A Security Analysis of Garage Door Remotes and the Danger of DOR Attacks

Sébastien Dudek

Introduction

Houses have physical access systems to keep intruders and home invaders out. These systems can take on many forms: radio frequency identification (RFID) systems, old-school lock and key, or even a receptionist to fend off social engineering tactics in a residential building. Needless to say, residents must be confident in the protection offered by these barriers to feel safe inside their homes.

In many homes, aside from the main door, the garage door serves as an initial barrier. However, an intruder might see the garage door as a discreet option for breaking into a house. Once past that, they can figure out several ways to go break in further, such as accessing an elevator in the building or even abusing emergency boxes to disable the locks of doors inside the house. But first they must enter the garage without attracting suspicion. One way they can do this is by using a remote to open garage doors.

Well-known experiments on automatic garage door security have been done before.[1] However, an opportunity to revisit this attack scenario presented itself in the form of a broken garage door remote that needed to be replaced — in this case, that of the researcher. This circumstance gave us the idea of testing two attack scenarios that bypass the rolling code protection implemented by most automatic garage door mechanisms.

We began the test by capturing and analyzing the signals sent by each remote button push using Software-Defined Radio (SDR). During the analysis, we show a hidden feature shared by several brands of remotes that is used in many installations. This hidden feature is essential in our first attack scenario where we see how it can be used to register a new remote to the receiver and get permanent access to the garage. We also introduce the rolling code mechanism used by the remotes and the known ways to bypass that protection. The last part of this article introduces PandwaRF, a tool that shows that it is possible to conduct such attacks in a very discreet and stealthy manner.

The Remotes

The remotes used for this article are S449-QZ2 remotes, which cost approximately €10 each (around US$12 as of this writing), and receivers that support these remotes, like the Cardin RMQ449200, which cost approximately €50 (around US$58 as of this writing), as seen in Figure 1. We chose Cardin because it is one of many systems that we use in one scenario, which we introduce in the next section.

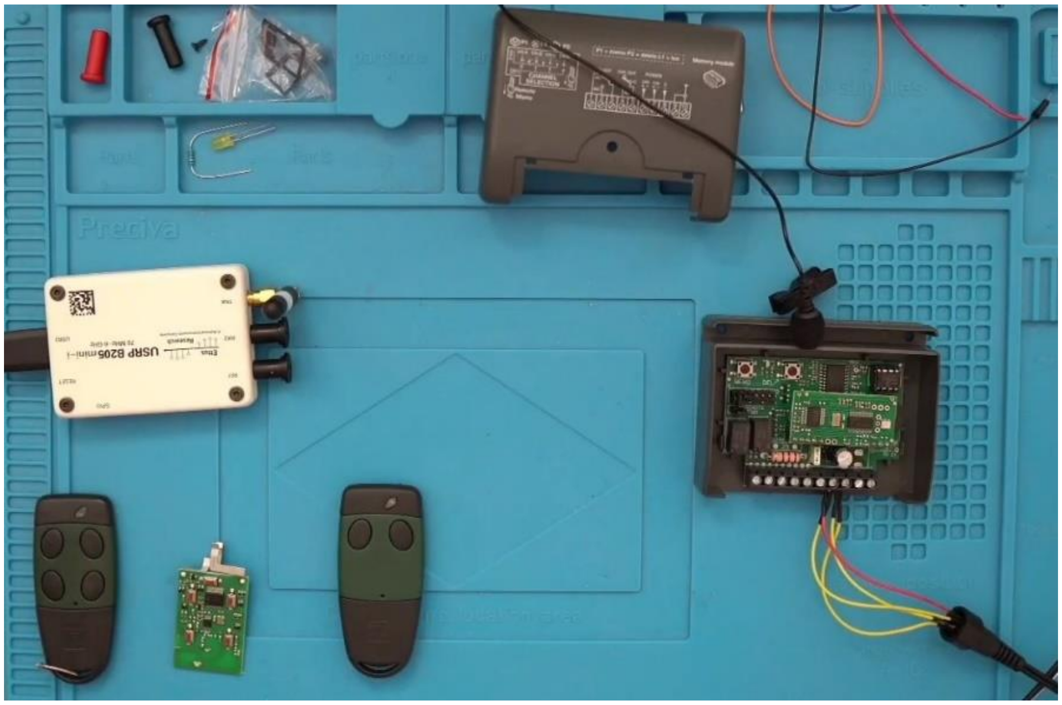


Figure 1. The setup composed of two garage Cardin S449-QZ2 test remotes and a Cardin RMQ449200 test receiver that will be analyzed against a USRP B205mini-i[2]

The two buttons serve as the authenticated remote, and the four-button remote serves as the remote we will introduce to the receiver later on. To begin, we need to record the first remote into the receiver's electrically erasable programmable read-only memory (EEPROM) using its manual "MEM" button and follow the step described in the provided manual.

For an attacker, this will be a straightforward step if the receiver is exposed outside without any physical security mechanism. Indeed, in such a scenario, it would be easy for an intruder to record his own remote and get in. But thankfully, receivers are typically kept in a metal box inside the locked garage, so potential intruders would need to first get inside by slipping in without being seen while a vehicle passes through — or they could find some other way of getting to the receiver without raising suspicion. It is only after doing so that they can eventually record a new remote to get permanent access or find a stealthy way of getting out of the premise without having to damage the mechanical opening.

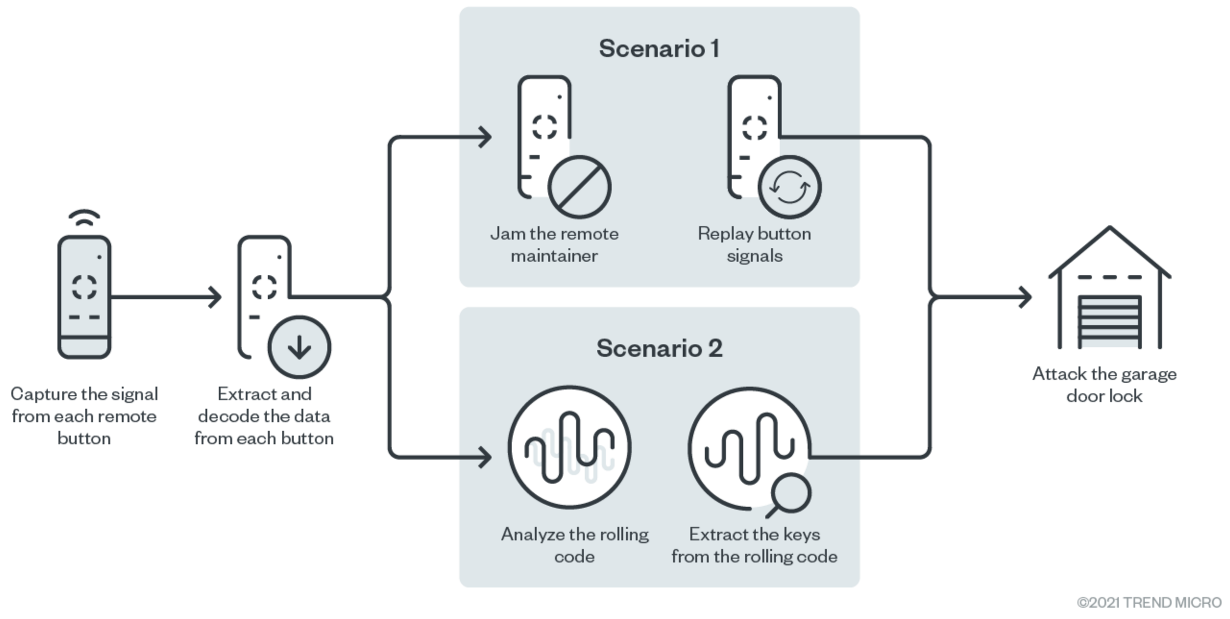

Scenario 1
Jam the remote maintainer
Replay button signals
Scenario 2
Analyze the rolling code
Extract the keys from the rolling code
Capture the signal from each remote button
Extract and decode the data from each button
Attack the garage door lock
©2021 TREND MICRO

Figure 2. The attack chain summarizing the two scenarios in this analysis

In this article, we study the signal sent by the remote and learn its possible uses. But first, let us introduce a feature that is present in many systems, called the DOR procedure.

The DOR Procedure

The meaning behind the acronym "DOR," which is used to name this feature, is not stated explicitly in the manual. However, after reaching out to the creator of PandwaRF, Djamil Elaidi, for our research, we now know that this acronym refers to "direct on receiver." The Cardin RMQ449200 manual (Figure 3) reveals how this is a procedure on how to remotely record a remote using a hidden" button. This hidden button is seen more clearly after disassembling the remote, as shown in Figure 4.

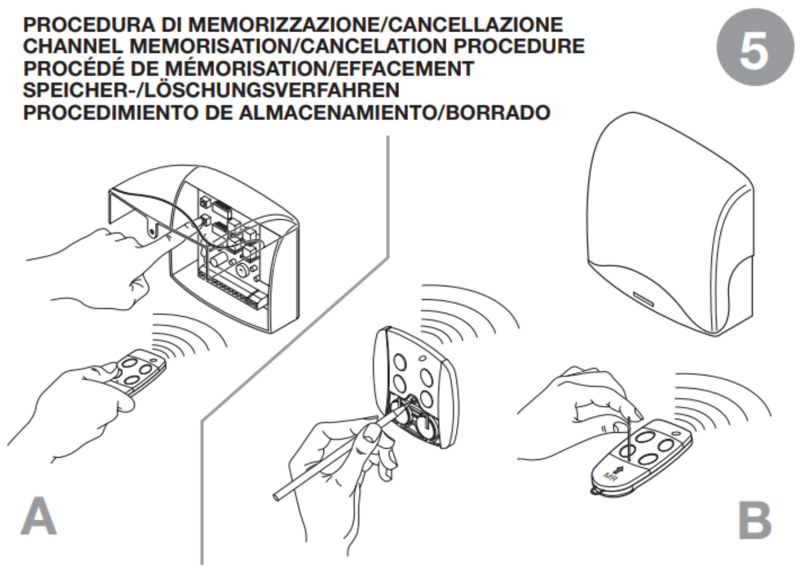

PROCEDURA DI MEMORIZZAZIONE/CANCELLAZIONE
CHANNEL MEMORISATION/CANCELATION PROCEDURE
PROCÉDÉ DE MÉMORISATION/EFFACEMENT
SPEICHER-/LÖSCHUNGSVERFAHREN
PROCEDIMIENTO DE ALMACENAMIENTO/BORRADO
5
A
B

Figure 3. Instructions for the DOR procedure with a hidden button[3]

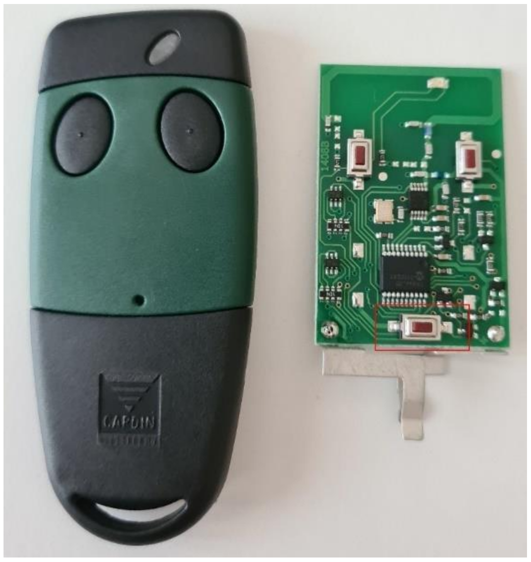


Figure 4. Image of the disassembled remote and the DOR's "hidden" button (in the red box)

The DOR procedure described in the manual is a fast and easy way to record a new remote with the help of an authentic one. If the DOR button is activated by a button push, the receiver should play a sound to notify users of this state. It is important to reiterate that the DOR functionality is relatively common and can be found in devices made by other manufacturers other than Cardin.

