Universal Radio Hacker: A Suite for Analyzing and Attacking Stateful Wireless Protocols

Johannes Pohl
University of Applied Sciences Stralsund, Germany

Andreas Noack
University of Applied Sciences Stralsund, Germany

Abstract

Proprietary wireless protocols used by IoT devices are designed under size and energy constraints, often neglecting the security. Therefore, attacks like opening wireless door locks or stealing cars are realistic threats. Software Defined Radios (SDR) propose a generic way to investigate such protocols as they can send and receive on nearly arbitrary frequencies. Most tools for SDR, however, focus on the HF side and offer little support for analyzing the actual protocol logic so custom tools or excel spreadsheets must be used. In this paper, we present the Universal Radio Hacker (URH), an open source tool which is designed for protocol analysis from the ground up and implements a full workflow including interfaces for SDRs, intuitive demodulation, customizable decodings, fuzzing support and a simulation component. URH splits the process down into the phases Interpretation, Analysis, Generation and Simulation, whereby results from one phase can be transferred to the other. The software offers all features needed for protocol investigation without overwhelming users with complexity. URH is developed with theoretic oriented researchers in mind who want to focus on protocol logic and try to avoid diving into the depths of HF and Digital Signal Processing.

1 Introduction

Internet of Things (IoT) brings comfort such as opening doors wirelessly without having to mess with mechanical keys or automatically closing roller blinds when the sun sets. Such IoT devices often communicate over proprietary wireless protocols which are designed under size and energy constraints whereby security is only a secondary factor. Weaknesses in such protocols pose critical privacy and security threats: Adversaries may, for example, track when victims leave their house and break their wireless door lock by radio without leaving traces.

Attacking a wireless protocol includes several steps that require knowledge in radio frequency (RF), coding theory, protocol design and sometimes even cryptography. Few research groups, however, include specialists from all these fields, especially a combination of applied (RF) and theoretic (cryptography) researchers is rare.

Given an unknown wireless protocol, we are interested in the transmitted bits and bytes. Thus, we record several signals with a Software Defined Radio (SDR) like HackRF [10] or USRP-N210 [6]. For further analysis, we need bit representations of the raw signals. Figure 1 shows an example for such a raw signal.

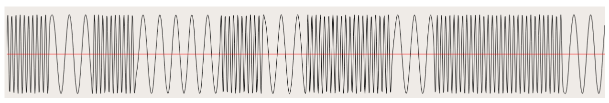


Figure 1: Sniffed signal, on 868MHz with USRP-N210

In order to extract the bit sequence `10100101101110` from this raw signal there are several steps needed:

1. Recognize modulation type (ASK, FSK, PSK, ...)
2. Demodulation to rectangular signal ...
3. Mapping of rectangular signal to bitstream
4. Optional: Remove encoding

After these steps, we concentrate on the *logical structure* of the protocol. The most promising approach is a target-oriented differential analysis of the bit representations from several signals. For example, comparing bit sequences from signals that activate and deactivate a light bulb may reveal the responsible on-off bit.

Present tools for wireless protocol investigation are specialized to certain parts of the process. Therefore, researchers need to combine applications such as GNU Radio [4] for demodulation and a wave editor like Audacity [2] to infer bits from recorded sine waves. Furthermore, manual effort or custom scripts are required to perform *logic analysis*, *apply fuzzing* or *simulate participants*.

Our main contribution is an open source software named Universal Radio Hacker (URH) which covers all aforementioned steps and is, to the best of our knowledge, the first complete suite for investigating stateful wireless protocols. The software provides abstracted and ergonomic user interfaces to comfort theoretical researchers and speed up protocol investigations. As a secondary contribution, we show attacks on three different IoT devices and how URH was used to find and perform them. The most complex attack (section 4.3) opens a wireless door lock which is protected by a challenge response procedure. The attack includes modeling the protocol state machine and extracting the encryption key from pairing messages.

2 Overview

In this section, we give an overview of the core ideas of our software and how it can be used in combination with external software for greater flexibility.

2.1 Approach

We break the investigation of partially or fully unknown wireless protocols down to three major steps:

- *Interpretation* phase covers demodulation of raw signals. The goal of this phase is to map received waves to digital information (bits).
- In *Analysis* phase bits are decoded and put into context. This phase targets to reveal the protocol logic, for example, by performing a differential analysis.
- The last phase depends on whether a stateful attack shall be performed.
  - For stateless attacks, *Generation* phase is the process of altering existing and/or generating new messages based on the revealed protocol logic. This includes *fuzzing*.
  - *Simulation* phase replaces the Generation phase for more complex protocols with multiple states. This phase allows to perform sophisticated attacks by simulating selected participants and, for example, apply a *stateful fuzzing* for certain protocol fields.

More details on each phase are given in section 3. Why is it advantageous to separate the protocol investigation into successive steps? First, the phases form a guideline that helps beginners to keep an overview. Second, experts benefit from clear phase definitions when organizing teamwork. Third, the separation gives users more flexibility when it comes to interaction with external programs.

2.2 Design principles

The development of URH is mainly driven by three design goals. First, **ergonomy** to provide an intuitive user interface which does not require deep RF knowledge, see appendix A to get an impression of this UI. Second, **functional completeness** to cover all steps necessary to investigate and attack a stateful wireless protocol - we describe this in section 3. Third, **extensibility** with external software to consider existing workflows. We achieve extensibility by offering interfaces for external programs in each phase of our design.

2.2.1 Interfaces For External Programs

We respect that experts already have software or even self-made scripts that are better suited for special cases; refer to section 5 for an overview of present tools. Our architecture, shown in fig. 2, considers this by offering *interfaces* to external programs in each phase so that certain tasks can be accomplished outside of our tool. For example, you can demodulate a signal with GNU Radio and pass the bits to our software in order to perform the Analysis phase afterwards. In our approach, we distinguish two ways of interacting with other software. First, for *static* investigations we use standard file formats for raw signals and binary data. Second, a socket based network interface enables *dynamic* exchange with other tools.

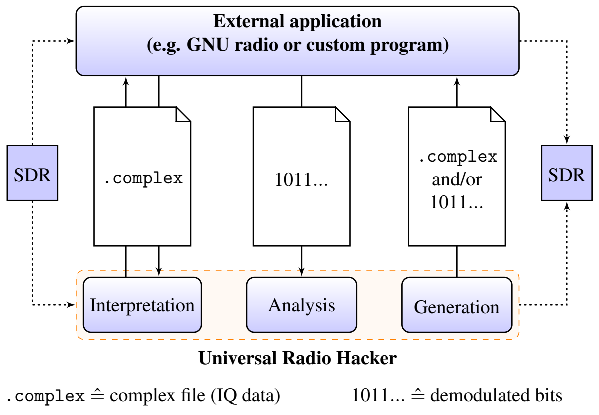


Figure 2: Architecture: Applications can work with URH using open file formats for raw samples and bits. For dynamic investigations, a socket based interface supports both receiving from and sending to external tools.

2.2.2 Device Backends

As shown in fig. 2 our software communicates with SDRs during Interpretation and Generation. We provide three distinct *device back-ends* to interface SDRs: native, external and network.

The *native device back-end* is a Cython [3] wrapper of the respective SDR driver. The advantage is that we have full control and can optimize it for our software. Furthermore, no additional dependencies are required for this back-end. Diving into the SDR drivers also gives us more insight and helps to optimize device configurations. Native support is available for a wide range of commonly used SDRs including HackRF, USRP and RTL-SDR. For users with unsupported SDRs it is, however, a laborious task to create a native wrapper on their own.

Therefore, our approach also covers *external back-ends* which work by interfacing external applications like GNU Radio or RfCat [1]. The most important external back-end is for GNU Radio that is implemented with simple Python files using GNU Radio device blocks and a ZeroMQ sink and source for sending and receiving, respectively. In case our software lacks support for a certain SDR, users may implement a GNU Radio back-end on their own as this is straightforward. A general drawback is that the respective external application needs to be installed for the back-end to work.

The *network back-end* is built on *TCP sockets* to communicate with applications in a generic way. It is not dedicated to a certain device and can be used in various cases. For example, you can receive samples from a remote system, or send the binary protocol to an external program that implements a state machine.

3 Universal Radio Hacker

The Universal Radio Hacker (URH) is the realization of our approach. URH is a cross platform application (Linux, Windows, OSX) implemented in Python and Cython/C++ for performance critical parts. In this section, we explain the phases Interpretation, Analysis, Generation and Simulation in a detailed but rather abstract way. Refer to appendix A for a visual breakdown of the user interface.

3.1 Interpretation

The Interpretation phase aims to extract binary information (bits) from the received waveform signals. In order to do this, data needs to be *demodulated* in the right way. In this section, we sketch how URH performs this demodulation behind the scenes.

3.1.1 Demodulation

If you record a wireless transmission and view its waveform in a wave editor like Audacity, it strongly depends on the used *modulation type* how the signal appears. We want to keep the demodulation process as uniform as possible, but since each modulation transports information with a different signal parameter, namely amplitude, frequency and phase, we have to use some kind of pre-processing for the selected modulation. Our demodulation pre-processing, no matter what modulation is used, results in a rectangular signal like the one in fig. 3. Next, we explain the pre-processing algorithms for Amplitude Shift Keying (ASK), Phase Shift Keying (PSK) and Frequency Shift Keying (FSK).

