but the "ANNWS.01.2.1.W.SYS" application note sets a maximum length of 1 000 bytes.

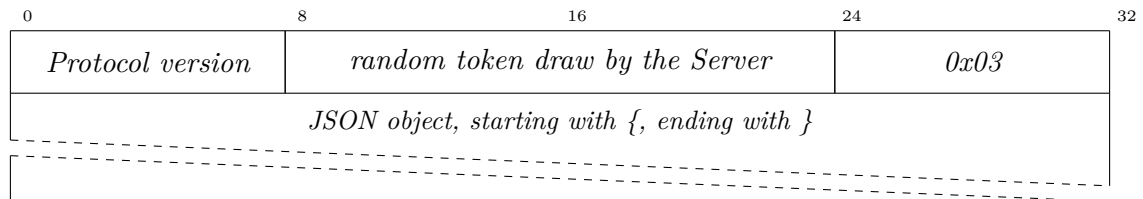


Figure 3.12: Downstream protocol: Pull Resp message (0x03)

As shown in Figure 3.12, there is the C string JSON Object field in addition to the common fields. The JSON Object contains the LoRa packet to be sent from the LoRa Gateway and all related information inside an item "txpk".

The "ANNWS.01.2.1.W.SYS" application note says that the JSON string may contain zero or more "txpk" items.

See Table 3.3 for a complete list of possible fields. Most fields are optional, default parameters are used when a field is not present.

```
{  "txpk": {... }
}
```

In the protocol version 2, the Pull Resp message is acknowledged by the LoRa Gateway by using the TX Ack (0x05) message.

Name	Type	Description
imme	bool	Send packet immediately (will ignore tmst & time)
tmst	number	Send packet on a certain timestamp value (will ignore time)
tmms	number	Send packet at a certain UTC time (UTC synchronization required). Warning: not present in "ANNWS.01.2.1.W.SYS" application note
freq	number	TX central frequency in MHz (unsigned float, Hz precision)
rfch	number	Concentrator "RF chain" used for TX (unsigned integer)
powe	number	TX output power in dBm (unsigned integer, dBm precision)
modu	string	Modulation identifier "LORA" or "FSK"
datr	string	LoRa datarate identifier (eg. SF12BW500)
	number	FSK datarate (unsigned, in bits per second)
codr	string	LoRa ECC coding rate identifier
fdev	number	FSK frequency deviation (unsigned integer, in Hz). Warning: not present in "ANNWS.01.2.1.W.SYS" application note.
ipol	bool	LoRa modulation polarization inversion.
prea	number	RF preamble size (unsigned integer). Warning: not present in "ANNWS.01.2.1.W.SYS" application note
size	number	RF packet payload size in bytes (unsigned integer)
data	string	Base64 encoded RF packet payload, padding optional
ncrc	bool	If true, disable the CRC of the physical layer (optional)

Table 3.3: Possible items of "txpk" JSON object (revision 1.4 & 1.5). I did not see this type of message on the wire. Information taken from github LoRa-net.

Operation ID 0x05 TX Ack

The TX Ack (0x05) message is sent from the LoRa Gateway to the LoRa Server to inform if a Pull Resp has been rejected or accepted (with or without errors/warnings).

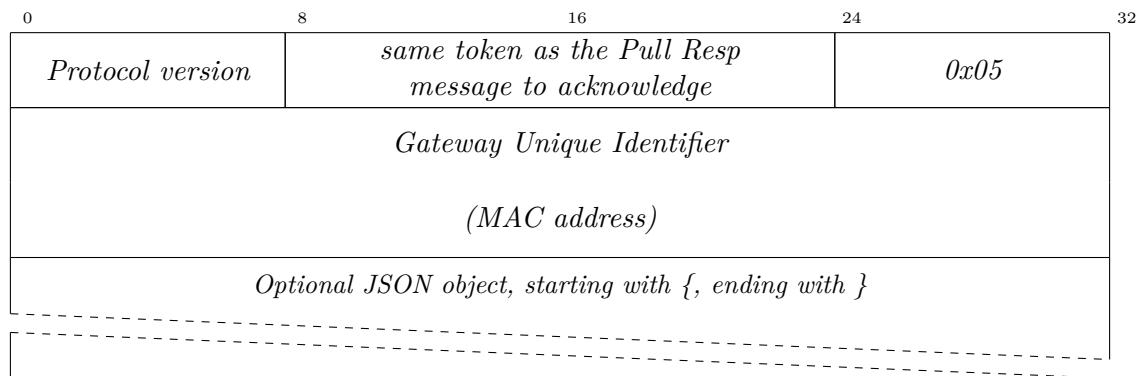


Figure 3.13: Downstream protocol: TX Ack message (0x05)

As shown in Figure 3.13, there are two other fields in addition to the common fields:

1. Gateway Unique Identifier: identify the gateway;
2. optional JSON Object: a ASCIIZ in JSON format;

The optional JSON Object is used to report errors. This JSON String can be empty⁶ to indicate the absence of errors/warning or includes a single "txpk_ack" item in case of one error or warning which contains the following fields:

1. The error code inside a "error" item with one of the following values:
 - (a) NONE: Packet has been programmed for downlink (removed in revision 1.5, not present in "ANNWS.01.2.1.W.SYS" application note);

⁶An ASCIIZ is empty when it starts with the End Of String character that is a single byte with value zero (NUL character, ASCII code 0, \0).

- (b) TOO_LATE: Rejected because it was already too late to program this packet for downlink;
- (c) TOO_EARLY: Rejected because downlink packet timestamp is too much in advance;
- (d) COLLISION_PACKET: Rejected because there was already a packet programmed in the requested timeframe;
- (e) COLLISION_BEACON: Rejected because there was already a beacon planned in the requested timeframe;
- (f) TX_FREQ: Rejected because requested frequency is not supported by TX RF chain;
- (g) TX_POWER: Rejected because requested power is not supported by the gateway (removed in revision 1.5, present in "ANNWS.01.2.1.W.SYS" application note);
- (h) GPS_UNLOCKED: Rejected because UTC is unlocked, so UTC timestamp cannot be used.

In the revision 1.5, the "NONE" value has been removed and the "TX_POWER" has been moved from "error" category to "warn" category of "txpk_ack" object.

```
{ "txpk_ack": { "error": "COLLISION_PACKET" } }
```

2. The "warn" and "value" items were introduced on revision 1.5 to specify a warning code and a value. The "warn" field contains one of the following codes:

- (a) TX_POWER: Rejected because requested power is not supported by gateway, the power actually used is given in the "value" item.


```
{ "txpk_ack": { "warn": "TX_POWER", "value": 20} }
```

3.4.3 Information extracted from network packet analysis

Based on the information previously described, all the binary and JSON text or Base64 encoded strings can be extracted from the network flow.

Other information can be computed exploiting the timestamp of the received packets, keeping in mind that the RF channels are shared with a lot of devices and the information can be sent with a delay of seconds from the real sent request.

```
{ "timestamp": "1584694285.068462", "ver": "2", "rnd": "8386", "opID": "0", "gw": "dcc025feffeb27b8",
  "value": { "rxpk": [ { "tmst": 4157095595, "time": "2020-03-20T08:51:24.056932Z", "tmms": 1268729503056,
    "chan": 1, "rfch": 1, "freq": 868.300000, "stat": 1, "modu": "LoRa", "datr": "SF7BW125",
    "codr": "4/5", "lsnr": 9.0, "rssi": -93, "size": 51,
    "data": "QAYoAf8AKAABUBwZaZ3p43YNK1Ww3hbU2f4gmT5TPypq24y6E6dKW90Bm2iCJOMEfvH0" } ] },
  "unpacked": { "size": "51", "devAddr": "ff012806", "fCnt": "2800" } }
{ "timestamp": "1584694271.810198", "ver": "2", "rnd": "33472", "opID": "0", "gw": "dcc025feffeb27b8",
  "value": { "stat": { "time": "2020-03-20 08:51:11 GMT", "lati": 36.55989, "long": 136.65347, "alti": 69,
    "rxnb": 1, "rxok": 0, "rxfw": 0, "ackr": 100.0, "dwnb": 0, "txnb": 0 } } }
{ "timestamp": "1584694271.851998", "ver": "2", "rnd": "33472", "opID": "1" }
{ "timestamp": "1584694272.164906", "ver": "2", "rnd": "36450", "opID": "2", "gw": "dcc025feffeb27b8" }
{ "timestamp": "1584694272.202189", "ver": "2", "rnd": "36450", "opID": "4" }
{ "timestamp": "1584694282.284887", "ver": "2", "rnd": "9379", "opID": "2", "gw": "dcc025feffeb27b8" }
{ "timestamp": "1584694282.321339", "ver": "2", "rnd": "9379", "opID": "4" }
{ "timestamp": "1584694285.068462", "ver": "2", "rnd": "8386", "opID": "0", "gw": "dcc025feffeb27b8",
  "value": { "rxpk": [ { "tmst": 4157095595, "time": "2020-03-20T08:51:24.056932Z", "tmms": 1268729503056,
    "chan": 1, "rfch": 1, "freq": 868.300000, "stat": 1, "modu": "LoRa", "datr": "SF7BW125",
    "codr": "4/5", "lsnr": 9.0, "rssi": -93, "size": 51,
    "data": "QAYoAf8AKAABUBwZaZ3p43YNK1Ww3hbU2f4gmT5TPypq24y6E6dKW90Bm2iCJOMEfvH0" } ] },
  "unpacked": { "size": "51", "devAddr": "ff012806", "fCnt": "2800" } }
{ "timestamp": "1584694285.109693", "ver": "2", "rnd": "8386", "opID": "1" }
{ "timestamp": "1584694292.414888", "ver": "2", "rnd": "3216", "opID": "2", "gw": "dcc025feffeb27b8" }
{ "timestamp": "1584694292.455639", "ver": "2", "rnd": "3216", "opID": "4" }
```

3.5 Analysis of captured data

The analysis of the captured data can be done from two different viewpoints:

1. Monitoring the connection between LoRa Gateway and LoRa Server;
2. Monitoring the connection between LoRa end nodes and LoRa Gateway (and LoRa Server);

It is possible to extract information on the periodicity of the information exchange from the analysis of the temporal sequence of the messages exchanged between the LoRa gateway and LoRa server.

Further information have been deduced from the first time analysis of the message timestamp sequence. The Pull Data (0x02) message used to keep open the network path between LoRa Gateway and Server is sent to the LoRa Server every 10 seconds. The Push Data (0x00) message is always followed by a Pull Ack (0x04) message. The Push Data (0x00) message containing a "stat" JSON Object is sent to the LoRa Server every 30 seconds. The Push Data (0x00) message containing "rxpk" LoRa Packed payload does not seem to respect any fixed scheduling (but they seem to be sent when a packet is received from LoRa MoDem). As expected the Push Data (0x00) message is always followed by an Push Ack (0x01) message.

These timing are stable during normal execution. My intention is to use these properties to detect an attack.

3.5.1 Monitoring the connection between LoRa Gateway and LoRa Server

It is possible to monitor the quality of the connection between LoRa Gateway and LoRa Server through the analysis of the Round Trip Time (RTT) between data transmissions and acks. The graph in Figure 3.14 shows a quite stable RTT. It also shows an increasing RTT near the origin. This can be related to network congestions or LoRa Server overloads. In the second graph, different colours have been used for each operation IDs.

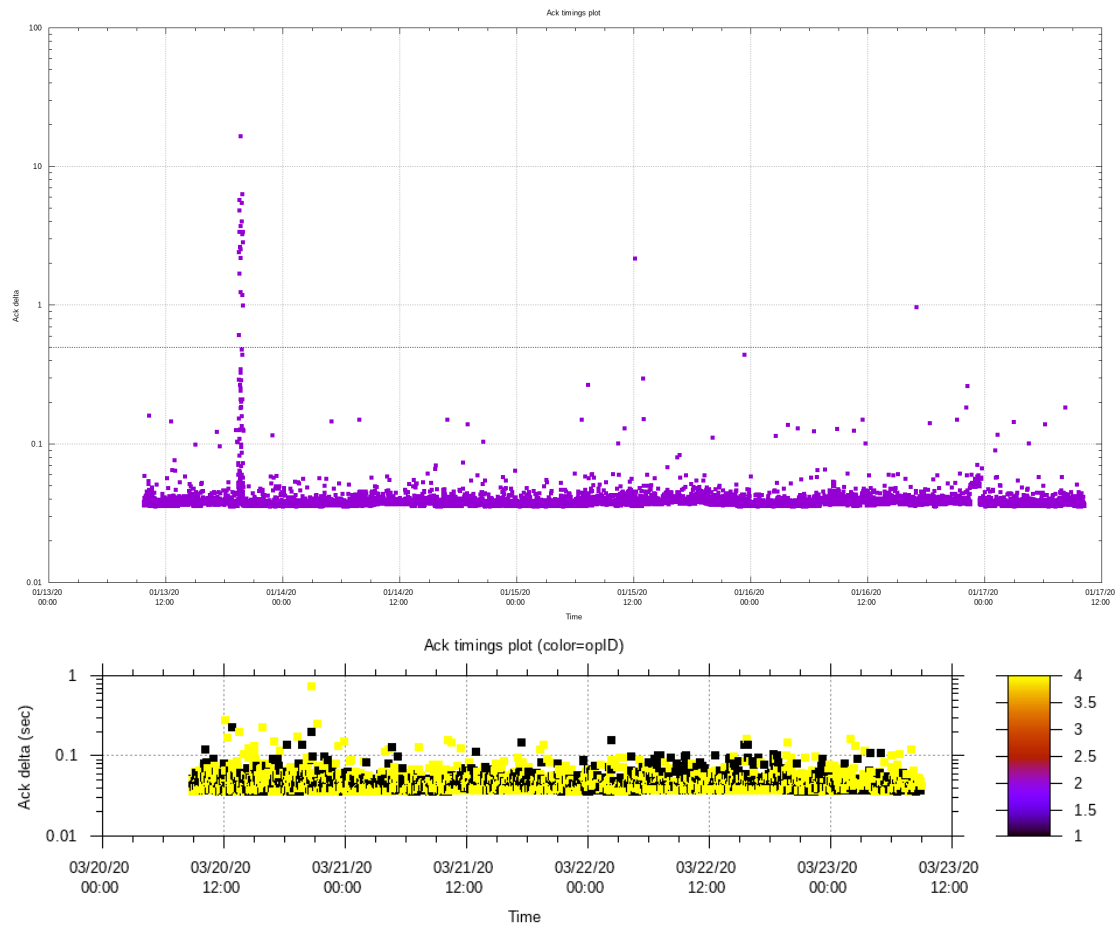


Figure 3.14: Round trip times between Request and Ack (logarithmic scale).

3.5.2 Monitoring the connection between end nodes and LoRa Gateway (and LoRa Server)

A lot of information is sent unencrypted by using textual format or base64 encoding. Using this information, I can keep track of the number of present devices, transmission rate, RSSI, datarate, RX and TX throughput, used frequency channels, CRC errors, etc.

Received signal strength indication

The intensity of the received LoRa packets (Figure 3.15 and Figure 3.16) changes throughout the day for all end nodes according to the environmental conditions. Also the orientation of the LoRa Gateway antenna can affect the RSSI depending on the position of the transmission thyroidal emissions.

The colour in these figures are proportional to the signal to noise level of the received LoRa packets.

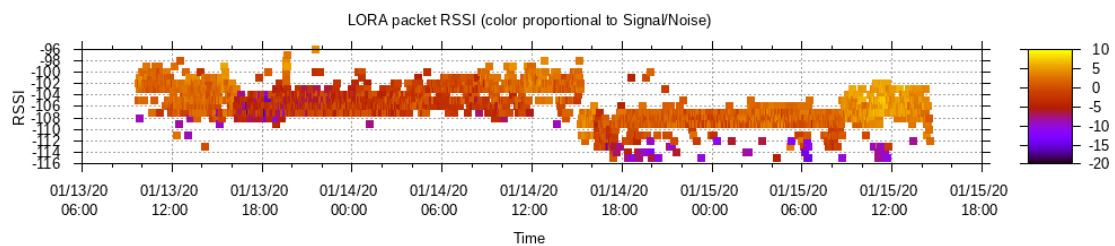


Figure 3.15: LoRa packets RSSI, indoor antenna.

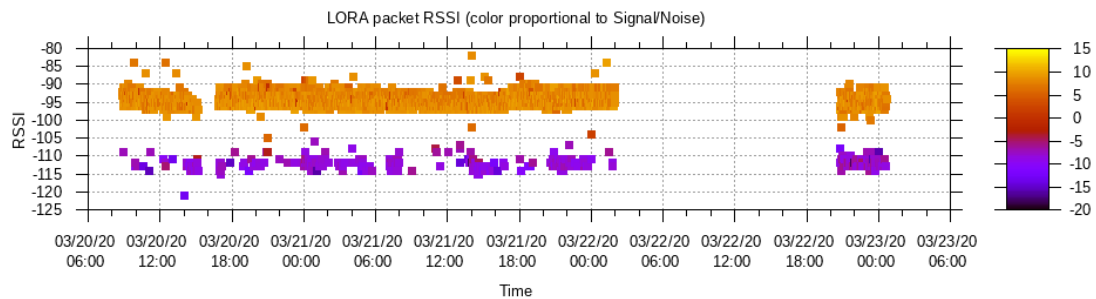


Figure 3.16: LoRa packets RSSI, antenna close to a window.

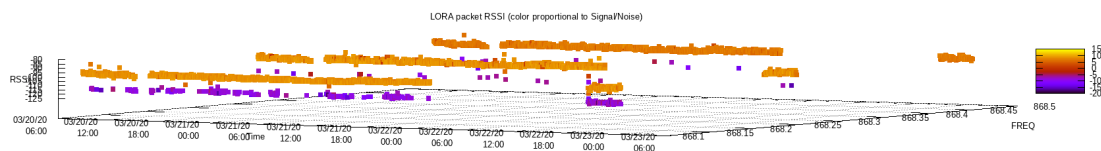
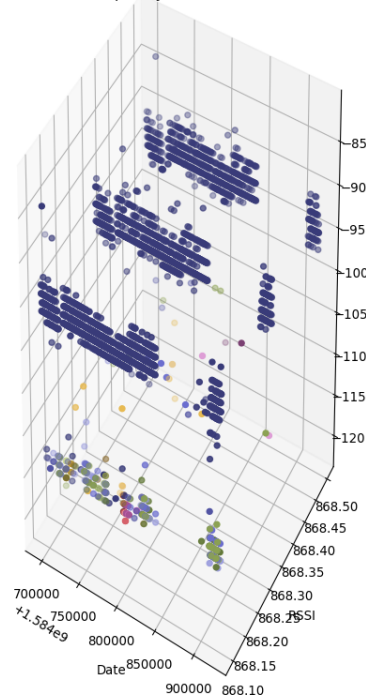


Figure 3.17: LoRa packets RSSI in the used frequencies.

The LoRa frequencies ranges used to transmit data are compact. The radio channel properties and the noise level are similar. The RSSI grouped by LoRa frequency show similar intensities.

Lora RSSI/frequency/time, color is device ID



Lora device ID transmission/total

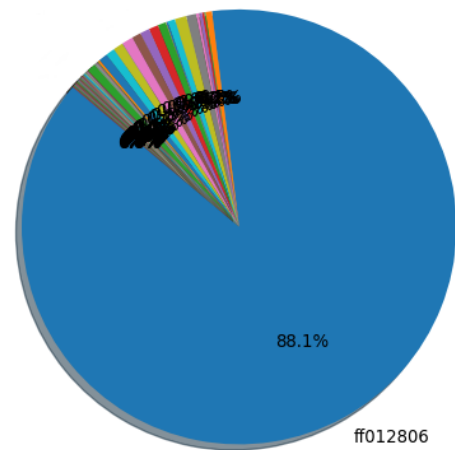


Figure 3.19: LoRa end node traffic

Figure 3.18: Utilization of LoRa frequency by device ID.

Usage pattern of the available LoRa frequencies

The graph in Figure 3.18 shows the frequency used and the RSSI of the received packets to estimate the presence of a frequency hopping rule.

LoRa datarate

The LoRa transmission datarate "datr" varies from end node to end node. Most of the analysed LoRa end nodes communications did not change their datarate

(Figure 3.20).

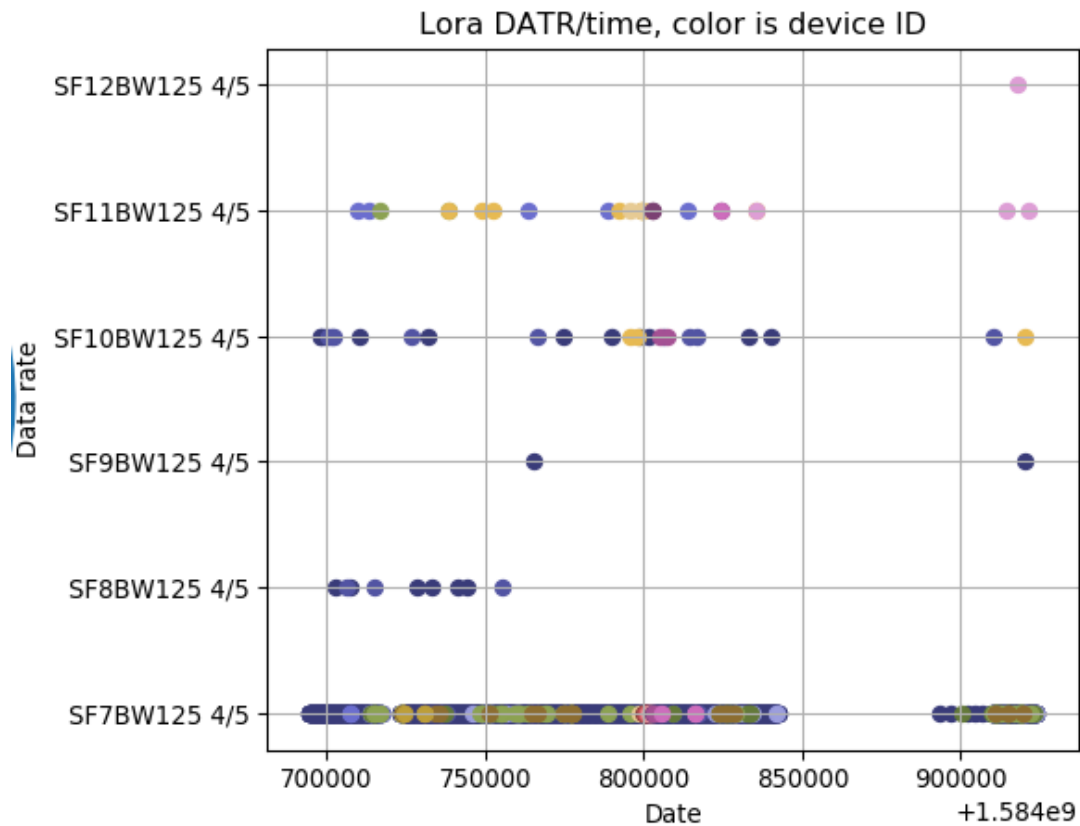


Figure 3.20: Use of Datr by LoRa devices.

