Rapid Radio Reversing

Michael Ossmann
Great Scott Gadgets

Abstract

Over the past several years, Software Defined Radio (SDR) has rendered a new generation of wireless security researchers capable of reverse engineering and exploring the attack surfaces of digital radio technologies that now pervade the world around us. While SDR is certainly the single greatest tool for reverse engineering wireless signals, a variety of non-SDR tools based on wireless transceiver Integrated Circuits (ICs) may also be found in the toolbox of the wireless reverse engineer. The use of a combination of both SDR and non-SDR tools often enables the quickest and easiest way to reverse a new wireless communication system. I provide an example method employing this hybrid approach.

1 Introduction

A Software Defined Radio is a truly universal radio. Regardless of the modulation and encoding of the target system, you can use an SDR platform with sufficient bandwidth and operating frequency to capture a signal. After acquiring the radio waveform, you can quickly discover the target's precise operating frequency and modulation characteristics. Additionally, you can use this captured waveform to execute a replay attack without any effort to understand the modulation and encoding.

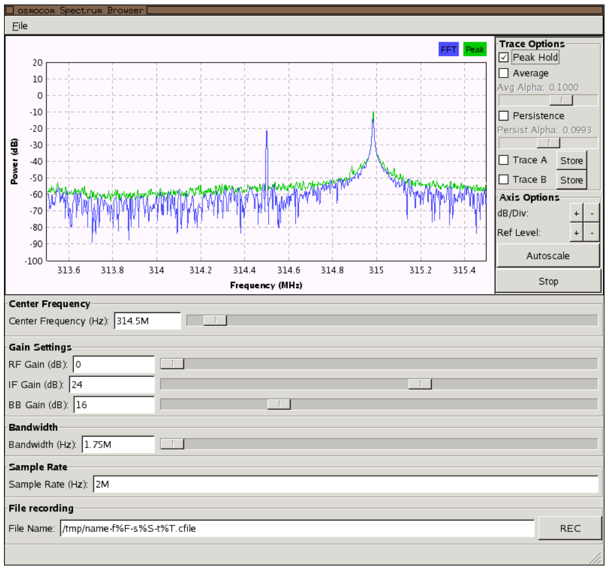
While the initial steps of reverse engineering a signal with SDR go very quickly, subsequent efforts to determine data encoding, byte alignment, forward error correction, checksums, encryption, and whitening often take much longer. A reverse engineer is just as likely to turn to pen and paper as to SDR tools during this phase of the project. Efforts to analyze the data encoding may be hampered, for example, by difficulty with recovery of the transmitter's symbol clock, causing a frustrating inability to reliably convert a waveform into bits.

Much of this frustration can be overcome by improving your SDR skills [7]. To become an expert wireless reverse engineer capable of analyzing rare or technologically advanced wireless systems, there is no substitute for becoming proficient in SDR development. However, SDR has a steep learning curve; it can take years to learn the Digital Signal Processing (DSP) and SDR techniques necessary to reverse engineer arbitrary wireless communication systems with a pure SDR approach. Fortunately, for many target wireless systems, the most significant advantages of SDR for reverse engineering are accessible through techniques that can be learned very quickly.

Instead of using SDR alone, for many reverse engineering efforts I recommend a hybrid approach using both SDR and non-SDR tools. Platforms such as YARD Stick One [5] provide a wireless transceiver IC that includes a digital modem implemented in hardware instead of software. Although these tools are less flexible than software implementations enabled by SDR, they are compatible with a wide range of digital radio systems, particularly low cost, low speed systems.

Wireless transceiver ICs do the job of clock recovery and can instantly provide binary symbol data. This makes it easy for the reverse engineer to capture multiple packets, compare them with one another, and explore the data encoding characteristics of the target system. These advantages are particularly attractive when considering the easier learning curve compared to SDR.