ASK pre-processing The pre-processing of ASK is simple, because we only need to evaluate the absolute values of the complex samples $x_c(n)$ [9, chap. 9.2]

$$R(n) = |x_c(n)| = \sqrt{x_r(n)^2 + x_i(n)^2}, \tag{1}$$

whereby $x_r(n)$ is the real and $x_i(n)$ the imaginary part. Performing this for all samples in the signal yields the rectangular curve $R(n)$.

PSK pre-processing We use a *Costas Loop* for PSK pre-processing. The idea behind Costas Loop is to use a numerical controlled oscillator to approximate the carrier signal and get the current phase shift by evaluating the phase difference between the current sample and the current carrier value. For further readings refer to Feigin [7] who gives a deep insight into Costas Loop.

FSK pre-processing FSK uses different frequencies for transporting information. Following [9, chap. 9.2] the instantaneous frequency $F(t)$ can be derived using

$$F(t) = \frac{d}{dt}\Phi(t), \tag{2}$$

with $\Phi(t) = 2\pi f t + \Theta$ being the *instantaneous phase* of the signal. As we sample at discrete timestamps n we can approximate $F(n)$:

$$F(n) = \Phi(n) - \Phi(n-1) \tag{3}$$

$$F(n) = \arctan\left(\frac{x_i(n)}{x_r(n)}\right) - \arctan\left(\frac{x_i(n-1)}{x_r(n-1)}\right) \tag{4}$$

where $x_r(n)$ are the real and $x_i(n)$ the imaginary parts of the complex samples $x_c(n)$. In case of FSK, instantaneous frequencies $F(n)$ form the rectangular signal $R(n)$.

The rectangular signal in fig. 3 is created from the FSK modulated signal in fig. 1 with FSK pre-processing. Pulses in the upper area will be interpreted as binary one, in the lower area as binary zero. The border between both areas is automatically detected by URH, but can also be adjusted manually to correct noisy data, if necessary. The automatic detection works by first performing a k-means clustering on the rectangular signal and, subsequently, calculate the average of the two most common center points. Given the rectangular pulse sequence and the border between areas, URH extracts the correct bit sequence `10100101101110`.

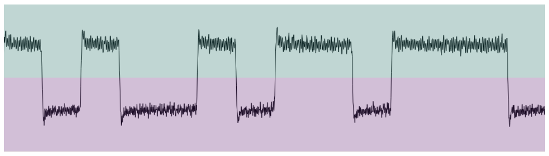


Figure 3: Signal from fig. 1 after FSK pre-processing

3.1.2 Control parameters

One benefit of URH is that you instantly see the resulting bits, when changing any of the modulation parameters. The Interpretation phase has four control parameters:

1. *Noise level* for suppressing noise. This is useful for recordings under non-ideal conditions.
2. *Center* means the border between ones and zeros, indicated by different background colors in fig. 3.
3. *Bit length* is the number of samples for one raw bit.
4. *Modulation type* whereby currently binary ASK, FSK/GFSK and PSK are supported.

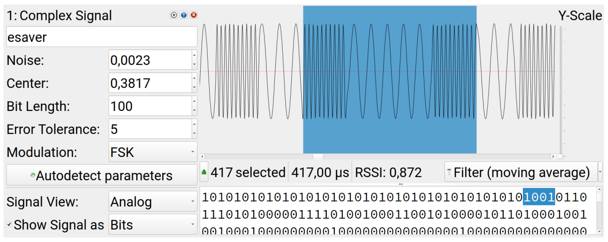
Note, that default values for these parameters are chosen by URH automatically based on heuristics, for example, the noise level is initially set to the average magnitude of the last 10% samples of the signal, as most recordings end with a pause. Therefore, this phase can usually be completed quickly for most signals.

3.1.3 Advanced features

Some signals may be harder to demodulate e.g. due to quantization errors induced by the SDR. Other devices may use *multiple channels* for communication which need to be separated before demodulation. URH offers some advanced features to deal with this and debug errors in the demodulation process.

Synchronization between data and signal Whenever a more closer look to a signal is required, URH allows zooming and even editing the captured signal. Furthermore, whenever a selection is made in signal or demodulated data the corresponding other part is also selected automatically as shown in fig. 4. This is helpful when it comes to debugging demodulation errors or get a better understanding of the signal.

Digital filters to improve signal quality Some SDRs may produce low quality captures due to quantization errors. URH allows filtering a signal to improve demodulation results. Filter parameters and types can be configured in a custom dialog.


1: Complex Signal

esaver

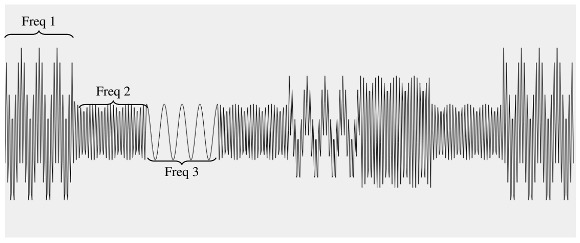
Noise: 0,0023
Center: 0,3817
Bit Length: 100
Error Tolerance: 5
Modulation: FSK
Signal View: Analog
Show Signal as: Bits

417 selected 417,00 μs RSSI: 0,872

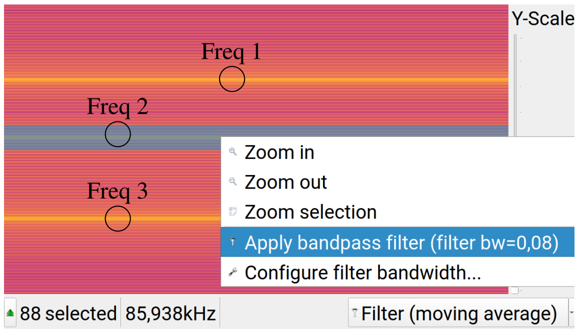
```
101010101010101010101010101010101010101010010110
111010100000111101001000110010100001011010001001
001000100000000001000000000000000010000000000000
```

Figure 4: Synchronized signal and data selection

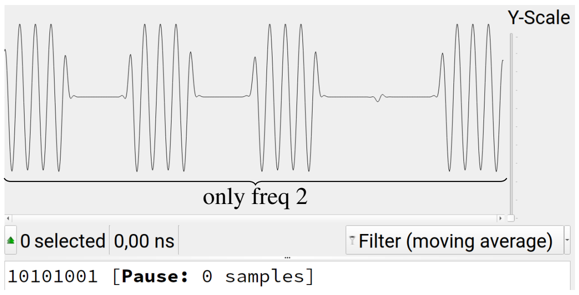
Spectrogram view and bandpass filters Advanced protocols may use different channels for transmitting data. Demodulation of a signal carrying multiple channels is hard at first sight as shown in fig. 5a. For this reason, URH supports a *spectrogram view* to identify and, furthermore, a customizable bandpass filter to extract channels as shown in fig. 5b.



(a) Signal with multiple data channels



(b) Identify and extract channels in spectrogram view



```
10101001 [Pause: 0 samples]
```
(c) Resulting signal can easily be demodulated

Figure 5: Example of signal with multiple channels and corresponding spectrogram

3.2 Analysis

Once we have obtained raw bits from the investigated wireless protocol, the next phase is analysis of the actual protocol logic. Reverse engineering a protocol is mainly a thinking process that strongly depends on the experience and methods of the individual research group. Our contribution to this process is a set of helpful tools and recommendations based on heuristics. In the following we introduce some of these features. For a first overview, fig. 6 shows the steps of Interpretation and Analysis.

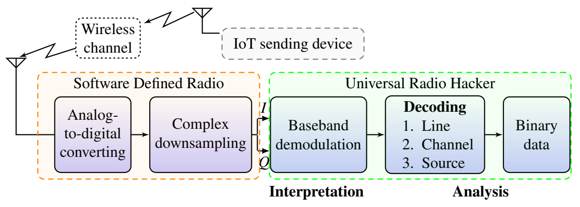


Figure 6: Overview of Interpretation and Analysis

3.2.1 Encodings

Wireless protocols can have various encodings which need to be removed before obtaining the actual bits. Our software offers *predefined encodings* and, additionally, enables users to build encodings using an *object based approach*: From a set of encoding primitives, such as *inverting* or *edge triggering*, any number of instances can be created and connected in arbitrary order. Furthermore, URH offers an API for external decoding programs for an even greater flexibility. Having selected or built an encoding, there are several ways to check whether this is the right encoding for your signal. First, some encodings do not allow special bit sequences, e.g., Manchester $\nrightarrow$ `000` or `111`, hence some encodings can be ruled out. Second, different encodings lead to different output lengths. If the protocol includes a length field, you can easily prove the decoded message length. Third, in case the protocol entails a checksum the correct encoding can be verified by recalculating the checksum value. For convenience reasons, URH displays the decoding errors for each message separately and offers capabilities for verifying checksums.

3.2.2 View, search and highlight data (VSH)

After decoding, the next step is to assign context to the bits and, finally, reveal the protocol logic. Different representations like bit, hex or ASCII view help to identify certain protocol fields, for example, a preamble on the physical layer, MAC addresses or a cyclic redundancy checksum (CRC). Moreover, some protocol elements simply become better visible with the right view type, e.g., `01010101` vs. `0xBEEF`.

An important approach is to bring different protocol messages in relation to each other. When comparing wireless protocol messages, a *differential view* will instantly disclose bit ranges that stay constant and bit ranges that differ. Together with context information such as: the same devices/addresses were used or messages were recorded in sequence, further information about the meaning of certain bit ranges can often be derived. These ranges may eventually turn out as addresses, sequence numbers, length fields or serial numbers.