LoRa SNR

For each of the received packet, the LoRa SNR shown in Figure 3.21 represents the ratio of the Signal power over the signal Noise in dB. This value depends on both signal and noise powers, so a low SNR value is expected for packets coming from long distance. However, a low SNR value can be expected also for signals sent from near devices when the noise power is high.

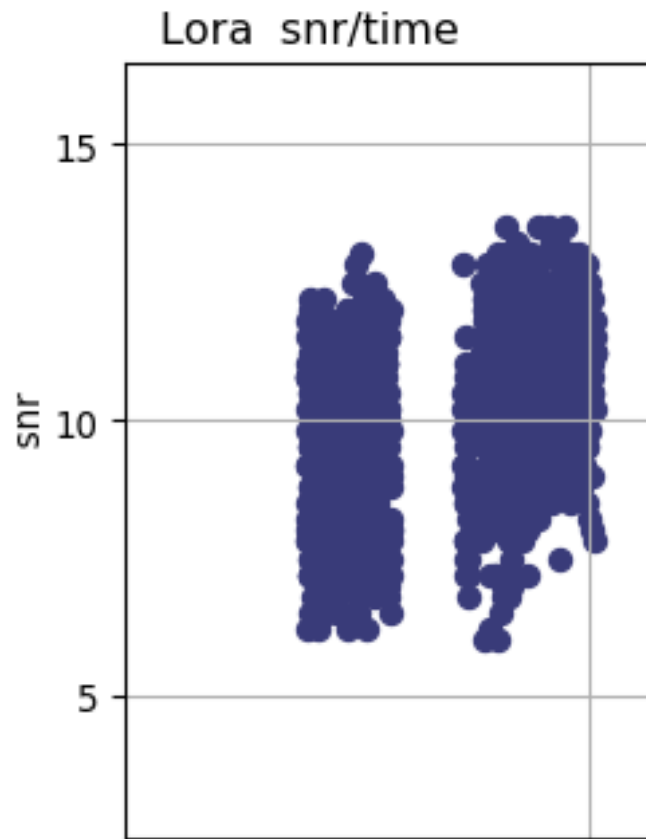


Figure 3.21: LoRa gateway SNR feature for normal traffic.

LoRa end node TX/RX throughput

In our tests, a single LoRa end node sent more than the 50% of the total traffic while the rest of devices make less transmission traffic (Figure 3.19).

The number of LoRa Radio packets received by the gateway remains constant (Figure 3.22). During only two or three days in the daytime, the LoRa Gateway received radio packet throughput graph shows a marked increase.

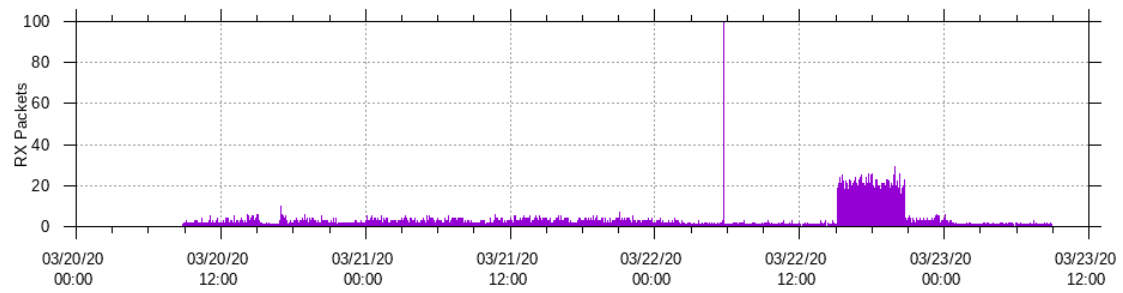


Figure 3.22: LoRa gateway received packets.

LoRa gateway inter-arrival time between two consecutive messages

The inter-arrival time between two consecutive messages remains constant in case of fixed periodic data transmission (Figure 3.23).

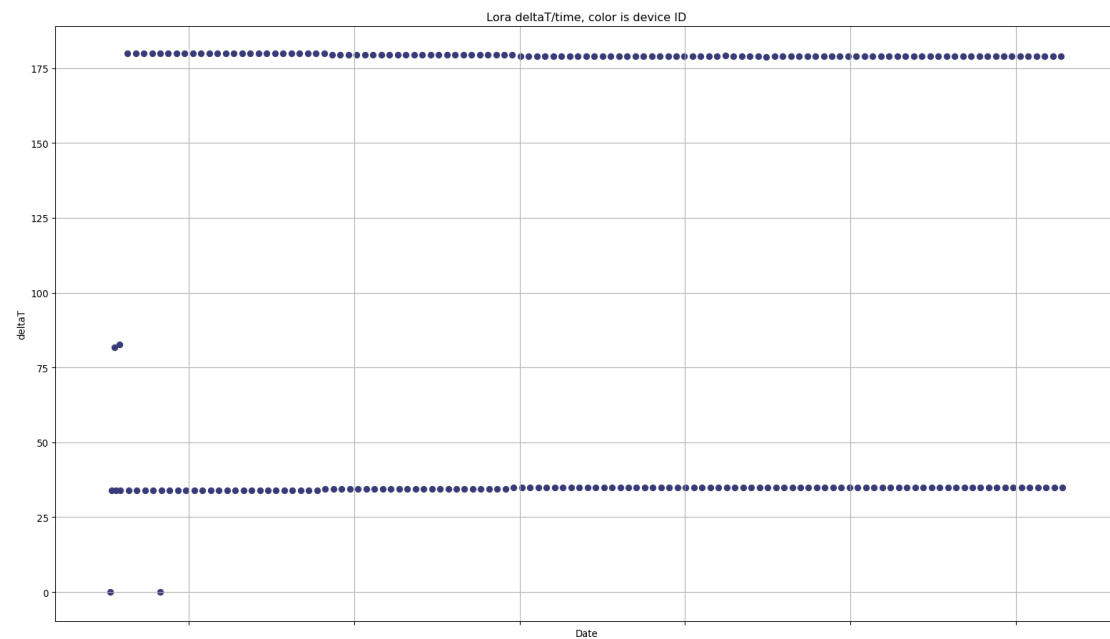


Figure 3.23: LoRa gateway inter-arrival time between two consecutive messages in case of fixed periodic data transmissions.

The number of packet received in 5 minutes, named "fiveMinPacketCount" from now on, can be used as a feature. In case of periodic transmissions, the number

of packets sent in an interval of time larger than the LoRa end node periodic transmission leads to a stable number of received packets.

LoRa packet physical CRC errors

The communication has a high number of physical CRC errors due to collisions between LoRa end nodes transmissions or noise problems. The number of physical CRC errors is higher than 30% for most of the time.

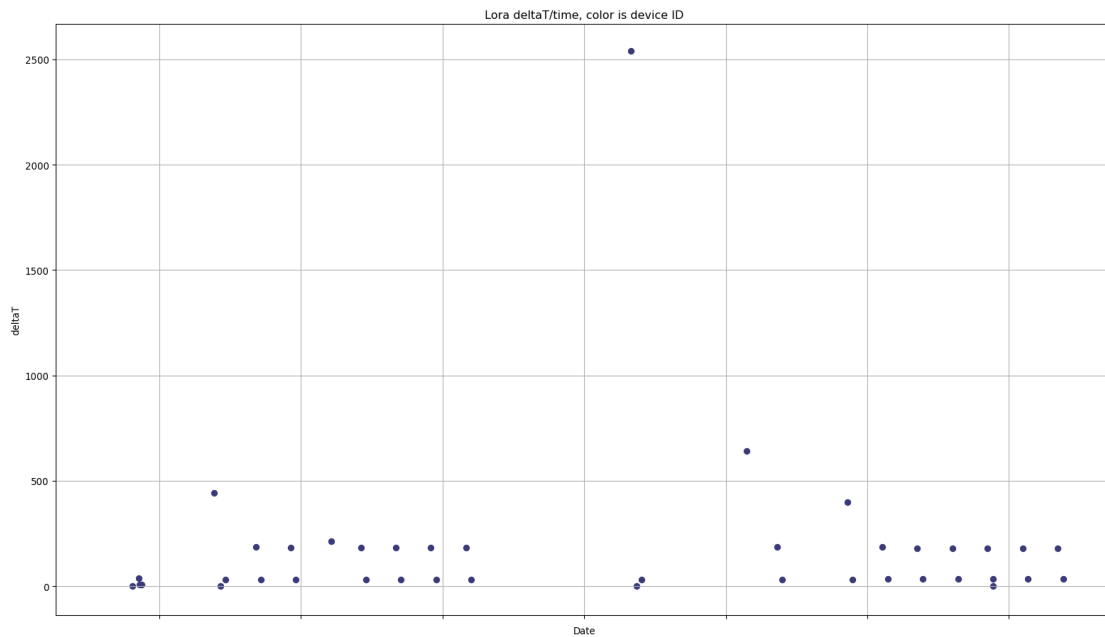


Figure 3.24: LoRa gateway inter-arrival time between two consecutive messages in case of fixed periodic data transmissions under attack by reactive jammer.

3.6 Software Defined Radio and LoRa physical layer

One or more cyber attacks have to be identified and replicated to verify if the identified features allow detecting a potential cyber attack situation. I did not find a freely available dataset containing attacks against the LoRa protocol, so I built a radio jammer device to disrupt the reception of LoRa packets.

I used an SDR card to sense, monitor, jam or disrupt a LoRaWAN network. SDR software, hardware and methods demonstrated a great flexibility and usefulness in different technological areas (Wireless protocol security, 5G radio access network, etc). In the last years, most of the radio equipment use SDR modules in the physical layer to provide the better flexibility granted by software-based demodulation and signal reconstruction.

The recent drop in the price of this kind of equipment generated a lot of small portable devices able to perform complex and dangerous attacks on commonly used radio transmission protocols.

The old analog signal processing has been substituted by Digital Signal Processing (DSP) components. In a similar way, also the software tools became simpler and the increased bandwidth of Universal Serial Bus (USB) 3.0 allowed increasing the SDR sampling throughput. The know-how level required to implement new attacks on unknown radio protocols is still high but is really simple to find on the Internet a lot of open source programs that can be used in a malicious way just pressing a few buttons.

The physical layer of LoRaWAN supports LoRa modulation and FSK modulation. The LoRa modulation is based on Chirp Spread Spectrum (CSS) which is known to be robust against interference and noise. But the Airtime of LoRa messages is long because the transmission bitrate is small. This causes a higher

probability of collisions with other signals. The Chirp signal is a sinusoidal tone with the instant frequency increasing (up-chirp) or decreasing (down-chirp) linearly at a constant rate over time and wrapped around a predefined frequency range.

The LoRa Packet Structure is composed of a preamble with several identical up-chirps followed by 2 sync word symbols and a Start Frame Delimiter (SFD) 2.25 symbols long. In the explicit header mode, the SFD is followed by the LoRa physical header and LoRa payload. The LoRa MoDem listens to the radio frequencies trying to detect a valid LoRa preamble and SFD. Then, we expect the LoRa physical header and payload.

If a jamming attack disrupts the preamble, the LoRa MoDem completely ignores the incoming packet.

LoRa jamming has been attracting attention in both academia and industry. I proposed a reactive jamming attack to interfere or disrupt the LoRa network by using two SDR cards and the GNU radio software.

When a single LoRa gateway is deployed in a large area, the LoRaWAN network forms a star topology. In this topology, the LoRa gateway can become a single point of failure if jammed by malicious attackers. Adding more LoRa gateway increases the resistance against jamming but the jamming attacks can be extended also to this additional gateways.

3.7 LoRa signal jamming testbed

The LoRa testbed (as show in figure 3.25) uses two or three Class A LoRa end node equipped with one temperature sensors (see section 3.7.2). The Intrusion Detection System (IDS) analyses the communication between LoRa Gateway and LoRa TTN Server working on network packets previously saved inside Pcap files.

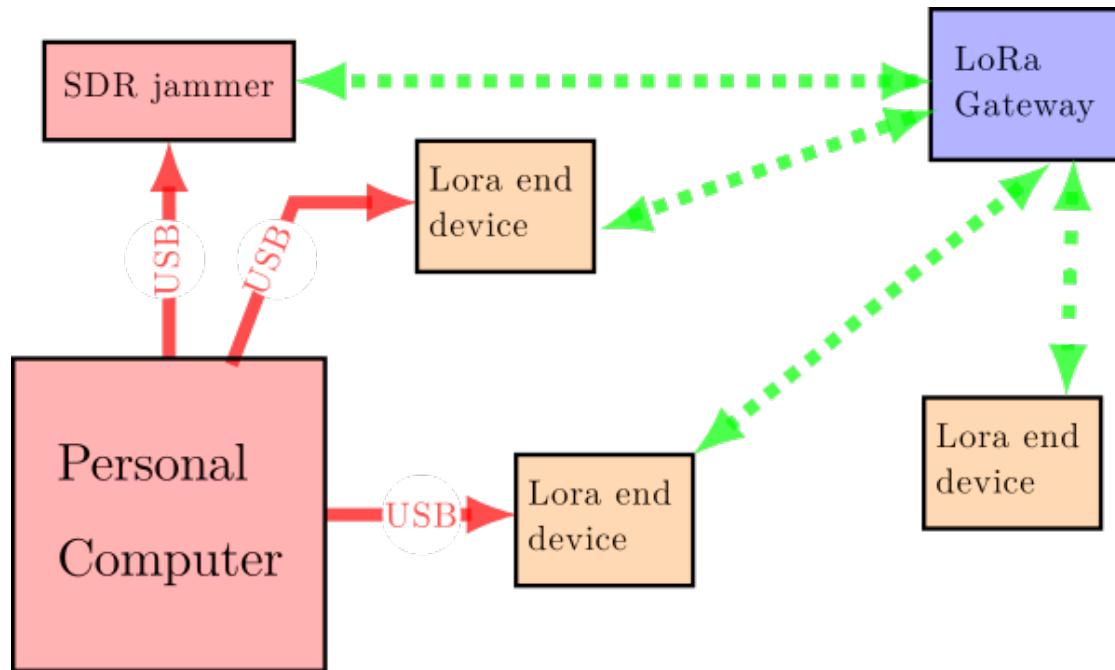


Figure 3.25: LoRa testbed

The reactive jammer consists in a Linux System with GNU Radio software and two SDR cards. The first SDR card senses the channel and the second one acts on the radio channel as show in Figure 3.26 It consists of two GNU radio flowgraphs

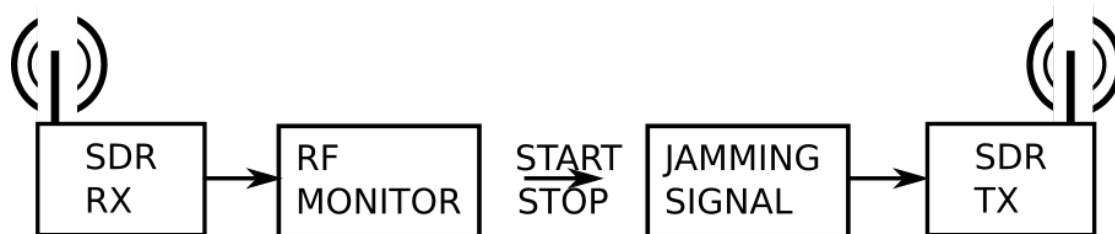


Figure 3.26: Logical structure of LoRa jammer

working together using a shared memory buffer. We present each flowgraph at page 101 and 108.

These flowgraphs use an HackRF SDR card to transmit the jamming signal and one RTL-SDR card to monitor the activity in the LoRa RF channels.

A multitude of software for Linux and Window OS are available to monitor the radio frequencies. In this section, we show images taken by using CubicSDR⁷ Linux software.

GNU radio LoRa jamming simulation from radio dump file

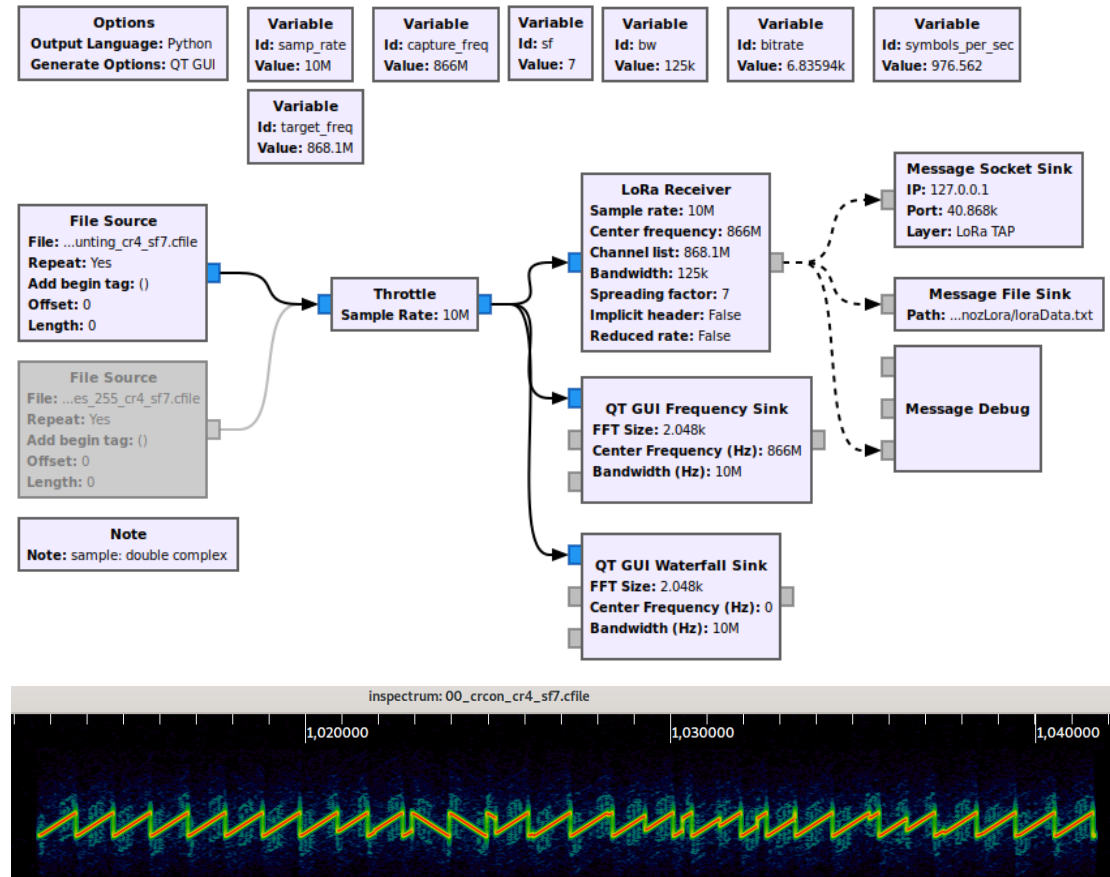


Figure 3.27: GNU radio LoRa detector from radio dump file (RX)

I realised the GNU radio flowgraph shown in Figure 3.27 to test the jamming of a LoRa transmission. This flowgraph detects and decodes the LoRa packets contained in a LoRa radio frequency dump file⁸. The dump file contains the radio

⁷<https://cubiccdr.com/>

⁸<https://github.com/rpp0/gr-LoRa-samples.git>, last change 5 Sep 2016

I and Q Components (IQ) samples saved as complex 32 bit floating point binary format as show in Figure 3.28.

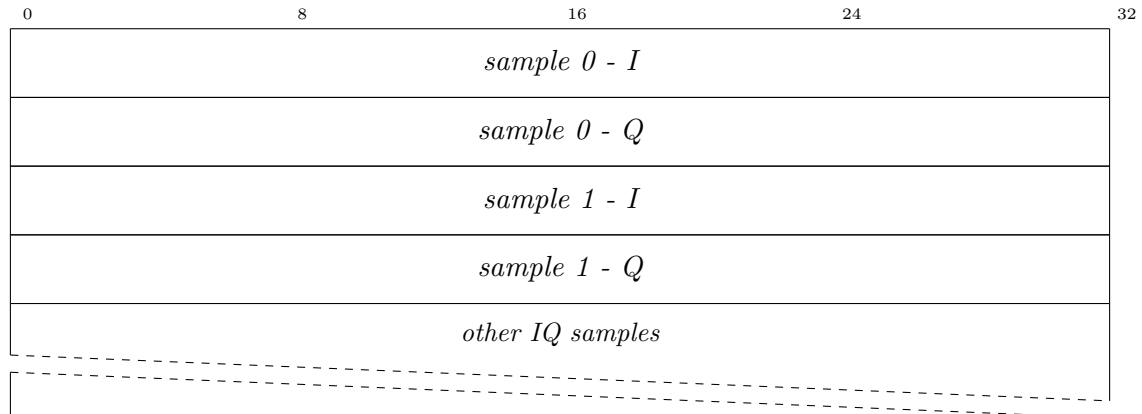


Figure 3.28: Binary format used by GNU Radio to store IQ samples on file

The jam signal is generated in real-time and added⁹ to the LoRa radio signals read from the dump file. The gr-LoRa Software Decoder¹⁰ tries to decode this distorted signal. The LoRa packets are demodulated using Gr-lora decoder and sent as datagrams to a specific IP address and UDP port (by default set to 127.0.0.1:40868).