Making Captures

Now that we have introduced the remotes and the DOR feature, we can move on to capturing the signal. To do so, we first identify the frequency range used by the devices. This information can be collected through a little investigating: by looking at the Radio Frequency (RF) chip used by the remote, which appears to be a TDA5150 (Figure 5) that is used for 300 to 928 MHz bands. Looking at the printed circuit board (PCB) itself narrows the range further, as it shows either 433 MHz or 868 MHz (Figure 6).

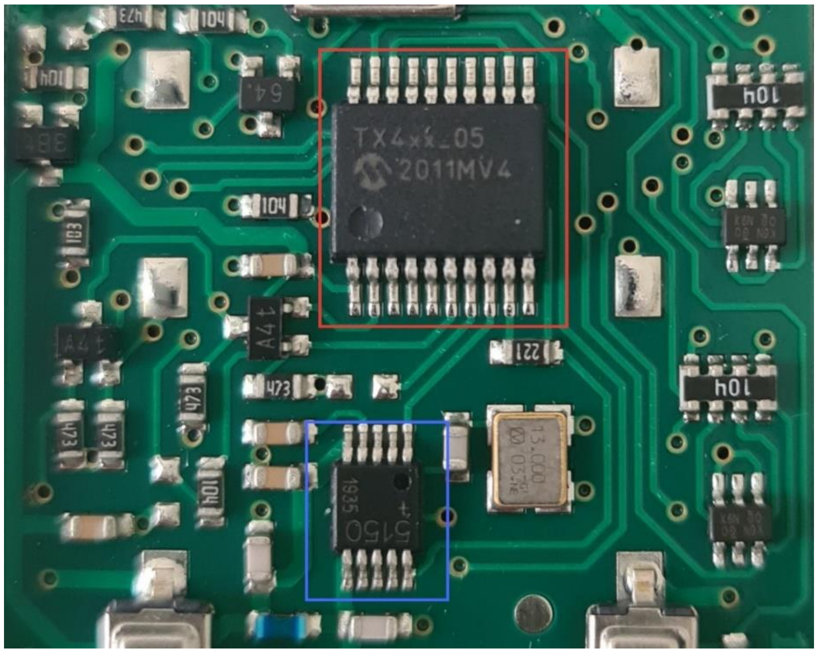


Figure 5. The PCB of the remote with a Microchip microcontroller (in the red box) and a TDA5150 transceiver (in the blue box)

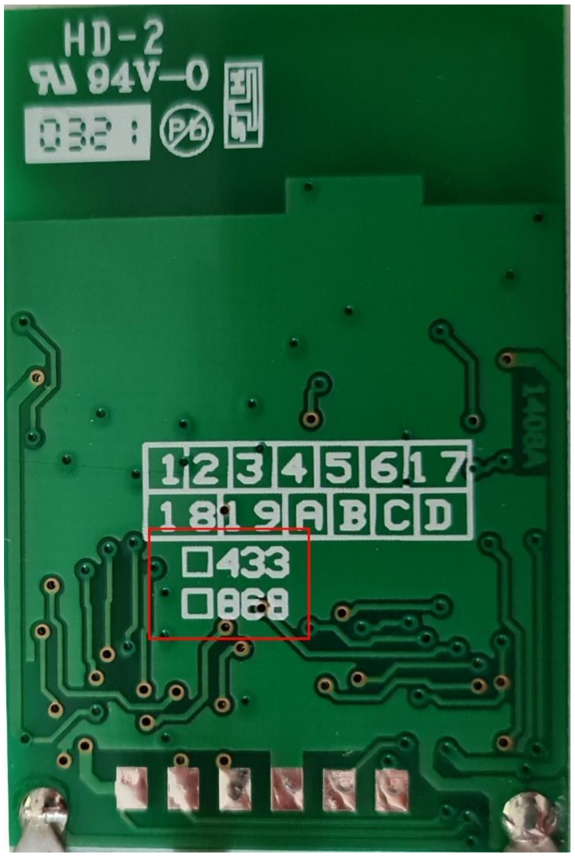


Figure 6. The back of the PCB with information about the possible bands

These central frequencies correspond to European industrial, scientific, and medical (ISM) bands, which are interesting to monitor, particularly in internet of things (IoT) devices.

Aside from these two indicators, sometimes the vendor can also directly just give the exact frequency of their product, as was the case for our test remote.

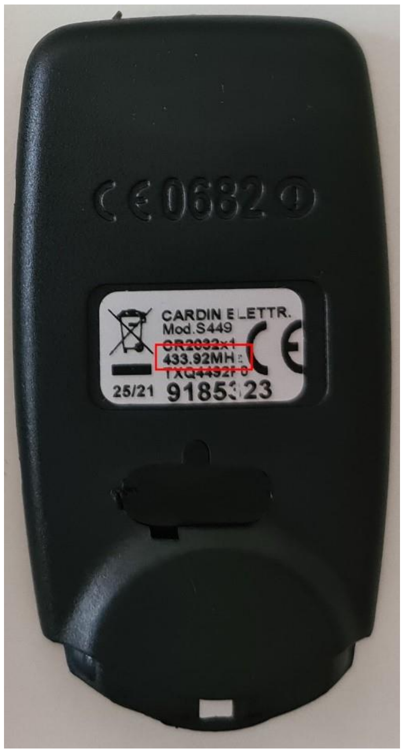


Figure 7. Our test remote's ID and the precise frequency it uses

With this information, we can now use a custom Software-Defined Radio (SDR) frequency analyzer with a simple flowgraph that will display a fast Fourier transform (FFT). There, we observed two spikes relative to a 2-FSK/BFSK by looking at the space and mark level, as seen in Figure 8.

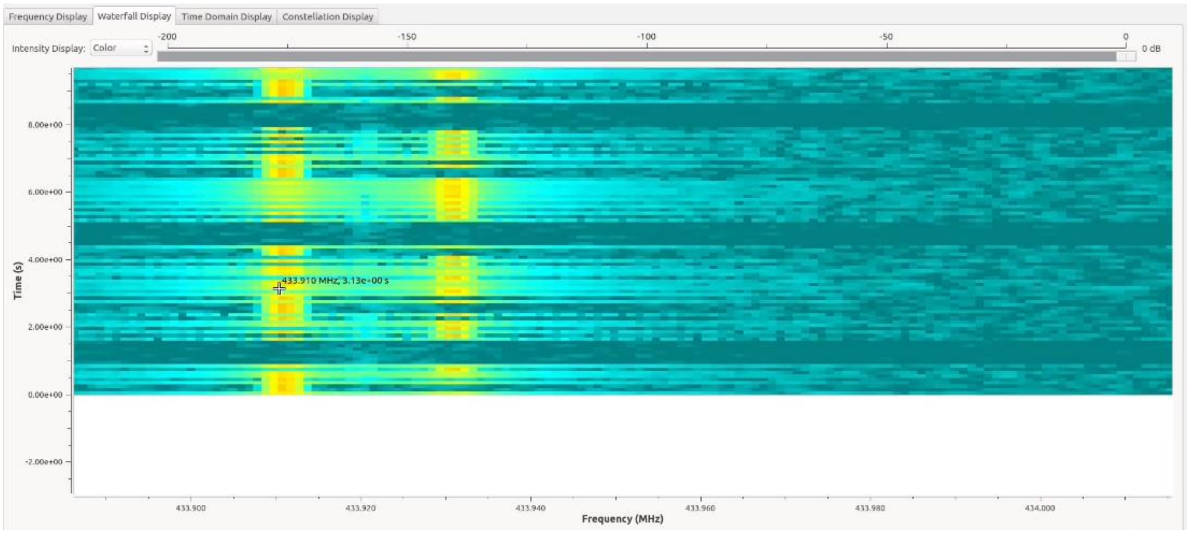


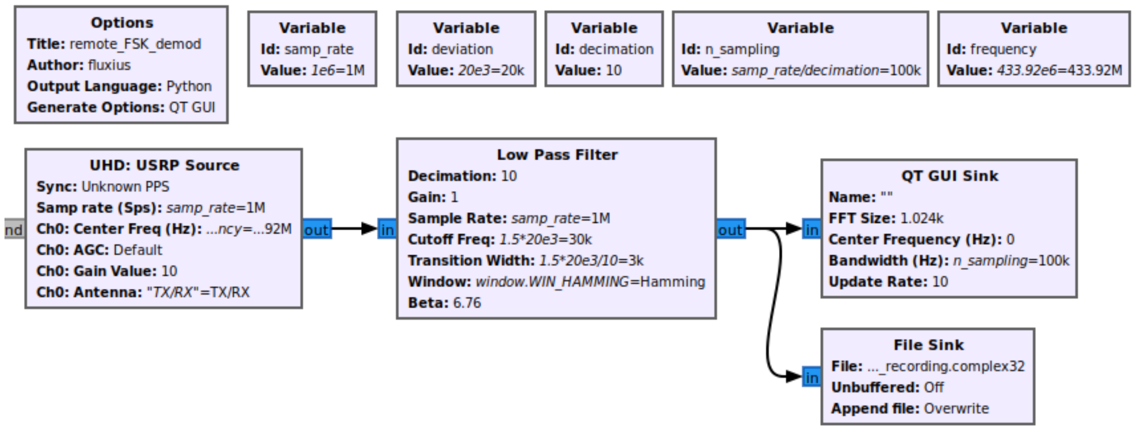
Figure 8. Waterfall showing the mark and the space of the 2-FSK transmission

This signal is also a deviation of 20 kHz, a piece of essential information that one needs to keep in mind when demodulating this signal. We can then record this signal using a complex file sink since we will use this for further analysis without having to push any button.

Note that we can try using a lower sampling rate to better match the bandwidth of our signal. However, SDR is limited to a minimum and a maximum sampling rate depending on the analog-to-digital converter (ADC) chip and the baseband (BB) antialiasing filters (if supported) to perform decimation.

For our case, using a Universal Software Radio Peripheral (USRP), we can begin with a sampling rate of 1 MSps, which works fine even with most SDR devices, and then decimate within a software filter. For optimization, we can try lowering the sampling rate to see if we will still get the same results.

As a result of using a filter and optionally decimating it by 10 to spare us the effort of extra computing, the GNU radio companion (GRC) flowgraph to capture our signal will therefore look as pictured in Figure 9.



Options
Title: remote_FSK_demod
Author: fluxius
Output Language: Python
Generate Options: QT GUI

Variable
Id: samp_rate
Value: 1e6=1M

Variable
Id: deviation
Value: 20e3=20k

Variable
Id: decimation
Value: 10

Variable
Id: n_sampling
Value: samp_rate/decimation=100k

Variable
Id: frequency
Value: 433.92e6=433.92M

UHD: USRP Source
Sync: Unknown PPS
Samp rate (Sps): samp_rate=1M
Ch0: Center Freq (Hz): ...ncy=...92M
Ch0: AGC: Default
Ch0: Gain Value: 10
Ch0: Antenna: "TX/RX"=TX/RX

Low Pass Filter
Decimation: 10
Gain: 1
Sample Rate: samp_rate=1M
Cutoff Freq: 1.5*20e3=30k
Transition Width: 1.5*20e3/10=3k
Window: window.WIN_HAMMING=Hamming
Beta: 6.76

QT GUI Sink
Name: ""
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): n_sampling=100k
Update Rate: 10

File Sink
File: ..._recording.complex32
Unbuffered: Off
Append file: Overwrite

Figure 9. GRC flowgraph to capture button presses in a complex file with decimation

Demodulating the Signal

Before demodulating the signal, let us look at the FFT to see how we can clean the signal with a squelch to remove some noise.