A hybrid approach of using both SDR and non-SDR tools has been employed for as long as wireless security researchers have used SDR. Despite SDR's increasing popularity, the use of non-SDR tools has not fallen out of favor. New advances in wireless security are just as likely to be made by a hybrid approach now as they were several years ago. Within the past year, for example, Samy Kamkar used the approach in reverse engineering remote keyless entry systems [6], and Mike Ryan and Richo Healey used it to reverse electric skateboard control interfaces [9].



☑ Peak Hold
☐ Average
Avg Alpha: 0.1000
☐ Persistence
Persist Alpha: 0.0993
☐ Trace A
☐ Trace B

Center Frequency
Center Frequency (Hz): 314.5M

Gain Settings
RF Gain (dB): 0
IF Gain (dB): 24
BB Gain (dB): 16

Bandwidth
Bandwidth (Hz): 1.75M

Sample Rate
Sample Rate (Hz): 2M

File recording
File Name: /tmp/name-f%F-s%S-t%T.cfile

Figure 1: osmocom_fft capturing an OOK signal

2 Signal Acquisition

To use the hybrid approach, you only need to know a little about SDR. The most important thing to understand is that an SDR receiver captures a contiguous range of radio frequencies for the duration of the acquisition. Just as a sound card captures any and all sounds in the audible range of frequencies while recording, an SDR receiver captures every radio signal present in the range of frequencies it is tuned to. Everything that happens in that chunk of spectrum is described by the digital waveform produced by the receiver. This is best demonstrated by storing a captured waveform and visualizing it with a spectrogram as we will do in this example.

To capture a signal, plug in a HackRF One [4] or other SDR platform and launch osmocom_fft, a simple software tool for real-time visualization and acquisition [11]. Set the frequency and sample rate such that your signal of interest is within the displayed bandwidth. The total captured bandwidth is equal to the sample rate, so there is some benefit in setting the sample rate as high as possible, especially if you are not certain of the target frequency and need to hunt for it. (To avoid hunting, you can often learn the frequency from online resources such as the FCC Equipment Authorization database [10].) Note that the acquisition bandwidth is entirely controlled by the sample rate setting; the separate bandwidth setting controls filtering in the analog domain, and you typically don't need to change it.

With osmocom_fft tuned appropriately, initiate a transmission on your target device and observe the signal in the FFT plot. If it isn't obvious right away, try placing your receive antenna very close to the target transmitter. Signals at very close range are received with much greater power than signals from more distant transmitters. Once you've found the signal of interest, spend some time adjusting the gain settings to give you the cleanest acquisition. If your gain is too low, the signal will be lost in the noise. If your gain is too high, the signal will be distorted, resulting in extra peaks showing up at frequencies other than the signal's true frequency. I usually try to arrange things so that the target signal is about -10 dB with respect to the highest power the SDR receiver can handle, but the most important thing to optimize is the signal to noise ratio.

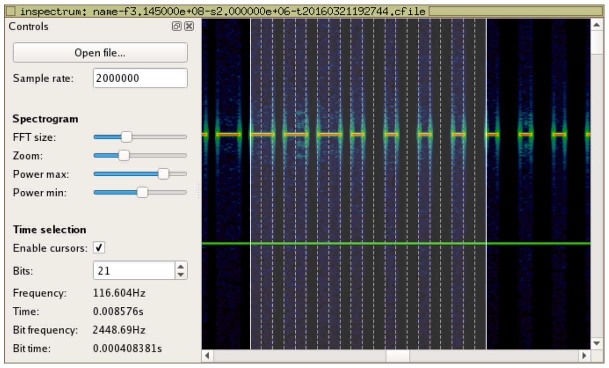
Like many SDR receivers, HackRF One has an architecture that results in a spike in the center of the captured bandwidth. Even if this spike has been hidden from you by hardware or software, it can affect the received signal. For this reason it is good to develop a habit of tuning to one side of your target frequency. You'll notice the importance of visualization as you go through the process of tuning the frequency and setting gains. These things are very difficult to do blindly!

Once everything is set up for optimal signal acquisition, click the record button in the lower right corner. A file containing the received waveform will be saved until you click stop. During acquisition, the file grows at a rate of 8 bytes per sample, so your maximum capture duration and sample rate may be limited by your storage medium.

