# Automatic Wireless Protocol Reverse Engineering

Johannes Pohl and Andreas Noack

University of Applied Sciences Stralsund

August 13, 2019

# Proprietary wireless protocols everywhere

## Example: Smart Home

- Increase comfort of users through wireless sockets, door locks, valve sensors ...
- Devices are designed under size and energy constraints
- Limited resources for cryptography

## Risks of Smart Home

- Manufactures design custom *proprietary wireless protocols*
- Hackers may take over households and, e.g., break in without physical traces

*How can we speed up the security investigation of proprietary wireless protocols?*

# Software Defined Radio

## Why Software Defined Radios?

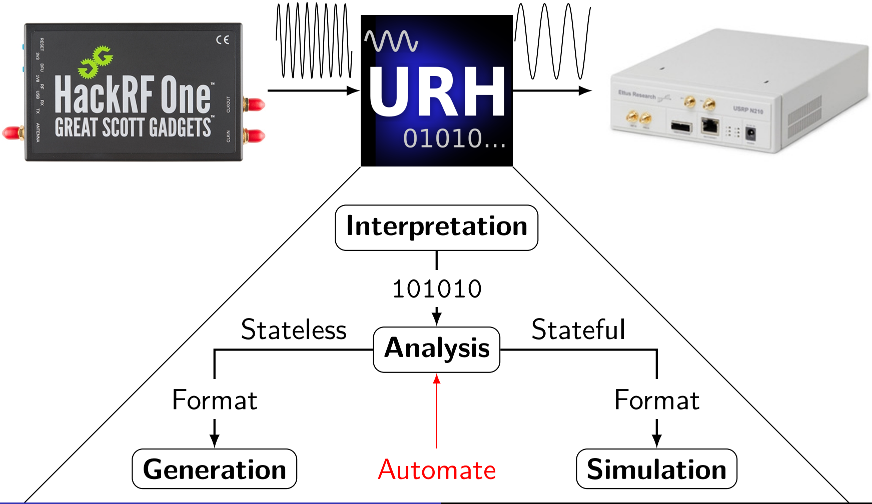
- Send and receive on nearly arbitrary frequencies[a]
- Flexibility and extendability with *custom software*

[a] e.g. HackRF: 1 MHz - 6 GHz


(a) USRP N210


(b) HackRF

# Universal Radio Hacker



# Structure of a Wireless Protocol

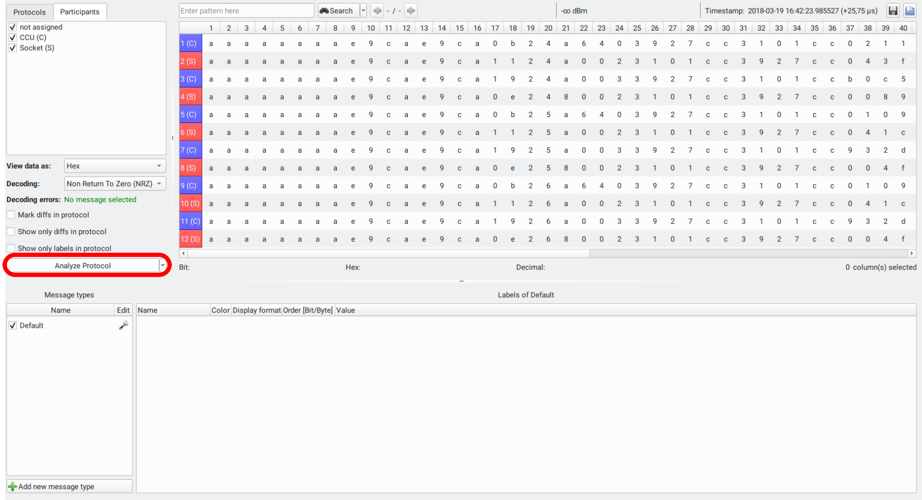
| Preamble | Synchronization | Length | SRC | DST | Sequence Number | ··· | Checksum |
|---|---|---|---|---|---|---|---|