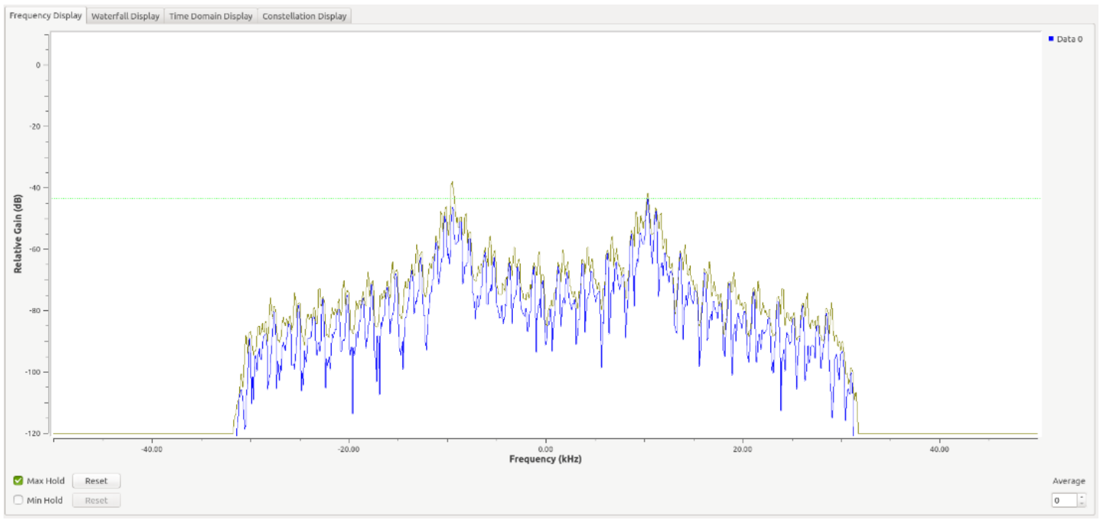
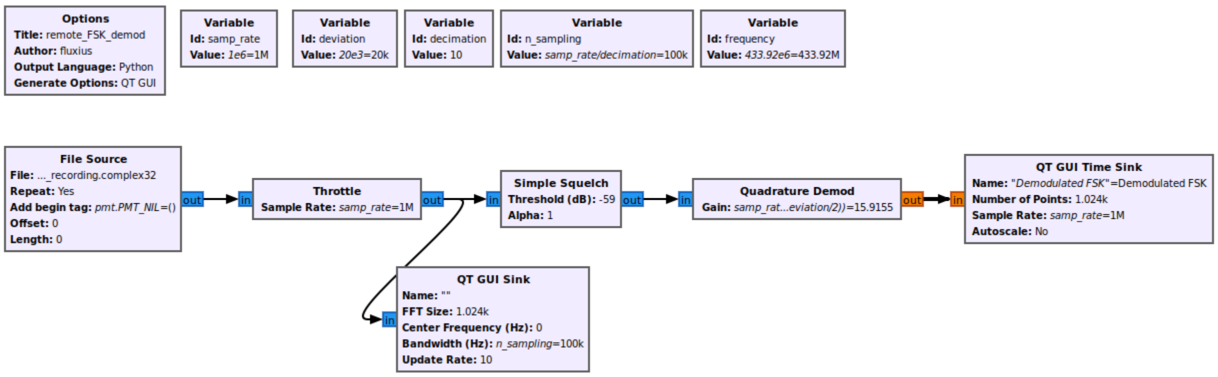


Figure 10. FFT of the captured signal

It should be noted that we can refine the low-pass IQ filter further to only let the desired band pass. We can then proceed to demodulating this signal with an FM or a quadrature demodulation block by defining a sensibility relative to the deviation as seen in the following equation:

$$Gain = \frac{1e^{6}}{2 \times \pi \times (\frac{20\text{kHz}}{2})}$$



Options
Title: remote_FSK_demod
Author: fluxius
Output Language: Python
Generate Options: QT GUI

Variable
Id: samp_rate
Value: *1e6*=1M

Variable
Id: deviation
Value: *20e3*=20k

Variable
Id: decimation
Value: 10

Variable
Id: n_sampling
Value: *samp_rate/decimation*=100k

Variable
Id: frequency
Value: *433.92e6*=433.92M

File Source
File: ..._recording.complex32
Repeat: Yes
Add begin tag: *pmt.PMT_NIL*=()
Offset: 0
Length: 0

Throttle
Sample Rate: *samp_rate*=1M

Simple Squelch
Threshold (dB): -59
Alpha: 1

Quadrature Demod
Gain: *samp_rat...eviation/2))*=15.9155

QT GUI Time Sink
Name: *"Demodulated FSK"*=Demodulated FSK
Number of Points: 1.024k
Sample Rate: *samp_rate*=1M
Autoscale: No

QT GUI Sink
Name: ""
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): *n_sampling*=100k
Update Rate: 10

Figure 11. The flowgraph to demodulate the captured signal

The result of the demodulation gives us some interesting data.

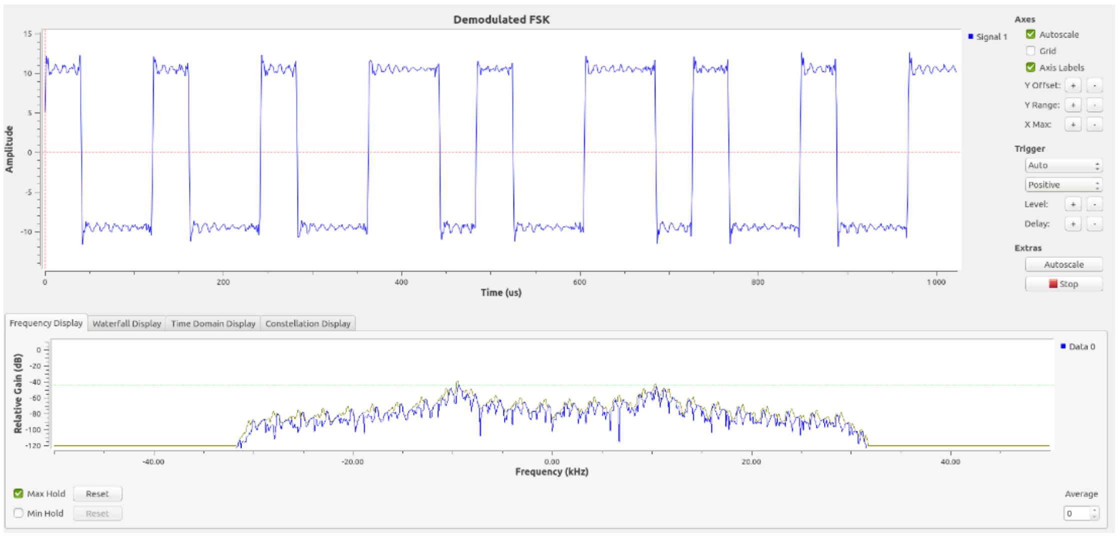
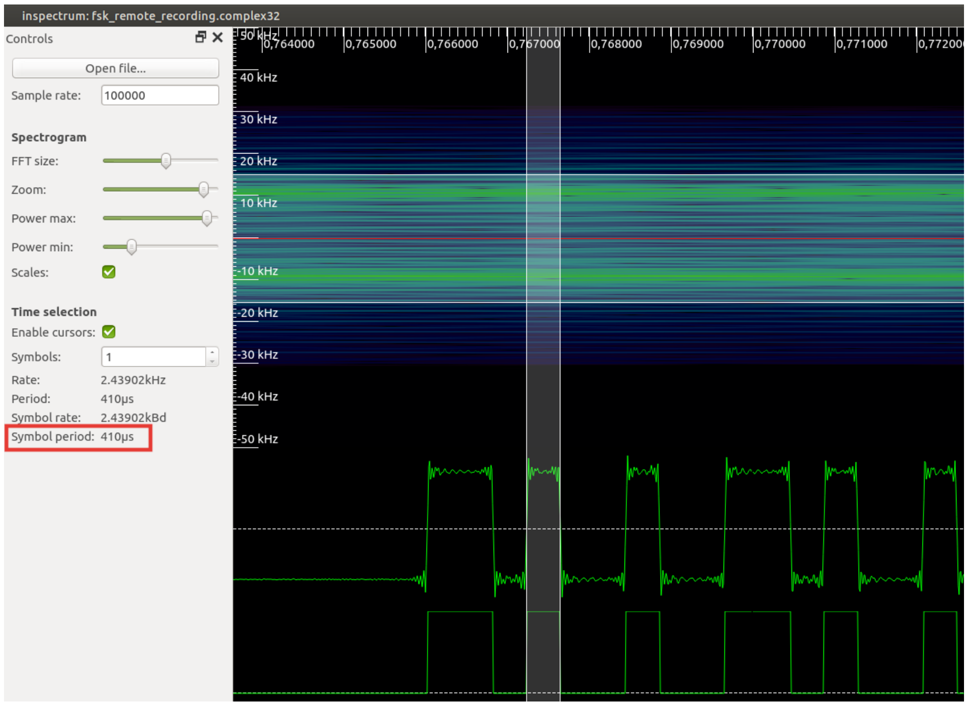


Figure 12. The demodulated signal

Now that we have demodulated the signal, we can now move on to decoding this little signal to get relevant data.

Extracting the Data

Before decoding the signal, we will have to find the samples per symbol rate. This is because both symbols, 0s and 1s, get repeated several times among the samples so that the receiver can synchronize with these symbols. To simulate that action, we can record the signal to "wav" and look at the number of samples it takes to get a "0" or a "1" through Audacity or another powerful tool like Inspectrum, which we used for this test.



inspectrum: fsk_remote_recording.complex32

Controls

Open file...

Sample rate: 100000

Spectrogram

FFT size:

Zoom:

Power max:

Power min:

Scales: ✓

Time selection

Enable cursors: ✓

Symbols: 1

Rate: 2.43902kHz

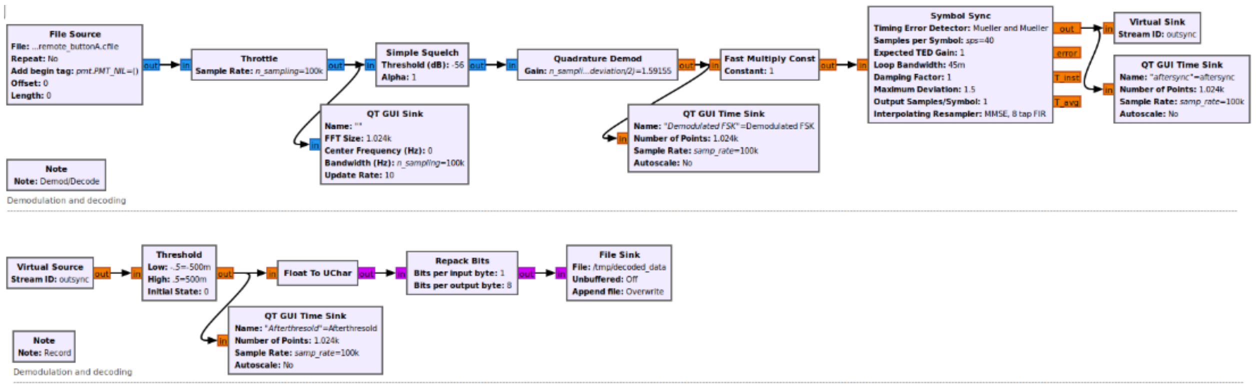
Period: 410µs

Symbol rate: 2.43902kBd

Symbol period: 410µs

Figure 13. Looking at samples per symbol rate in Inspectrum

We can clearly see that our symbol period is around 410 µs (most likely 400 µs), so the samples per symbol rate should be 40. We can then use this value in the Symbol Sync block, replacing the Clock Recovery Mueller and Mueller (M&M) block in the new versions of GNU Radio. Then we use a threshold value to extract clear 0 and 1 symbols, pack all bits to bytes, and record them into a text file. This flow can be seen in Figure 14. The data recorded into our file sink at /tmp/decoded_data can be seen in Figure 15.