3 Signal Visualization and Analysis

My favorite tool for visualizing a saved waveform is inspectrum, a new software tool for signal analysis [12]. The primary feature of inspectrum is a spectrogram that allows you to explore everything that took place in the received bandwidth during the acquisition period. A spectrogram displays time on one axis and frequency on another. Signal power is indicated by color or brightness of pixels in this two-dimensional domain. You've probably seen a spectrogram in the form of a waterfall display. Unlike most waterfalls, the spectrogram in inspectrum has a continuous time axis showing every signal detected during the acquisition, even very brief events. Recent versions of inspectrum display frequency on the vertical axis, so it will appear sideways compared to the frequency domain plot in osmocom_fft.

Using inspectrum, try to identify the precise operating frequency, modulation, and symbol rate of the target transmission. The frequency is displayed relative to osmocom_fft's configured center frequency. The modulation and symbol rate can usually be identified visually for transmissions with lower data rates. Put the cursor in multi-bit mode and drag it over the signal to measure the



```
inspectrum: name-f3.145000e+08-s2.000000e+06-t20160321192744.cfile
Controls
Sample rate:     2000000
Spectrogram
Time selection
Enable cursors:  ✓
Bits:            21
Frequency:       116.604Hz
Time:            0.008576s
Bit frequency:   2448.69Hz
Bit time:        0.000408381s
```

Figure 2: inspectrum with multi-bit cursor measuring OOK/PWM symbol periods

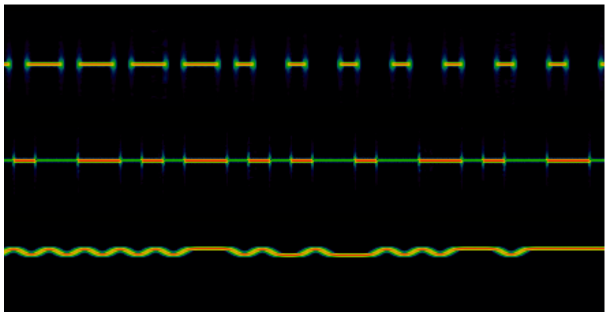


Figure 3: spectrogram comparison of OOK, ASK, and FSK, top to bottom

symbol rate.

Some modulations are easier to identify than others, but most can be determined by answering a couple of questions: Does the amplitude change over time? Does the frequency change over time?

On-Off Keying (OOK) is one of the most common modulations of low speed digital wireless systems, and it is also the easiest to identify. It looks like a train of distinct pulses, all at a constant frequency. Amplitude Shift Keying (ASK) looks similar except the intervals between pulses have low amplitude instead of zero amplitude. A Frequency Shift Keying (FSK) signal typically has constant amplitude, but it wiggles back and forth between distinct frequencies. If you suspect FSK but the frequency deviations appear blurred together, try reducing inspectrum's FFT size.

Most modulations supported by wireless transceiver ICs can be identified visually in a spectrogram. If you can't tell what the modulation is in inspectrum, it is probably FSK with a very fast symbol rate or a more sophisticated modulation that is unlikely to be supported by a general purpose wireless transceiver IC. You might have better luck capturing at a higher sample rate, but you may have to abandon the hybrid approach and use SDR alone.

At this point you should look for characteristics that indicate how data bits are encoded into raw symbols. The most trivial encoding is Non-Return-to-Zero (NRZ) in which high symbols represent a one and low symbols represent a zero. (It can be the other way around, of course, but an incorrect assumption can be fixed later by simply inverting all bits. That goes for all the encoding schemes below as well.) NRZ is a common scheme for FSK, but it is quite uncommon for OOK. One of the reasons NRZ is unpopular for OOK is that it makes a long sequence of zeros indistinguishable from a gap between packets.