If you have prior knowledge about the investigated protocol messages, e.g., a wireless thermometer showing 21 degree Celsius, it is a promising approach to *search* for specific values in the data. This search should be available for different views on the raw bits like decimal, hexadecimal or ASCII. Furthermore, it is sometimes even valuable to highlight every appearance of a specific value, for example, to verify an assumption about a binary delimiter. As well, *hiding* already revealed or unimportant parts of the protocol makes it easier to focus on relevant parts.

The Universal Radio Hacker incorporates all view, search and highlight features in its Analysis component. URH aligns recorded messages under each other so that differences can easily be seen. For structuring purposes, you can even organize your messages in groups and toggle their visibility.

3.2.3 Bringing messages into context

By reviewing the protocol or deploying URHs automatic logic analyzer you gain knowledge about the position of fields like serial numbers or addresses. When you attach text labels to particular positions, our software assigns a specific color and adds the fields to a *Wireshark-like preview*. Each value in this preview can have a different view type, for example, a hexadecimal view is suited for addresses while length values and counters are most obvious in a decimal view.

Knowing which *participant* sends a protocol message is essential for investigating complex wireless protocols. Our software allows assigning participants in Interpretation and Analysis phase *by hand* or *automatically* based on the Received Signal Strength Indicator (RSSI) of a message.

Larger protocols tend to have different *message types* such as *data* or *ACK* messages that have their own fields and semantics. URH allows creating, deleting and assigning message types to messages. This can either be done manually or automatically based on configurable rules, for instance, *if byte on position 7 is 0x42 and message length equals 21 then assign message type* ACK. In summary, Interpretation and Analysis are tied closely together and ideally result in an estimation of the protocol logic.

3.3 Generation

After finishing Interpretation and Analysis you have hypotheses for the position of protocol fields maybe even for the complete protocol. To test these hypotheses you can generate according data and watch how the target device reacts on it. You may also modify certain parts of the protocol like addresses or type codes to induce errors or unusual behavior. All these steps form the *Generation* phase. The Universal Radio Hacker offers two ways for changing messages: First, you can manually edit single bits or nibbles. This is a good starting point for verifying the examined logic. For example, if you captured a message from a light bulb remote that turns this light bulb on, you could test for the on-off bit in the message. Second, URH comes with a fuzzing component.

The *fuzzing component* is designed to perform more sophisticated protocol editing. Assume, you want to fuzz an eight bit counter. Thanks to the fuzzing component, you do not need to edit 256 messages by hand, you just specify the counter position in the protocol, enter an ascending range from 0-255 and hit the button. Besides iterative fuzzing, URH also supports to pick random values from a finite set and to test the numerical limits of a field, i.e. 0 and $2^{16} - 1$ for 16 bit. When you have created some fuzzing messages, URH applies the chosen encoding and performs the selected modulation like ASK or FSK as shown in fig. 7. Aside from saving the fuzzing messages in a complex standard format, they can be transmitted via common SDRs directly.

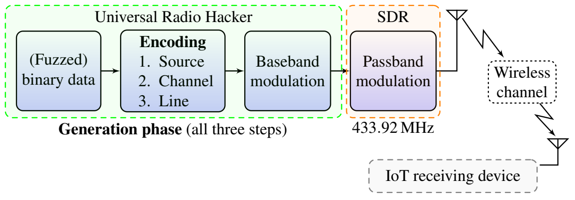
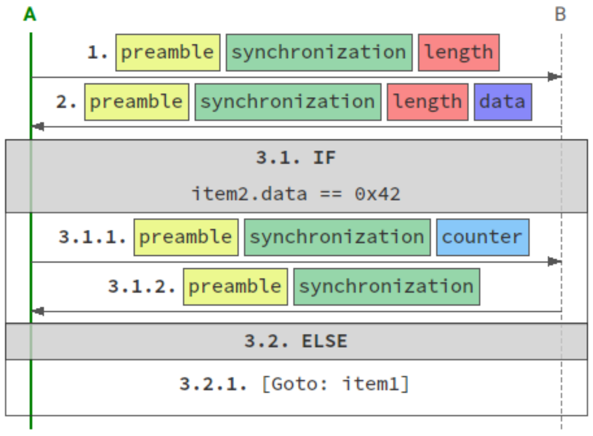


Figure 7: In Generation, manipulated bits are encoded and modulated to communicate with the target.

3.4 Simulation

A major limitation of the Generation phase is that stateful protocols cannot be attacked since some values like sequence numbers or cryptographic challenges need to be determined and manipulated live during interacting with the devices. In order to address this issue, URH offers a Simulation tab. The simulator of URH is mainly configured through a flow graph which also shows a visual representation of the protocol flow. See fig. 8 for an example with two participants *A* and *B*. At a glance, the flow of the protocol becomes apparent. Here we simulate *A* and keep sending messages to *B* until *B* responds with a message that transports the `data 0x42`. The number and names of participants are fully customizable to match the specific scenario.



A
B
1. preamble synchronization length
2. preamble synchronization length data
3.1. IF
item2.data == 0x42
3.1.1. preamble synchronization counter
3.1.2. preamble synchronization
3.2. ELSE
3.2.1. [Goto: item1]

Figure 8: Example flow graph of a simulation

The flow graph is a high level view on the protocol as it only shows the fields of messages and not the underlying bits. These bits can be inspected or changed in another sub tab, if necessary. At this stage, however, you mainly want to operate on the logical level so this representation helps to keep an overview. The flow graph is interactive, i.e., users can add or delete messages or, for example, change source or destination of a message. Messages can be bootstrapped based on the labels that were assigned in Analysis (section 3.2) or created from scratch.

The simulator also offers an API for external programs, which will get the message bits that were sent and received until calling the external program, i.e the partial transcript. Results of external programs can be accessed in the simulator and, e.g., be inserted as value for a protocol label to deal with encrypted data.

As also shown in fig. 8, our simulator supports conditions. This way, URH can react to certain events at simulation time such as the return codes of external commands or values of labels. Furthermore, you can add `Goto` items which allow jumping back and forth to arbitrary places in the flow graph. This way, loops become possible and you can, for example, wait for a certain message to be sent and, ultimately, model state machines.

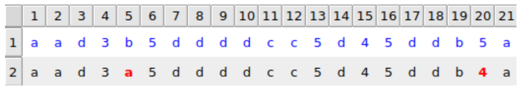
Values for labels may either be static or learned live during simulation. One may also refer to the value of previous labels with formulas, e.g., set the sequence number of message 4 to `item3.sequence_number + 1`. Combined with the feature to learn the values of labels live during simulation, this enables modeling protocol state machines and develop more sophisticated attacks than in Generation phase.

4 Practical Examples

As a secondary contribution, we show three examples how URH assisted us finding weaknesses in IoT protocols whereby the first two examples are rather simple. The third example is a more complicated attack and uses our simulator to generate a response for a cryptographic challenge. We informed manufacturers and assured time for fixing, so we will not mention product names.

4.1 Example 1: Destroy a device wireless

We acquired \$150 wireless sockets (WS) and a compatible remote control implementing the EnOcean protocol. While analyzing their wireless protocol, we found out that we can physically destroy the wireless sockets with nothing more than a SDR in receiving range. Details about the hacking history are presented in the following: We start by investigating whether the WS is prone to *replay attacks*. Therefore, we sniff the process of switching it on and off using the remote control. Figure 9 shows the corresponding messages in hex format. Note the differing hex values in column 5 and 20. The on-off information is located here and is protected by some kind of redundancy, that is, a cyclic redundancy checksum (CRC), for detecting and eliminating transmission errors.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	a	a	d	3	b	5	d	d	d	d	c	c	5	d	4	5	d	d	b	5	a
2	a	a	d	3	a	5	d	d	d	d	c	c	5	d	4	5	d	d	b	4	a

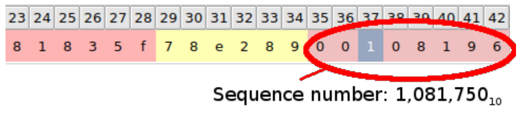
Figure 9: Messages for on (row 1) and off (row 2)

The protocol does not seem to include any real protection like counters or encryption. Therefore, we feed the data from fig. 9 into our Generator (section 3.3) and can watch the WS socket switching – replay attack works! Even when rapidly pressing the remotes on-off buttons manually, we could measure a pause of at least 150 ms between the on and off message. We use our program to shorten this pause to 5 ms. We enqueue this manipulated sequence for sending in an infinite loop using a HackRF (SDR) and can watch the WS switching certain times. However, after a while the WS stops switching. For further investigations we turned the HackRF off, but we could not switch the WS with the remote control anymore. It could not even be reset or switched with the physical button on the WS itself or by disconnecting it from the power supply. The socket was broken, a classical warranty case. Our hope that we just had a defective socket was wiped out by the replacement devices. It is critical when \$150 IoT devices are not designed to withstand wireless attacks, but the risk is even greater when it is possible to physically destroy devices from distance.

4.2 Example 2: Device desynchronization

Our second example is a smart home system involving a central unit (CU) that registers, manages and switches authenticated devices. The CU and its paired devices communicate via a proprietary modification of the BidCos protocol. We can not perform a simple replay attack because messages include a sequence number and the communication is AES encrypted. Furthermore, the protocol uses *data whitening* [5] which consists of a PN9 generator that produces a pseudo random bit stream that is XOR'd with the cleartext data. Consequently, we need to apply a de-whitening beforehand.