File Source
File: ...remote_buttonA.cfile
Repeat: No
Add begin tag: pmt.PMT_NIL=()
Offset: 0
Length: 0

Throttle
Sample Rate: n_sampling=100k

Simple Squelch
Threshold (dB): -56
Alpha: 1

Quadrature Demod
Gain: n_sampl...deviation/2)=1.59155

Fast Multiply Const
Constant: 1

Symbol Sync
Timing Error Detector: Mueller and Mueller
Samples per Symbol: sps=40
Expected TED Gain: 1
Loop Bandwidth: 45m
Damping Factor: 1
Maximum Deviation: 1.5
Output Samples/Symbol: 1
Interpolating Resampler: MMSE, 8 tap FIR

Virtual Sink
Stream ID: outsync

QT GUI Time Sink
Name: "aftersync"=aftersync
Number of Points: 1.024k
Sample Rate: samp_rate=100k
Autoscale: No

QT GUI Sink
Name: ""
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): n_sampling=100k
Update Rate: 10

QT GUI Time Sink
Name: "Demodulated FSK"=Demodulated FSK
Number of Points: 1.024k
Sample Rate: samp_rate=100k
Autoscale: No

Note
Note: Demod/Decode

Demodulation and decoding

Virtual Source
Stream ID: outsync

Threshold
Low: -.5=-500m
High: .5=500m
Initial State: 0

Float To UChar

Repack Bits
Bits per input byte: 1
Bits per output byte: 8

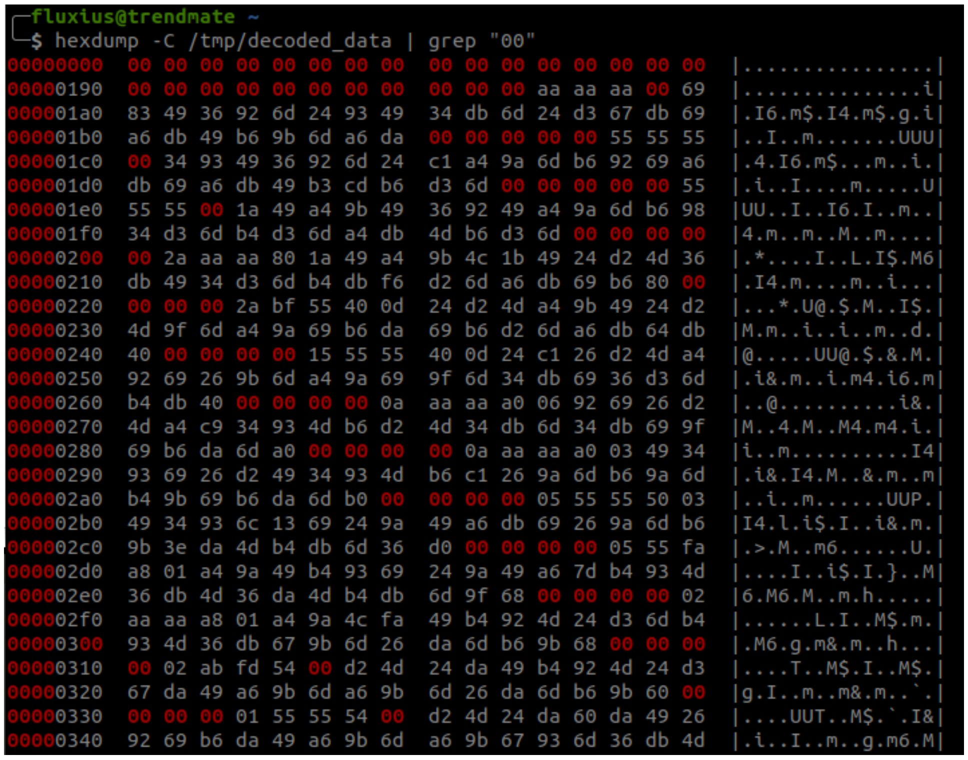
File Sink
File: /tmp/decoded_data
Unbuffered: Off
Append file: Overwrite

QT GUI Time Sink
Name: "Afterthresold"=Afterthresold
Number of Points: 1.024k
Sample Rate: samp_rate=100k
Autoscale: No

Note
Note: Record

Demodulation and decoding

Figure 14. The flowgraph to extract the relevant data into a binary file



```
┌─fluxius@trendmate ~
└─$ hexdump -C /tmp/decoded_data | grep "00"
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000190  00 00 00 00 00 00 00 00  00 00 00 aa aa aa 00 69  |...............i|
000001a0  83 49 36 92 6d 24 93 49  34 db 6d 24 d3 67 db 69  |.I6.m$.I4.m$.g.i|
000001b0  a6 db 49 b6 9b 6d a6 da  00 00 00 00 00 55 55 55  |..I..m.......UUU|
000001c0  00 34 93 49 36 92 6d 24  c1 a4 9a 6d b6 92 69 a6  |.4.I6.m$...m..i.|
000001d0  db 69 a6 db 49 b3 cd b6  d3 6d 00 00 00 00 00 55  |.i..I....m.....U|
000001e0  55 55 00 1a 49 a4 9b 49  36 92 49 a4 9a 6d b6 98  |UU..I..I6.I..m..|
000001f0  34 d3 6d b4 d3 6d a4 db  4d b6 d3 6d 00 00 00 00  |4.m..m..M..m....|
00000200  00 2a aa aa 80 1a 49 a4  9b 4c 1b 49 24 d2 4d 36  |.*....I..L.I$.M6|
00000210  db 49 34 d3 6d b4 db f6  d2 6d a6 db 69 b6 80 00  |.I4.m....m..i...|
00000220  00 00 00 2a bf 55 40 0d  24 d2 4d a4 9b 49 24 d2  |...*.U@.$.M..I$.|
00000230  4d 9f 6d a4 9a 69 b6 da  69 b6 d2 6d a6 db 64 db  |M.m..i..i..m..d.|
00000240  40 00 00 00 00 15 55 55  40 0d 24 c1 26 d2 4d a4  |@.....UU@.$.&.M.|
00000250  92 69 26 9b 6d a4 9a 69  9f 6d 34 db 69 36 d3 6d  |.i&.m..i.m4.i6.m|
00000260  b4 db 40 00 00 00 00 0a  aa aa a0 06 92 69 26 d2  |..@..........i&.|
00000270  4d a4 c9 34 93 4d b6 d2  4d 34 db 6d 34 db 69 9f  |M..4.M..M4.m4.i.|
00000280  69 b6 da 6d a0 00 00 00  00 0a aa aa a0 03 49 34  |i..m..........I4|
00000290  93 69 26 d2 49 34 93 4d  b6 c1 26 9a 6d b6 9a 6d  |.i&.I4.M..&.m..m|
000002a0  b4 9b 69 b6 da 6d b0 00  00 00 00 05 55 55 50 03  |..i..m......UUP.|
000002b0  49 34 93 6c 13 69 24 9a  49 a6 db 69 26 9a 6d b6  |I4.l.i$.I..i&.m.|
000002c0  9b 3e da 4d b4 db 6d 36  d0 00 00 00 00 05 55 fa  |.>.M..m6......U.|
000002d0  a8 01 a4 9a 49 b4 93 69  24 9a 49 a6 7d b4 93 4d  |....I..i$.I.}..M|
000002e0  36 db 4d 36 da 4d b4 db  6d 9f 68 00 00 00 00 02  |6.M6.M..m.h.....|
000002f0  aa aa a8 01 a4 9a 4c fa  49 b4 92 4d 24 d3 6d b4  |......L.I..M$.m.|
00000300  93 4d 36 db 67 9b 6d 26  da 6d b6 9b 68 00 00 00  |.M6.g.m&.m..h...|
00000310  00 02 ab fd 54 00 d2 4d  24 da 49 b4 92 4d 24 d3  |....T..M$.I..M$.|
00000320  67 da 49 a6 9b 6d a6 9b  6d 26 da 6d b6 9b 60 00  |g.I..m..m&.m..`.|
00000330  00 00 00 01 55 55 54 00  d2 4d 24 da 60 da 49 26  |....UUT..M$.`.I&|
00000340  92 69 b6 da 49 a6 9b 6d  a6 9b 67 93 6d 36 db 4d  |.i..I..m..g.m6.M|
```

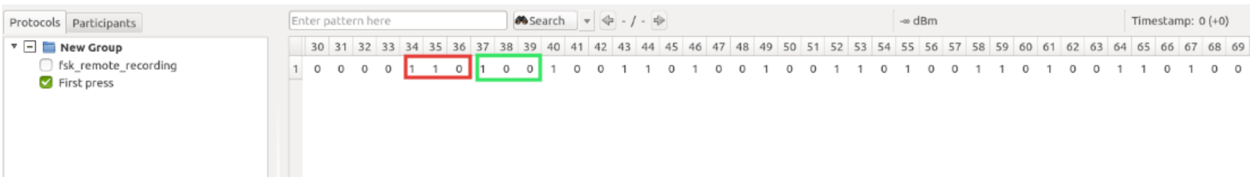
Figure 15. The extracted data from the signal

Note that the squelch and threshold should also be adjusted to the captured signal gain and spike drops. Optionally, an automatic gain control (AGC), which helps control the received signal strength, can be put before the synchronization if the signal is weak or the gain is unstable. We can also choose to leverage the weak signal manually. Afterward, it is advised to convert bytes into binary and apply a good alignment based on the preamble.

We can then try all commands and compare each button push to figure out the different fields in the packets in blind, but doing this work manually can be time-consuming. Rather, there exist tools that would allow the reversal of the different fields and compare pushes more efficiently.

Decoding the Data

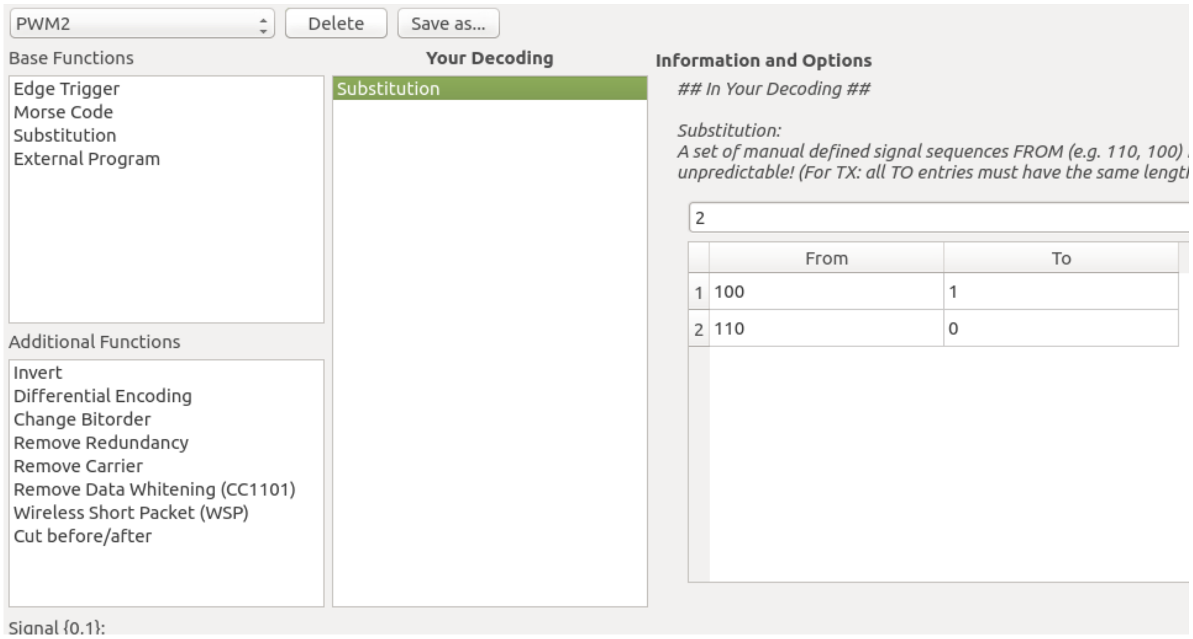
To decode the data, we use the Universal Radio Hacker (URH)[4] tool, which would allow us to record, demodulate, and compare different pushes. But first, let us analyze and guess the potential fields using URH.



	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69
1	0	0	0	0	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	1	1	0	1	0	0	1	1	0	1	0	0

Figure 16. First button press analysis in URH

We can see that many instances of "100" and "110" appear in the packet after the preamble, leading us to use the popular Pulse Width Modulation (PWM) encoding instead of Manchester encoding. Therefore, using URH, we can create a decoder that will substitute "110" with "0" and "100" with "1." The result can be seen in Figure 18.