A common encoding, especially for OOK, is Pulse Width Modulation (PWM) in which a short pulse represents a zero and a long pulse represents a one. Some PWM systems always use the same interval duration between every pulse, but it is more common to see long intervals follow short pulses and short intervals follow long pulses, resulting in a consistent total time per data bit.

A slightly more sophisticated but less common scheme is Pulse Interval and Width Modulation (PIWM) in which the duration of both pulses and the intervals between pulses carry data. If pulses and intervals both vary in duration but there is no correlation between the length of intervals and the length of adjacent pulses, then PIWM is likely. Unlike most modulations, PIWM does not feature a consistent time per data bit.

Another encoding scheme sometimes seen in OOK transmissions is Pulse Position Modulation (PPM) in which pulses of a consistent duration appear early to represent a zero or late to represent a one. In some cases this might look like inverted PWM, but PPM often includes unmodulated reference pulses between the modulated pulses, making it distinct from any PWM variation.

Manchester (or bi-phase) encoding is popular for FSK and is also often found in OOK and ASK systems. In Manchester encoding, a data bit is encoded by a pair of differing symbols. For example, the binary symbol pair 10 may represent a zero while 01 represents a one. You can quickly recognize Manchester encoding because it looks like NRZ but has the particular characteristic that you never see more than two like symbols in a row. It is possible for other encoding schemes to result in this characteristic, however, so it is important to check for valid Manchester. For example, the symbol stream 0110010110 (data bits 1, 0, 1, 1, 0) is valid but the symbol stream 0110110100 is not. If you see something that looks at first like Manchester but isn't valid Manchester, it might consist of Manchester encoded payloads separated by non-Manchester framing, or it might be something completely different such as PIWM.

After identifying the encoding scheme, make note of

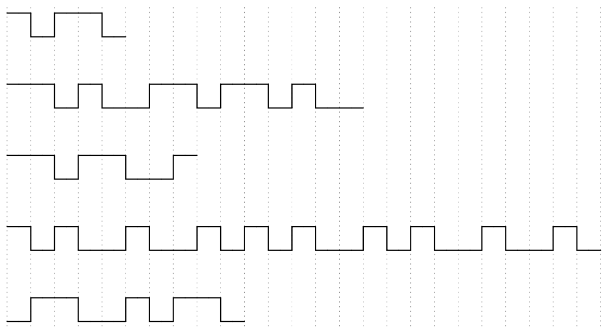


Figure 4: data bits (1, 0, 1, 1, 0) encoded with NRZ, PWM, PIWM, PPM, and Manchester, top to bottom

the difference between the symbol rate and the data rate. In NRZ, the symbol rate, also called a baud rate, is equal to the data rate. In Manchester encoding, the symbol rate is twice the data rate. PWM and other encodings are typically made up of shorter symbol periods with a consistent duration. For example, a long pulse might be encoded as the three symbols 110 while a short pulse is encoded as 100. (The word "symbol" may be used to describe the entire pulse that is three time units long, but I often find it more useful to use the word to describe the features of the encoding having the shortest observable time unit.) The multi-bit cursor in inspectrum is quite handy for this. I can usually find an integer number of symbols between three and eight that makes PWM pulses line up perfectly.

The only other thing you'll need to observe in inspectrum is how packets start and can be recognized. In most cases there will be an obvious preamble of alternating ones and zeros at the beginning of a packet. How many symbols long is the preamble? Is it made up of alternating symbols at the rate you observed earlier (e.g. where three symbols 110 make up a PWM pulse period)? If so, then you may be able to use a preamble detection feature of your wireless IC.

After the preamble, packets typically start with a synchronization word, sometimes called an access code or start of frame delimiter. Unlike the preamble, the sync word is composed of non-repeating groups of bits, although sometimes you may find that an entire sync word is repeated. Sync words give the receiver the opportunity to determine precisely which symbol contains the start of the packet data. In most systems, all transmitters and receivers use the same sync word, but occasionally you may find a system which uses a device's unique address instead of a global sync word. Either way, you'll need to manually decode the sync word once so that you can use it to configure your wireless IC to detect similar packets.