We can use Tshark to show a detailed explanation of decoded LoRa packets as shown in Figure 3.29.

We can print on a Linux console an hexadecimal dump of the decoded LoRa data by using the “hexdump” command and the “netcat”¹¹ tool to open the destination UDP socket as show in Figure 3.30.

I tested different types of jamming signal sources by using different DSP blocks in GNU Radio:

⁹to simulate an additive radio channel

¹⁰ <https://github.com/rpp0/gr-LoRa.git>, last change 6 Feb 2021

¹¹<https://nmap.org/netcat/>

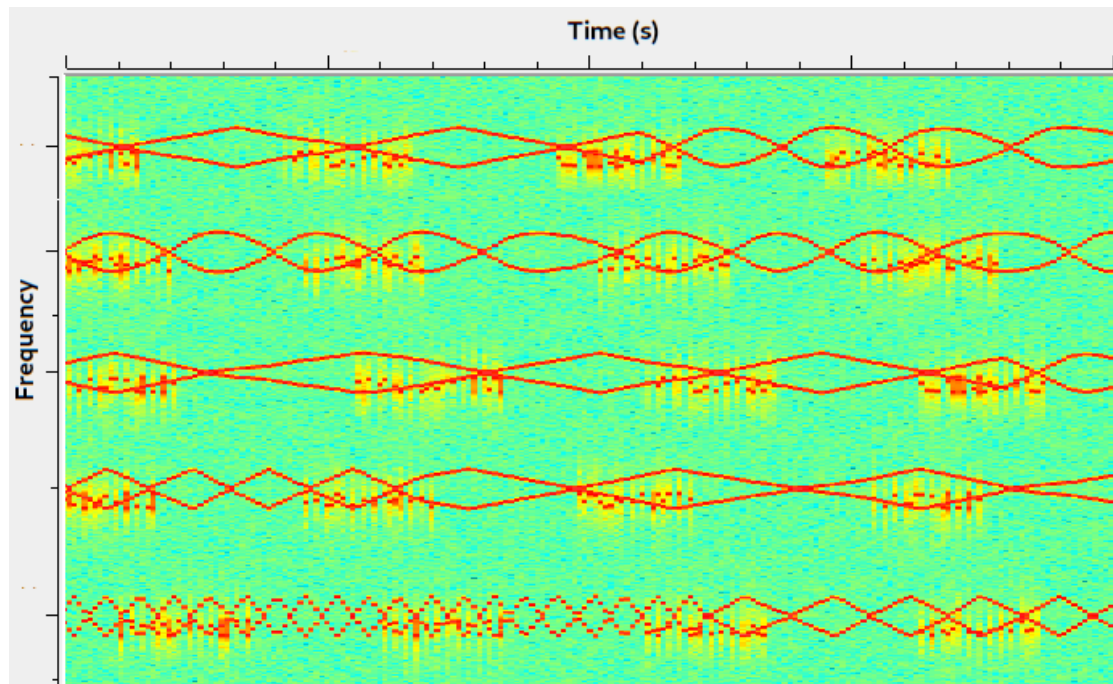
1. **VCO:** I used Voltage Controlled Oscillator (VCO) block to generate a sinusoidal tone moving in the frequency domain on a sinusoidal, triangular and saw shaped signals. The most effective jamming is obtained through sinusoidal and triangular controlled movements of the sinusoidal tone as shown in Figure 3.31;
2. **WBFM, FM, and phase modulation:** The most effective jamming is obtained by using Wide Band Frequency Modulation (WBFM) and Frequency Modulation (FM) modulated random noise. WBFM modulation generates a continuous noise in a wider frequency range (200 kHz) and needs a higher signal intensity to overcome LoRa chirps, while FM modulation generates fast changes in the jam frequencies causing problems on parts of the LoRa packets as show in Figure 3.32. This jamming is better if compared with VCO ones;
3. **gr-radar chirp source:** A continuous upchirp and downchirp chirp signal generated by gr-radar block as show in Figure 3.33 causes a complete and permanent failure of the GNU radio LoRa software decoder.

The jamming signal waveform that guaranteed the best chance of success is the chirp modulation.

GNU radio LoRa reactive jammer (RX/TX)

The aim of the jammer is trying to destroy the LoRa signal preamble and the following LoRa data symbols. This GNU radio flowgraph sends for one second a continuous up-chirp and down-chirp signal to jam the LoRa frequencies only when a transmission is detected.

This jamming system is composed by two GNU radio flowgraphs working together. The first one detects the presence of one transmission and sends a floating

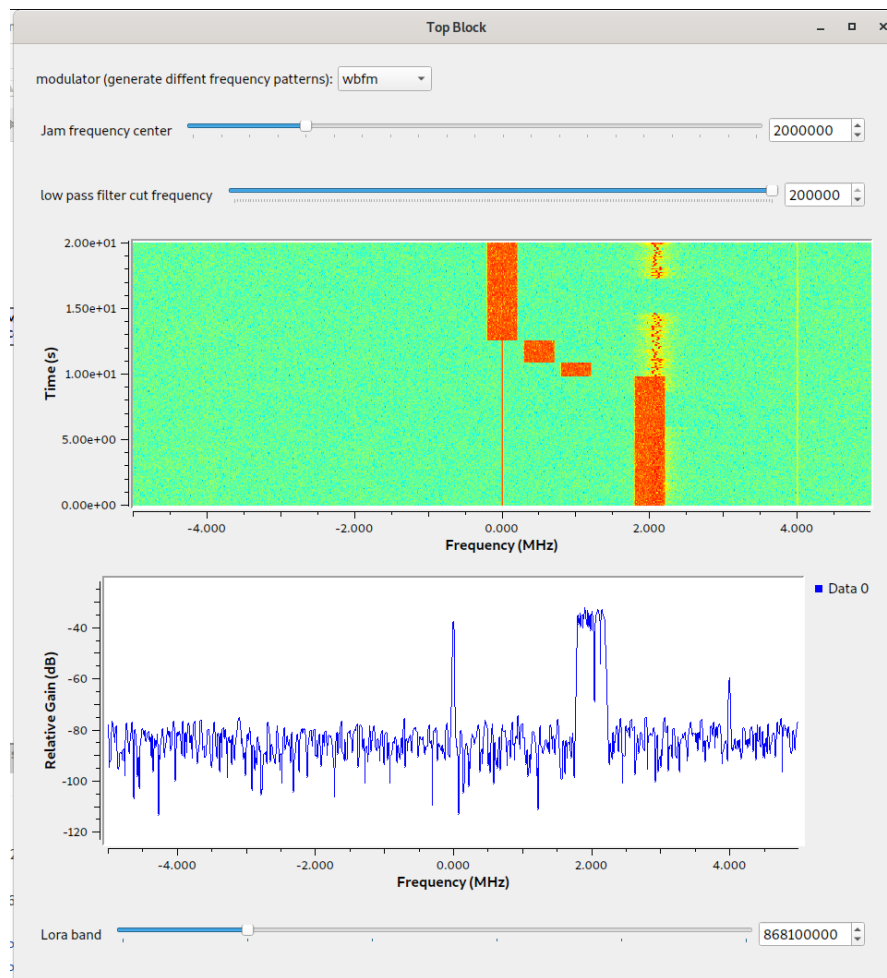


```

Bits (nominal) per symbol: 3.5      Bins per symbol: 128
Samples per symbol: 1024          Decimation: 8
17 91 a0 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 b8 73 ( !"s)
17 91 a0 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 fd e5 ( )
17 91 a0 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 a3 69 (i)
17 91 a0 00 01 02 03 04 05 06 8b 4c 45 64 1f 3b 06 f9 0f 04 f6 b5 90 df c6 e1 34 83 (LEd;4)
17 91 a0 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 b8 73 ( !"s)
88 88 88 88 88 88 88 88 fd e5 ( )
17 91 a0 8a 40 a8 c0 f5 4f ad ec 25 99 79 06 4c 2d eb c7 ee eb c6 a2 de 6e b3 ec 86 (@0%yL-n)
17 91 a0 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 a3 69 (i)
17 91 a0 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 b8 73 ( !"s)
17 91 a0 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 fd e5 ( )
17 91 a0 12 12 12 12 12 12 12 12 a0 05 42 51 d9 3b b3 ac 6d a0 71 27 bb b7 f1 58 bf 96 (BQ;mq'X)
17 91 a0 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 b8 73 ( !"s)
17 91 a0 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 fd e5 ( )
17 91 a0 a6 de 79 28 ac 70 23 95 5a a6 21 9d 00 f1 9a 4d ea fd d1 6a c3 e0 01 7c 84 (y(p#Z!Mj|)
17 91 a0 12 12 12 12 12 12 12 12 9a 92 9a de b8 c4 a5 bc 7e e3 c4 3e 98 78 9f 54 ea f8 e2 e5 ec (->xT)
17 91 a0 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 9a dd 7a ce 7c ee f1 dd 26 c3 c9 79 f1 ad (z|&y)
17 91 a0 a5 30 5a b0 6e e5 23 f4 8b 96 63 9e 18 ba 5b f9 cf ec d3 2f c3 58 9d e5 a0 (0Zn#c[/X)

```

Figure 3.31: Results of VCO Jam on GNU radio LoRa software decoding

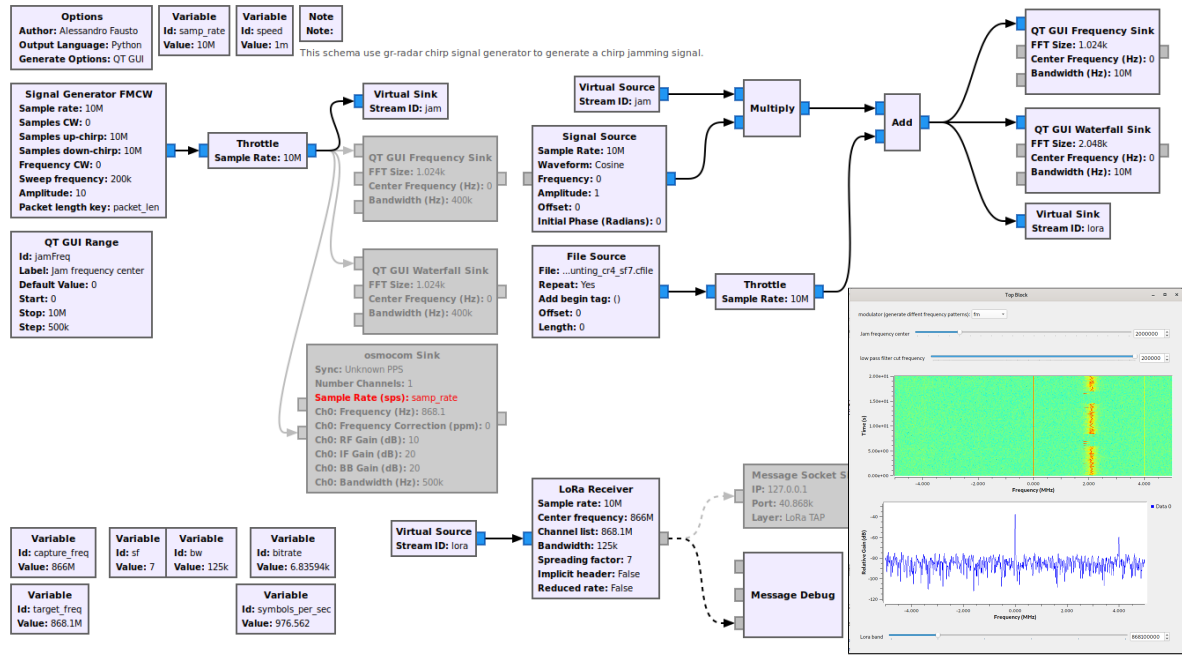


```

17 91 a0 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 fd e5 ( )
17 91 a0 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 a3 69 (i)
34 8d 80 00 81 a3 d7 5f c6 3c 91 7b ef d6 2c e1 6b 46 ec 90 1f 7a fd 22 4f ec 5f f8 14 6e f7 93
 0b e8 7d f4 bf 02 2e bc 13 62 5c 98 55 85 f5 2f 26 59 55 07 dd 21 6d (<,kFz"0_n.b
U/&YU!m)
17 91 a0 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 a3 69 (i)

```

Figure 3.32: GNU radio LoRa jammer simulator by using WBFM or FM modulated signals



```

17 91 a0 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 fd e5 ( )
17 91 a0 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 82 1a 34 68 d0 a0 40 80 (4h0)
00 00 00 ( )
( )
( )

```

Figure 3.33: GNU radio LoRa jammer simulator results by using gr-radar Chirp module

point trigger value [0.0 or 1.0] able to switch the jamming signal On or Off. This flowgraphs uses various embedded python blocks¹² to implement dedicated logic and operations.

The second flowgraph creates the jamming signal, multiplies it by the trigger value to switch the signal On/Off, and transmits the result. When the received trigger value is 0.0, the jamming signal is cancelled, otherwise, is modulated in amplitude.

We need two different flowgraph because the RTLSDR card used to listen the radio channel cannot get more than 2 million samples per seconds, while the HackRF used to transmit the jamming signal runs at 10 million samples per second.

The communication of the trigger value is done by using a shared memory buffer. The current value of the trigger can be monitored by using a command line tool as show in Figure 3.34.

```

Triggered Off at 04/30/2021 12:21:53
Triggered On at 04/30/2021 12:21:53
Triggered Off at 04/30/2021 12:22:33
Triggered On at 04/30/2021 12:22:34
Triggered Off at 04/30/2021 12:23:14
Triggered On at 04/30/2021 12:23:14
Triggered Off at 04/30/2021 12:23:54
Triggered On at 04/30/2021 12:23:55
Triggered Off at 04/30/2021 12:24:05
[ ON AIR ] trigger 0.000000

Triggered On at 04/30/2021 12:21:43
Triggered Off at 04/30/2021 12:21:53
Triggered On at 04/30/2021 12:21:53
Triggered Off at 04/30/2021 12:22:33
Triggered On at 04/30/2021 12:22:34
Triggered Off at 04/30/2021 12:23:14
Triggered On at 04/30/2021 12:23:14
[ ON AIR ] trigger 1.000000

```

Figure 3.34: Visual monitoring of trigger value stored in shared memory

This IPC done by using shared memory is the fastest possible but it must be used on the same computer. The value of the trigger changes only when a signal is detected and when the trigger returns to the normal untriggered state. The trigger value is read by the jamming flowgraph at each loop of the GNU radio flowgraph. Figure 3.35 shows the jamming signal in jamming and silent operation.

¹²More info at https://wiki.GNURadio.org/index.php/Embedded_Python_Block

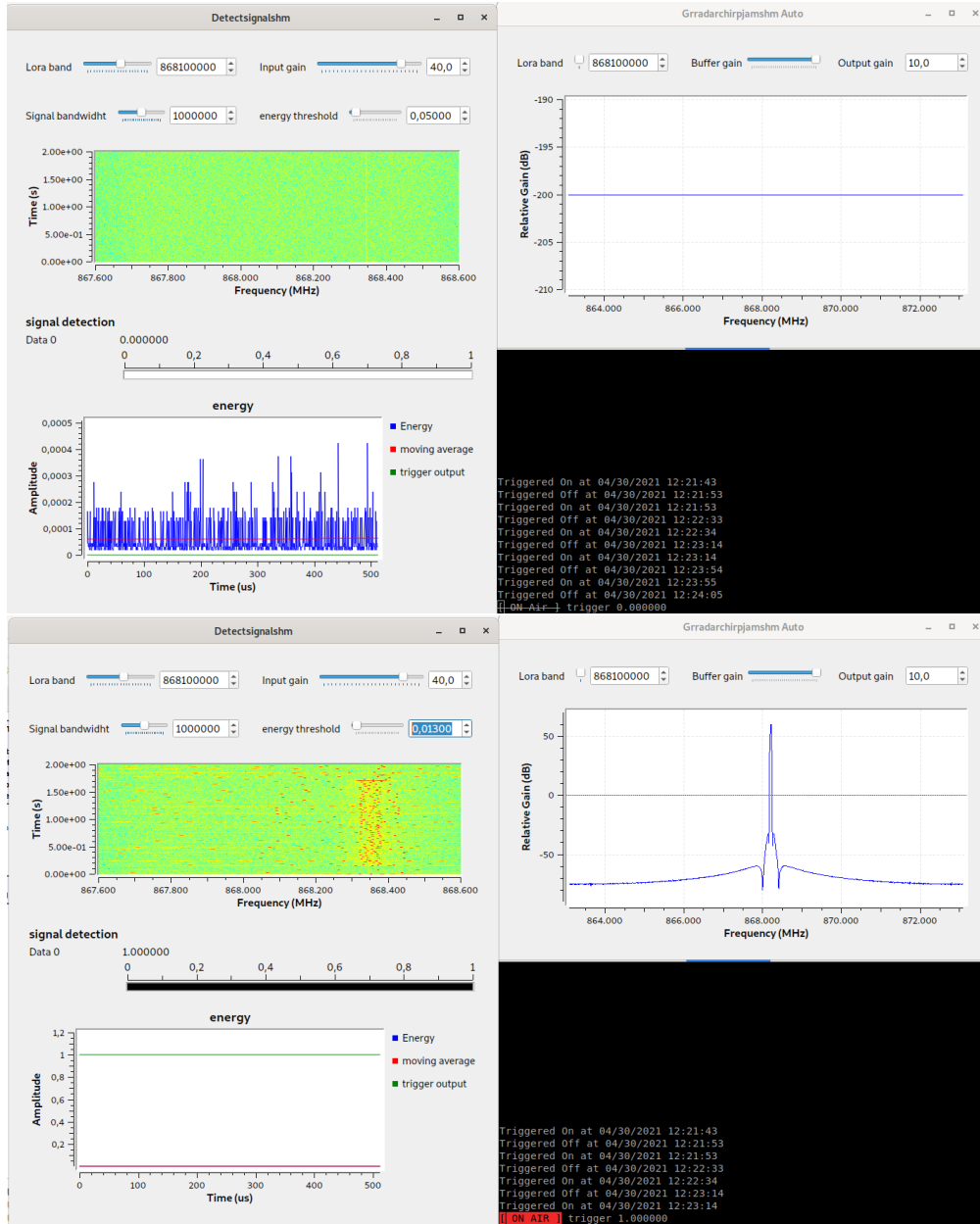


Figure 3.35: reactive jamming in rest mode (lower image) and jam (Upper image)

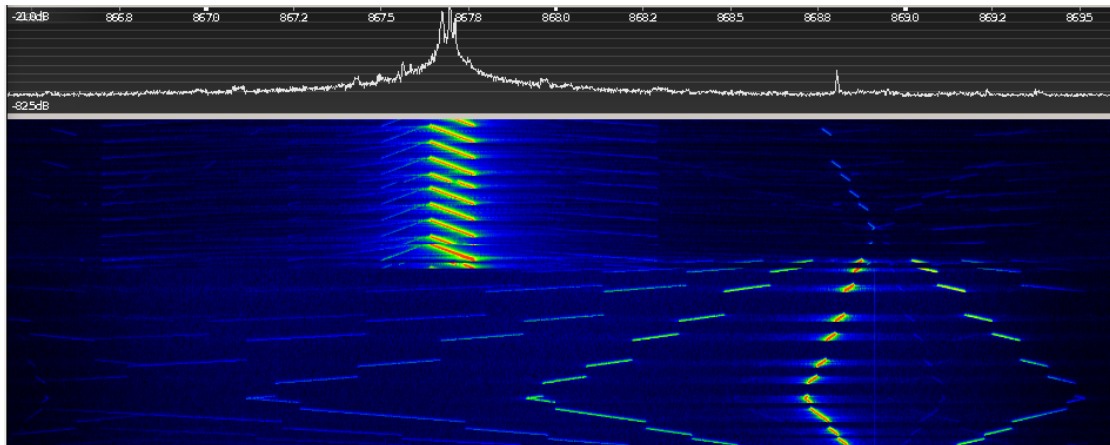


Figure 3.36: RF Spectrum WaterFall showing both LoRa and jamming signals

Monitoring Radio Frequencies channels

The attacker can monitor all RF LoRa channels by using CubicSDR or similar software and tries to estimate the power needed to jam the signals. By viewing the RF spectrum in real time, it is possible to see the differences between the amplitude of LoRa and jamming signals (Figure 3.36).

The peak detection of power spectrum graph can be used to estimate the actual intensity of noise in specific portion of the radio frequencies (Figure 3.37).

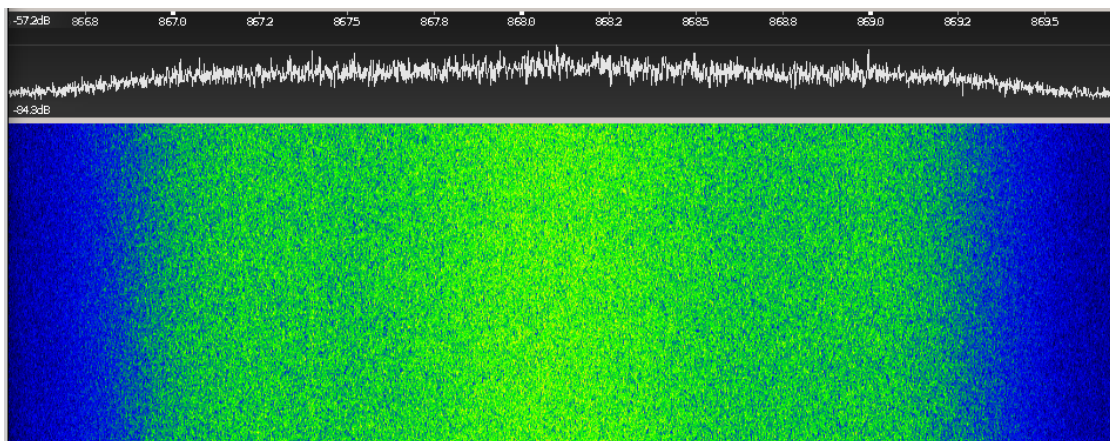


Figure 3.37: RF Spectrum WaterFall showing the noise power level

We can track the differences between the overall RF amplitude of LoRa and jamming signals by using peak detection (Figure 3.38).