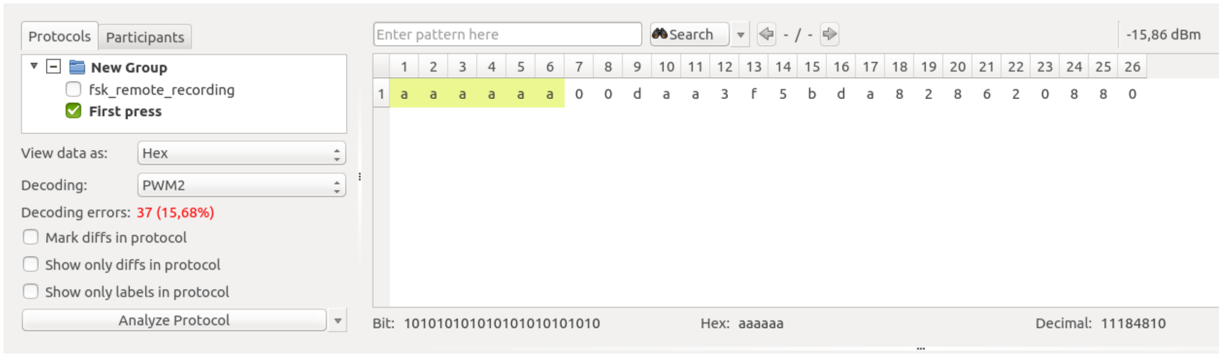
## In Your Decoding ##

Substitution:
A set of manual defined signal sequences FROM (e.g. 110, 100)
unpredictable! (For TX: all TO entries must have the same lengt

2

	From	To
1	100	1
2	110	0

Figure 17. Making a decoder in URH



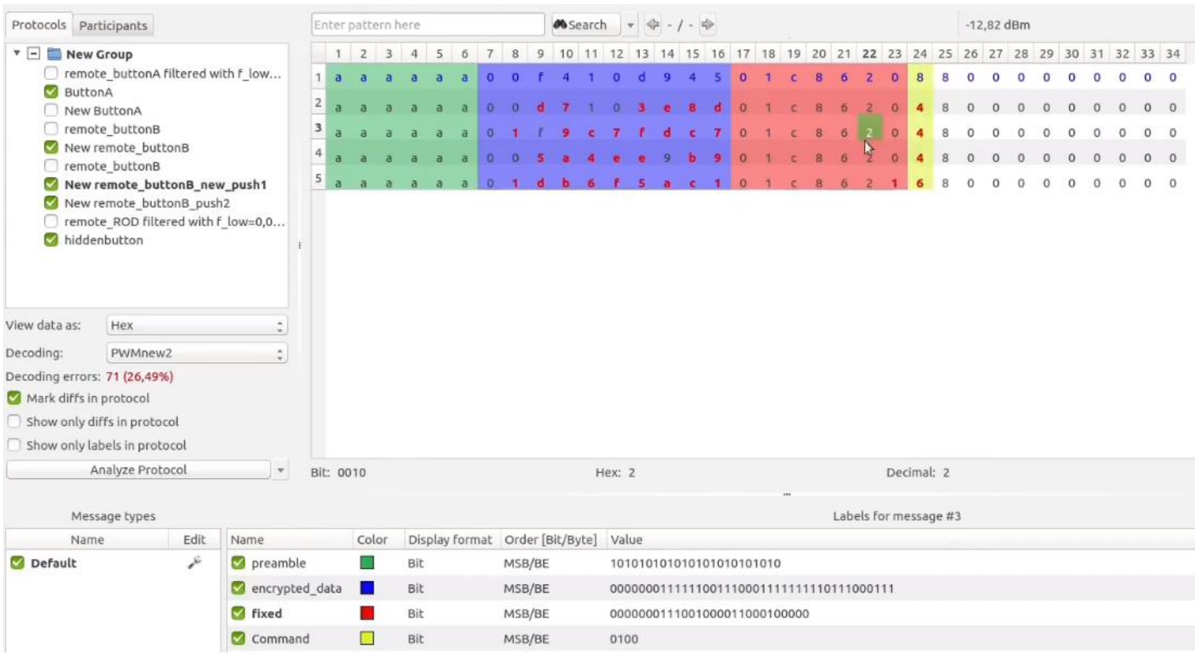
View data as: Hex
Decoding: PWM2
Decoding errors: 37 (15,68%)
Mark diffs in protocol
Show only diffs in protocol
Show only labels in protocol

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1	a	a	a	a	a	a	0	0	d	a	a	3	f	5	b	d	a	8	2	8	6	2	0	8	8	0

Bit: 101010101010101010101010 Hex: aaaaaa Decimal: 11184810

Figure 18. The PWM-decoded button press

We have been able to identify the preamble with only one sample. Therefore, we will have to keep going by pushing the same button again to observe the presence of the rolling code and try to guess other fields using an additional remote.


Protocols | Participants

- ☐ New Group
  - ☐ remote_buttonA filtered with f_low...
  - ✓ ButtonA
  - ☐ New ButtonA
  - ☐ remote_buttonB
  - ✓ New remote_buttonB
  - ☐ remote_buttonB
  - ✓ **New remote_buttonB_new_push1**
  - ✓ New remote_buttonB_push2
  - ☐ remote_ROD filtered with f_low=0,0...
  - ✓ hiddenbutton

View data as: Hex
Decoding: PWMnew2
Decoding errors: 71 (26,49%)
✓ Mark diffs in protocol
☐ Show only diffs in protocol
☐ Show only labels in protocol
Analyze Protocol

Enter pattern here | Search | - / - | -12,82 dBm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
1	a	a	a	a	a	a	0	0	f	4	1	0	d	9	4	5	0	1	c	8	6	2	0	8	8	0	0	0	0	0	0	0	0	0
2	a	a	a	a	a	a	0	0	d	7	1	0	3	e	8	d	0	1	c	8	6	2	0	4	8	0	0	0	0	0	0	0	0	0
3	a	a	a	a	a	a	0	1	f	9	c	7	f	d	c	7	0	1	c	8	6	2	0	4	8	0	0	0	0	0	0	0	0	0
4	a	a	a	a	a	a	0	0	5	a	4	e	e	9	b	9	0	1	c	8	6	2	0	4	8	0	0	0	0	0	0	0	0	0
5	a	a	a	a	a	a	0	1	d	b	6	f	5	a	c	1	0	1	c	8	6	2	1	6	8	0	0	0	0	0	0	0	0	0

Bit: 0010 Hex: 2 Decimal: 2

Message types

Name	Edit
✓ **Default**	

Labels for message #3

Name	Color	Display format	Order [Bit/Byte]	Value
✓ preamble	🟩	Bit	MSB/BE	1010101010101010101010
✓ encrypted_data	🟦	Bit	MSB/BE	00000001111110011100011111111110111000111
✓ fixed	🟥	Bit	MSB/BE	0000000111001000011000100000
✓ Command	🟨	Bit	MSB/BE	0100

Figure 19. Decoding a different button push

Note that we do this for the two buttons that we will label as A and B and the hidden button related to the DOR feature mentioned earlier.

After recording several different pushes, we can identify fields like the command field, the fixed part, and an encrypted part, which indicates that the remote is protected by a rolling/hopping code mechanism.

Looking again at the microcontroller chip, we can see an "M" logo, indicating that this chip is probably made by Microchip. While it is hard to find the reference and get the datasheet of this chip, we can refer to the KeeLoq algorithm, which Microchip natively uses for a rolling/hopping code.

Analysis of the DOR Command

Comparing the DOR command to the buttons we have labeled as A, B, C, and D of a remote — the latter two are only seen in the four-button test remote — shows no difference in the structure. We have the fixed part, an encrypted ID, and another encrypted part with the hopping/rolling code. But there is something strange with the DOR command: We can replay it.

If we try to send this same command multiple times, we can see that we can replay it without a problem for the receiver, which will accept it only until the rolling code is exhausted by any of the A, B, C, or D buttons.

This process is what allows for a plausible attack plan, as illustrated in Figure 20.

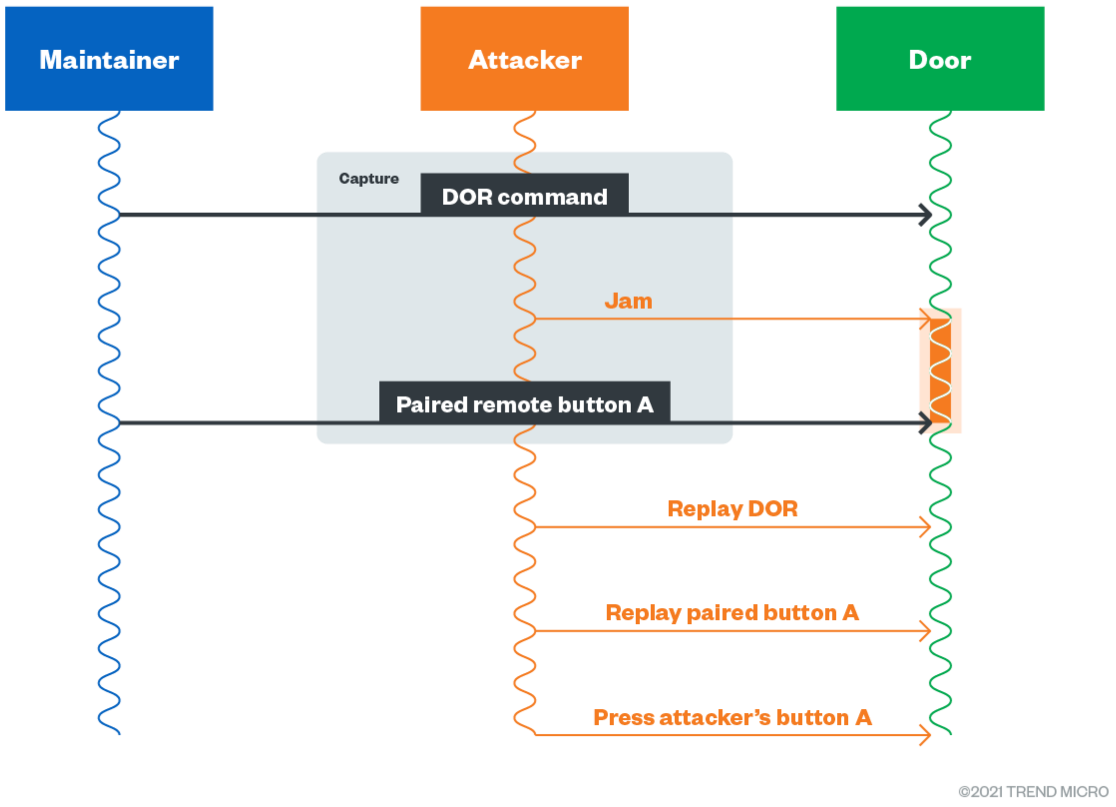


Figure 20. The initial DOR attack plan

As described in Figure 20, we need to sniff the DOR command, then block the first button press by jamming it while also recording it. The procedure will fail as a result, which should allow us to directly replay the DOR button of the authenticated remote, play its button A, and record our remote by sending its button A, B, C, and D signal.

We can also skip the first pushes and jam the remote maintainer to register and record our remote instead. But this technique would require us to capture both a valid button A and DOR key presses which, in a real scenario, would be rare unless the would-be intruder has physical access to the remote of a resident (even for just few seconds), or if they had the chance to meet the maintenance personnel tasked with programming a new remote.

The Rolling Code

Now we proceed to testing the second scenario. Let us see if and how we can go against the rolling code in use. Much of this might already be familiar to those in the industry, but it can perhaps serve as an introduction to others who are studying similar threat scenarios.

Microchip's KeeLoq

As mentioned earlier, Microchip uses its own proprietary KeeLoq algorithm to protect the packet from being replayed and decoded. There are two types of KeeLoq mechanisms: a hopping code, which we use in our case, and an Identify Friend or Foe (IFF), which relies on a challenge-response.

In our context, a rolling/hopping code is generated in the format illustrated in Figure 21.