## Purpose of Fields

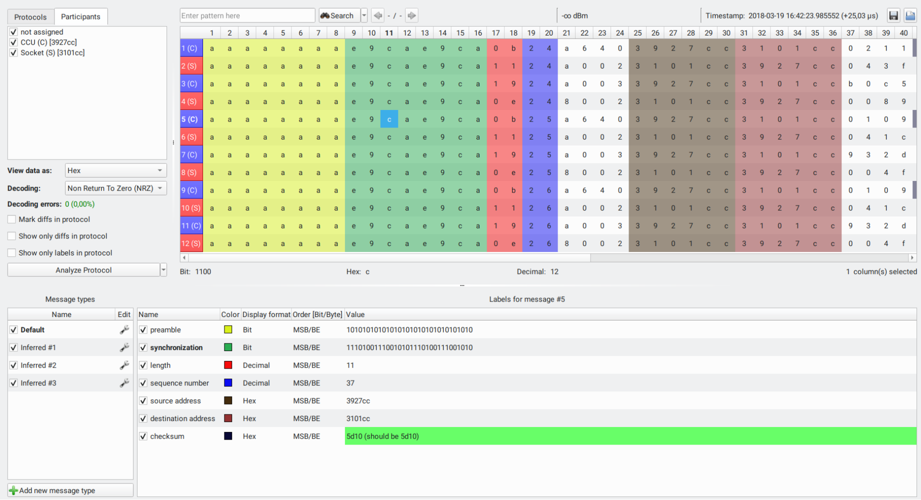
- **Preamble:** Synchronize clocks with fixed preamble pattern, e.g., 101010...
- **Synchronization:** Indicate start of transmission with sync sequence, e.g., `0x9a7d`
- **Length:** Contains the size of following data, usually in bytes
- **SRC/DST**: Source / Destination addresses of communicating devices
- **Sequence Number**: Increasing counters used for flow control and freshness
- **Checksum**: Verify integrity of received data (recognize transmission errors)

The **message format** determines the order and type of fields for a message. The **message type** describes which message format to use. A protocol can contain various message types such as DATA and ACK (acknowledgement).

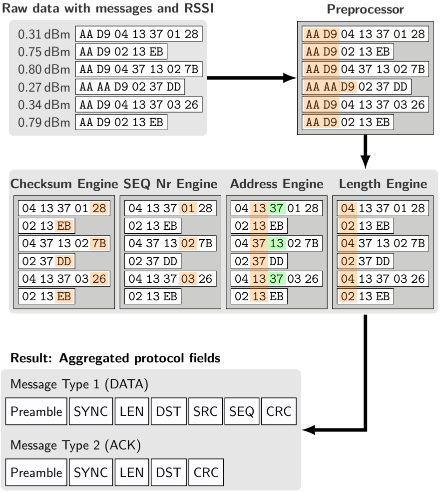
# Example Protocol: Communication between two Smart Home Devices



# Example Protocol after hitting the Analyze Protocol Button



# Overview of Proposed Algorithm




## Design Goals

- Work on limited number of messages
- Tolerant against transmission errors
- Bootstrap unknown protocols but also consider **prior knowledge**
- Work solely on captured messages, i.e., program binary is not accessible

# Preprocessing: Align messages on (unknown) sync words

**Preprocessor**

```
AA D9 04 13 37 01 28
AA D9 02 13 EB
AA D9 04 37 13 02 7B
AA AA D9 02 37 DD
AA D9 04 13 37 03 26
AA D9 02 13 EB
```

## Purpose of Preprocessing

- Identify **preamble**: $(a^n b^m)^k$ with $a, b \in \{0, 1\}$, $a \neq b$ and $n, m, k \in \mathbb{N}^+$, e.g., 10101010 with $n = m = 1$ and $k = 4$.
- Identify **sync word(s)**
- **Align** messages on sync word(s): Pass only data behind sync to subsequent engines

# Overview Field Type Inference

1. Assign messages to engine-specific **clusters**. For example, the length engine clusters messages based on their physical length in bytes.
2. Find **common ranges** inside and/or between clusters.

$m_1$ : AB CD EF 11
$m_2$ : 12 34 EF 22
$m_3$ : 56 78 EF 37

Common Range

Start: 5
Length: 2
Value: EF
Messages: 1,2,3

   1. Score common ranges with an engine-specific *scoring function*.
   2. Return common ranges with highest score if they surpass a minimum score $s_{\text{min}}$.
   3. If possible, merge the resulting ranges.
3. Add found labels to the current *message type* or create a new one, if necessary.