4 Receiving with a Wireless IC

Once you have made notes about the frequency, modulation, symbol rate, preamble, and sync word, you can configure your wireless transceiver IC to receive packets from the target system. If you use YARD Stick One or another RfCat-compatible wireless IC dongle, your next steps can be done rapidly in an interactive Python shell.

Plug in your YARD Stick One and start RfCat [1] with:

```
rfcat -r
```

RfCat will check to make sure that there is a compatible USB dongle plugged in, and then it will drop you into an interactive Python shell. You'll have an object called "d" that gives you an interface to control the dongle. You can access help with:

```
help(d)
```

Set the frequency in Hz with:

```
d.setFreq(314980000)
```

In this example command and others below, I am using values taken from a live demonstration I've done with a particular target device [8]. Choose values for different targets based on your analysis in inspectrum.

For OOK or ASK, set the modulation with:

```
d.setMdmModulation(MOD_ASK_OOK)
```

For FSK, instead use:

```
d.setMdmModulation(MOD_2FSK)
```

Set the symbol rate in symbols per second:

```
d.setMdmDRate(2450)
```

If a compatible preamble is not present at the start of the packets you are trying to detect, set the preamble quality threshold to zero, disabling preamble checking entirely:

```
d.setPktPQT(0)
```

Specify the synchronization word with:

```
d.setMdmSyncWord(0b0000011011011011)
```

Choosing the best sync word is often the most difficult part of the configuration. With YARD Stick One (and other platforms based on the CC11xx wireless transceiver IC), the sync word must be either 16 bits (the default) or 32 bits long, and the 32 bit mode only works as a 16 bit sync word repeated. Sometimes these modes are compatible with a target system, and sometimes they aren't. Fortunately there are some tricks you can use to handle situations where the supported modes don't exactly meet your needs.

If the target has a 24 bit sync word, for example, it is straightforward to configure only 16 out of those 24 bits as the sync word in RfCat. This increases the probability of false positive matches but otherwise works well.

If the target has a sync word shorter than 16 bits, you can configure the sync word to include some of the preamble. For example, if the target uses an eleven bit sync word of 0b11100010010, configure RfCat with a sixteen bit sync word of 0b0101011100010010. If there is no preamble prior to the sync word, you can use 0b0000011100010010 instead.

If the target does not have a consistent sync word, or if you haven't yet been able to identify it, you can employ the Goodspeed method [3] by configuring your sync word to match the preamble. For example, you might set it to 0b1010101010101010 if the preamble is at least 16 bits long. Unless the preamble is much longer than 16 bits, you may need to disable preamble quality checking when using this technique. If nothing else works, you can even disable sync word checking entirely and detect packets solely on the basis of carrier detection with:

```
d.setMdmSyncMode(SYNCM_CARRIER)
```

Some of these tricks will result in the detection of false positive packets or packets that are inconsistently aligned, but they can still be quite useful. In some cases your configuration might only detect a small percentage of packets, but that can be enough to capture a few packets from which you can learn better settings.

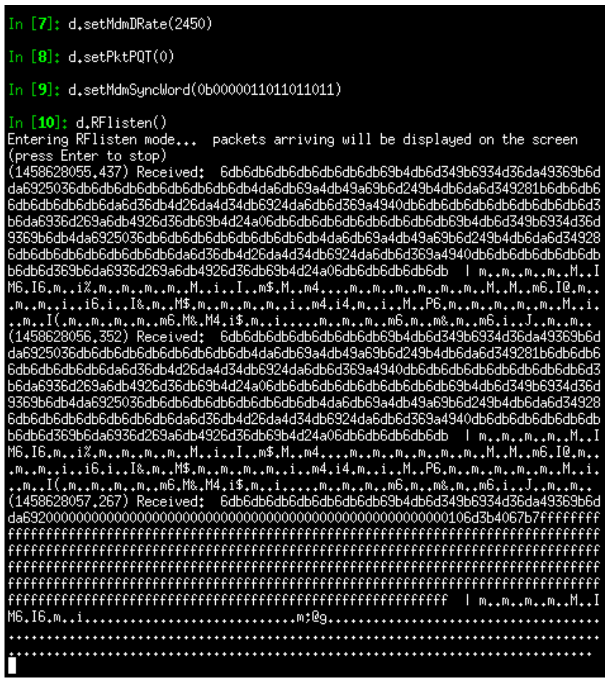
Once you have RfCat configured for your target, try detecting packets with:

```
d.RFlisten()
```