We found out that we can *desynchronize* certain devices from the central unit (CU). To illustrate this, a captured message from the CU to a wireless socket (WS) is partially shown in fig. 10. The message part starts with three byte destination address `0x81835f` of the WS, followed by three byte source address `0x78e289` of the CU and ends with a 4 byte sequence number `0x00108196`.



23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
8	1	8	3	5	f	7	8	e	2	8	9	0	0	1	0	8	1	9	6

Sequence number: $1{,}081{,}750_{10}$

Figure 10: Relevant protocol part for desync attack

Among other link layer information, such as type, length and device addresses, the sequence number is transmitted in plain text. We supposed that all link layer information are (integrity) protected by a MAC-like primitive in the AES encrypted part. Nevertheless, we incremented nibble 37 by one, that is, we increased the sequence number by about a million, and injected the message again. Subsequently, the WS could not be switched from the CU anymore and was indicated as unreachable due to "wireless problems". We assume, that certain devices (WS in this case) increase their internal sequence number *before* checking the integrity of the unencrypted protocol part. To validate our assumption, we put the WS back to factory settings and performed the attack again. This time, we increased the sequence number only by 8 and eavesdropped the regular communication attempts between the CU and WS. The devices were synchronized again after eight messages were sent. The same attacking message can be taken for desynchronizing other device connections by changing the source address present in nibbles 29-34 from the message part in fig. 10.

This protocol behavior is a serious vulnerability for a smart home system. Attacked devices become practically unusable by inserting sufficiently high values as sequence number. Only with a factory reset and re-attaching them, devices can be switched again. Non-professional users will probably make a warranty case of it when they observe ongoing "wireless problems". Hence, manufacturers reputation may suffer from the demonstrated device desynchronization attack when customers consider their devices as broken.

4.3 Example 3: Break a door lock wireless

For demonstration of the simulator, we sketch an attack on a wireless door lock which is paired with a smart home central. The communication between door lock (*D*) and smart home central (*C*) uses a challenge response authentication with AES. The AES encryption key can be configured through a Web UI in the smart home central. Note, the following attack allows hackers to break into houses with SDRs and, since it is based on a protocol flaw, there is no easy way to fix this for the manufacturer. Therefore, we will not mention the product name and focus on the role of the simulator in this attack.

All messages in the protocol are *pseudo-encrypted* by XORing each byte with its predecessor while the first byte is XOR'd with a well-known constant. Although this does not strengthen the protocol security, it increases the hurdle for attacking it with a SDR. Moreover, all messages are encoded with a *data whitening* [5] to optimize them for radio transmission. So when receiving a message during simulation, URH performs the following steps in the background **Demodulation** ⇒ **De-Whitening** ⇒ **Pseudo-Decryption**. Obviously, when sending a message during simulation these steps are performed in reverse order and replaced by their counterparts.

In fig. 11 you can see the flow graph that was used for this attack. We will simulate participant *A* for attacker. The first action is to trigger an external program which looks if the AES key was already found by our tool. If not, it will wait for messages matching the structure of message 3. These messages transport parts of the AES key to new devices and are always sent when a new device is paired with C, even when this new device is configured to not (!) use encryption. So at this stage, we simply wait for a new device to pair and then extract the AES key from the key transport messages. Once the key was extracted, the simulation goes on to send message 6 which includes the *open* command. This message will trigger the door lock to send message 7 which includes a cryptographic challenge. Message 8 is then generated by the simulator using this challenge and the previously extracted AES key to calculate the correct response. After this, the door will open and send message 9 for confirmation which we just included for completeness in the flow graph. It is not required for the attack. Breaking such a critical device demonstrates the power of the simulator and the need for more secure IoT protocols: With just two SDRs a hacker can open a door lock and enter a victims house, given the hacker waits long enough until victim pairs a new device.

5 Discussion and Related Work

Security of IoT systems is a hot topic of international research groups. There are strong security analyses, for instance, Garcia et al. [8] dealt with the security of smart cards for wireless payment or Strobel et al. [15] broke a digital locking and access control system that is widely used in larger companies and universities. Both studies involve opening the device hardware to reverse engineer the wireless protocol by connecting the wireless chip to a logic analyzer. We would like to enable less hardware oriented researchers access to these analyses. Software Defined Radios are the answer to this problem, because they can send and receive on nearly arbitrary frequencies. But SDRs have to be operated as well: software is needed to process received data or to prepare it for sending.

GNU Radio [4] is the de facto standard for this task. Flow graphs based on hardware-related building blocks, that allow processing of analog and digital signals and interacting with SDRs, can be created within a GUI. As GNU Radio works on a quite low level, knowledge about Digital Signal Processing (DSP) is necessary to get the right information out of wireless signals. You might save a recorded signal as a wave file (`.wav`) for further processing, since raw complex files cannot be handled by most analyzing applications. Audacity [2] is a wave editor that can be used to manually extract the raw bits from the sine waves, laborious but easy. A direct approach within GNU Radio is using a *binary slicer* which can output bits directly. Flow graphs become complex, are not trivial to create and specific for the investigated protocol, in short, not suited for DSP beginners.

Zillner [16] found serious security leaks in ZigBee using *SecBee*. SecBee [16], based on scapy-radio, is a specialized solution for testing the ZigBee protocol with SDRs. Zillner's great contribution is limited to the ZigBee protocol and cannot be used for other protocols. Another specific solution is *Ubertooth* [11], based on GNU Radio, that implements a partial Bluetooth stack and enables interacting with Bluetooth devices using a SDR. Further research has been done on this project by Ryan [14], but as well as SecBee, it is limited to a particular application.

Generic wireless protocol analyzers like URH are rare. To the best of our knowledge, there are only two generic analyzers available that are comparable to our solution: GNU Radio and scapy-radio. Picod et al. [12] developed *scapy-radio* that combines GNU Radio and the Python scapy library for monitoring wireless protocols. Their application is supposed to be used for security assessments. Since their solution is script based, you can also conditionally inject packets and thereby simulate participants. However, scapy-radio requires a complete GNU Radio flow graph for demodulating and decoding the examined protocol.

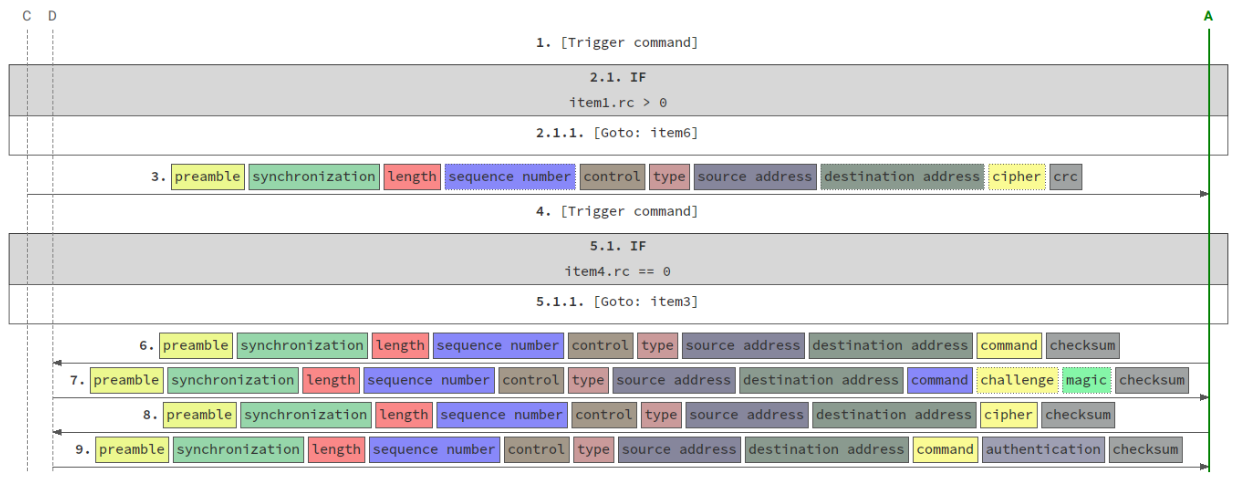

C
D
A
1. [Trigger command]
2.1. IF
item1.rc > 0
2.1.1. [Goto: item6]
3. preamble synchronization length sequence number control type source address destination address cipher crc
4. [Trigger command]
5.1. IF
item4.rc == 0
5.1.1. [Goto: item3]
6. preamble synchronization length sequence number control type source address destination address command checksum
7. preamble synchronization length sequence number control type source address destination address command challenge magic checksum
8. preamble synchronization length sequence number control type source address destination address cipher checksum
9. preamble synchronization length sequence number control type source address destination address command authentication checksum

Figure 11: Flow graph of an attack to open a wireless door lock.

We believe, that this is a significant hurdle for theoretic researchers. For this reason, our tool abstracts most of the DSP components and requires very little knowledge about DSP in general. In contrast to scapy-radio, our contribution does not require any preliminary information, such as demodulation or encoding, about the protocol. Due to the generalist approach of our contribution, it is considerably better suited for unknown and proprietary protocols. URH also contains a simulation environment that can be used to simulate protocol participants. This feature may be helpful for the wireless security community, as it allows observing and interfering with complete protocol runs using own parameters. With very little effort, chosen plaintext and ciphertext attacks are possible.