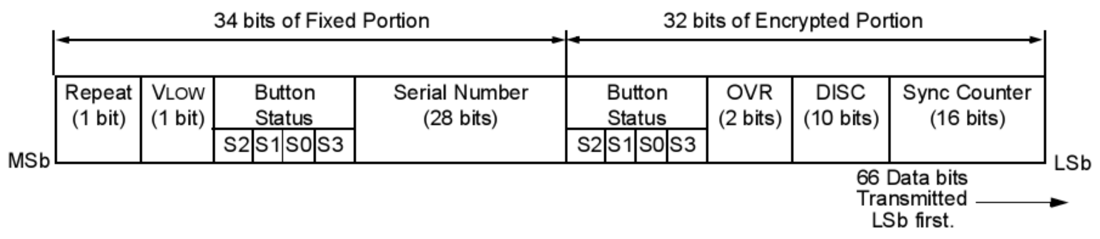


Figure 21. Format of KeeLoq encoded messages[5]

To encode the messages, the transmitter has several elements to fetch from its on-chip EEPROM, some of which are the following:

- The 64-bit encryption key
  Synchronization counter
  Serial number

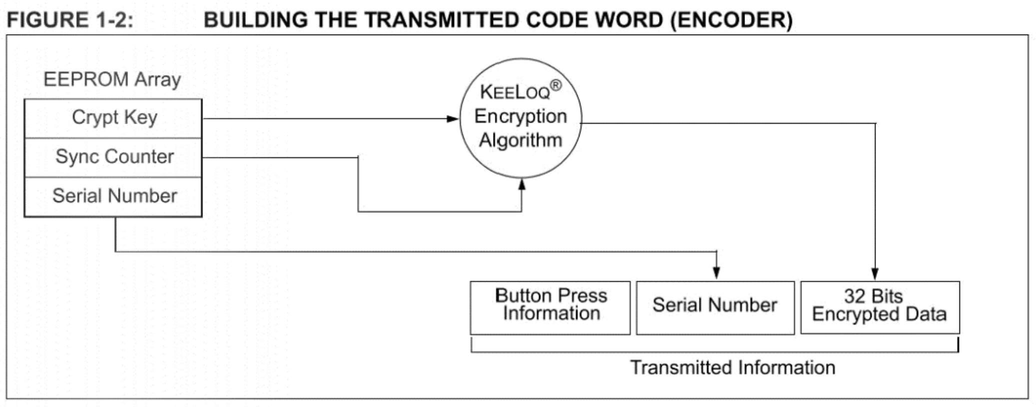


Figure 22. How KeeLoq encodes the messages[6]

For each keystroke, a message is generated with fresh, new encrypted data. Packets also cannot be replayed, as the receiver (in our case) will look for the keystroke with a sync counter +1.

Microchip used to propose to the vendors an algorithm depending on the serial number and a 64-bit manufacturer key to generate the crypt key. Using the crypt key, it is possible to encrypt or decrypt the second part of the message.

Moreover, the source of KeeLoq has been published and released to the public in the past. As a result, although it is now considered confidential and subject to a license, it is still possible to find the leaked source code of the algorithm.

Attacks on KeeLoq

Like many rolling/hopping code mechanisms, KeeLoq does not have any nonce or timestamp preventing an attacker from recording two successive pushes by jamming them, replaying the first one, then using the second one to discreetly open a door. This attack was demonstrated by Andrew Macpherson and Mike Davis in a video.[7] This attack was then re-implemented by several researchers, such as Samy Kamkar with his RollJam tool.[8]

KeeLoq itself is also suffering from cryptanalysis attacks like the following:

- Sliding techniques and linear approximation[9]
- Sliding and algebraic method[10]
- Side channels attacks with power analysis[11]

Moreover, after Microchip introduced a seed to improve KeeLoq security, COPACOBANA's parallel computing attack[12] was also published to get required secrets.

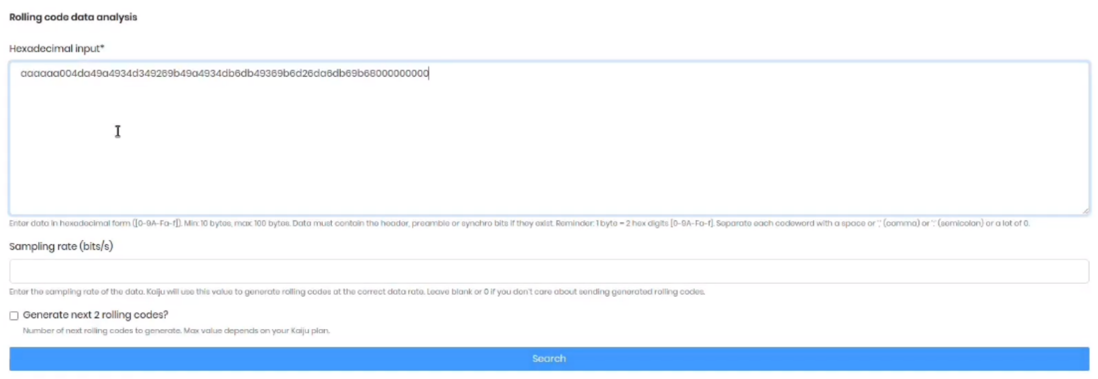
Among all these methods to extract the key is another way: dumping the content of the on-chip EEPROM through the in-circuit serial programming (ICSP) if the memory is not read-protected.

There are also interesting services that support many remote types. These would allow an intruder to quickly generate commands from a captured message and see how they can use these to get access.

Opening Doors with Kaiju

Kaiju is a service developed by the creators of PandwaRF,[13] a portable device with a nice form factor that we elaborate on later. This service allows users to send hex or binary payloads to analyze the rolling code of many devices. A complete list of its supported devices is available on the Kaiju project website.[14]

To use the service, we take the hexadecimal of one of the commands we captured and copy it to the form, seen in Figure 23.



Rolling code data analysis

Hexadecimal input*

aaaaaa004da49a4934d349269b49a4934db6db49369b6d26da6db69b6800000000

Enter data in hexadecimal form ([0-9A-Fa-f]). Min: 10 bytes, max: 100 bytes. Data must contain the header, preamble or synchro bits if they exist. Reminder: 1 byte = 2 hex digits [0-9A-Fa-f]. Separate each codeword with a space or ',' (comma) or ';' (semicolon) or a lot of 0.

Sampling rate (bits/s)

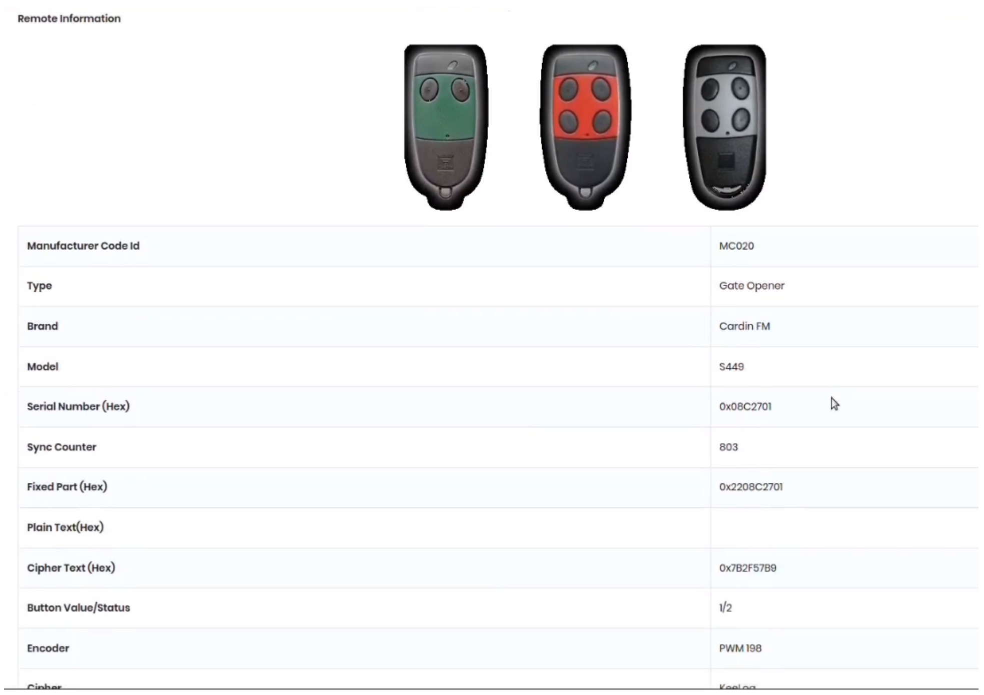
Enter the sampling rate of the data. Kaiju will use this value to generate rolling codes at the correct data rate. Leave blank or 0 if you don't care about sending generated rolling codes.

☐ Generate next 2 rolling codes?

Number of next rolling codes to generate. Max value depends on your Kaiju plan.

Figure 23. Sending a remote button push to Kaiju

After sending this command, we can successfully decode it and gain useful information that would help us decode the payload we captured.



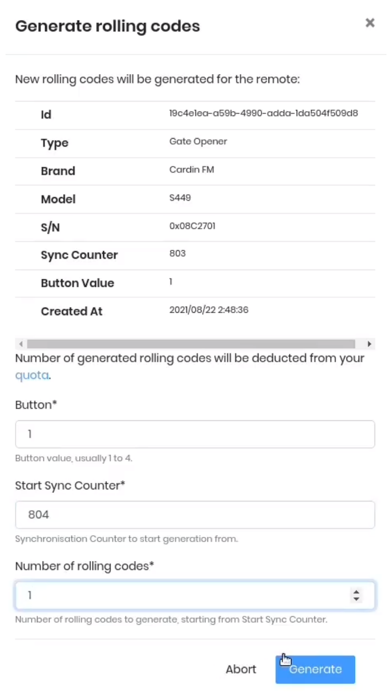
Remote Information

Manufacturer Code Id	MC020
Type	Gate Opener
Brand	Cardin FM
Model	S449
Serial Number (Hex)	0x08C2701
Sync Counter	803
Fixed Part (Hex)	0x2208C2701
Plain Text(Hex)	
Cipher Text (Hex)	0x7B2F57B9
Button Value/Status	1/2
Encoder	PWM 198
Cipher	KeeLoq

Figure 24. Remote information from our captured button push

We can see information such as the hexadecimal value that refers to the remote's unique identifier (UID), how the signal is encoded, which button was used, the synchronization counter state, and the cipher algorithm (KeeLoq for this case).

Taking note of the "Sync counter" state, we can now generate the next push of a button on the same page, as seen in Figures 25 and 26.



Generate rolling codes

New rolling codes will be generated for the remote:

Id	19c4e1ea-a59b-4990-adda-1da504f509d8
Type	Gate Opener
Brand	Cardin FM
Model	S449
S/N	0x08C2701
Sync Counter	803
Button Value	1
Created At	2021/08/22 2:48:36

Number of generated rolling codes will be deducted from your quota.

Button*: 1

Button value, usually 1 to 4.

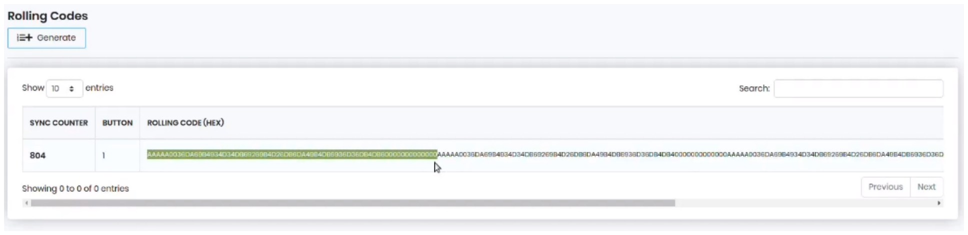
Start Sync Counter*: 804

Synchronisation Counter to start generation from.

Number of rolling codes*: 1

Number of rolling codes to generate, starting from Start Sync Counter.

Figure 25. Generating the next button push rolling code with Kaiju



Rolling Codes

Generate

Show 10 entries Search:

SYNC COUNTER	BUTTON	ROLLING CODE (HEX)
804	1	AAAAA0036DA69B4934D34DB69269B4D26DB6DA49B4DB6936D36DB4DB40000000000000AAAAA0036DA69B4934D34DB69269B4D26DB6DA49B4DB6936D36DB4DB40000000000000AAAAA0036DA69B4934D34DB69269B4D26DB6DA49B4DB6936D36D

Showing 0 to 0 of 0 entries

Previous Next

Figure 26. The generated rolling code payload

We need the first part of the rolling code to send it over the air. To do so, we can copy the hexadecimal value to URH. However, natively this will be difficult to do because URH does not seem to take text files in its generator tab. Therefore, to avoid modifying the URH Python code, let us do this on GRC.