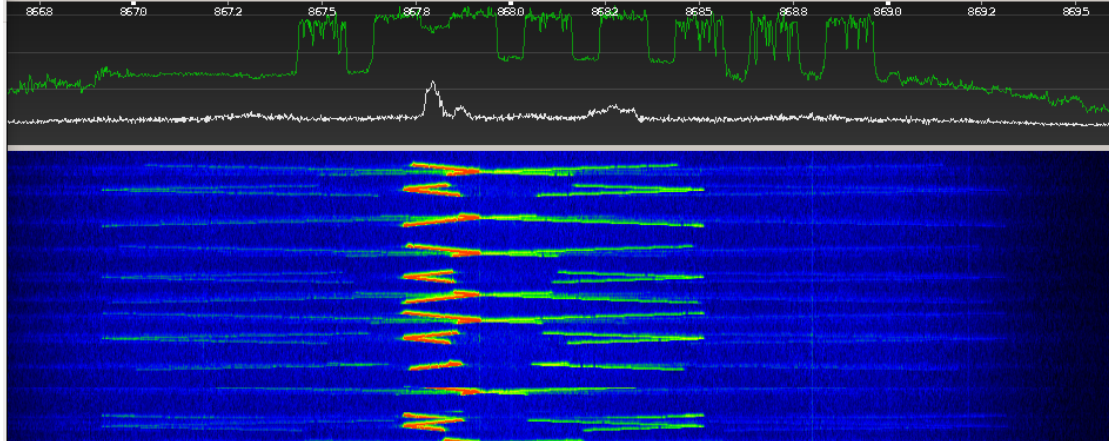


Figure 3.38: RF Spectrum WaterFall showing the jamming and the power peaks of LoRa signals

The attacker can also use a monitor device to see if the jamming has the expected effects on the LoRa signals. A successful jam signal completely disrupts the LoRa signal, as show in the right part of Figure 3.39. In the left part of the same figure, we show a LoRa signal sent on frequencies different from the disturbed ones.

Signal presence detection

The presence or absence of a signal in a specific frequency channel can be estimated by using the power of the signal in that channel.

The signal $x(t)$ captured with the SDR card is filtered through a Bandpass Filter to remove all unwanted frequencies, obtaining the $y(t)$ signal. The power of this digital signal can be calculated by using the formula $P = |y(t)|^2$.

Figure 3.40 shows the part of the GNU radio flowgraph that carries out the signal presence detection.

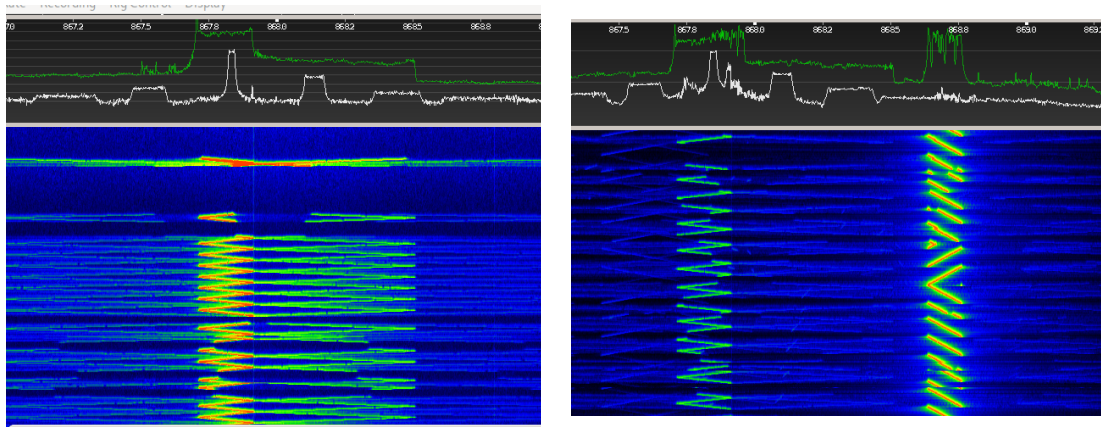


Figure 3.39: RF spectrum of a successful (right image) or unsuccessful (left image) jam of a LoRa signal

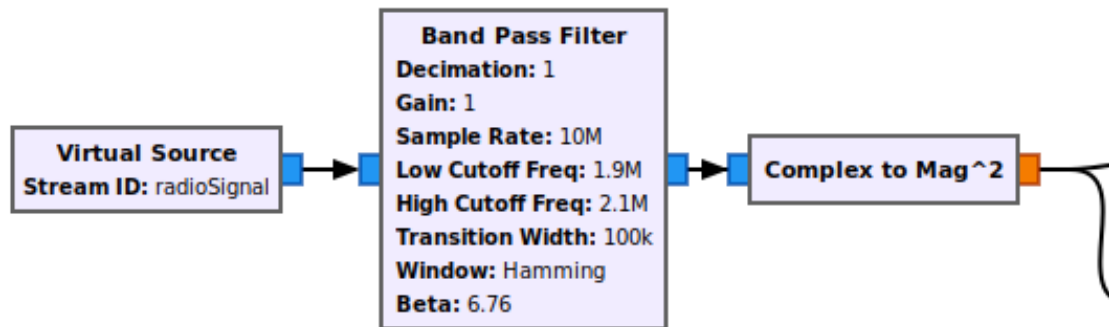


Figure 3.40: Signal presence detection by using GNU radio script

A threshold energy value P_{THR} can be used to estimate the presence or absence of a transmission. When there is no transmission on the channel, the power of the signal $y(t)$ can be near to the power of the channel noise (see Figure 3.41), but some electromagnetic noise spike can temporarily increase the power value P over the expected threshold value P_{THR} .

To avoid triggering the jamming process on isolated spikes, we compute a moving average over the last N values of the estimated signal power. This operation has the effect of smoothing the transitions (see Figure 3.42) and the power spikes

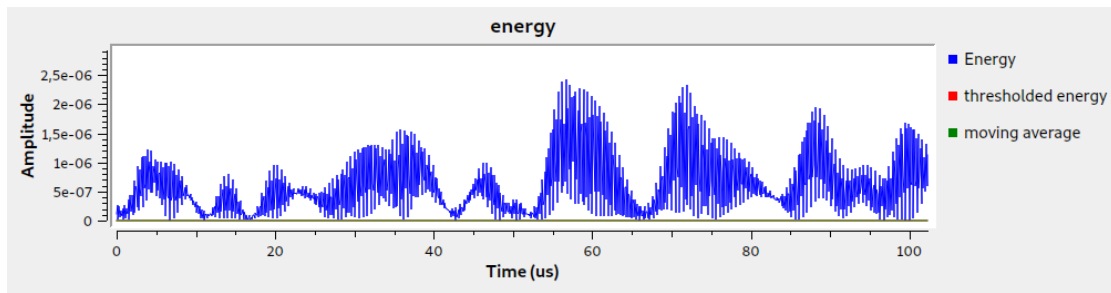


Figure 3.41: Power of the noise detected by using GNU radio script

avoiding false detections, but introducing some delay on the signal detection.

A transmission is considered present on the monitored frequency channel when the signal power P_{AVG} moving average is greater than the Power threshold P_{THR} . This threshold depends on the noise power level and changes over time.

This approach has been used to build the GNU radio flowchart in Figure 3.43. The embedded Python "Trigger" block has a triggered timer and the output is used to switch On or Off the jamming signal generation. This block triggers when the P_{AVG} moving average of the sensed signal power is above a user selectable threshold. If the block is triggered, the output changes to 1.0, otherwise, the output is set to 0.0. The block remains triggered for a selectable amount of seconds (1 second in the test). After this "ON" period, the trigger ignores the moving average for a selectable amount of seconds (0.1 second in the test) to avoid to be triggered by the tail of the jamming signal energy. In the tests, the parameters are set as follows: a trigger ON timer of 1 second; a ignore timer of 0.1 second; output 0.0 when not triggered and 1.0 when triggered. The core logic of the code can be seen in listing 3.1.

Multiplying the jam signal by the output of this block to switch off the jamming signal when a signal is not detected as show in Figure 3.44 for a Baseband jamming signal.

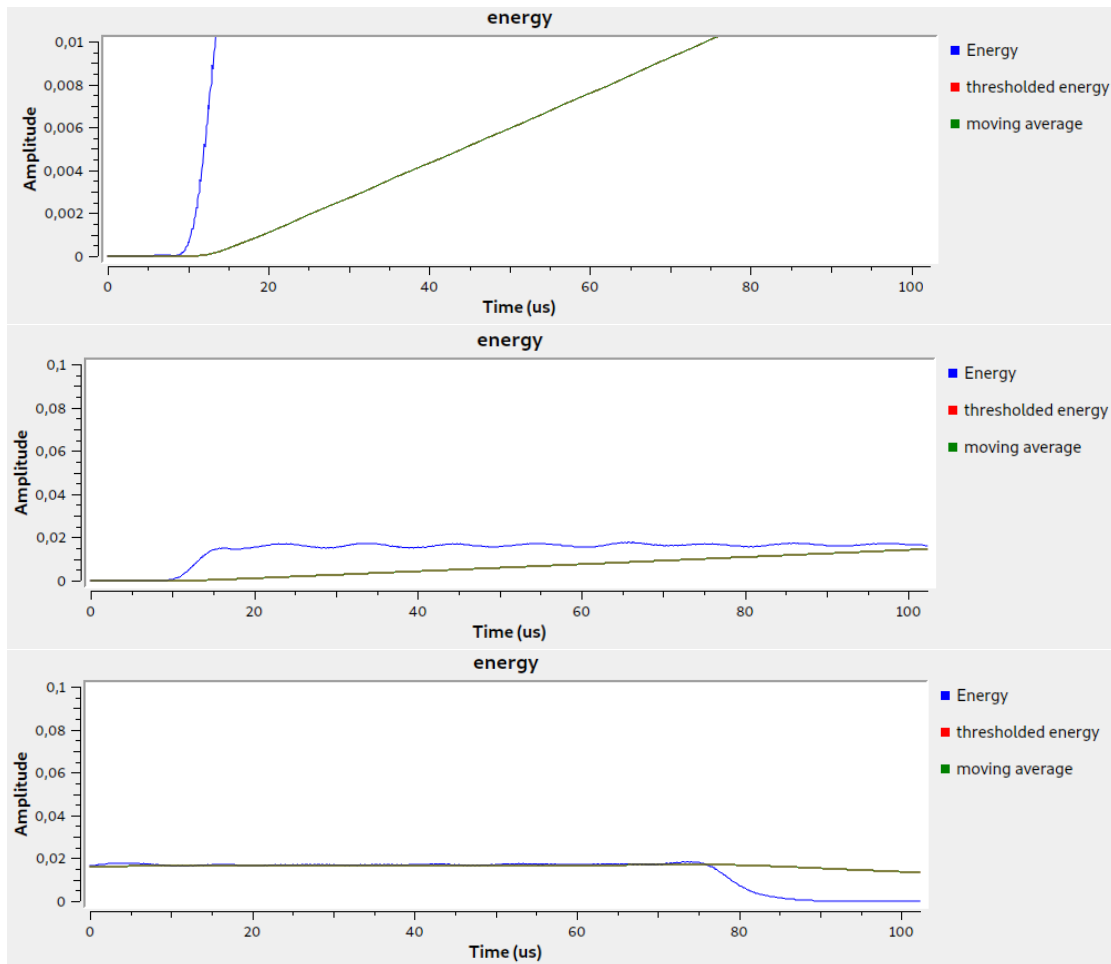


Figure 3.42: Moving average smoothing effect when signal start or end

```
if self.triggered:
    if time.time() >= self.timestamp:
        # switch off when timer is expired
        print("Done")
        self.shm.buf[0:4] = struct.pack('f', self.offValue)
        self.triggered = False
        self.timestamp = time.time() + self.ignore_timer
elif time.time() >= self.timestamp:
    # trigger ?
    m = min(input_items[0])
    if m >= self.threshold:
        self.triggered = True
        self.timestamp = time.time() + self.trigger_timer
        print("T %f %s" % (input_items[0].all(), time.strftime(
            '%m/%d/%Y %H:%M:%S', time.localtime(self.timestamp))))
        self.shm.buf[0:4] = struct.pack('f', self.onValue)

if self.triggered:
    for i in range(0, len(output_items[0])):
        output_items[0][i] = self.onValue
else:
    for i in range(0, len(output_items[0])):
        output_items[0][i] = self.offValue
return len(output_items[0])
```

Listing 3.1: Code extracted from Trigger block to show the trigger delay logic

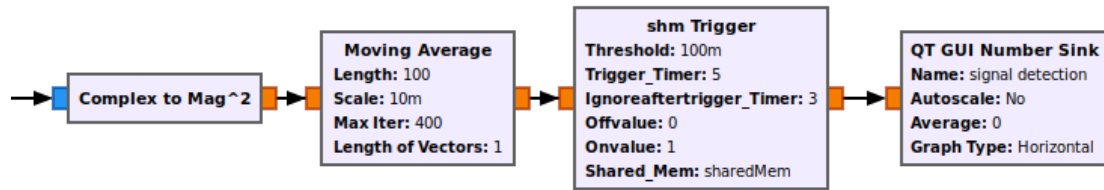


Figure 3.43: GNU radio flowgraph for signal presence detection with embedded python block

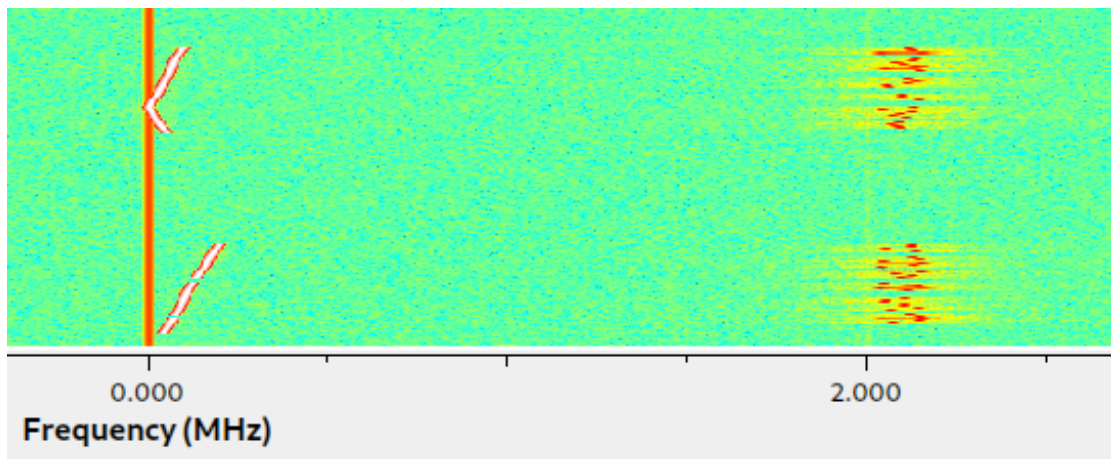


Figure 3.44: Jamming signal multiplied by the signal present trigger output

Compiling GNU radio by using PyBOMBS

To run the jammer prototype, I used the GNU Radio radio version 3.8. To compile GNU radio 3.8 (last current version 3.8.2) by using PyBOMBS, I followed the instructions in <https://github.com/GNURadio/GNURadio> and presented in listing 3.2.

Then, I installed the missing blocks following the instructions found on https://sdr-setup-notes.readthedocs.io/en/latest/software_grc.html.

Some modules need to explicitly set the 3.8 branch by using the following command:

```
pybombs config -P gr-osmosdr gitbranch gr3.8.
```

```
sudo -H pip3 install PyBOMBS
pybombs auto-config
pybombs recipes add-defaults
pybombs prefix init ~/GNU Radio -R GNU Radio-default

# Compile other needed modules
pybombs install gr-LoRa gr-LoRa2 gr-radar
sudo ldconfig
volk_profile # make a profile for optimized fft on local system
```

Listing 3.2: Commands used to build GNU Radio version 3.8

3.7.1 Software Defined Radio useful tools

Some LoRa packets can be captured for offline analysis by using the following command in the HackRF card:

```
hackrf_transfer -r LoRa12sf200bw.cu8 -f 868000000 -s 8000000
```

The frequency band to tune is 868.00 MHz and is specified in Hz after the '-f' option of "hackrf_transfer" tool. Table 3.4 reports the frequency bands used by the LoRa signals. The sample rate is set to 8 000 000 samples per seconds, each sample is composed of a IQ pair of 8 bit value (complex byte, see figure 3.45).

For the Nyquist–Shannon sampling theorem, the maximum visible frequency range is ± 4 MHz, centered in 868 MHz. So, for each second of capture, the software writes $2 \frac{\text{Bytes}}{\text{samples}} * 8 * 10^6 \text{ samples} = 16 \text{ MBytes}$ of data. We can use “dd” command to extract part of the samples. We can use the formula $ofs = 2 * int(T * 8 * 10^6)$ to convert the sample position from second to sample position inside the file (byte offset), where $int()$ extracts the integer part of a floating point number. The opposite conversion (byte offset to time position) can be calculated

LoRa Frequency Band	LoRa Channel Frequency
863 to 870 MHz	868.10 MHz (used by Gateway to listen)
	868.30 MHz (used by Gateway to listen)
	868.50 MHz (used by Gateway to listen)
	864.10 MHz (used by end node to transmit Join Request)
	864.30 MHz (used by end node to transmit Join Request)
	864.50 MHz (used by end node to transmit Join Request)
	868.10 MHz (used by end node to transmit Join Request)
	868.30 MHz (used by end node to transmit Join Request)
	868.50 MHz (used by end node to transmit Join Request)

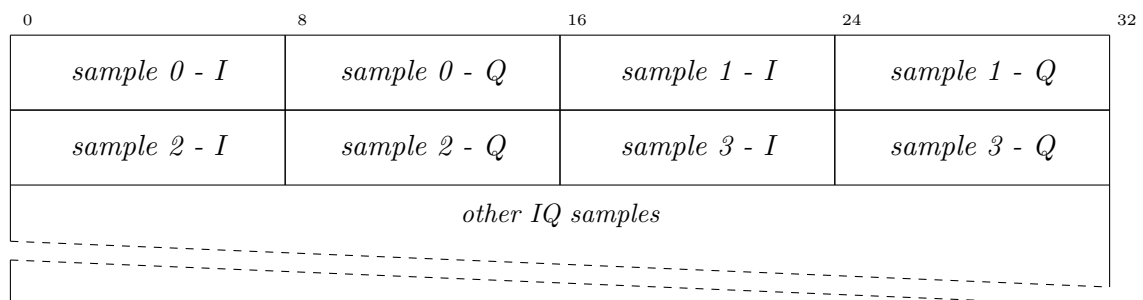
Table 3.4: LoRa frequency table¹³

Figure 3.45: Binary format used by hackrf_transfer to store IQ samples on file

as $T = \frac{\text{int}(ofs/2)}{8*10^6}$.

Different open source tools are available to plot the spectrum of the captured signal, such as `inspectrum`¹⁴. Figure 3.46 shows a waterfall spectrum graph of a LoRa packet captured with "hackrf_transfer" and plotted with "inspectrum" tool.

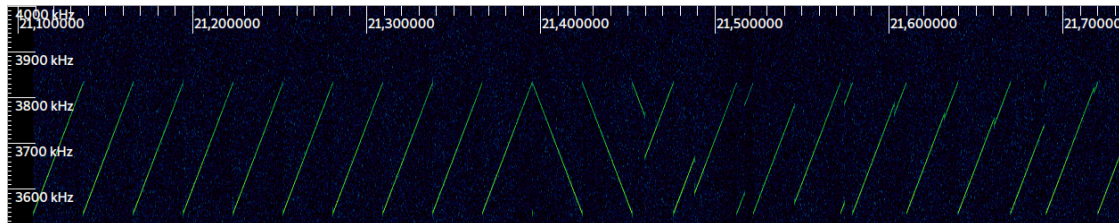


Figure 3.46: Inspectrum showing packet captured from `hackrf_transfer`

A multitude of different Open Source programs, such as `CubicSDR` and `gqrx`, can show in real-time the RF spectrum of the signals captured with the SDR card.

3.7.2 LoRa temperature sensing IoT demoboard

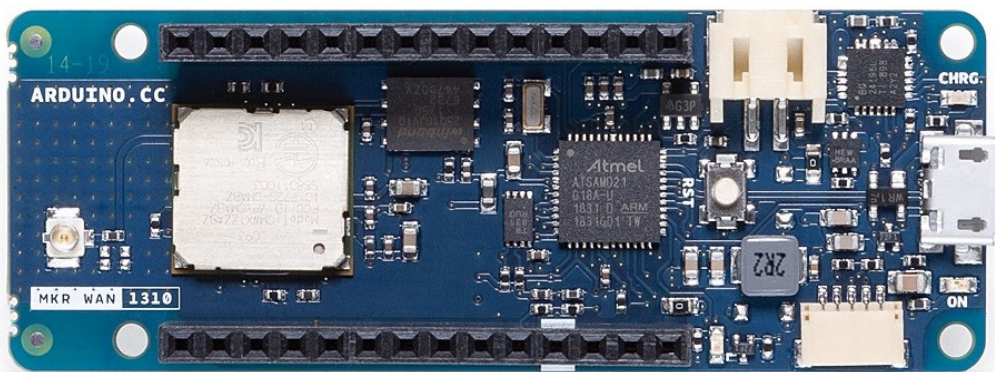



Figure 3.47: Arduino MKR WAN 1310 LoRa board

I built a LoRa end node interactive demoboard able to send a LoRa packet each time a mechanical button is pressed or every 30 seconds. This device is based

¹⁴available as `dpkg` in Ubuntu or through git  <https://github.com/miek/inspectrum.git>

on Arduino MKR WAN1300 or MKR WAN1310 (see Figure 3.47) connected to a daughter board (see Figure 3.50, schematics in Figure 3.48, board layout in Figure 3.49) with one button and four Light Emitting Diode (LED). The device can be controlled through the USB serial port.

Figure 3.50 pictures the demo board.