# Included Engines

### Length Engine

- Cluster messages by physical length
- Give higher score to ranges those decimal value matches physical length

### Address Engine

- Assign a participant to every message
- Infer **participant address candidates**
- Find fields with address candidates

### Sequence Number Engine

- Calculate a matrix $E$ of decimal differences between adjacent messages
- Evaluate columns of $E$ with only constants or zeros (when `SeqNr` is constant between some messages)

### Checksum Engine

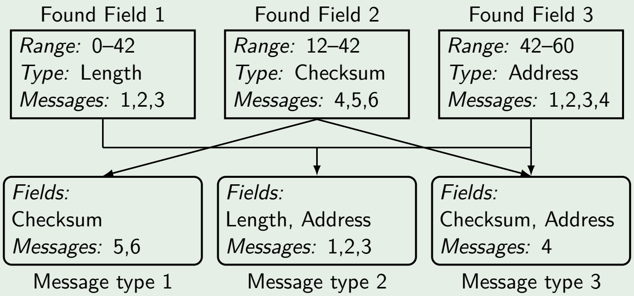
- Find checksums such as CRC16-CCITT by testing common CRC parameters and other checksums
- CRC testing algorithm uses a CRC cache for increased performance

# Message Type Creation and Assignment

## Assignment of Message Types

- Engines return a set of labels
- Group these labels into message types based on their **message indices**
- Create message type for non-overlapping fields with matching message indices
- In conflict case: Choose range(s) that maximize the total score of message type

## Example for three found fields



# Considering Prior Knowledge

## Rules for prior knowledge

1. Labels must not be changed.
2. Labels must not be removed from a message type.
3. Messages must keep their assigned message type.

## Dealing with prior knowledge

- Skip engines of already present fields
- Engines ignore all ranges of a message that are already labeled
- If new message type needs to be created: **split** original one (=copy over all labels)

## Run this for each message type to consider prior knowledge

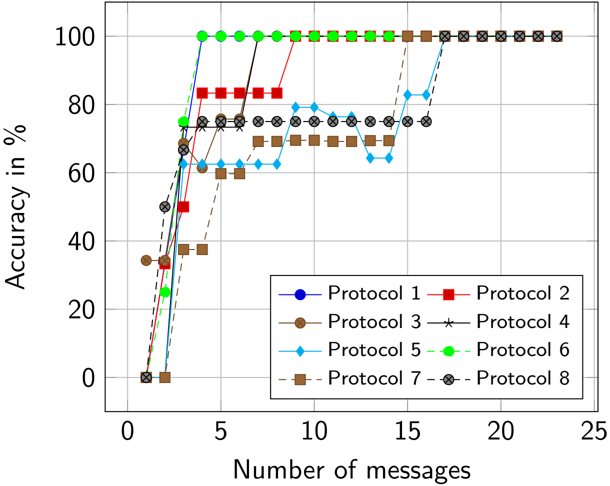
```
while new_fields_found and max_iteration_not_exceeded:
  for mt in existing_message_types:
    new_fields = []
    for engine in engines:
      if field_of_engine not in mt:
        new_fields.extend(engine.run(mt))
    add_to_message_type(mt, new_fields) # Split message type if necessary
```

# Overview of Generated Protocols

Table: Properties of tested protocols whereby × means field is not present and $N_P$ is the number of participants.

| # | Comment | $N_P$ | Message Type | Even/odd message data | Size of field in bit (BE=Big Endian, LE=Little Endian) Preamble | Sync | Length | SRC | DST | SEQ Nr | CRC |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | common protocol | 2 | data | 8/64 byte | 8 | 16 | 8 | 16 | 16 | 8 | × |
| 2 | unusual field sizes | 2 | data | 8/64 byte | **72** | 16 | 8 | **24** | **24** | 16 (BE) | × |
| 3 | contains ack and CRC8 CCITT | 2 | data | 10/10 byte | 16 | 16 | 8 | 16 | 16 | 8 | **8** |
| | | | ack | × | 16 | 16 | 8 | × | 16 | × | **8** |
| 4 | contains ack and CRC16 CCITT | 2 | data | 8/64 byte | 16 | 16 | 8 | 16 | 16 | × | **16** |
| | | | ack | × | 16 | 16 | 8 | × | 16 | × | **16** |
| 5 | three participants with ack frame | **3** | data | 8/64 byte | 16 | 16 | 8 | 16 | 16 | 8 | × |
| | | | ack | × | 16 | 16 | 8 | × | 16 | × | × |
| 6 | short address | 2 | data | 8/64 byte | × | 16 | 8 | **8** | × | 8 | × |
| 7 | four participants, varying preamble size, varying sync words | **4** | data | 8/8 byte | **16** | **16** | 8 | **24** | **24** | × | 16 |
| | | | ack | × | **8** | **16** | × | × | **24** | × | 16 |
| | | | kex | 64/64 byte | **24** | **16** | × | **24** | **24** | × | 16 |
| 8 | nibble fields + LE | 1 | data | 542/260 byte | **4** | **4** | 16 (**LE**) | × | × | 16 (**LE**) | × |