All in all, the Universal Radio Hacker has some similarities with GNU Radio, especially in Interpretation, but differs in many other ways and targets a different audience and focuses on the logical protocol level. The most notable differences are that URH offers assistance for protocol analysis, can generate and check CRCs, includes customizable decodings, entails a fuzzing component and comes with a simulation environment.

6 Conclusion

In this paper, we have introduced wireless hacking using Software Defined Radios. Our main contribution is an open source software named *Universal Radio Hacker* dedicated to wireless protocol investigation. URH is, to our knowledge, the first software to provide a complete suite for investigating stateless and stateful wireless protocols.

- It breaks down DSP including demodulation and filtering of raw signals to accessible GUI dialogs. We refer to this as *Interpretation* phase.
- It supports logic analysis of protocols and provides enhanced features like checksum generation besides full customizable decodings. We call this step the *Analysis* phase.
- It includes a fuzzing component for efficiently finding vulnerabilities and applies modulation and encoding to crafted messages. We call this *Generation*.
- For more complex protocols that use, e.g., cryptographic handshakes it allows simulating participants and, therefore, dynamically react to messages.

The new software has a modular architecture including a plugin system for easy extensions to address even more sophisticated problems in the future. Interfaces for common SDRs and a standard file format allow an exchange with existing hard- and software. Hence, researchers familiar with GNU Radio or DSP experts can benefit from URH without fundamentally changing their existing workflows. While abstracting complex mathematics, URH is intended to be easy accessible, so we focused on making it *ergonomic* and *intuitive*. The *Universal Radio Hacker* including source code and documentation can be found at GitHub [13]. Refer to appendix A to get an impression of URHs graphical user interface.

Besides our main contribution, we gave three practical examples of wireless protocol hacking. First, we showed a wireless signal that physically destroys hardware of an international manufacturer. Second, we desynchronized a device from its control unit. Third, we broke the protection of a wireless door lock.

Future work is the enhancement of the automatic logic analyzer of URH so protocol labels like `preamble` and `length` can be bootstrapped with the click of a button. Likewise, further disclosures of IoT security vulnerabilities using the *Universal Radio Hacker* are planned.

References

[1] Atlas. *RfCat*. https://bitbucket.org/atlas0fd00m/rfcat. 2017.

[2] Audacity Team. *Audacity®*. http://www.audacityteam.org. 2016.

[3] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. "Cython: The best of both worlds". In: *Computing in Science and Engineering* 13.2 (2011).

[4] E. Blossom. "GNU radio: tools for exploring the radio frequency spectrum". In: *Linux journal* 2004 (2004).

[5] B. G. Christiansen. *Design Note DN509 Data Whitening and Random TX Mode*.

[6] Ettus. *USRP N210*. https://www.ettus.com/product/details/UN210-KIT. 2016.

[7] J. Feigin. "Practical Costas loop design". In: *RF signal processing* January (2002).

[8] F. D. Garcia, G. De Koning Gans, R. Muijrers, P. Van Rossum, R. Verdult, R. W. Schreur, and B. Jacobs. "Dismantling MIFARE classic". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5283 LNCS (2008).

[9] R. G. Lyons. *Understanding Digital Signal Processing*. Third. 2011.

[10] M. Ossmann. *HackRF One – an open source SDR platform*. https://greatscottgadgets.com/hackrf/. 2016.

[11] M. Ossmann, D. Spill, M. Ryan, W. Code, and J. Boone. *Ubertooth – open source wireless development platform suitable for Bluetooth experimentation*. https://github.com/greatscottgadgets/ubertooth. 2015.

[12] J.-m. Picod, A. Lebrun, and J.-c. Demay. "Bringing Software Defined Radio to the Penetration Testing Community". In: *Black Hat USA* 2014 (2014).

[13] J. Pohl and A. Noack. *Universal Radio Hacker*. https://github.com/jopohl/urh. 2018.

[14] M. Ryan. "Bluetooth: With Low Energy Comes Low Security." In: *WOOT*. 2013.

[15] D. Strobel, B. Driessen, T. Kasper, G. Leander, D. Oswald, F. Schellenberg, and C. Paar. "Fuming acid and cryptanalysis: Handy tools for overcoming a digital locking and access control system". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8042 LNCS.PART 1 (2013).

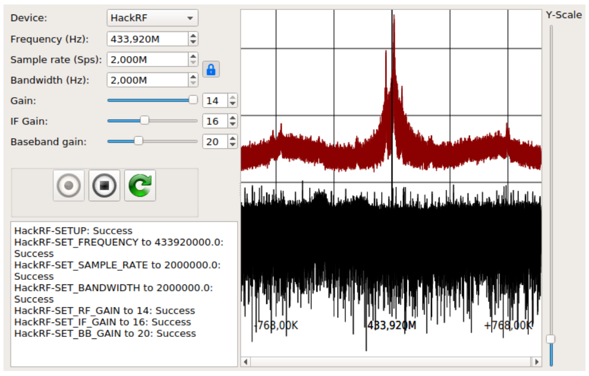
[16] T. Zillner. "ZigBee Exploited - The Good, the Bad and the Ugly". In: *Black Hat USA* 2015 (2015).

A Appendix – UI breakdown

In this section, we show important UI elements of the Universal Radio Hacker to support section 3 and demonstrate how the abstract concepts look in reality.

A.1 Device Interaction

The following dialogs enable interaction with SDRs. With the spectrum analyzer from fig. 12 you can find the center frequency of the investigated device. The dialog is interactive, that is, clicking on a certain point will let URH tune the SDR to the corresponding frequency.

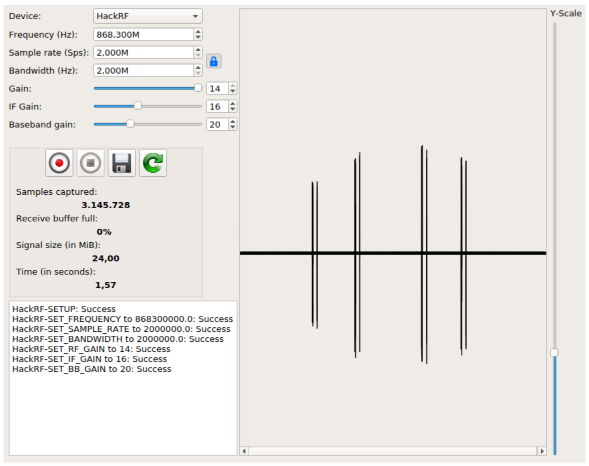


Device: HackRF
Frequency (Hz): 433,920M
Sample rate (Sps): 2,000M
Bandwidth (Hz): 2,000M
Gain: 14
IF Gain: 16
Baseband gain: 20

```
HackRF-SETUP: Success
HackRF-SET_FREQUENCY to 433920000.0:
Success
HackRF-SET_SAMPLE_RATE to 2000000.0:
Success
HackRF-SET_BANDWIDTH to 2000000.0:
Success
HackRF-SET_RF_GAIN to 14: Success
HackRF-SET_IF_GAIN to 16: Success
HackRF-SET_BB_GAIN to 20: Success
```

Figure 12: Spectrum analyzer with maximum in red

Having found the target frequency, the next step is to receive sample signals. This can be done right from URH with the receive dialog from fig. 13. A preview of the received signal is drawn live so you instantly see the order and strength of received signals at first sight.



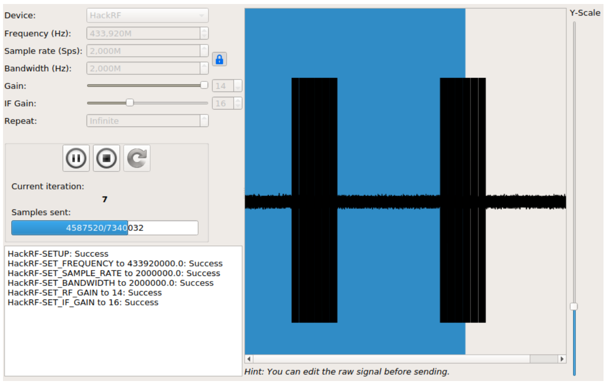
Device: HackRF
Frequency (Hz): 868,300M
Sample rate (Sps): 2,000M
Bandwidth (Hz): 2,000M
Gain: 14
IF Gain: 16
Baseband gain: 20

Samples captured: **3.145.728**
Receive buffer full: **0%**
Signal size (in MiB): **24,00**
Time (in seconds): **1,57**

```
HackRF-SETUP: Success
HackRF-SET_FREQUENCY to 868300000.0: Success
HackRF-SET_SAMPLE_RATE to 2000000.0: Success
HackRF-SET_BANDWIDTH to 2000000.0: Success
HackRF-SET_RF_GAIN to 14: Success
HackRF-SET_IF_GAIN to 16: Success
HackRF-SET_BB_GAIN to 20: Success
```

Figure 13: Receive dialog with live signal preview

At certain stages during protocol investigation, signals need to be sent back to the target. This might be in Interpretation to perform a replay attack or at the end of Generation to send manipulated messages. Sending is done with the dialog shown in fig. 14. The signal in this dialog is editable so users can delete or mute parts or add configurable sine waves before sending. The blue progress bar indicates the current sent signal part so users can watch the sending progress. This is useful to identify messages that trigger desired actions on the target device.