To avoid software deadlocks, the sketch uses a Watchdog to automatically reboot the device avoiding software crashes. The Watchdog timeout is set to 10 seconds but is temporally disabled during the network join operation. The sketch is able to send a LoRa packet when a button is pressed or every 30 seconds.

The onboard LED is used to have a feedback of the operation done by the Arduino main code. During the setup procedure, it is on, then, during the normal operation, toggle on and off states every second. The demoboard has one LM35 analogue sensor used to sense the environmental temperature and one button able to do different actions depending on how much time it is pressed. When pressed for a short time, the device immediately sends a LoRa packet, when pressed for more than 5 seconds, the device re-starts the LoRa Over The Air Authentication (OTAA) Join procedure. Also the other LED on the demoboard are used to have some visual feedbacks on the status of the LoRa Network and of the LoRa transmissions:

- LED D1 yellow: on when the board is actively operating LoRa transmissions, such as when it sends a data packet or during the join network process.
- LED D2 orange: on when the LoRa device is connected to the LoRa network;
- LED D3 red: on when the LoRa packet transmission failed;
- LED D4 green: on when the LoRa packet transmission succeed;

The demoboard has also Transistor-Transistor Logic (TTL) level Serial, Inter Integrated Circuit (I2C), Serial Peripheral Interface (SPI), 3.3V and signal ground

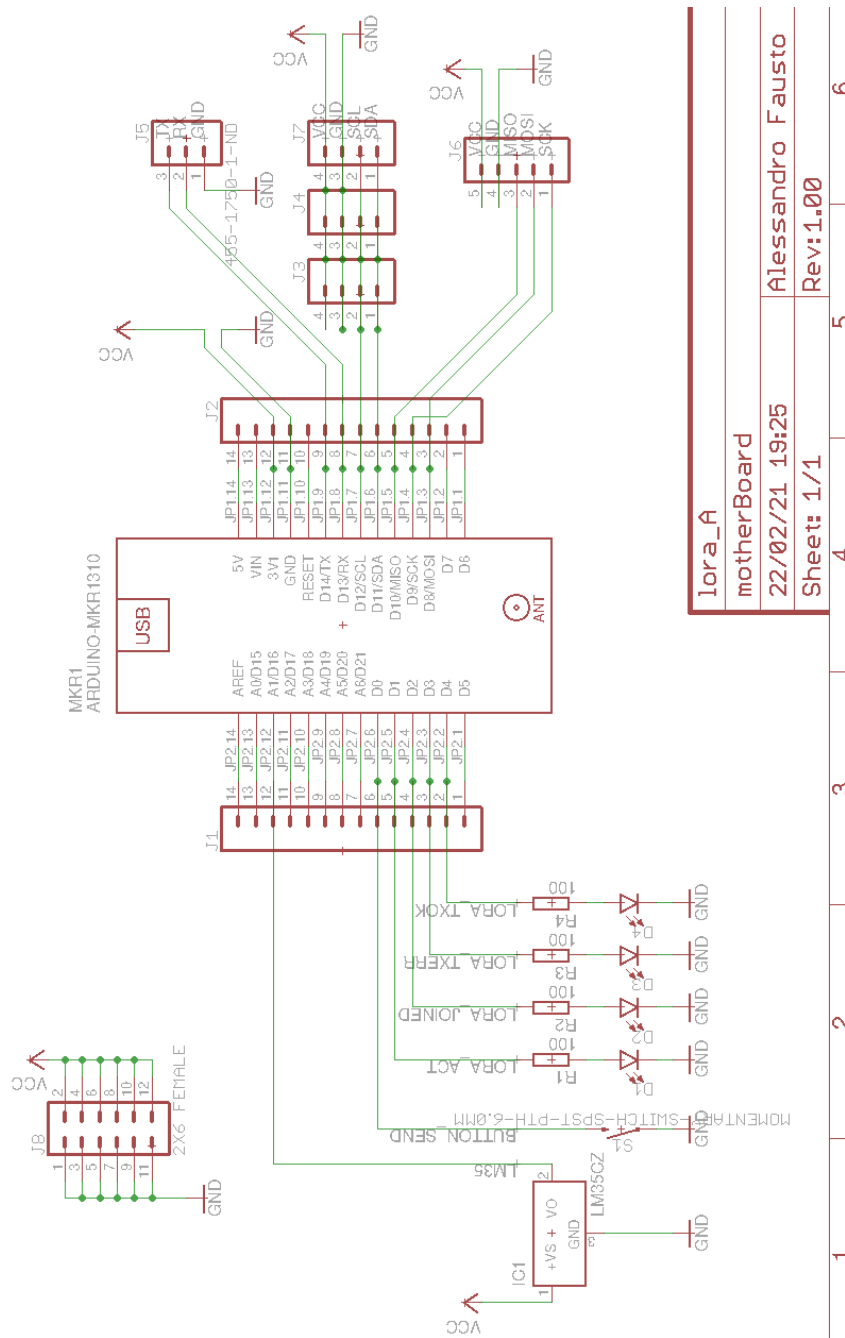


Figure 3.48: Schematic of testbed board LoRa_A Release 1.00

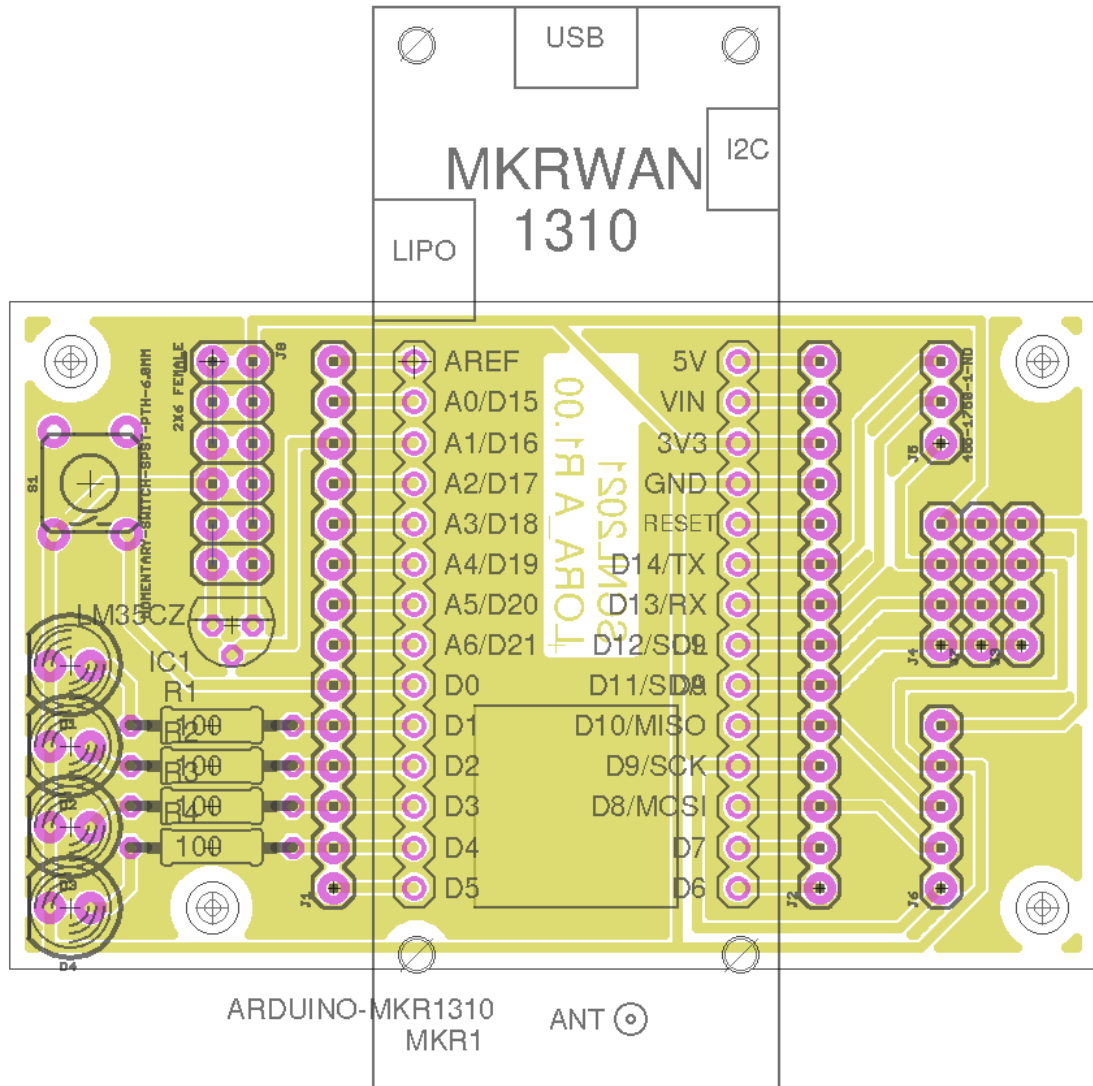


Figure 3.49: Board layout of testbed LoRa_A Release 1.00

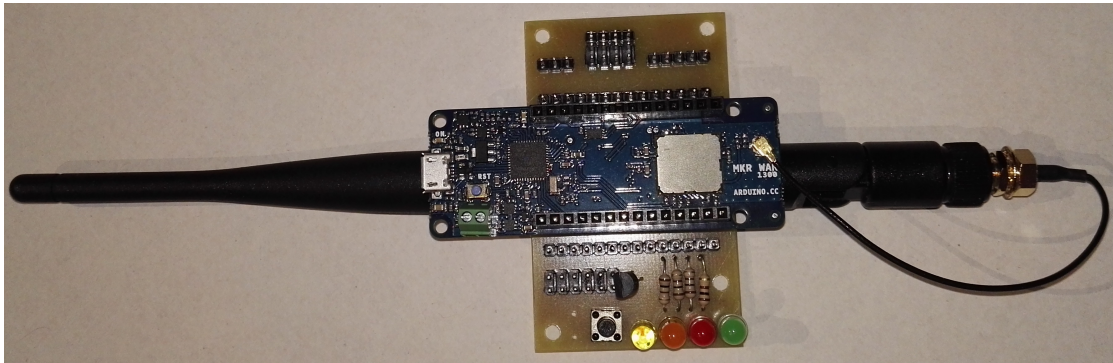


Figure 3.50: Fully working prototype of LoRa_A Release 1.00

connectors for future expansion.

The sent LoRa packet contains as a payload the environment temperature measured by the LM35 temperature sensor converted in string format and padded with some "_" character to reach the user defined packet length.

The device can be controlled through the USB virtual serial port (ttyACM in Linux) by using a normal serial terminal (Putty in Windows or Minicom in Linux). The device listens on the serial port and performs some actions when it receives the following characters:

' ' print the module version, the device EUI, and the information menu;

'+' or '-' increase or decrease the packet length;

's' send a LoRa packet now;

'm' reset the LoRa MoDem;

'r' restart the device;

'j' rejoin the network by using the OTAA Join procedure.

Carriage Return (CR), Line Feed (LF), and any other characters will be ignored.

The device sends debug information on the serial port (see Figure 3.3). It sends one single char every second to signal that it is still working (not crashed). It prints the timestamp (in uptime seconds), the payload length, the payload content, the transmission result (succeeded or failed), and the time needed by the library to complete the data packet creation (in milliseconds) for each packet when it has been sent. It also prints the request result (succeeded or failed) and the time needed by the LoRa library to complete the request creation (in milliseconds) for each OTAA Join network request.

3.8 Analysis of features during jamming

The network traffic was memorized during the tests carried out with the jammer and analyzed to compare the performance of the extracted features with and without active jamming.

Significant differences emerged in three of the features. These features are directly linked to the actions performed by the jammer on the radio communications of the LoRaWAN network.

Switching on the jammer causes a sharp increase in the inter-arrival time between two consecutive messages (Figure 3.51).

At the same time, the LoRa packets that reach the LoRa Gateway have a much lower SNR than that detected during the tests in the absence of jamming (Figure 3.52).

The Scatterplot of the "SNR" and "fiveMinPacketCount" features in Figure 3.53 shows two clusters well separated even if not perfectly defined. The cluster in the top of the graph contains the features in the normal situation and the cluster in the bottom part contains the features in anomalous situation.

```

Serial configured for 115200 baud.
Reset the controller setting the speed of virtual serial to 1200 by using the following command
sudo stty -F /dev/ttyACM0 speed 1200 cs8 -cstopb -parenb
Set the watchdog timeout to 10 seconds
Modem module correctly started
Your module version is: YYY-999 4.5.0
Your device EUI is: XXXXXXXXXXXXXXXX
Temporarily disable the watchdog
Start join OTAA procedure by using XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX key, please wait.
LoRa device is connected to the network (6240).
Set the watchdog timeout to 10 seconds
Setup completed.
bcdefghijklmnopqrstuvwxyzabcd
uptime 29 s , payload(62): 29.814272_-----
Message sent correctly (58ms)!
efghi
BUTTON pressed, send packet!

uptime 34 s , payload(62): 29.814272_-----
Message sent correctly (49ms)!
jklmnopqrstu
BUTTON pressed for long time, rejoin!
Temporarily disable the watchdog
Start join OTAA procedure by using XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX key, please wait.
⌘.....⌘
Uptime: 710 minutes.
mnopqrstuvwxyzabcdefghijklmnop
uptime 42629 s , payload(62): 28.836754_-----
Error sending message (11ms).
qrstuvwxyzabcdefghijklmnopqrst

```

Listing 3.3: Dump of Arduino demoboard serial output

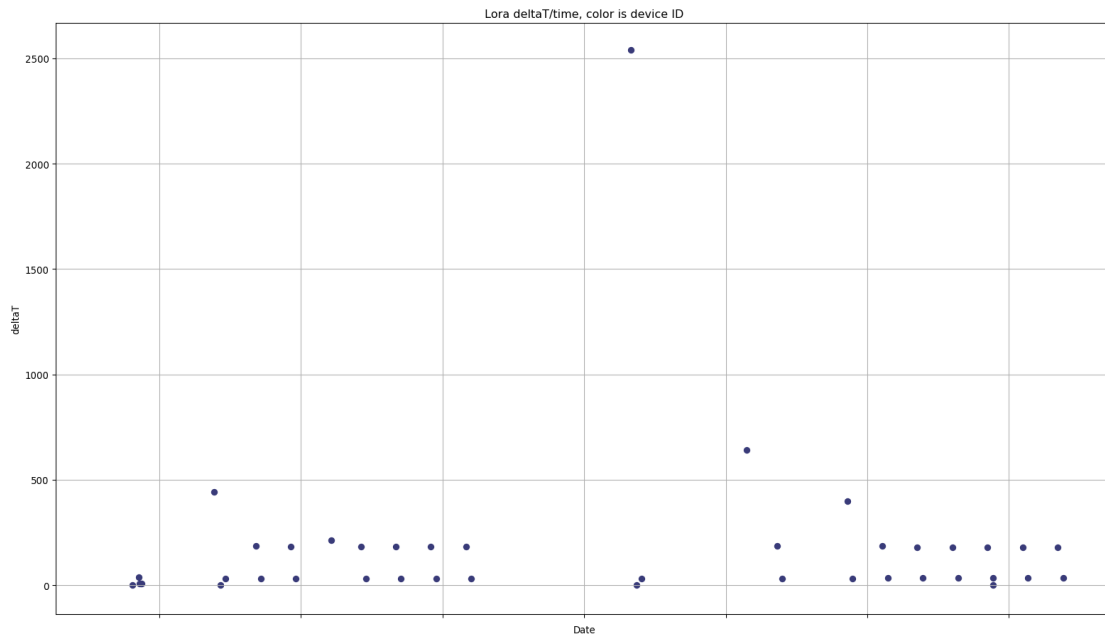


Figure 3.51: Graph of the inter-arrival time between two consecutive packets. Notice the "holes" in periodicity due to the action of the jammer

3.9 LoRa IDS to detect RF jamming

The LoRa Jam Intrusion Detection System (LoRaJamIDS) was built by using the Python language, the scapy library for the extraction of network data and the scikit-learn library for the ML algorithms. The system must be able to decide if a new sample of features belongs to normal traffic (inlier), or should be considered different (outlier). The novelty detection algorithms are able to detect whether a new set of features is an outliers (also called a novelty) which is defined as observations that are far from the others. Four novelty detection ML algorithms have been tested: Robust covariance, One Class Support Vector Machine (OCSVM), Isolation Forest (iForest) and Local Outlier Factor (LOF).

These algorithms during the training phase learn a close frontier approximately delimiting the N-dimensional contour (bidimensional in our case) of the initial

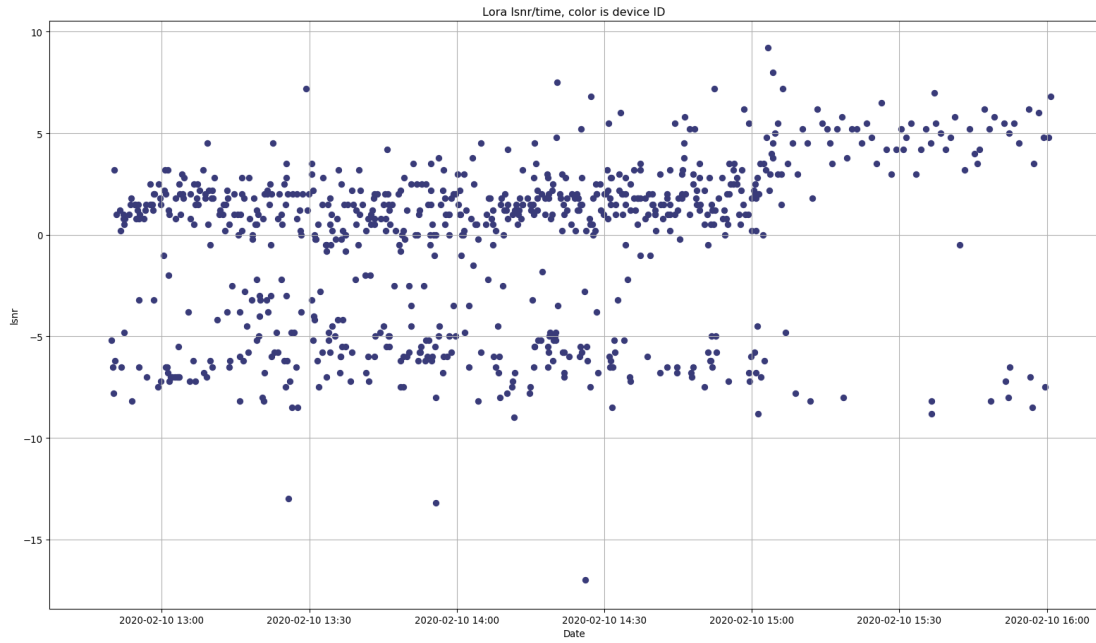


Figure 3.52: Graph of the SNR feature under the jamming action

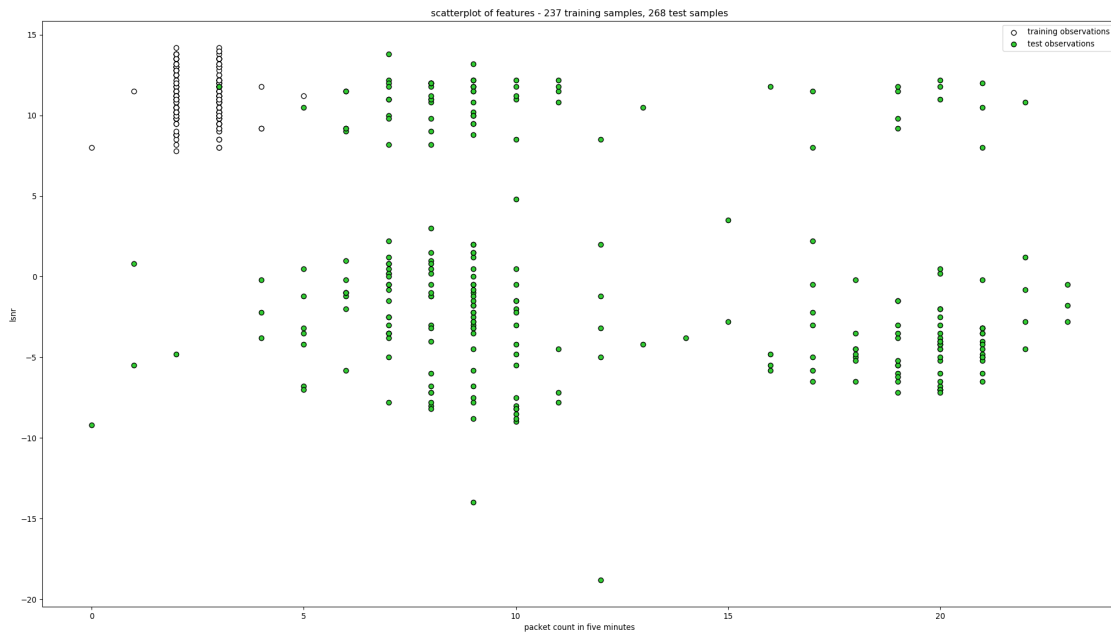


Figure 3.53: Scatterplot of "SNR" and "fiveMinPacketCount" features showing clusters of features in normal and anomalous states

training dataset.

When new samples fall inside the frontier, they are considered as coming from a communication made in "normal" situation. Otherwise, if they falls outside the frontier, we can say with some degree of certainty that they are novel and coming from transmissions made under jamming attack.

One common way of performing novelty detection is to assume that the normal dataset (inlier) comes from a known distribution (by example Gaussian). Then we generally try to define the "shape" of the data in the n-dimensional feature space, and the samples which stand far enough from the frontier contour can be defined as outlier/novelty.

The scikit-learn provides a Robust Covariance method that applies a robust covariance estimation to the training dataset, and surrounds with an ellipse the central part of the data points ignoring all points outside the central part. The distance from this elliptical frontier is considered to classify the samples as inlier or outlier/novelty.

The OCSVM is an unsupervised outlier/novelty detection ML algorithm and has been introduced by [22]. A classic Support Vector Machine (SVM) ML algorithm search an hyperplane that separates two classes of samples keeping the maximum possible distance from all n-dimensional samples. The OCSVM finds a hyper-plane that separates the given dataset from the n-dimensional origin keeping the hyperplane at the minimum distance from all datapoints. To be able to encompass a non-linear frontier, we use an Radial Basis Function (RBF) as kernel.