On GRC, we take a text file string, unpack bytes into bits, map them to have a mark and a space that consider a 20kHz deviation, (noted earlier when we first captured the signal), and center to DC-Centered to get our 2-FSK final signal. The result is the flowgraph seen in Figure 27.

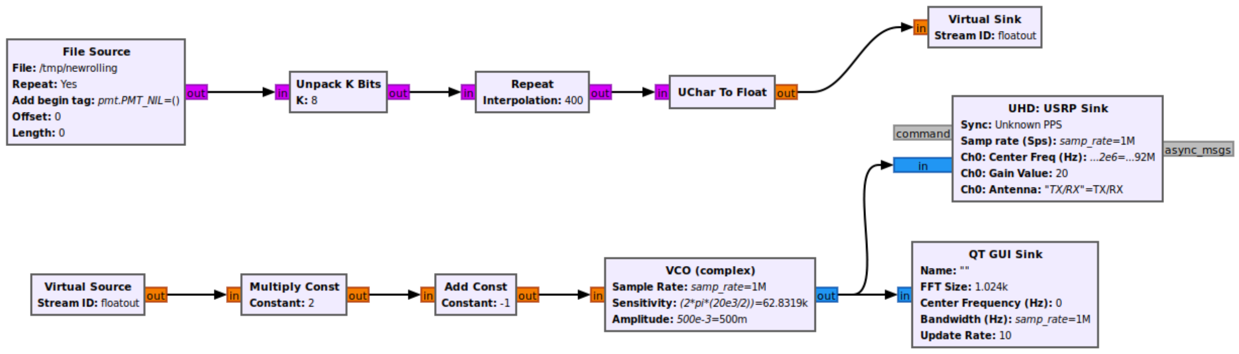
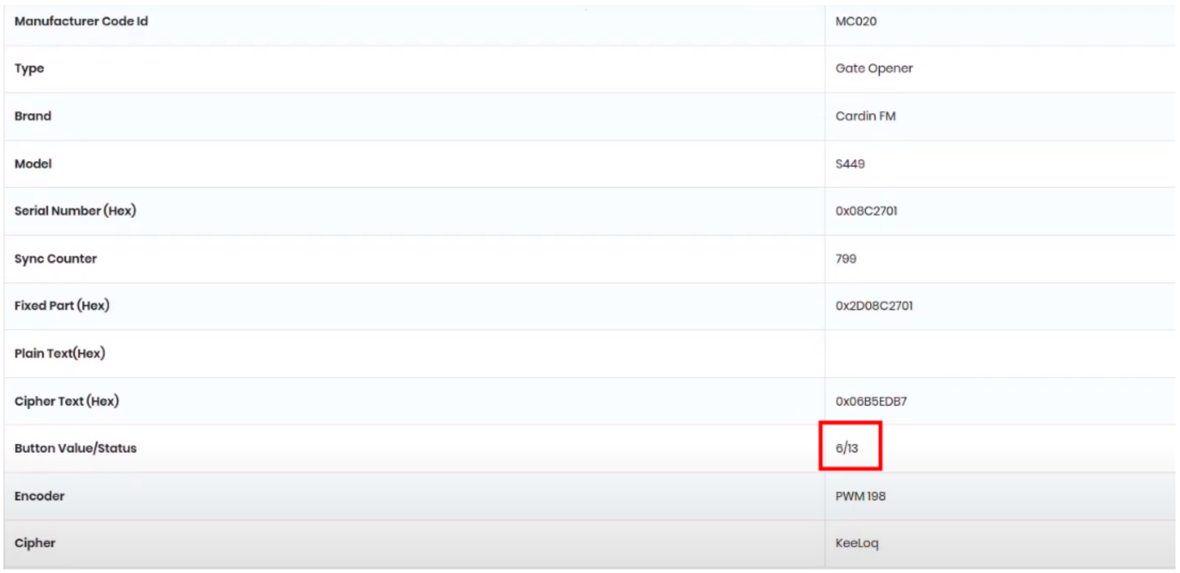


Figure 27. The transmitting flowgraph

With this, the receiver should acknowledge the command, allowing us to open the garage with this quick technique. Now, however, we have a different question to ask: What would happen if we mixed the DOR command with the rolling code generation?

Mixing DOR With Rolling Code Brute Force

At this point, Kaiju is not yet available meaning we are not allowed to generate the sixth push of the button as decoded and seen in Figure 28.



Manufacturer Code Id	MC020
Type	Gate Opener
Brand	Cardin FM
Model	S449
Serial Number (Hex)	0x08C2701
Sync Counter	799
Fixed Part (Hex)	0x2D08C2701
Plain Text(Hex)	
Cipher Text (Hex)	0x06B5EDB7
Button Value/Status	6/13
Encoder	PWM 198
Cipher	KeeLoq

Figure 28. Analyzing a DOR command in Kaiju

After talking to the creators of Kaiju, we learned that the buttons pushed are limited to a value of "4" maximum but this will be unlocked for LEA users, which will prevent every user both from accessing and having permanent access easily.

However since KeeLoq’s implementations are available on the internet, this does not mean the end for our test. Nevertheless, we will still have to encode the payload properly and get the manufacturer key to generate the payload on our own. Furthermore, we will need to identify the chip family and use a PIC ICSP programmer to get the memory, if this memory is not protected on the targeted chip.

Another method that we can use, which would allow us to get a few master keys of other vendors, would be to use the memory dump of the WHY EVO remote, which is based on a STMicroelectronics chip. Afterward, we can interface with this chip through the single wire interface module (SWIM) communication protocol using an ST-LINK/V2 to dump the program and the keys.

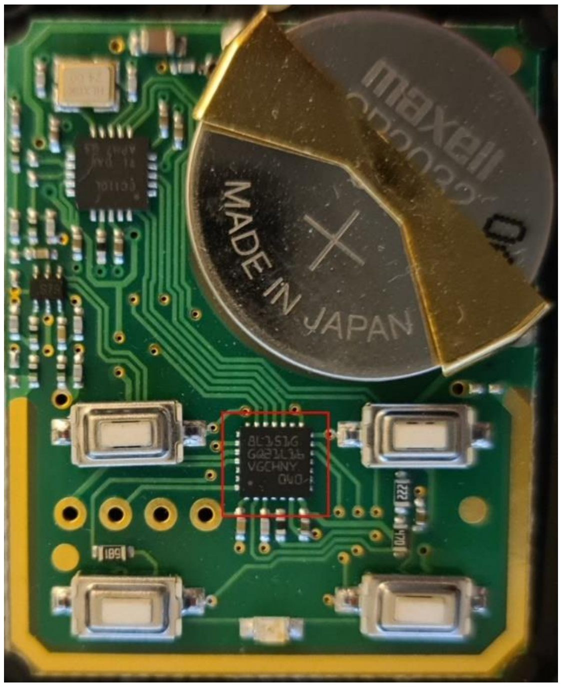


Figure 29. The WHY EVO circuit

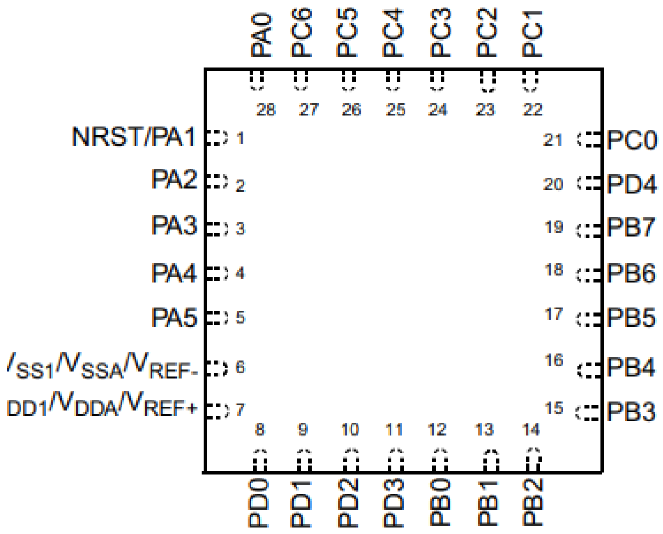


Figure 30. The pinout of the STM8L151Gx chip

Minimizing the Setup With PandwaRF

Knowing the frequency and the data sent by the remote, we can minimize the setup of an attack using a frequency analyzer with a compact form factor called PandwaRF.

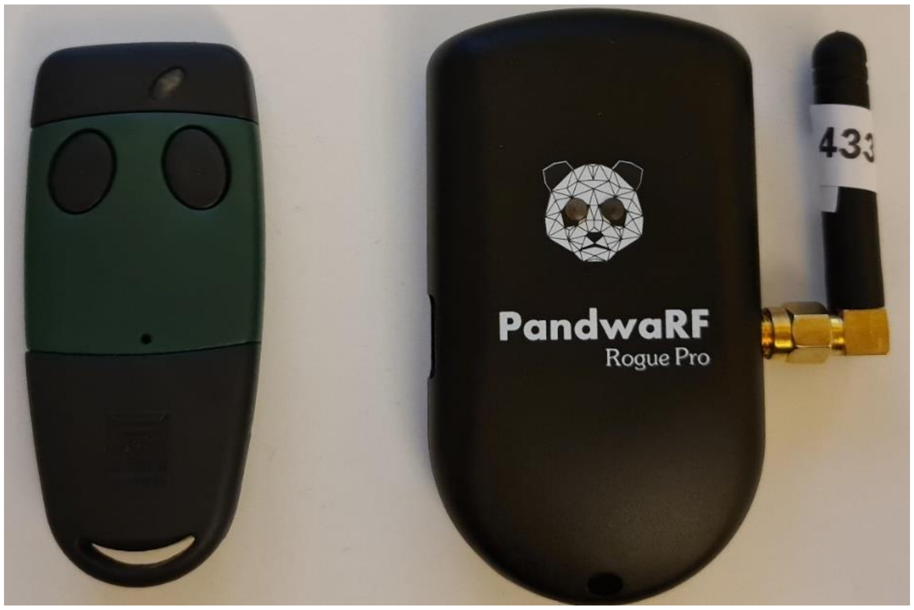
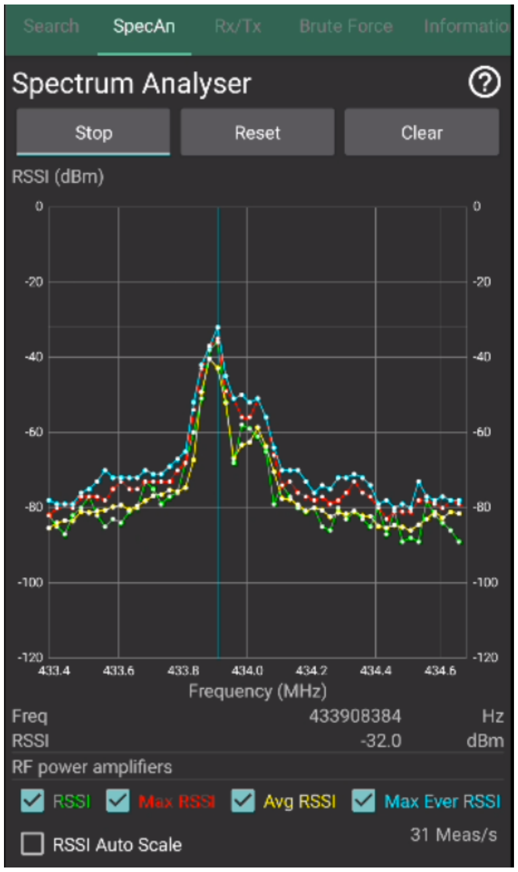


Figure 31. The PandwaRF shown close to a remote to see its compact form factor

PandwaRF is indeed an interesting tool in that it can be used by threat actors to conduct stealthy attacks. They would also have to analyze the target as we had done to understand the data we discussed earlier.