Device: HackRF
Frequency (Hz): 433,920M
Sample rate (Sps): 2,000M
Bandwidth (Hz): 2,000M
Gain: 14
IF Gain: 16
Repeat: Infinite
Current iteration: 7
Samples sent: 4587520/7340032

```
HackRF-SETUP: Success
HackRF-SET_FREQUENCY to 433920000.0: Success
HackRF-SET_SAMPLE_RATE to 2000000.0: Success
HackRF-SET_BANDWIDTH to 2000000.0: Success
HackRF-SET_RF_GAIN to 14: Success
HackRF-SET_IF_GAIN to 16: Success
```

Figure 14: Send dialog with current send indicator

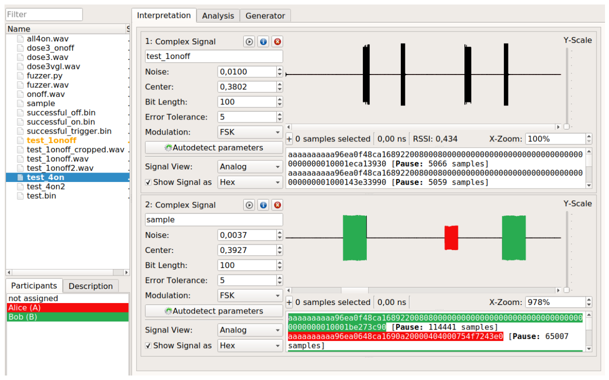
A.2 Interpretation Tab

After recording signals, they are automatically added in the Interpretation tab as shown in fig. 15. Parameters like noise, center, bit length and error tolerance are detected automatically but can also be adjusted by the user. Additionally, users can choose from a set of modulations. For a better understanding of the signal the user can switch between the analog view and the preprocessed rectangular curve of the signal (section 3.1.1) that in most cases gives a better impression of the signal and helps to confirm the modulation type.

In fig. 15 you can see that messages of the lower signal are colored differently. The colors indicate *participants* of the protocol. The participants are automatically detected based on the Received Signal Strength Indicator (RSSI) but can be changed or set by hand if necessary.

Every adjustment influences the generated preview of demodulated binary data visible at the bottom of fig. 16. Furthermore, selection of demodulated data is synchronized with the raw signal as shown in fig. 16. This way, users can conveniently explore the signal.

Each signal is also *editable*, that is, you can delete or mute selections or zoom into interesting parts of the signal. Moreover, users can copy and paste signal areas or even insert custom sine waves using the dialog from fig. 17 to operate on the raw signal level.



all4on.wav
dose3_onoff
dose3.wav
dose3vgl.wav
fuzzer.py
fuzzer.wav
onoff.wav
sample
successful_off.bin
successful_on.bin
successful_trigger.bin
test_1onoff
test_1onoff_cropped.wav
test_1onoff.wav
test_1onoff2.wav
test_4on
test_4on2
test.bin

not assigned
Alice (A)
Bob (B)

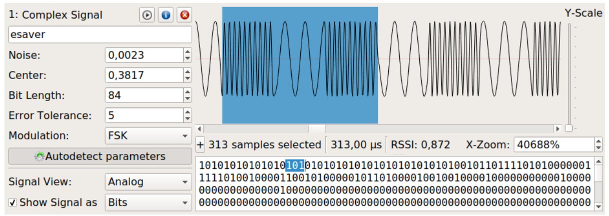
1: Complex Signal
test_1onoff
Noise: 0,0100
Center: 0,3802
Bit Length: 100
Error Tolerance: 5
Modulation: FSK
Signal View: Analog
✓ Show Signal as Hex
0 samples selected | 0,00 ns | RSSI: 0,434 | X-Zoom: 100%

```
aaaaaaaaaa96ea0f48ca16892200800080000000000000000000000000000000
0000000010001eca13930 [Pause: 5066 samples]
aaaaaaaaaa96ea0f48ca16892200800080000000000000000000000000000000
000000001000143e33990 [Pause: 5059 samples]
```

2: Complex Signal
sample
Noise: 0,0037
Center: 0,3927
Bit Length: 100
Error Tolerance: 5
Modulation: FSK
Signal View: Analog
✓ Show Signal as Hex
0 samples selected | 0,00 ns | X-Zoom: 978%

```
aaaaaaaaaa96ea0f48ca16892200808000000000000000000000000000000000
0000000010001be273c90 [Pause: 114441 samples]
aaaaaaaaaa96ea0648ca1690a20000404000754f7243e0 [Pause: 65007
samples]
```

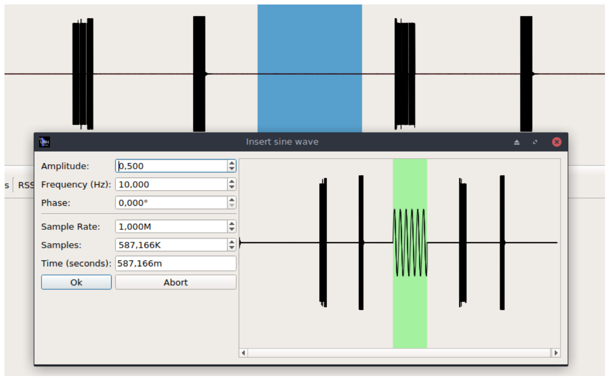
Figure 15: Overview of Interpretation tab



1: Complex Signal
esaver
Noise: 0,0023
Center: 0,3817
Bit Length: 84
Error Tolerance: 5
Modulation: FSK
Signal View: Analog
✓ Show Signal as Bits
313 samples selected | 313,00 µs | RSSI: 0,872 | X-Zoom: 40688%

```
1010101010101010101010101010101010101010100101101111010100000001
1111010010000110010100000101101000010010010000100000000000010000
0000000000001000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Figure 16: Synchronize bit and signal view



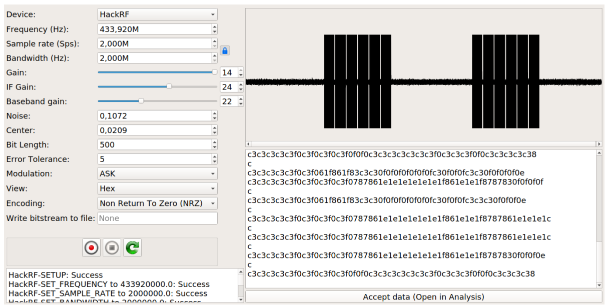
Amplitude: 0,500
Frequency (Hz): 10,000
Phase: 0,000°
Sample Rate: 1,000M
Samples: 587,166K
Time (seconds): 587,166m

Figure 17: Insert sine waves into signal

The *protocol sniff dialog* from fig. 18 is an advanced feature that can be used if modulation parameters of a protocol are known. In this dialog, demodulation is performed live by URH and you will only see the messages in binary form in the Analysis tab. Therefore, large signals do not have to be saved in a complex format which saves a lot of space in RAM or on disk.

A.3 Analysis Tab

All demodulated bits from Interpretation tab are automatically inserted into the table on *Analysis tab* shown in fig. 19. The view type is set to hex in this example. Every row header shows the regarding participant by including its configurable short name and color to keep an overview



Device: HackRF
Frequency (Hz): 433,920M
Sample rate (Sps): 2,000M
Bandwidth (Hz): 2,000M
Gain: 14
IF Gain: 24
Baseband gain: 22
Noise: 0,1072
Center: 0,0209
Bit Length: 500
Error Tolerance: 5
Modulation: ASK
View: Hex
Encoding: Non Return To Zero (NRZ)
Write bitstream to file: None

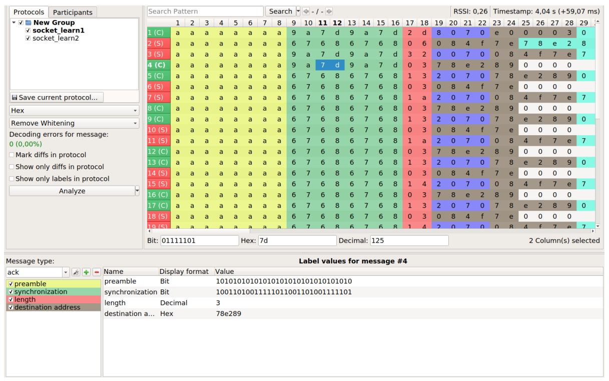
```
HackRF-SETUP: Success
HackRF-SET_FREQUENCY to 433920000.0: Success
HackRF-SET_SAMPLE_RATE to 2000000.0: Success
```