One efficient way of performing novelty detection is to use the iForest ML algorithm introduced by [23]. The samples are 'isolated' by using a recursive partitioning of the feature hyperplane by randomly selecting both a feature and a split value between its maximum and minimum values. The recursive partitioning can be seen as a decision tree. The number of splittings required to isolate a sample

correspond to length of the path from the tree root node to the leaf node. This measure, averaged over an ensemble of trees (a forest), can be used to discriminate between a normal or abnormal/novel sample because the random partitioning produces shorter paths for anomalies.

The LOF ML algorithm identifies anomalous/novel samples measuring the local density deviation of a given data point with respect to its neighbours. A normal sample is expected to have a local density similar to the one of its neighbours, while abnormal data are expected to have much smaller local density. The number k of neighbours considered can be selected by the user. The question is not how isolated the sample is, but how isolated it is with respect to the surrounding neighbourhood.

Table 3.5 summarises the features considered to be used with a ML algorithm for the novelty identification. The features "fiveMinPacketCount" and "interarrivalTime", both described at page 93, are linearly dependent so it is not useful to use these features together. We can try grouping the remaining features in order to generate, if possible, "normal" and "anomalous" clusters well separated on the feature plane. We can use the "SNR" feature, both described at page 91, together with "fiveMinPacketCount" or "interarrivalTime" to build two groups of features $A = [\text{"fiveMinPacketCount"}, \text{"SNR"}]$ and $B = [\text{"interarrivalTime"}, \text{"SNR"}]$. Both groups take into consideration the communication periodicity and the electromagnetic noise of the radio channel where the LoRa transmission takes place. The "fiveMinPacketCount" feature keeps track of the number of packets received in the last 5 minutes while the "interarrival" feature measures the time between two consecutive messages. Both of these features allow you to keep track of the traffic volume of LoRa packets and the timing properties (periodicity, minimum ΔT) of sending data from the LoRa end node. In case of periodic transmissions, the number of packets sent in an interval of time larger than the LoRa end node

Features	Description	Notes
"fiveMinPacketCount"	number of packet received in 5 minutes.	In case of periodic transmissions the number of packets sent in an interval of time larger than the LoRa end node periodic transmission leads to a stable number of sent packets.
"interarrivalTime"	interarrival time between two consecutive LoRa packets	remains constant in case of fixed periodic data transmission.
"SNR"	ratio of the Signal power over the signal Noise in dB	depends on both signal and noise powers, so a low "SNR" value is expected for packets coming from long distance. However, a low "SNR" value can be expected also for signals sent from near devices when the noise power is high.

Table 3.5: Considered features for the detection of RF jamming

periodic transmission leads to a stable number of sent packets. The "SNR" feature is extracted directly from the "lsnr" field present in the Push Data message (Operation ID 0x00) of the LoRa protocol inside the JSON item "rxpk" (see chapter 3.4.1)

When a jammer succeed in jamming all LoRa transmissions, the number of packets received in 5 minutes drop to zero and the "interarrivalTime" between consecutive packets grows reaching a value of tens of seconds. In the case of a jamming attack, sometimes a packet overcomes the high level of noise and the distortions suffered by the signal waveform. These packets successfully complete the demodulation and decoding phase of the LoRa protocol by reaching the application layer and then forwarded to the LoRa server via the LoRa forward protocol. At the same time, the LoRa forward protocol is intercepted and analysed by the LoRa-

JamIDS which extracts the "SNR" features and updates the "fiveMinPacketCount" and "interarrivalTime" features. The same jamming operation can potentially be detected by both groups of features. The identification occurs for the *A* feature group when both the "SNR" feature and the "fiveMinPacketCount" feature have very low values. For the *B* group it happens when the "SNR" feature has very low values and at the same time the "interarrivalTime" feature reaches very high values.

All the four tested algorithms allow, through the contamination parameter, to specify the proportion of outliers in the training dataset. In the case that the training dataset contains a certain percentage of anomalous samples, linked for example to transmissions received from very distant sensors that have low "SNR", the training was repeated with different contamination values (as show in Table 3.6) to verify which value guarantees the better performance. The features extracted from the communications captured in the absence of jamming (Figures 3.23 and 3.21) has been used as training dataset.

Then the LoRaJamIDS was tested by using the features extracted from the communications captured in the presence of jamming. (Figures 3.51 and 3.52).

Figures 3.54 and 3.55 show the results changing the group of features (*A* or *B*), ML algorithm and contamination factor (0.01, 0.05, 0.1, 0.5).

The presence of separated clusters of features eases the work of some ML algorithms, for example as happens in the Robust Covariance in Figure 3.56. During the training phase of this algorithm the learned elliptical barrier can encompass the "normal" cluster avoiding to keep inside a vast part of the empty areas. The group *A* or *B* can lead to a better clustering of the feature array depending on the type of decision surface (hyperplane or rules) as show in Figures from 3.56 to 3.59. The group *A*, composed of "fiveMinPacketCount" and "SNR" features, has a better shape (more compact) so the learned barrier of tested ML algorithms can

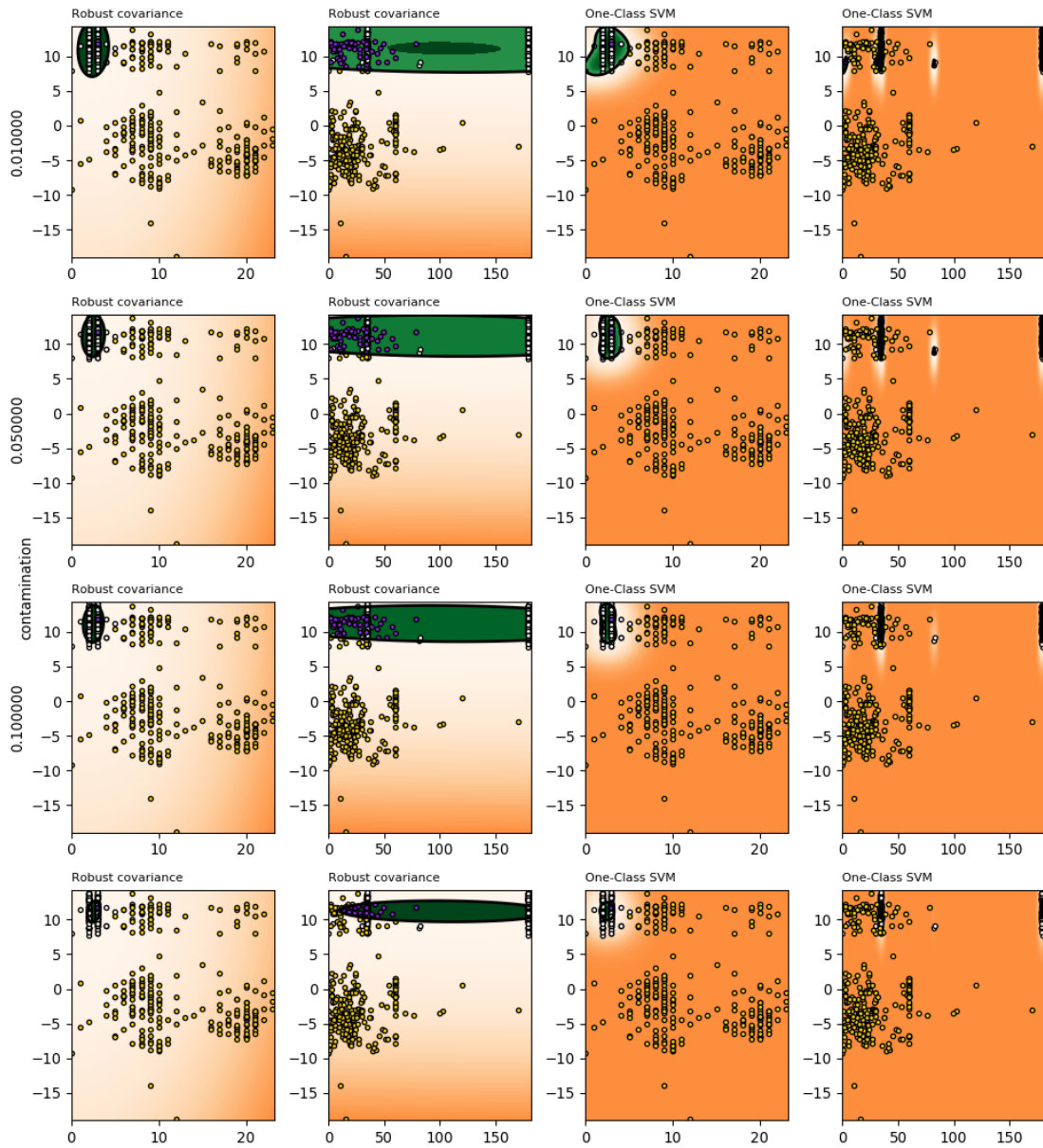


Figure 3.54: Robust covariance and One Class SVM, changing contamination factor and feature group (*A* for Odd columns and *B* for even columns).

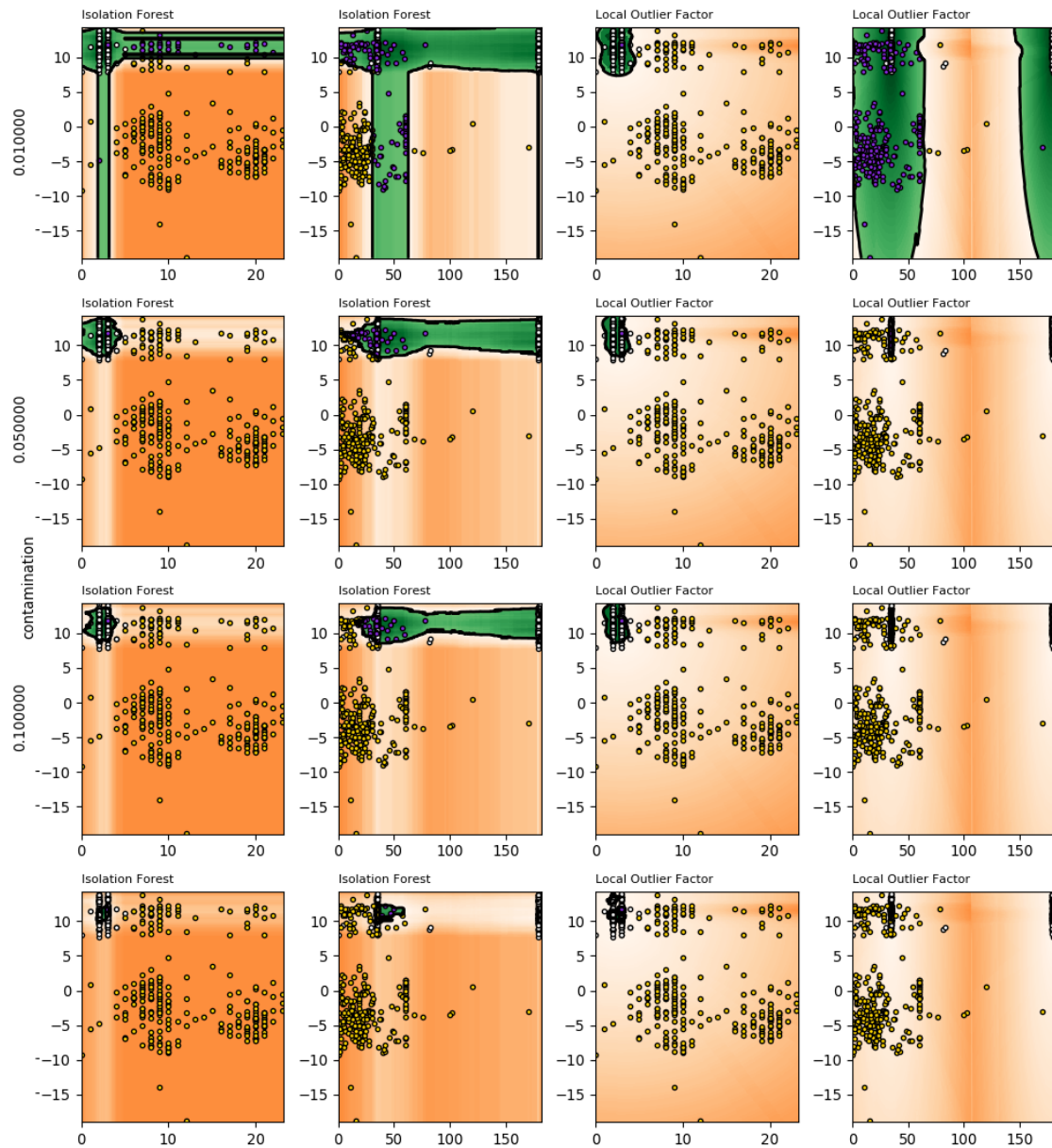


Figure 3.55: Isolation Forest and Local Outlier Factor, changing contamination factor and feature group (*A* for Odd columns and *B* for even columns).

Algorithm	parameters	dataset contamination
Robust covariance	Elliptical frontier shape	0.01
		0.05
		0.10
		0.50
Isolation Forest	100 samples used to train each base estimator	0.01
		0.01
		0.10
		0.50
One Class SVM	RBF kernel with gamma 0.1	0.01
		0.05
		0.10
		0.50
Local Outlier Factor	35 neighbours analyzed during k-neighbours search	0.01
		0.05
		0.10
		0.50

Table 3.6: Tested ML algorithms with parameters

fit well the cluster of "normal" state features.

The ML algorithm that guarantees the best performance for the identification of anomalies/novelty remains to be determined. Figures from 3.56 to 3.59 show how each algorithm separates the samples with similar property on what they have seen during the training and all other sample candidates to be marked as "novel". They also show an empirical visual estimate of the best fit as the ML algorithms vary. As expected, the learned frontier is reduced increasing the value of contamination, leading to an higher rate of false negative. For all algorithm the better precision is achieved with a contamination factor of 0.01. The decision surface of LOF, show in figure 3.59, seem the better to match the characteristics

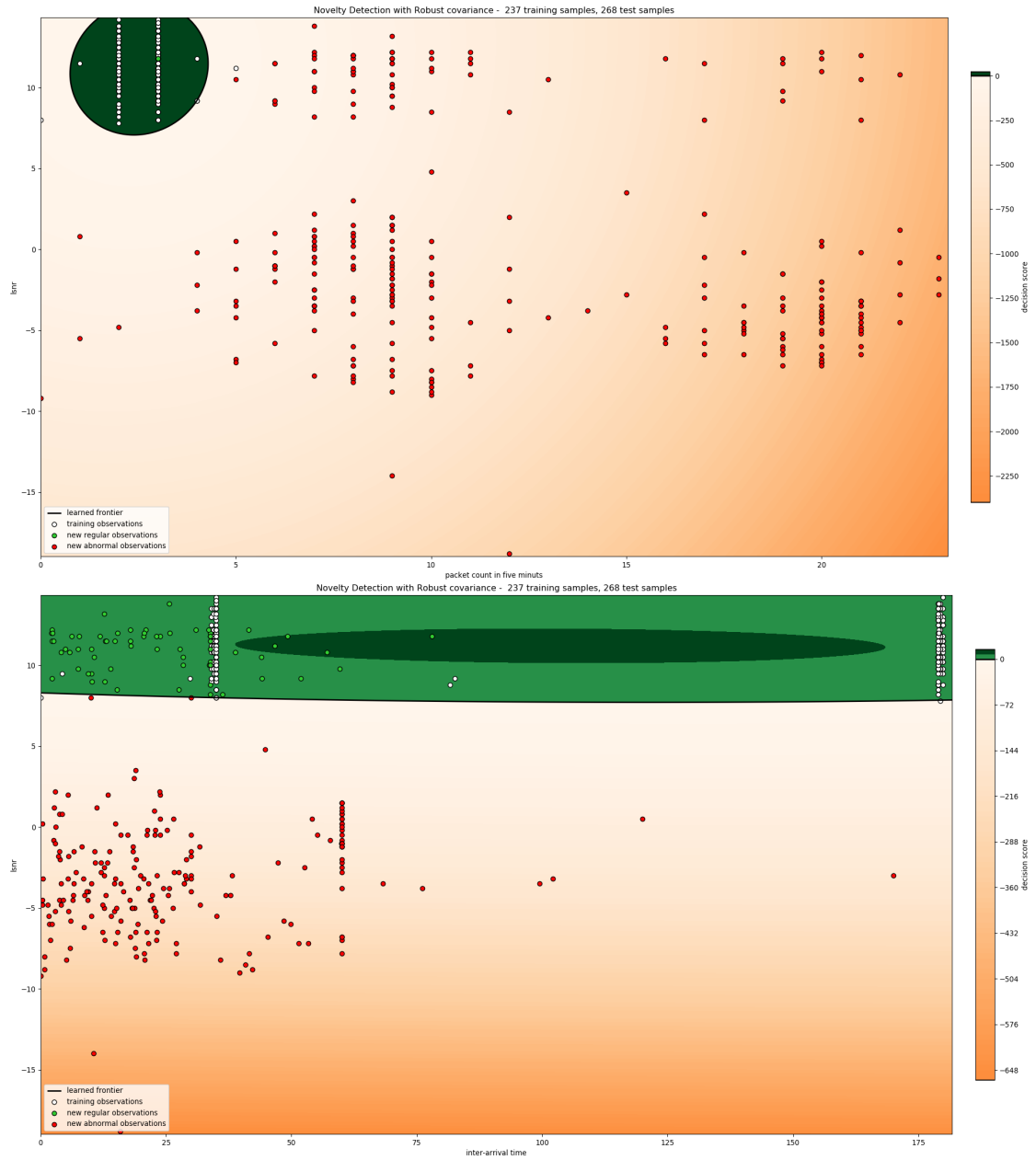


Figure 3.56: Robust covariance on feature group A (above) and group B (below).

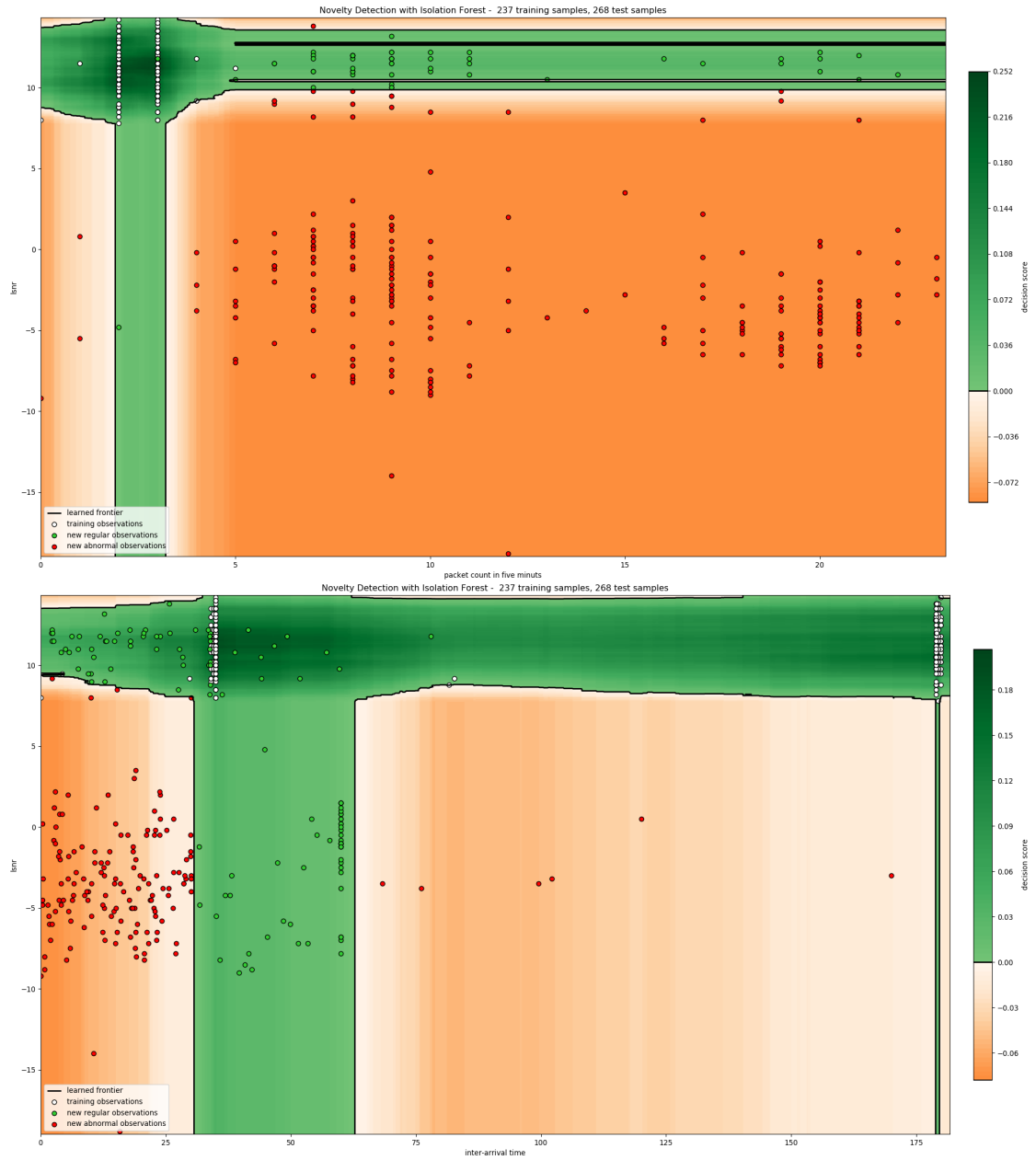


Figure 3.57: Isolation Forest on feature group A (above) and group B (below).