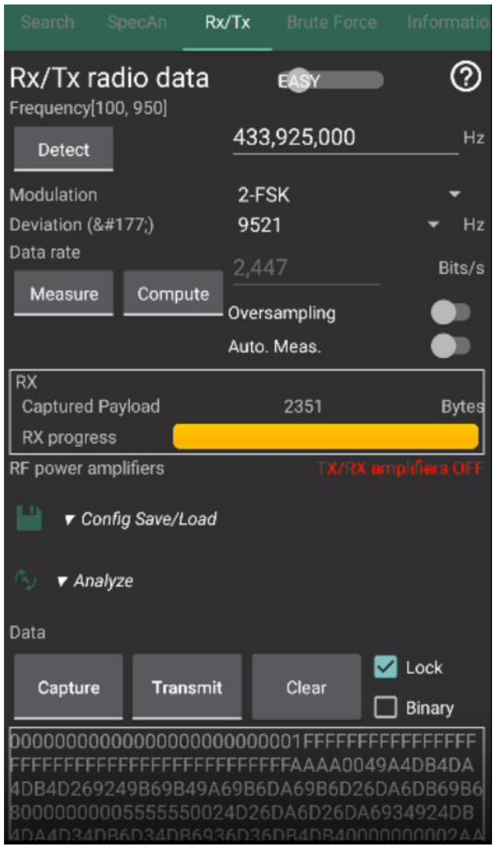
Nevertheless, the PandwaRF Android application allows us to see what goes over the air with a proper frequency analyzer that spots the remote sending a command.



Freq	433908384	Hz
RSSI	-32.0	dBm

Figure 32. The frequency analyzer of the PandwaRF tool showing our remote's signal

It is much more difficult to tell what modulation we are facing using this device, even when zooming in on the signal. However, PandwaRF is still well featured enough to detect it; we just have to follow a few steps to do so. First, we need the right antenna to get enough energy, and then we need to detect the right frequency (+ possible shift). Lastly, we need to force parameters if the auto-measurement fails to capture the data. Following these steps, we can recover what we were able to decode earlier.



Rx/Tx radio data

Frequency[100, 950] 433,925,000 Hz

Modulation 2-FSK

Deviation (&#177;) 9521 Hz

Data rate 2,447 Bits/s

Oversampling

Auto. Meas.

RX

Captured Payload 2351 Bytes

RX progress

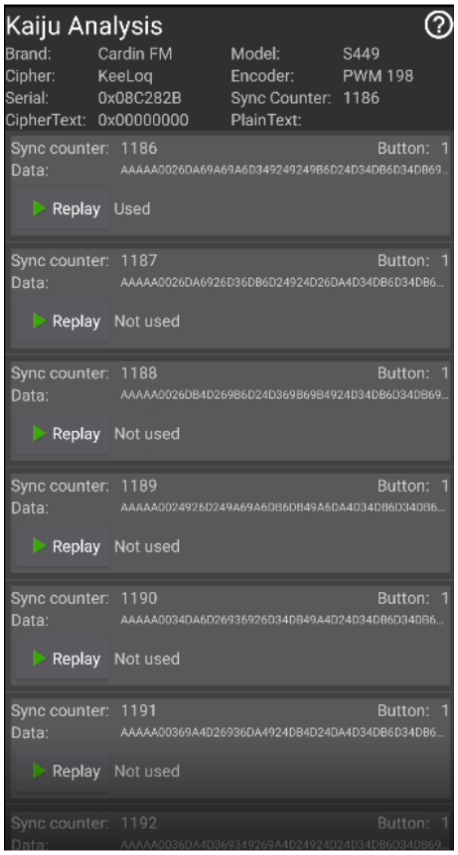
RF power amplifiers TX/RX amplifiers OFF

Data

[✓] Lock

[ ] Binary

```
0000000000000000000000000001FFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFAAAA0049A4DB4DA
4DB4D269249B69B49A69B6DA69B6D26DA6DB69B6
B0000000005555550024D26DA6D26DA6934924DB
4DA4D34DB6D34DB6936D36DB4DB40000000002AA
```



Kaiju Analysis

Brand: Cardin FM Model: S449
Cipher: KeeLoq Encoder: PWM 198
Serial: 0x08C282B Sync Counter: 1186
CipherText: 0x00000000 PlainText:

```
Sync counter:  1186                                    Button:  1
Data:          AAAAA0026DA69A69A6D349249249B6D24D34DB6D34DB69...
Replay  Used

Sync counter:  1187                                    Button:  1
Data:          AAAAA0026DA6926D36DB6D24924D26DA4D34DB6D34DB6...
Replay  Not used

Sync counter:  1188                                    Button:  1
Data:          AAAAA0026DB4D269B6D24D369B69B4924D34DB6D34DB69...
Replay  Not used

Sync counter:  1189                                    Button:  1
Data:          AAAAA0024926D249A69A6DB6DB49A6DA4D34DB6D34DB6...
Replay  Not used

Sync counter:  1190                                    Button:  1
Data:          AAAAA0034DA6D26936926D34DB49A4D24D34DB6D34DB6...
Replay  Not used

Sync counter:  1191                                    Button:  1
Data:          AAAAA00369A4D26936DA4924DB4D24DA4D34DB6D34DB6...
Replay  Not used

Sync counter:  1192                                    Button:  1
Data:          AAAAA0036DA4D369349269A4D24924D24D34DB6D34DB69...
```

Figure 33. Signal analysis (left) and the generated rolling code (right) using the PandwaRF APK and Kaiju service

After this, we can directly send it to Kaiju because the Android application for this device can connect us directly to this service. Kaiju will then automatically generate rolling codes for us to test. It should be noted that Kaiju also processes incoming data if parameters like the data rate are incorrect on PandwaRF.

Having generated the rolling code, we can send the next commands to the door and open it.

Going back to the DOR command, since PandwaRF allows us to use our own API, we can generate new A and DOR buttons, making it possible to register a new remote to gain permanent access to the mechanism.

Conclusion

Through our demonstration, we have shown how it is possible to gain access to a garage door by brute forcing the rolling code of a garage door mechanism. Not only that, but by using the DOR function, we demonstrated how a potential intruder can gain permanent access and proved how a powerful yet compact device such as PandwaRF can make such an attack stealthier to execute.

Cars, garage doors, and Remote Keyless Entry (RKE) systems[15] suffer from not having timestamps when faced with jamming and replay attacks. Looking deeper into the structure of a frame captured over the radio, decoding it, and getting the secrets of its unknown parameters allow attackers to generate codes themselves. Combined with a DOR procedure and similar features in other remotes, attackers need only perform an attack once and then reuse its own recorded remote.

It is only a matter of time before rolling code attacks such as this become a reality that homeowners face. Once an algorithm is known and the master keys are recovered, it would be possible to quickly brute force the remaining parameters within a decent period. Manufacturers, on the other hand, are faced with the challenge to avoid this issue in their own products. They must ensure that their microcontrollers or Secure Elements (SEs) will keep their secrets. In the face of such a possibility, chips must be tested against glitches and side-channel attacks.

Recommendations

In order to prevent such an attack from becoming a reality, steps should be taken to make them more complex to execute. This means adding several mechanisms in addition to the rolling code, such as the following:

1. Using a different manufacturer key per remote and introducing diversification so that an attacker would have to find out the generation algorithm of each key, even after dumping the master key
2. Debugging interfaces that are physically disabled on remotes and receivers
3. Implementing memory protection on remotes and receivers to avoid possible leaks
4. Using a seed when adding to the sync counter to complexify the brute-force process

For their part, homeowners can also take steps to prevent such attacks by ensuring that their remote and receivers are secured and adding other security measures. These best practices include:

1. Never leaving open garages unattended for long periods
2. Being mindful of where garage remotes are kept
3. Considering the use of traditional locks such as padlocks and deadlocks to secure garages, especially when one is out of town
4. Installing additional security measures such as sensors and cameras

As this demonstration shows, users should become more mindful of hidden features in their remotes to disable them if necessary. For example, the DOR feature can be manually disabled by removing a jumper on the receiver. Users should also keep the receiver hidden, protected against opening, and strengthened against simple lock-picking attacks.

References

[1] Andy Greenberg. (Aug. 6, 2015). *The Wired*. "This Hacker's Tiny Device Unlocks Cars And Opens Garages." Accessed on Sept. 29, 2021, at https://www.wired.com/2015/08/hackers-tiny-device-unlocks-cars-opens-garages/.

[2] Sébastien Dudek. (Aug. 23, 2021). *YouTube*. "RollD.O.R Attacks - Getting permanent accesses in garages." Accessed on Sep. 29, 2021, at https://www.youtube.com/watch?v=vI9UsjFCzaE.

[3] Cardin Electronica. (n.d.). *Cardin Electronica*. "Radiocomando Digitale A Codico Rolling S449." Accessed on Sept. 29, 2021, at https://www.cardin-europe.com/index.php?controller=attachment&id_attachment=626.

[4] Johannes Pohl. (July 7, 2021). *GitHub*. "Universal Radio Hacker." Accessed on Sept. 29, 2021, at https://github.com/jopohl/urh.

[5] Microchip. (n.d.). *Microchip*. "KEELOQ® Code Hopping Encoder." Accessed on Sept. 29, 2021, at http://ww1.microchip.com/downloads/en/devicedoc/21143b.pdf.

[6] Microchip. (n.d.). *Microchip*. "KEELOQ® Code Hopping Encoder." Accessed on Sept. 29, 2021, at http://ww1.microchip.com/downloads/en/devicedoc/21143b.pdf.

[7] Andrew Macpherson and Mike Davis. (Feb. 5, 2016). *AndrewNohawk*. "Bypassing Rolling Code Systems." Accessed on Sept. 29, 2021, at andrewmohawk.com/2016/02/05/bypassing-rolling-code-systems/.

[8] Andy Greenberg. (Aug. 6, 2015). *The Wired*. "This Hacker's Tiny Device Unlocks Cars And Opens Garages." Accessed on Sept. 29, 2021, at https://www.wired.com/2015/08/hackers-tiny-device-unlocks-cars-opens-garages/.

[9] Andrey Bogdanov. (Feb 16, 2007). *International Association for Cryptologic Research*. "Cryptanalysis of the KeeLoq block cipher." Accessed on Sept. 29, 2021, at https://www.iacr.org/cryptodb/data/paper.php?pubkey=13337.

[10] Nicolas T. Courtois, Gregory V. Bard, and David Wagner. (Aug. 15, 2007). *The Cryptology ePrint Archive*. "Algebraic and Slide Attacks on KeeLoq." Accessed on Sept. 29, 2021, at https://eprint.iacr.org/2007/062.pdf.

[11] Thomas Eisenbarth and Timo Kasper. (Dec. 27, 2008). *Ruhr-Universität Bochum Chair for Embedded Security*. "Messing around with Garage Doors Breaking KeeLoq with Power Analysis." Accessed on Sept. 29, 2021, at https://www.emsec.ruhr-uni-bochum.de/media/crypto/attachments/files/2011/05/KeeLoq25C3_opt.pdf.

[12] Martin Novotn and Timo Kasper. (Sept. 9, 2009). *Ruhr-Universität Bochum Chair for Embedded Security*. "Cryptanalysis of KeeLoq with COPACOBANA." Accessed on Sept. 29, 2021, at https://www.emsec.ruhr-uni-bochum.de/research/publications/cryptanalysis-keeloq-copacobana/.

[13] PandwaRF. (n.d.). *PandwaRF*. "PandwaRF." Accessed on Sept. 29, 2021, at https://pandwarf.com/.

[14] PandwaRF. (n.d.). *PandwaRF*. "Kaiju - Rolling code analyzer & generator." Accessed on Sept. 29, 2021, at https://rolling.pandwarf.com/.

[15] Ryad Benadjila, Mathieu Renard, José Lopes-Esteves, and Chaouki Kasmi. (June 8, 2017). *SSTIC*. “From Academia to Real World: a Practical Guide to Hitag-2 RKE System Analysis.” Accessed on Sept. 29, 2021, at https://www.sstic.org/2017/presentation/from_academia_to_real_world_a_practical_guide_to_hitag-2_rke_system_analysis/.