```
c3c3c3c3f0c3f0c3f0c3f0f0f0c3c3c3c3c3c3f0c3c3c3f0f0c3c3c3c3c38
c
c3c3c3c3f0c3f061f861f83c3c30f0f0f0f0f0f0fc30f0f0fc3c3c30f0f0f0e
c3c3c3c3f0c3f0c3f0c3f0787861e1e1e1e1e1e1f861e1e1f8787830f0f0f0f
c
c3c3c3c3f0c3f061f861f83c3c30f0f0f0f0f0f0fc30f0f0fc3c3c30f0f0f0e
c
c3c3c3c3f0c3f0c3f0c3f0787861e1e1e1e1e1e1f861e1e1f8787861e1e1e1c
c
c3c3c3c3f0c3f0c3f0c3f0787861e1e1e1e1e1e1f861e1e1f8787861e1e1e1c
c
c3c3c3c3f0c3f0c3f0c3f0787861e1e1e1e1e1e1f861e1e1f8787830f0f0f0e
c
c3c3c3c3f0c3f0c3f0c3f0f0f0c3c3c3c3c3c3f0c3c3c3f0f0f0c3c3c3c38
```

Accept data (Open in Analysis)

Figure 18: Protocol Sniff dialog

for larger protocols with many participants. Note the tree view on the upper left, where all signal names from Interpretation are listed and get highlighted if the user selects corresponding data in the table. Interpretation and Analysis are strongly tied together in URH. For example, you can select an area in the table and choose *Show in Interpretation* in context menu. The software will jump back to Interpretation and show the respective raw signal part. Furthermore, signals can be arranged in any order and assigned to configurable groups for better overview.



Protocols | Participants
New Group
✓ socket_learn1
✓ socket_learn2
Save current protocol...
Hex
Remove Whitening
Decoding errors for message: 0 (0,00%)
☐ Mark diffs in protocol
☐ Show only diffs in protocol
☐ Show only labels in protocol
Analyze

Search Pattern | Search | RSSI: 0,26 | Timestamp: 4,04 s (+59,07 ms)

	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
1 (C)	a	a	a	a	a	a	a	a	9	a	7	d	9	a	7	d	2	d	8	0	7	0	e	0	0	0	0	3	0
2 (S)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	0	6	0	8	4	f	7	e	7	8	e	2	8
3 (S)	a	a	a	a	a	a	a	a	9	a	7	d	9	a	7	d	3	2	0	0	7	0	0	8	4	f	7	e	7
4 (C)	a	a	a	a	a	a	a	a	9	a	7	d	9	a	7	d	0	3	7	8	e	2	8	9	0	0	0	0	
5 (C)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	1	3	2	0	7	0	7	8	e	2	8	9	0
6 (S)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	0	3	0	8	4	f	7	e	0	0	0	0	
7 (S)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	1	a	2	0	7	0	0	8	4	f	7	e	7
8 (C)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	0	3	7	8	e	2	8	9	0	0	0	0	
9 (C)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	1	3	2	0	7	0	7	8	e	2	8	9	0
10 (S)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	0	3	0	8	4	f	7	e	0	0	0	0	
11 (S)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	1	a	2	0	7	0	0	8	4	f	7	e	7
12 (C)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	0	3	7	8	e	2	8	9	0	0	0	0	
13 (C)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	1	3	2	0	7	0	7	8	e	2	8	9	0
14 (S)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	0	3	0	8	4	f	7	e	0	0	0	0	
15 (S)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	1	4	2	0	7	0	0	8	4	f	7	e	7
16 (C)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	0	3	7	8	e	2	8	9	0	0	0	0	
17 (C)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	1	3	2	0	7	0	7	8	e	2	8	9	0
18 (S)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	0	3	0	8	4	f	7	e	0	0	0	0	
19 (S)	a	a	a	a	a	a	a	a	6	7	6	8	6	7	6	8	1	4	2	0	7	0	0	8	4	f	7	e	7

Bit: 01111101 Hex: 7d Decimal: 125 2 Column(s) selected

Message type: ack
✓ preamble
✓ synchronization
✓ length
✓ destination address

Label values for message #4

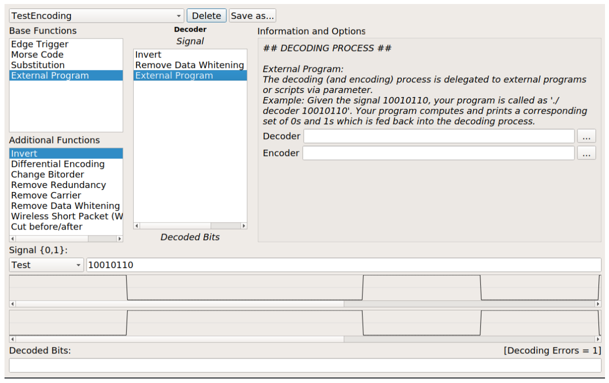
Name	Display format	Value
preamble	Bit	10101010101010101010101010101010
synchronization	Bit	10011010011111011001101001111101
length	Decimal	3
destination a...	Hex	78e289

Figure 19: Analysis tab overview

To start a protocol investigation we need to decode the raw bits first. URH offers a large set of predefined encodings that can be configured in the dialog from fig. 20. You can arrange the decoding primitives from the left in an arbitrary order to craft more sophisticated encodings. If a primitive is missing, users can also trigger external programs that implement these special encodings.

Once the data is decoded, we enable the differential view that produces a result shown in fig. 21. At a glance, we see that nibbles at position 5 and 15 change, that is, carry certain information. In this case, these are the on-off bit of a wireless socket and corresponding checksum.

More sophisticated protocols like the one from fig. 19 have several *message types*. With URH custom message types can be added and named whereby each message type has its own protocol fields. The assigning of a message type to a message can either be done manually or by using the dialog from fig. 22. Users can define custom rules for a message type which is then assigned to every message matching these rules.



TestEncoding | Delete | Save as...

Base Functions: Edge Trigger, Morse Code, Substitution, External Program

Additional Functions: Invert, Differential Encoding, Change Bitorder, Remove Redundancy, Remove Carrier, Remove Data Whitening, Wireless Short Packet (W, Cut before/after

Decoder — *Signal*: Invert, Remove Data Whitening, External Program — *Decoded Bits*

Information and Options

## DECODING PROCESS ##

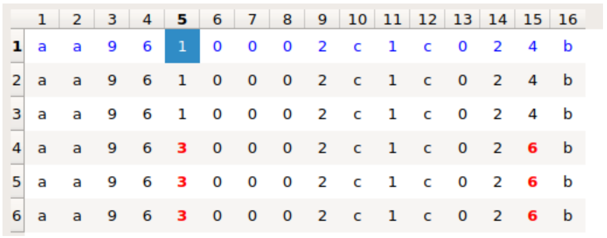
External Program:
The decoding (and encoding) process is delegated to external programs or scripts via parameter.
Example: Given the signal 10010110, your program is called as './decoder 10010110'. Your program computes and prints a corresponding set of 0s and 1s which is fed back into the decoding process.

Decoder:
Encoder:

Signal {0,1}: Test 10010110

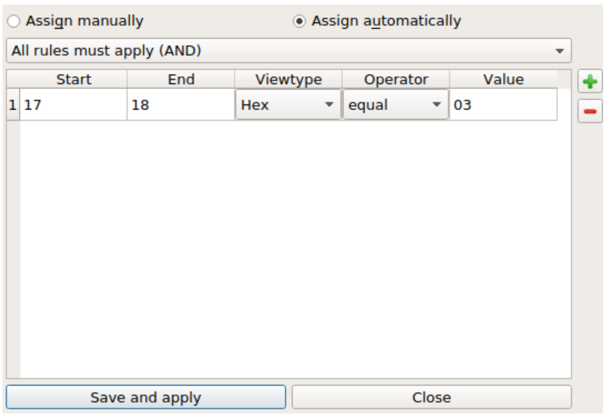
Decoded Bits: [Decoding Errors = 1]

Figure 20: Configure custom decodings in this dialog



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	a	a	9	6	1	0	0	0	2	c	1	c	0	2	4	b
2	a	a	9	6	1	0	0	0	2	c	1	c	0	2	4	b
3	a	a	9	6	1	0	0	0	2	c	1	c	0	2	4	b
4	a	a	9	6	3	0	0	0	2	c	1	c	0	2	6	b
5	a	a	9	6	3	0	0	0	2	c	1	c	0	2	6	b
6	a	a	9	6	3	0	0	0	2	c	1	c	0	2	6	b

Figure 21: Differential view to identify interesting areas



Assign manually [✓ Assign automatically]

All rules must apply (AND)

	Start	End	Viewtype	Operator	Value
1	17	18	Hex	equal	03

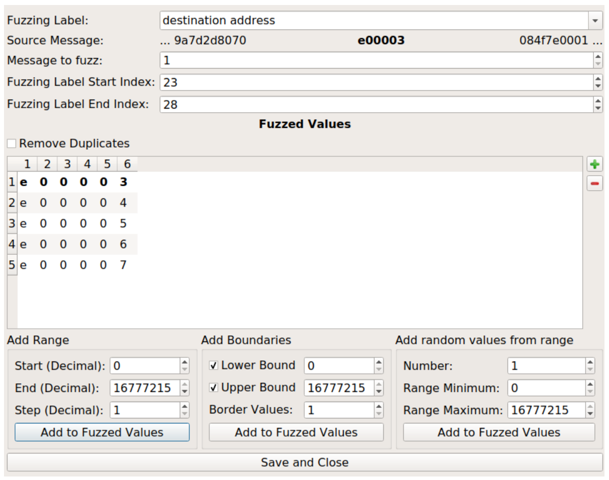
Save and apply | Close

Figure 22: Assign message types automatically based on custom rules

A very helpful feature is the possibility to attach colored text labels to protocol parts. Thereby, already reverse-engineered parts of a protocol can be tagged. This is useful to get a better understanding of the complete protocol. For each label you can configure a display format, that is, bit, hex, decimal or ASCII view. When vertically traversing through different messages, you can see the interpreted values from each label in a Wireshark-like view in the bottom of the application (see fig. 19). URH is also able to find particular labels automatically.

A.4 Generation Tab

In *Generation tab* messages can be crafted and send back via a SDR to observe reactions of investigated devices. In order to do this, you can create an arbitrary set of bit sequences by choosing from the previously analyzed signals and modify them as desired. It is, however, a laborious task to change messages manually. To address this problem, URH has a powerful *fuzzing component*. Each label can be individually configured using the fuzzing dialog shown in fig. 23.