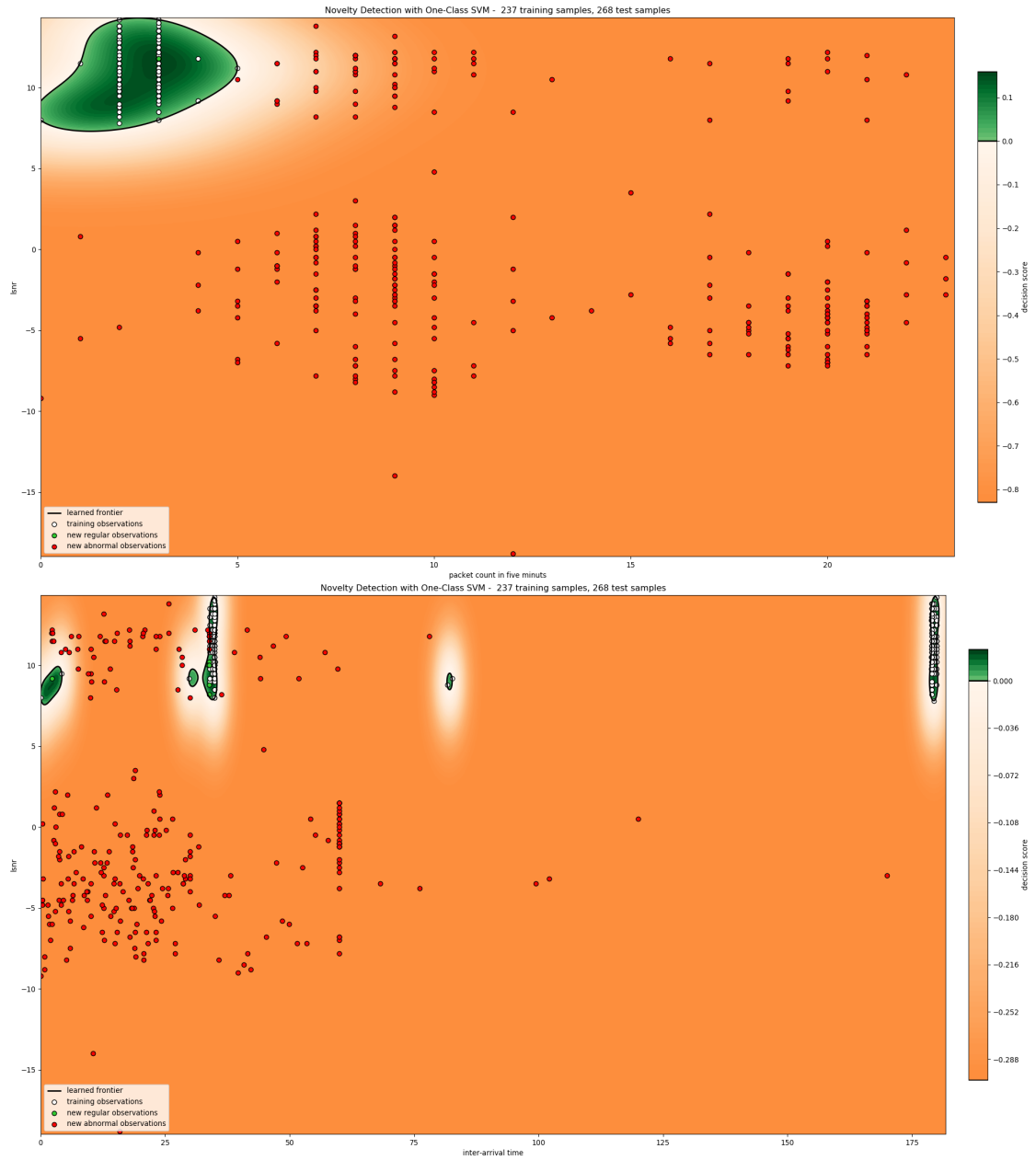


Figure 3.58: One class SVM on feature group A (above) and group B (below).

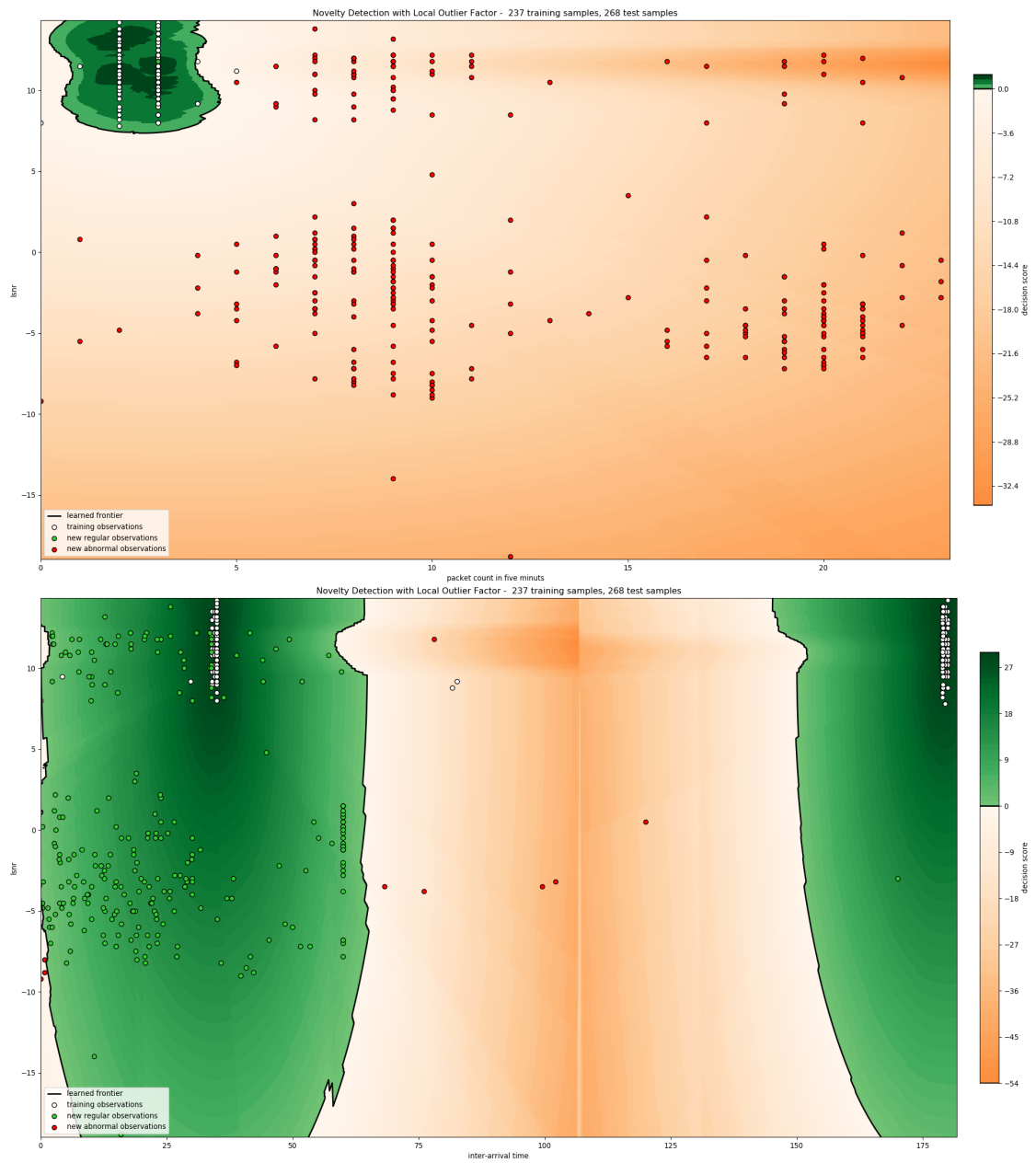


Figure 3.59: Local Outlier Factor on feature group A (above) and group B (below).

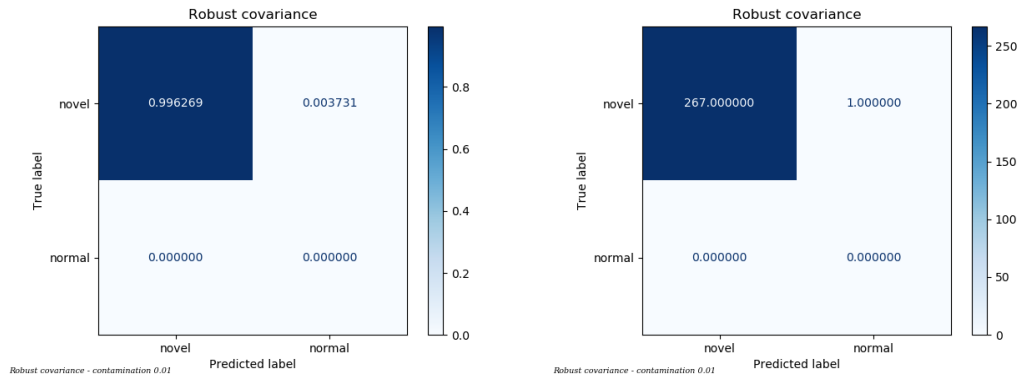


Figure 3.60: Confusion matrix for Robust covariance algorithm

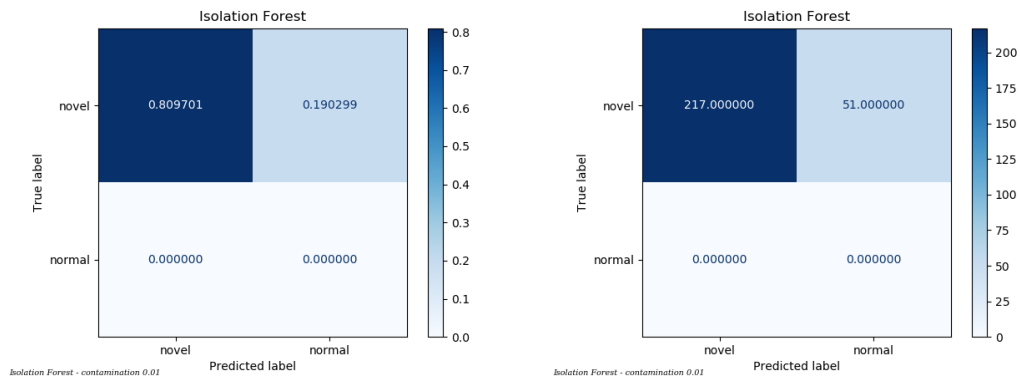


Figure 3.61: Confusion matrix for Isolation Forest algorithm

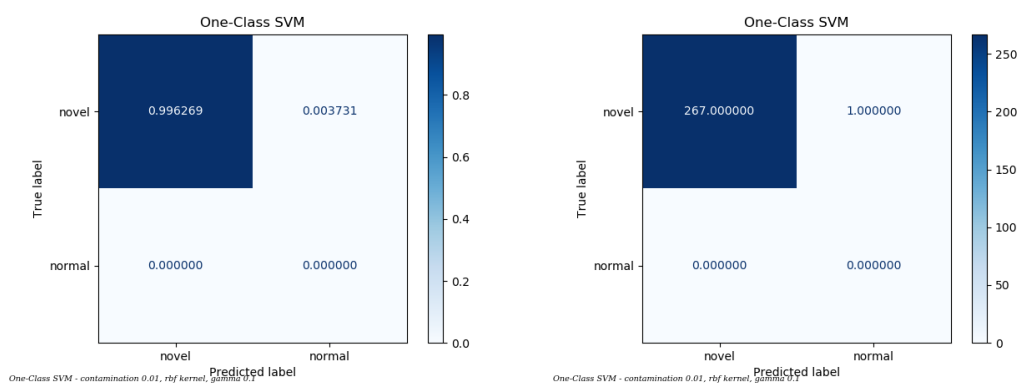


Figure 3.62: Confusion matrix for One-Class SVM algorithm

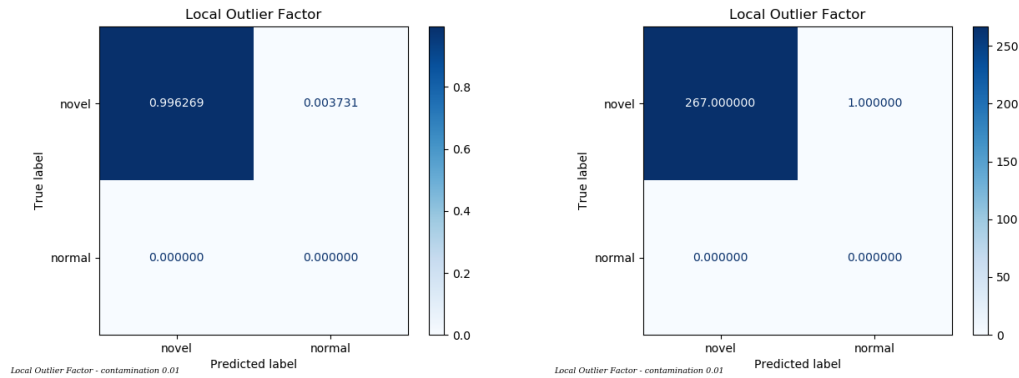


Figure 3.63: Confusion matrix for Local Outlier Factor algorithm

of normal behaviour of the selected features. To confirm the facts, the confusion matrix was calculated for each of the considered ML algorithms in order to identify the one with the best performance. The confusion matrices are shown in Figures from 3.60 to 3.63 in both normalised (on the left) and not normalised form (on the right). The confusion matrix of the iForest ML algorithm shown in Figure 3.61 has a high rate of false negatives (19%) and cannot be considered reliable when compared with the others algorithm. With the exclusion of the iForest ML algorithm, all the others algorithm can be used to detect novelty.

An example of the LoRaJamIDS output is show in Figure 3.64. The first column contains the tree features of the sample to be classified followed by the classification results of the fourMLalgorithm.

```
[ 9 60.002824  1.5] Robust cov -1.0 OC-SVM -1.0 iForest -1.0 LOF -1.0
[10 4.92456 11.]   Robust cov -1.0 OC-SVM -1.0 iForest  1.0 LOF -1.0
[11 33.84149 11.5] Robust cov -1.0 OC-SVM -1.0 iForest  1.0 LOF -1.0
[12 15.831349 -18.8] Robust cov -1.0 OC-SVM -1.0 iForest -1.0 LOF -1.0
[12 5.402017  2.]  Robust cov -1.0 OC-SVM -1.0 iForest -1.0 LOF -1.0
```

Figure 3.64: Textual dump of LoRa IDS classification output. False Negatives are show in red color.

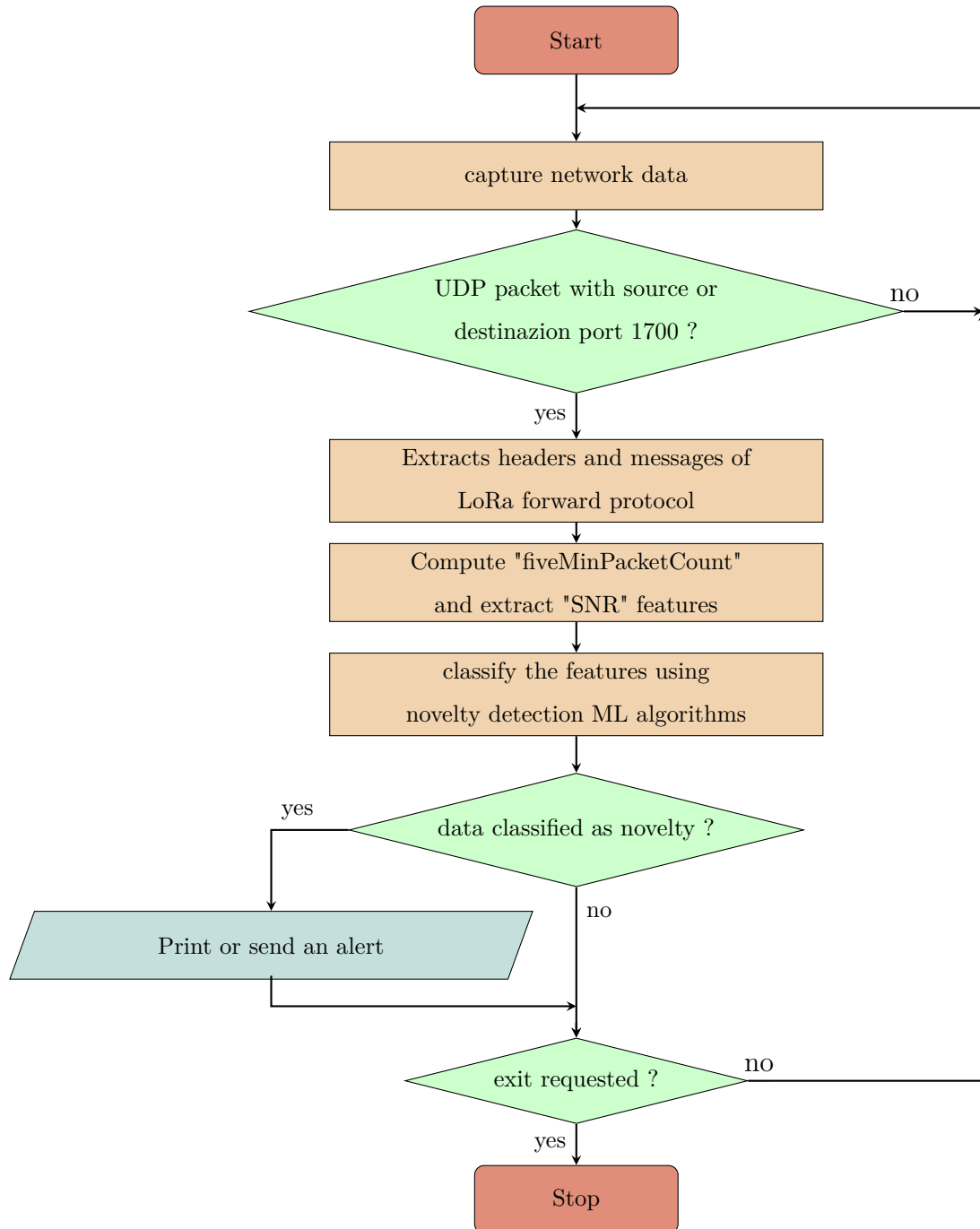
The LoRaJamIDS can be executed locally, inside the LoRa gateway, or remotely. The logic is the same, the only difference is in the network packet capture phase where the remotely executed LoRaJamIDS has the necessity to receive a copy of the LoRa forward protocol traffic. This traffic can be captured inside the gateway using an SSH channel or duplicated by any network appliance (by example switch, router or firewall) through which the communication passes.

This paragraph describes the logic of LoRaJamIDS running locally on the gateway. The flowchart is shown in figure 3.65.

The IDS begins its execution with a network packet capture cycle. This cycle is concluded at the request of the user. For each packet captured, the program analyses only those belonging to the UDP protocol with source or destination ports set to 1700 which characterise the LoRa forward protocol. Each of these communication packets is analysed and all the unencrypted fields presented in Chapter 3.4.1 are extracted. The program extracts the "lsnr" field from the "rxpk" information and updates the number of packets received in the last five minutes. These two values are the group of features A which are passed to one of the previously selected and trained novelty detection ML algorithms. If these features are classified as "novelty", the program prints an alert on the console. This action can be extended by alerting an external monitoring system. At the end of this cycle, if the user has requested to exit the program, the execution ends by releasing all resources. Otherwise the cycle resumes waiting for new network packets.

3.10 Possible industrial applications

Many companies use sensors with LoRa technologies to gather and send to the headquarter various measurements necessary for remote monitoring of the process. For example, in water or gas pipelines the pressure is monitored at different

**Figure 3.65:** Flowchart of the Lora Jam IDS

points in the network, even far away from the control room. In the management of smart reefer many physical measurements are carried out at various locations and sent back to the company headquarter. In addition to this information, it is possible to receive automatic reports relating to failures in the refrigerant system or security breaches of the container. In this context, the use of LoRa technology to communicate sensor data overcomes the problems of other wireless technologies such as Bluetooth, which are not robust enough to penetrate the insulation of the refrigerated container. During land transport of smart reefer, the LoRa gateway services can be provided by truck or railway infrastructure. Port infrastructures can guarantee the forwarding of LoRa radio packets by providing integrated gateways on the quays. During maritime transport, even on the high seas, the ship can provide a LoRa gateway service through its network with satellite Internet uplink. Each of these applications can be subject to RF jamming which, as presented in the 3.7 chapter, is easy to implement with low cost hardware.

Many players may be interested in deploying systems capable of detecting this type of threat. LoRa gateway systems manufacturers may be interested in providing a system capable of detecting and reporting this type of threat. The same service can be provided, at the request of the customer, by any data network service provider that conveys the LoRa forward protocol communications, for example Internet Service Providers (ISP) or cellular networks, as the information used to generate the features is not encrypted. The LoRaJamIDS can be installed in many locations within an IP network as show in figure 3.66. Upon authorisation of the customer, the ISP can run it within its own network wherever he wants (for example on the edge or the core network). The vendors of LoRa gateway systems can provide the onboard service by reporting any attacks also through onboard LED or acoustic signals. Finally, the LoRaJamIDS can also reside within the LoRa application server.