Detected packets will be dumped in hexadecimal, making it easy to compare one against another and verify that you are receiving what you intend. By default, RfCat assumes that all packets are 255 bytes long. This is longer than the packets of most targets, so you'll see some garbage (or maybe subsequent packets) at the end of each packet. Once you've learned the actual maximum packet length of your target, you can set that length in bytes with:

```
d.makePktFLEN(30)
```

At this point you should be able to reliably receive packets with RfCat, displaying the raw symbol data in hexadecimal. If the target is using an encoding such as PWM, PIWM, PPM, or Manchester, you can now make a small Python function to turn those raw symbols into bits.



```
In [7]: d.setMdmDRate(2450)

In [8]: d.setPktPQT(0)

In [9]: d.setMdmSyncWord(0b0000011011011011)

In [10]: d.RFlisten()
Entering RFlisten mode...  packets arriving will be displayed on the screen
(press Enter to stop)
(1458628055.437) Received:  6db6db6db6db6db6db6db69b4db6d349b6934d36da49369b6d
da6925036db6db6db6db6db6db6db6db4da6db69a4db49a69b6d249b4db6da6d349281b6db6db6
6db6db6db6db6da6d36db4d26da4d34db6924da6db6d369a4940db6db6db6db6db6db6db6db6d3
b6da6936d269a6db4926d36db69b4d24a06db6db6db6db6db6db6db6db69b4db6d349b6934d36d
9369b6db4da6925036db6db6db6db6db6db6db6db4da6db69a4db49a69b6d249b4db6da6d34928
6db6db6db6db6db6db6db6da6d36db4d26da4d34db6924da6db6d369a4940db6db6db6db6db6db
b6db6d369b6da6936d269a6db4926d36db69b4d24a06db6db6db6db6db  | m..m..m..m..M..I
M6.I6.m..i%.m..m..m..m..M..i..I..m$.M..m4....m..m..m..m..m..m..M..M..m6.I@.m..
.m..m..i..i6.i..I&.m..M$.m..m..m..m..i..m4.i4.m..i..M..P6.m..m..m..m..m..M..i.
..m..I(.m..m..m..m..m6.M&.M4.i$.m..i.....m..m..m..m6.m..m&.m..m6.i..J..m..m..
(1458628056.352) Received:  6db6db6db6db6db6db6db69b4db6d349b6934d36da49369b6d
da6925036db6db6db6db6db6db6db6db4da6db69a4db49a69b6d249b4db6da6d349281b6db6db6
6db6db6db6db6da6d36db4d26da4d34db6924da6db6d369a4940db6db6db6db6db6db6db6db6d3
b6da6936d269a6db4926d36db69b4d24a06db6db6db6db6db6db6db6db69b4db6d349b6934d36d
9369b6db4da6925036db6db6db6db6db6db6db6db4da6db69a4db49a69b6d249b4db6da6d34928
6db6db6db6db6db6db6db6da6d36db4d26da4d34db6924da6db6d369a4940db6db6db6db6db6db
b6db6d369b6da6936d269a6db4926d36db69b4d24a06db6db6db6db6db  | m..m..m..m..M..I
M6.I6.m..i%.m..m..m..m..M..i..I..m$.M..m4....m..m..m..m..m..m..M..M..m6.I@.m..
.m..m..i..i6.i..I&.m..M$.m..m..m..m..i..m4.i4.m..i..M..P6.m..m..m..m..m..M..i.
..m..I(.m..m..m..m..m6.M&.M4.i$.m..i.....m..m..m..m6.m..m&.m..m6.i..J..m..m..
(1458628057.267) Received:  6db6db6db6db6db6db6db69b4db6d349b6934d36da49369b6d
da69200000000000000000000000000000000000000000000000000000106d3b4067b7ffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffff  | m..m..m..m..M..I
M6.I6.m..i............................m;@g....................................
..............................................................................
.............................................................................
```