Fuzzing Label:	destination address
Source Message:	... 9a7d2d8070 **e00003** 084f7e0001 ...
Message to fuzz:	1
Fuzzing Label Start Index:	23
Fuzzing Label End Index:	28

Fuzzed Values

☐ Remove Duplicates

	1	2	3	4	5	6
1	**e**	**0**	**0**	**0**	**0**	**3**
2	e	0	0	0	0	4
3	e	0	0	0	0	5
4	e	0	0	0	0	6
5	e	0	0	0	0	7

Add Range
Start (Decimal): 0
End (Decimal): 16777215
Step (Decimal): 1

Add Boundaries
✓ Lower Bound 0
✓ Upper Bound 16777215
Border Values: 1

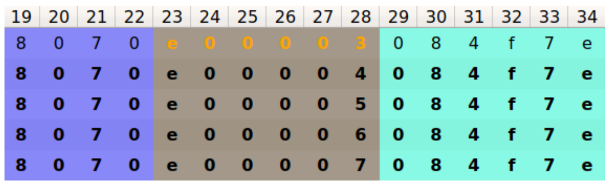
Add random values from range
Number: 1
Range Minimum: 0
Range Maximum: 16777215

Figure 23: Editing fuzzing settings in this dialog

Having configured the desired values for fuzzing, URH creates the corresponding messages. The configuration from fig. 23 produces the result from fig. 24. Note, that only values for the fuzzing label (nibble 23-28) change, while the rest of the message is repeated. Depending on how many protocol fields should be fuzzed, you can configure message generation to fuzz one field at a time (*subsequent*), all fields in parallel by selecting the next value for each field in each subsequent message (*concurrent*), or all fields in parallel by testing all combinations of values for each field (*exhaustive*). The encoding that was previously selected for decoding the input data will automatically be applied in reverse order.

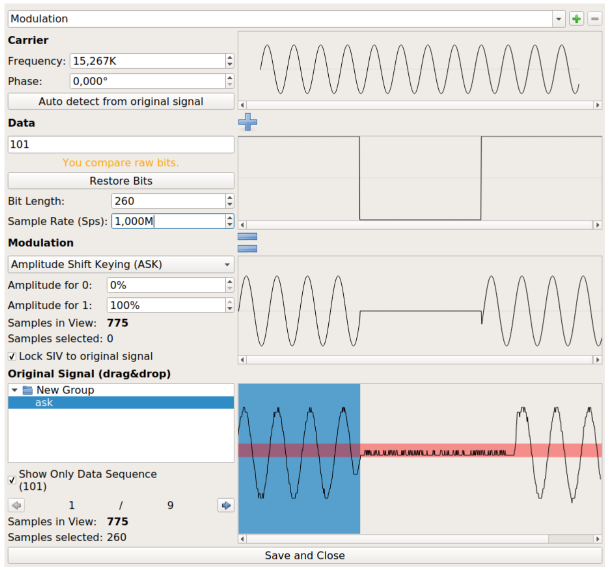
The final step in the Generation tab is to set the modulation that transforms bit sequences to actual signals. Modulations can be edited using dialog shown in fig. 25. Here, users can configure modulation parameters while comparing the respective output with recorded signals.

The process of configuring a modulation gets sped up by an auto detect button. Furthermore, values for bit length, sample rate and modulation default to the current project configuration. The correct modulation parameters are often found without further manual adjustments.



19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
8	0	7	0	e	0	0	0	0	3	0	8	4	f	7	e
8	**0**	**7**	**0**	**e**	**0**	**0**	**0**	**0**	**4**	**0**	**8**	**4**	**f**	**7**	**e**
8	**0**	**7**	**0**	**e**	**0**	**0**	**0**	**0**	**5**	**0**	**8**	**4**	**f**	**7**	**e**
8	**0**	**7**	**0**	**e**	**0**	**0**	**0**	**0**	**6**	**0**	**8**	**4**	**f**	**7**	**e**
8	**0**	**7**	**0**	**e**	**0**	**0**	**0**	**0**	**7**	**0**	**8**	**4**	**f**	**7**	**e**

Figure 24: Each label can be fuzzed



Modulation

Carrier
Frequency: 15,267K
Phase: 0,000°

Data
101
You compare raw bits.
Bit Length: 260
Sample Rate (Sps): 1,000M

Modulation
Amplitude Shift Keying (ASK)
Amplitude for 0: 0%
Amplitude for 1: 100%
Samples in View: **775**
Samples selected: 0
✓ Lock SIV to original signal

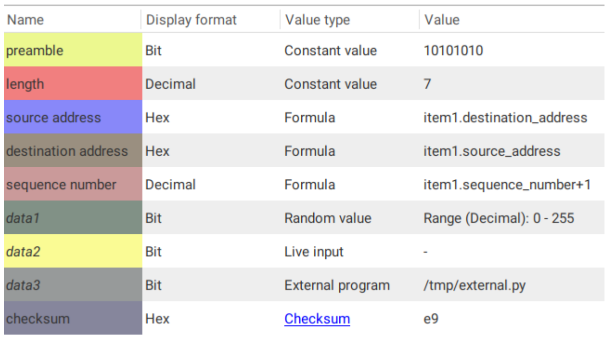
Original Signal (drag&drop)
New Group
ask
✓ Show Only Data Sequence (101)
1 / 9
Samples in View: **775**
Samples selected: 260

Figure 25: Manage modulations with this dialog

A.5 Simulator Tab

With the *Simulator tab* protocol state machines can be modeled which allows attacks on stateful wireless protocols. The simulator is mainly configured through a flow graph, as already described in section 3.4. In this section, we will show other important parts of the simulator.

Users can define the behavior for protocol labels during simulation time in a table as shown in fig. 26. Each label can have one of five *value types* which control how the resulting value will be created. The resulting value will be matched against when receiving a message and for a mismatch the simulation will retry to receive the message or end when the (configurable) number of maximum retries was reached. When sending a message the value will be inserted for this field. The possible value types are *Constant*, *Live input*, *Formula*, *Random value* and *External program*. Checksum fields are a special case, the software will automatically calculate the corresponding checksum during simulation.

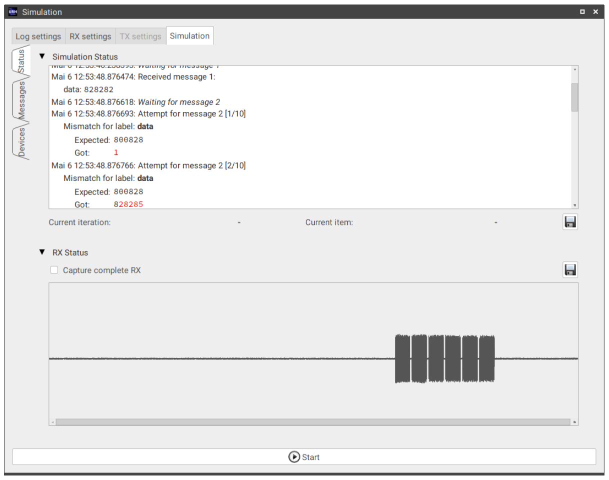


Name	Display format	Value type	Value
preamble	Bit	Constant value	10101010
length	Decimal	Constant value	7
source address	Hex	Formula	item1.destination_address
destination address	Hex	Formula	item1.source_address
sequence number	Decimal	Formula	item1.sequence_number+1
data1	Bit	Random value	Range (Decimal): 0 - 255
data2	Bit	Live input	-
data3	Bit	External program	/tmp/external.py
checksum	Hex	Checksum	e9

Figure 26: Configure labels in this table

These value types allow to reach the desired protocol state and create stateful answers to messages. For example, a sequence number can be learned live during simulation (*Live Input*) for message 1 and then be increased by one with a *Formula* in message 2 as shown for the sequence number in fig. 26. Such formulas can be arbitrary Python expressions to cover a variety of use cases. For even more complex tasks such as AES encryption a simple API for *External programs* allows to perform these tasks outside of URH and use the result during simulation.

Having configured a SDR for sending and receiving the simulation can be started. The status will be shown in the dialog from fig. 27 which helps to keep an overview and trace problems with the simulation in two ways. First, it shows the expected and actual value for each field in a message. Second, it shows a preview of the current RX status at the bottom to quickly track down problems resulting, e.g., from poor signal strength.



```
Mai 6 12:53:48.876474: Received message 1:
    data: 828282
Mai 6 12:53:48.876618: Waiting for message 2
Mai 6 12:53:48.876693: Attempt for message 2 [1/10]
    Mismatch for label: data
        Expected: 800828
        Got:      1
Mai 6 12:53:48.876766: Attempt for message 2 [2/10]
    Mismatch for label: data
        Expected: 800828
        Got:      828285
```

Figure 27: Simulator dialog

The transcript of the simulation can be found in the *Messages* tab of the Simulator dialog for further analysis. With Universal Radio Hacker researchers can investigate arbitrary stateful wireless protocols in separate phases: Interpretation, Analysis, Generation and Simulation. The knowledge acquired from the first two phases is brought to practice in Generation and/or Simulation phase which allow to test your hypotheses about the investigated protocol against the physical system. Using this concept and the particular support of our software, we have reverse engineered more than ten protocols.

Due to the modular concept and plugin system of URH, new features can be added easily. In this way, CC1101 data whitening and CRC-16/32 checking have been included in URH, both because of a specific demand. More features and plugins will follow, by us or by other researchers that like to provide new features to the open source solution *Universal Radio Hacker*.