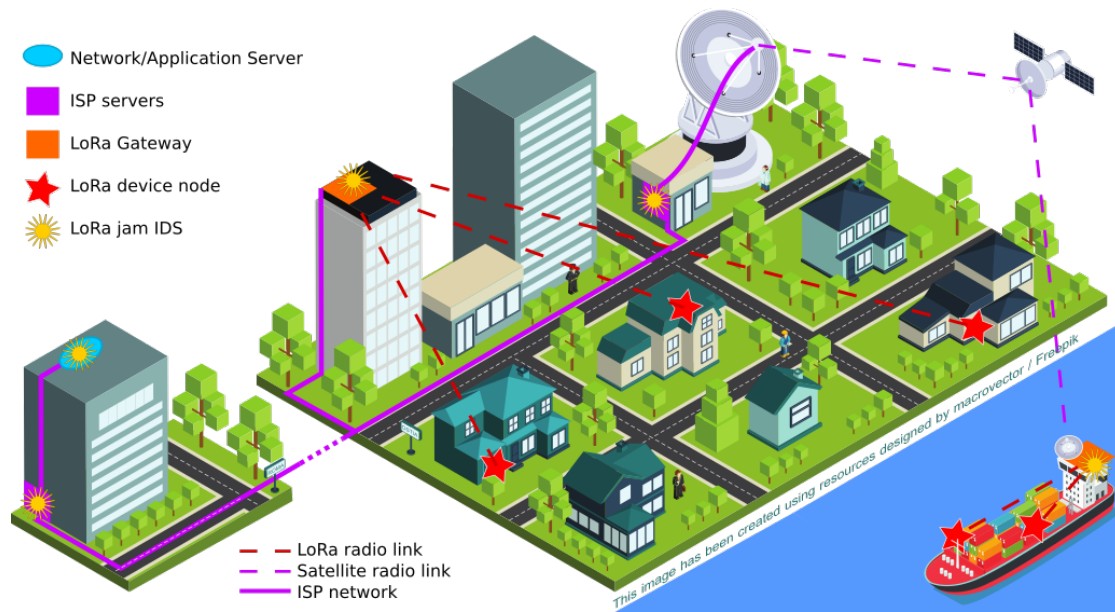


Figure 3.66: potential position of services offered for IDS jamming detection

When the LoRaJamIDS is integrated into the LoRa gateway, it must be kept in mind that any problem in the Internet connection would make impossible to receive the alerts. To avoid this possibility, it is necessary to constantly monitor the accessibility of the LoRa gateway through the Internet. Furthermore, when designing the gateway, the LoRaJamIDS requests (albeit minimal) for computing resources, memory, disk space and network communication must be taken into account. In the event that the communication takes place through network equipment owned by the same company, it is possible to forward, by using port mirroring or similar technology, a copy of the UDP packets with source or destination port equal to 1700 to an external monitoring system where the LoRaJamIDS is executed without any computational burden for the gateway. If it is not possible to intercept these packets when they pass through the data network, it is still possible to capture them inside the LoRa gateway and forward them directly to the monitoring server. However, in this case the communication would be duplicated

and the number data to be sent to the internet would increase.

When the LoRaJamIDS is run remotely and is necessary to limit the number of data sent through the Internet, the program can be easily split into two modules. A probe can reside inside the LoRa gateway taking care of the interception of the LoRa forward protocol and the extraction of the features ["fiveMinPacketCount", "SNR"]. The probe would communicate only the features (3 floating point values) to a remote analysis and alerting module capable of processing them and sending reports. Furthermore, the remote module would automatically detect any interruption in the connection between the probe and the internet. This minimises the use of particularly expensive or slow internet connections.

3.11 Related work

The authors of [24] presents the design goals and the techniques, which different LPWA technologies exploit to offer wide-area coverage to low- power devices at the expense of low data rates. They also present as future challenges the interference control and mitigation. They point out that the increase in the number of devices expected in the next years combined with the use of the simple ALOHA scheme to content the RF channel access will cause a deterioration in performance and higher interference.

An overview of the capabilities and the limitations of LoRaWAN are discussed in [25]. The authors present different use case scenario with positive and negative issues. One issue is that LoRa to cope with higher interference levels use of larger Spreading Factor and this could cause an increase in collision probability.

Different studies present the problem related to the fact that the same transmissions can unintentionally collide with each other when the number of devices present in the area is high. [26] demonstrate that the collision resolution approach

used in LoRawan is inefficient with a high number of lora end notes and leads to collision avalanches. [27] shows that in a dense LoRa multicell system the intercell interference cause non-negligible performance degradation. [28] present a theoretical expressions for both the collision and the packet loss probabilities in a LoRaWAN network.

The Wireless Communication jamming method are discussed in [29], [30] and [31]. Four jamming method are described: constant, deceptive, random and reactive jammer. The constant jammer continuously sends jamming signal with a waveform unrelated to the modulation used by the legitimate traffic. The deceptive jammer continuously sends legitimate packets. This can be seen as a wireless media congestion by the media channel access logic controllers. The random jammer activate it jamming signal at random time. It stays On for a user selectable period of time, then it switches Off and stays silent for another user selectable period of time. The reactive jammer continuously monitor the radio channel and switch on the jammer signal only when a legitimate traffic is detected. The authors of [32] describe the LoRa network stack and present different types of attacks using COTS hardware, including an selective jamming attack. The selective jammer continuously monitor the radio channels and switch on the jammer signal only when a legitimate traffic sent by a specific devices (identified by its EUI) is detected. In [33] several possible threads agaist LoRaWAN are analyzed, including RF jamming where the authors call attention to the difficulty of identifying a selective jamming.

The authors of [24] employ a FM signal with 200 kHz bandwidth to interfere with one specific LoRa channel; A similar jamming attack on LoRa Infrastructure is proposed in [34] but use a valid LoRa Radio Packet as jamming signal. A measure of packet loss under white gaussian noise interference signal was discussed in [35]. The study highlight the robustness of the LoRa communication proto-

col versus different level of White Gaussian depending on is the values of LoRa parameters Bandwidth, Spreading Factor, transmission power and coding rate.

A detailed list of current solutions and open issues of the LoRa and LoRa network is presented in [36].

The author of [37] propose an IDS for the detection of jamming attacks in a LoRaWAN network based on the analysis of some property of the Join procedure affected by the jamming signal. To use this IDS we must have a LoRa end node always performing a join procedure (every 30 seconds) to be able to detect the presence of a jammer. They also present the realisation of a reactive jammer by using the Channel Activity Detection (CAD) mode of LoRa RF transceiver SX1271. The CAD is used for preamble detection of LoRa preamble. The detection of a valid preamble is used to switch On the jamming signal to interfere with the data transmission in LoRaWAN. In [38] the same approach is used to detect the presence of a Wireless Power Transfer (WPT) near the LoRa end node. This can be done because the WPT signal causes the same effect of a jamming on the RSSI. The RSSI is used to generate the random number used as random number (Nonce) in the Join procedure. The detection used by these two papers [37] and [38] identifies the presence of a jammer or WPT monitoring the randomness of the random number used as Nonce in the join procedure. An experimental evaluation of the jamming threat in LoRaWAN is presented [39]. The writers use normal LoRa Commercial off-the-shelf (COTS) as jammer device. In [40] is present a deep analysis of LoRaWAN physical layer jamming using chirp signals. The authors also present a countermeasure able to separate jamming chirp from legitimate chirp signals and correctly decode the packet.

3.12 Conclusion

In this chapter, the idea presented in Chapter 1 has been extended to LoRaWAN systems. I put under analysis the LoRaWAN network because the packet is encrypted and deep packet analysis is impossible to do without knowing the secret encryption keys, so an approach based on statistical features is the only feasible way to build an IDS.

The idea born from the analysis of SDN-SF-IDS latency done at the end of the first chapter can be applied to characterise some properties of the radio channels used to transmit LoRa packets or features intrinsic to the transmission sequences logic used by an IoT application. These property can be used to characterise the normal operation state of a LoRa network trying to detect anomalies related to specific type of malicious attacks.

In my experimental setup, I realised a scenario of temperature monitoring for smart agriculture or something similar use cases. The IoT application logic expects to receive the environmental temperature from the LoRa end node every 30 seconds. When the radio channel is in a normal state, the number of packets sent and received in 5 minutes is constant. This packet rate or the packet inter-arrival time can change in case of LoRa end node hardware problems (battery exhausted, hardware fault) or problem on the radio channel (signal collisions, jamming attack, etc). Also features strictly related on the quality of the radio communication channel (RSSI or SNR) can be useful to detect problems related to the radio media. These two features have been used together to realise the LoRa IDS able to detect a radio jamming. This is detected as an anomaly (novel data) by a correctly trained LOF algorithm.

Differences can arise also when a new device enter the system or when an already seen device changes its throughput. The quality of the training dataset is crucial to minimise the false anomalies. Otherwise, the presence of a valid traffic

not present on the training dataset can be detected as outlier.

The reactive jammer needs to process received signals and react in real-time as soon as possible to be able to disrupt also some parts of the LoRa preamble. This greatly increases the probability of completely jam the transmission because the gateway is not able to detect a valid preamble anymore. To be able to detect and start to jam as soon as possible, the reaction delay of the jammer prototype must be kept as small as possible. Otherwise, the time passed between the signal detection and the start of the jamming can be longer than the LoRa signal preamble. The jammer still working thanks to a strong corruption of the physical header and payload symbols. The CRC used in LoRa is not able to correct the decoding errors when the number of corrupted symbols is high. These reaction time is a sum of latencies generated by different causes. It depends from the size of the SDR acquisition buffer and sampling rate, the time needed to transfer the samples on USB bus, the DSP operations used to detect the presence of a signal and to create the jamming signal, the copy of sample in computer memory between different GNU Radio blocks, the kernel scheduling and so on. The jamming prototype implemented on a personal computer with Intel i5 CPU and 16GB of RAM is able to reach good jamming performance but the CPU load is high.

3.13 Future work

During this research, I collected both normal traffic and traffic under attack and uses novelty detection ML algorithms to detect abnormal situations.

The feature analysis can be extended and investigated in different directions. Comparing the LoRa forward protocol RTT with the delay of other network connections, it would be possible to discriminate between network congestion, local system overloads or other possibility. Sudden changes of datarate can be related

to an unauthorised upgraded of the firmware.

The presence of a deceptive jammer¹⁵ can be detect through an increase of the received LoRa end node messages.

The presence of a reactive jammer¹⁶ or shot noise-based intelligent jammer¹⁷ can be detected through an increase of the Physical CRC errors. The presence of one constant jammer can be detected by a SDR analysing the Power Spectrum of the LoRa frequencies.

This jamming algorithm can be implemented inside the SDR card giving unexpected reaction speed, but the required skill can be really high.

¹⁵constantly transmitting illegitimate packets in order to keep the channel busy

¹⁶acting only when a LoRa packet is transmitted

¹⁷transmits repeated impulses to corrupt the CRC used by packets, so that they are discarded by the Gateway

Chapter 4

Conclusions

Time-based properties related to repetitive and stable operation done by a digital system can be used to detect anomalies. Any difference on expected values of these time-based property can be due to changes in the software, to hardware problems or to malicious attacks.

In the first chapter, I measured the ΔT_{TOT} latency added by the SDN-SF-IDS to the analyzed network packet. These timings can be seen under another point of view. It is possible to analyze the SDN-SF-IDS latency to separate the network packet in two groups: packets with delay under 20 ms and packets with delays over 20 ms.

The majority of packets takes a short time to pass through the SDN-SF-IDS system, falling into the first group. These packets consist of all network packets except the first packet of each new TCP or UDP flow detected by the SDN-SF-IDS.

All first packets of the new TCP or UDP flows fall in the second group, because the number of operations performed before forwarding these packets is so greater than the operations normally performed by the SDN-SF-IDS system, that it is not possible for the two groups to overlap.

The gap between these two groups is higher than the fluctuations of ΔT_{TOT}

delays related to normal OS operation.

An overload of the OS during the forwarding of a packet of the first group can add enough delay to enter in the second group. Thanks to the current high computational resources of personal computer, these OS overloads are rare and can be ignored.

In the second chapter, this idea has been extended to LoRaWAN systems. I analysed a LoRaWAN network, one of the most used LPWAN solutions. My interest is mainly due to the fact that the packet is encrypted and deep packet analysis is impossible to do without knowing the secret encryption keys. An approach based on statistical features is the only feasible way to build an IDS. The idea born from the analysis of SDN-SF-IDS latency done at the end of the first chapter can be applied to characterize some properties of the radio channels used to transmit LoRa packet or property intrinsic to the transmission sequences logic used by an IoT application. These property can be used to characterize the normal operation state of a LoRa network trying to detect anomalies related to specific type of malicious attacks.

In my experimental setup, I realized a scenario of temperature monitoring. The employed LoRa end nodes send environmental temperature every 30 seconds. When the radio channel is in a normal state, the number of packets sent and received in 5 minutes is constant. This packet rate or the packet inter-arrival time can change in case of LoRa end node hardware problems (battery exhausted, hardware fault) or problem on the radio channel (signal collisions, jamming attack, etc). Also feature strictly related on the quality of the radio communication channel (RSSI or SNR) can be usefull to detect problems related to the radio media. These two features have been used together to realize the presented LoRa IDS able to detect a radio jamming. Radio jamming is detected as an anomaly (novel data) by a correctly trained LOF algorithm.

Differences can arise also when a new device enters the system or when an already seen device changes its throughput. The quality of the training dataset is crucial to minimize the false anomalies. Otherwise, the presence of a valid traffic not present on the training dataset can be detected as outlier.

4.1 Future Work

This idea of characterizing the normal state of a digital system can be extended to any type of system with known or unknown internal logic. Features based on packet rate or delay can be used to detect reactive jamming and signal collision. An unauthorized change in a firmware of an industrial system can lead to changes in the behaviour of the system itself.

For example, when the use of the system CPU unexpectedly increases, it can lead to higher temperature in the system case, and a different number of operations performed within a firmware procedure can change the execution times or the duration of the system calls. These properties can be measured passively or actively also for systems and protocols based on proprietary solutions. All these parameters can be monitored through novelty detection or anomaly detection algorithms in order to detect any unexpected change as an anomaly.

Bibliography

- [1] F. Bigotto, L. Boero, M. Marchese, and S. Zappatore, “Statistical fingerprint-based ids in sdn architecture,” in *SummerSim-SPECTS - Society for Modeling & Simulation International (SCS)*, Bordeaux, FR, France, Jul. 2018.
- [2] L. Boero, M. Cello, M. Marchese, E. Mariconti, T. Naqash, and S. Zappatore, “Statistical fingerprint-based intrusion detection system (sf-ids),” *International Journal of Communication Systems*, vol. 30, no. 10, 2016.
- [3] S. Seeber, L. Stiemert, and G. D. Rodosek, “Towards an sdn-enabled ids environment,” in *2015 IEEE Conference on Communications and Network Security (CNS)*, 2015, pp. 751–752.
- [4] S. Seeber and G. D. Rodosek, “Towards an adaptive and effective ids using openflow,” in *AIMS*, 2015.
- [5] P. Mishra, V. Varadharajan, U. Tupakula, and E. S. Pilli, “A detailed investigation and analysis of using machine learning techniques for intrusion detection,” *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 686–728, Firstquarter 2019.
- [6] L. N. Tidjon, M. Frappier, and A. Mammar, “Intrusion detection systems: A cross-domain overview,” *IEEE Communications Surveys Tutorials*, pp. 1–1, 2019.

- [7] Y. Chi, T. Jiang, X. Li, and C. Gao, "Design and implementation of cloud platform intrusion prevention system based on sdn," in *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*, 2017, pp. 847–852.
- [8] A. M. Quingueni and N. Kitsuwat, "Reduction of traffic between switches and ids for prevention of dos attack in sdn," in *2019 19th International Symposium on Communications and Information Technologies (ISCIT)*, 2019, pp. 277–281.
- [9] H. Hendrawan, P. Sukarno, and M. A. Nugroho, "Quality of service (qos) comparison analysis of snort ids and bro ids application in software define network (sdn) architecture," in *2019 7th International Conference on Information and Communication Technology (ICoICT)*, 2019, pp. 1–7.
- [10] N. Sultana, N. K. Chilamkurti, W. Peng, and R. Alhadad, "Survey on sdn based network intrusion detection system using machine learning approaches," *Peer-to-Peer Networking and Applications*, vol. 12, pp. 493–501, 2019.
- [11] D. P and M. S. K, "Comparative study on ids using machine learning approaches for software defined networks," *International Journal of Intelligent Enterprise*, 2019.
- [12] Y. Li, X. Guo, X. Pang, B. Peng, X. Li, and P. Zhang, "Performance analysis of floodlight and ryu sdn controllers under mininet simulator," *2020 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*, pp. 85–90, 2020.
- [13] M. Paliwal, D. Shrimankar, and O. Tembhurne, "Controllers in sdn: A review report," *IEEE Access*, vol. 6, pp. 36 256–36 270, 2018.
- [14] J. Xie, D. Guo, X. Li, Y. Shen, and X. Jiang, "Cutting long-tail latency of

- routing response in software defined networks,” *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 384–396, March 2018.
- [15] M. A. Albahar, “Recurrent neural network model based on a new regularization technique for real-time intrusion detection in sdn environments,” *Security and Communication Networks*, vol. 2019, p. 8939041, Nov 2019. [Online]. Available: <https://doi.org/10.1155/2019/8939041>
- [16] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, “Deep recurrent neural network for intrusion detection in sdn-based networks,” in *2018 4th IEEE Conference on Network Softwarization and Workshops (Net-Soft)*, 2018, pp. 202–206.
- [17] D.-M. Ngo, C. Pham-Quoc, and T. N. Thanh, “Heterogeneous hardware-based network intrusion detection system with multiple approaches for sdn,” *Mobile Networks and Applications*, vol. 25, pp. 1178–1192, 2020.
- [18] C. Hong, K. Lee, J. Hwang, H. Park, and C. Yoo, “Kafe: Can os kernels forward packets fast enough for software routers?” *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2734–2747, Dec 2018.
- [19] S. Sharma, “Substation communication with iec 61850 and application examples,” Dec. 2016, page 18. [Online]. Available: [https://web.archive.org/web/20180218233431/http://www04.abb.com/global/seitp/seitp202.nsf/0/4d1c836b9e7fdb67c12580870047d7c8/\\$file/1.Chile_+ABB+_Substatio+c+ommunication+with+IEC+61850+and+application+examples.pdf](https://web.archive.org/web/20180218233431/http://www04.abb.com/global/seitp/seitp202.nsf/0/4d1c836b9e7fdb67c12580870047d7c8/$file/1.Chile_+ABB+_Substatio+c+ommunication+with+IEC+61850+and+application+examples.pdf)
- [20] V. Sushil Joshi, ABB Ltd, “Utilization of goose in mv substation,” in *16th national power systems conference*, Hyderabad, IN, India, Dec. 2010, table II. [Online]. Available: <http://www.iitk.ac.in/npsc/Papers/NPSC2010/6114.pdf>

- [21] S. Chelluri, “Iec 61850 ... the electrical scada standard and integration with ddcemis,” 2015, page 37. [Online]. Available: <https://nebula.wsimg.com/a49e00efad15d7b63f58b0ff8bd94956?AccessKeyId=1C24E49FE84FF4D32384&disposition=0&alloworigin=1>
- [22] B. Schölkopf, J. Platt, J. Shawe-Taylor, A. Smola, and R. Williamson, “Estimating support of a high-dimensional distribution,” *Neural Computation*, vol. 13, pp. 1443–1471, 07 2001.
- [23] F. T. Liu, K. Ting, and Z.-H. Zhou, “Isolation forest,” in *Isolation Forest*, 01 2009, pp. 413 – 422.
- [24] L. E. Marquez, A. Osorio, M. Calle, J. C. Velez, A. Serrano, and J. E. Candelo-Becerra, “On the Use of LoRaWAN in Smart Cities: A Study With Blocking Interference,” *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 2806–2815, 2020.
- [25] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, “Understanding the limits of lorawan,” *IEEE Communications Magazine*, vol. 55, no. 9, pp. 34–40, 2017.
- [26] D. Bankov, E. Khorov, and A. Lyakhov, “On the limits of lorawan channel access,” in *2016 International Conference on Engineering and Telecommunication (EnT)*, 2016, pp. 10–14.
- [27] L. Beltramelli, A. Mahmood, M. Gidlund, P. Österberg, and U. Jennehag, “Interference modelling in a multi-cell lora system,” in *2018 14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2018, pp. 1–8.
- [28] G. Ferre, “Collision and packet loss analysis in a lorawan network,” in *2017*

- 25th European Signal Processing Conference (EUSIPCO)*, 2017, pp. 2586–2590.
- [29] W. Xu, W. Trappe, Y. Zhang, and T. Wood, “The feasibility of launching and detecting jamming attacks in wireless networks,” in *The Feasibility of Launching and Detecting Jamming Attacks in Wireless Networks*, 05 2005.
- [30] A. Mpitziopoulos, D. Gavalas, C. Konstantopoulos, and G. Pantziou, “A survey on jamming attacks and countermeasures in wsns,” *Communications Surveys & Tutorials, IEEE*, vol. 11, pp. 42 – 56, 01 2009.
- [31] K. Pelechrinis, M. Iliofotou, and S. Krishnamurthy, “Denial of service attacks in wireless networks: The case of jammers,” *Communications Surveys & Tutorials, IEEE*, vol. 13, pp. 245 – 257, 06 2011.
- [32] E. Aras, G. S. Ramachandran, P. Lawrence, and D. Hughes, “Exploring the security vulnerabilities of lora,” in *2017 3rd IEEE International Conference on Cybernetics (CYBCONF)*, 2017, pp. 1–6.
- [33] I. Butun, N. S. A. Pereira, and M. Gidlund, “Analysis of lorawan v1.1 security: research paper,” *Proceedings of the 4th ACM MobiHoc Workshop on Experiences with the Design and Implementation of Smart Objects*, 2018.
- [34] P. V. Wadtkar, B. S. Chaudhari, and M. Zennaro, “Impact of interference on lorawan link performance,” in *2019 5th International Conference On Computing, Communication, Control And Automation (ICCUBEA)*, 2019, pp. 1–5.
- [35] L. Angrisani, P. Arpaia, F. Bonavolontà, M. Conti, and A. Liccardo, “Lora protocol performance assessment in critical noise conditions,” *2017 IEEE 3rd International Forum on Research and Technologies for Society and Industry (RTSI)*, pp. 1–5, 2017.

- [36] J. P. Shanmuga Sundaram, W. Du, and Z. Zhao, "A survey on lora networking: Research problems, current solutions, and open issues," *IEEE Communications Surveys Tutorials*, vol. 22, no. 1, pp. 371–388, 2020.
- [37] S. M. Danish, A. Nasir, H. K. Qureshi, A. B. Ashfaq, S. Mumtaz, and J. Rodriguez, "Network intrusion detection system for jamming attack in lorawan join procedure," in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–6.
- [38] S. M. Danish, H. K. Qureshi, and S. Jangsher, "Jamming attack analysis of wireless power transfer on lorawan join procedure," in *2018 IEEE Globecom Workshops (GC Wkshps)*, 2018, pp. 1–6.
- [39] C.-Y. Huang, C.-W. Lin, R.-G. Cheng, S. J. Yang, and S.-T. Sheu, "Experimental evaluation of jamming threat in lorawan," in *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*, 2019, pp. 1–6.
- [40] N. Hou, X. Xia, and Y. Zheng, "Jamming of lora phy and countermeasure," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.