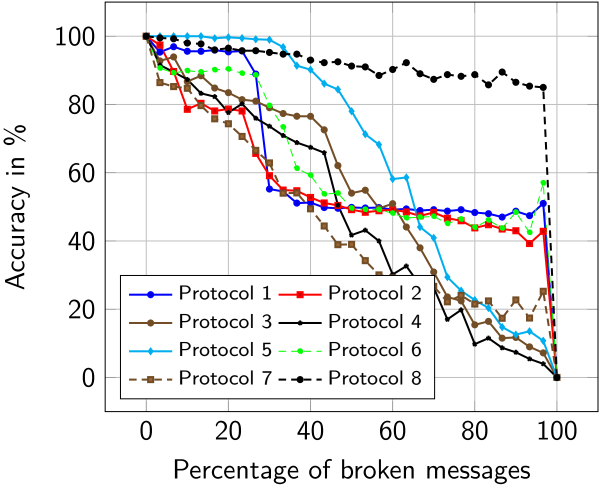
# Test against number of messages




## Results

- 100% accuracy for all protocols when more than 17 messages are available
- 100% accuracy for five out of eighth protocols when at least 7 messages are available
- Protocol 5 and 7 have more participants involved so algorithm needs more messages to infer address fields correctly

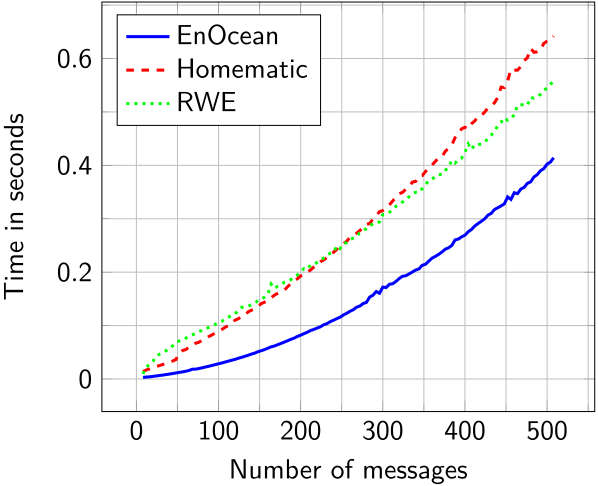
# Test against errors




## Setup and Results

- Break messages by setting bits to random values beginning at a random position (30 messages total)
- Worst case for the algorithm because some data remains valid in broken messages
- Majority of protocols are labeled with more than 80% accuracy when 20% of messages are broken

# Performance measurement with real-world smart home protocols




## Measurement Setup

- Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
- 16 GB RAM
- Message length between 8 and 61 bytes
- For every number of messages perform 100 performance measurements and take the mean performance

# Conclusion and Future Work

## Conclusion

- Framework for automatic reverse engineering of proprietary wireless protocols
- Dedicated engines to find `Preamble`, `Synchronization`, `Length`, `Sequence Number`, `Address` and `Checksum` fields
- **Bootstrap** unknown protocols but also able to consider **prior knowledge**
- Verified with simulated and real-world protocols

## Future Work

- Suggestion of attacks based on the found fields
- Detection of cryptography in message payload
- Ultimate goal: automated security score based on found cryptography and protocol complexity for initial security assessment right from captured messages

URH 01010...

https://github.com/jopohl/urh

## Contact

- E-Mail: `Johannes.Pohl90@gmail.com`
  E-Mail: `Andreas.Noack@hochschule-stralsund.de`
- Slack: `https://bit.ly/2LGpsra`
- GitHub: `https://github.com/jopohl`