Figure 5: receiving packets with RfCat

5 Next Steps

A little Python knowledge can also help a great deal with your next challenges:

- bit order
- checksum/CRC
- encryption
- whitening
- byte alignment
- Forward Error Correction (FEC)

These challenges are very similar to those you may encounter when reverse engineering any digital communication protocol, regardless of medium. Now that you have packet data at your fingertips, you can turn to tools such as your favorite programming language to explore the target protocol.

6 Conclusion

Reverse engineers of the future may find it easier to use SDR alone rather than a hybrid approach, but it depends upon the development of better software tools for the task. We might have better spectrum monitoring software, making it even easier to detect the frequency of operation of a target system. We might have better clock recovery implementations that more reliably synchronize to symbols in packets, even if the packets have no preamble. SDR software frameworks like GNU Radio [2] might make it just as convenient to experiment with packet data as it is to manipulate stream data. Signal visualization tools like inspectrum may include functions to select and demodulate packets [13], and we may even have software that automatically classifies modulations and other parameters required for packet decoding.

In the future we may also have better software tools for reverse engineering with wireless transceiver ICs. We might have software that automatically detects modulations, for example, by simply trying every supported modulation and checking to see which yields the best results. Such tricks are certainly possible, but they will always be more limited in capability than SDR.

Eventually, SDR will likely become the tool of choice for almost every wireless reverse engineering task. Until then it is best to have multiple tools in your toolbox.

7 Acknowledgments

I thank atlas, Mike Walters, Dimitri Stolnikov, and the many other open source software developers who have made these techniques possible. Thank you to the denizens of #hackrf and #rfcat on freenode who have shared many reverse engineering challenges and even provided a waveform or two used in this paper.

References

[1] ATLAS. RfCat. https://bitbucket.org/atlas0fd00m/rfcat.

[2] FREE SOFTWARE FOUNDATION. GNU Radio. http://gnuradio.org/.

[3] GOODSPEED, T. Promiscuity is the nRF24L01+'s Duty. *Travis Goodspeed's Blog* (2011). http://travisgoodspeed.blogspot.com/2011/02/promiscuity-is-nrf24l01s-duty.html.

[4] GREAT SCOTT GADGETS. HackRF One. http://greatscottgadgets.com/hackrf/.

[5] GREAT SCOTT GADGETS. YARD Stick One. http://greatscottgadgets.com/yardstickone/.

[6] KAMKAR, S. Drive It Like You Hacked It: New Attacks and Tools to Wirelessly Steal Cars. *DEF CON* (2015). https://www.defcon.org/html/defcon-23/dc-23-speakers.html#Kamkar.

[7] OSSMANN, M. Software Defined Radio with HackRF. http://greatscottgadgets.com/sdr/.

[8] OSSMANN, M. Rapid Radio Reversing. *ToorCon* (2015). http://greatscottgadgets.com/2015/12-29-rapid-radio-reversing-toorcon-2015/.

[9] RYAN, M., AND HEALEY, R. Hacking Electric Skateboards: Vehicle Research For Mortals. *DEF CON* (2015). https://www.defcon.org/html/defcon-23/dc-23-speakers.html#Ryan.

[10] SPILL, D. fcc.io: FCC ID Search and Redirection. https://fcc.io/.

[11] STOLNIKOV, D., ET AL. GrOsmoSDR. http://sdr.osmocom.org/trac/wiki/GrOsmoSDR.

[12] WALTERS, M. inspectrum. https://github.com/miek/inspectrum.

[13] WALTERS, M. inspectrum tuner demo. https://www.youtube.com/watch?v=9QyGKjt8zkE.