Automatic Wireless Protocol Reverse Engineering

Johannes Pohl
University of Applied Sciences Stralsund, Germany

Andreas Noack
University of Applied Sciences Stralsund, Germany

Abstract

Internet of Things manufacturers often implement their own wireless protocols in order to save licensing fees. Deviating from standard, however, sometimes paves the way for critical attacks such as stolen cars or house breaks without physical traces. For a security analysis of such proprietary protocols, researchers use Software Defined Radios and dedicated demodulation tools. But when reverse engineering is necessary, researchers are left alone and need to find protocol fields manually in a time-consuming and tedious process.

We contribute a framework designed for field inference of wireless protocols. In contrast to previous research, our algorithm operates on the physical layer and, moreover, takes wireless specifics such as Received Signal Strength Indicators into account. Furthermore, the algorithm is robust against errors that are common in wireless communication. Our contribution not only performs a bootstrap of completely unknown protocols but also considers prior knowledge such as participant addresses or known field positions in order to increase accuracy. An implementation is published as part of the open source software Universal Radio Hacker and is a first step towards a default security analysis for proprietary wireless protocols similar like a port-scan is for traditional security.

1 Introduction

Global manufacturers flood the Internet of Things (IoT) market with a huge number of devices such as smart light bulbs or door locks. The wireless communication between devices often uses proprietary protocols designed under size and energy constraints whereby security is only a secondary factor. This leads to serious threats for customers ranging from stolen cars to burglars silently breaking into their houses.

The security investigation of unknown wireless protocols involves many challenges. Researchers capture the device communication with a Software Defined Radio (SDR) and, subsequently, demodulate the signal with dedicated software solutions [5, 17]. Those tools provide communicated bits to researchers but leave them alone with reverse engineering of the actual protocol. This is a tedious process and a significant obstacle before the real security analysis begins.

The problem of *automatic* protocol reverse engineering is well studied by several authors as summarized in the survey of Narayan et al. [15]. The published solutions work with high accuracy, but cannot be applied to the *wireless* setting for two reasons. First, most of them are designed for *text-based* protocols like HTTP while wireless IoT protocols are rather *binary*. Second, all of them work on the *application layer*, i.e., they rely on information from lower layers like IP addresses.

Since the related work is not designed for the specific wireless setting, we present an algorithm that can infer protocol fields from network traces of IoT devices. Our algorithm considers specifics of wireless communications like Received Signal Strength Indicator (RSSI) and uses statistical methods in combination with heuristics to find physical layer fields.

The main goal of our algorithm is to perform a bootstrap of an IoT protocol in order to simplify the subsequent manual analysis. Moreover, the algorithm is able to work with prior knowledge, that is, find missing fields with an incomplete protocol specification. We make the following contributions:

- An extendable framework for inferring message formats of physical layer protocols, prepared for the specifics of wireless protocols.
- A framework that performs a complete **bootstrap** of unknown protocols and also considers **prior knowledge** such as participant addresses to enhance accuracy.
- Heuristics for field semantics `Preamble`, `Sync`, `Length`, `Address`, `Sequence Number` and `Checksum`.

We provide an initial step to find security leaks in proprietary wireless protocols in the same way port scanning is a standard procedure for security analysis of conventional systems. Many IoT attacks can be evaluated once the physical structure of the protocol is known, for example, manipulating the sequence number of a message may lead to desynchronization of devices under certain conditions.

2 Terminology and Design Assumptions

The main objective of protocol reverse engineering is to infer the **protocol format** that involves the message format and the protocol state machine. In this paper we focus on inferring the message formats and leave inferring the state machine for future work. The **message format** defines the order and type of **fields** in a message. A message field is a finite information sequence with known length and position, and a certain **field semantic** such as address, checksum or length in bytes. We further introduce the term **label**. We define label as an estimation for a field, that means a label aims to have same start, end and semantic as a field.

Problem Definition Our goal is to infer the message formats from captured messages (network traces) of a wireless communication. We make the following design assumptions:

- All protocol messages include a specific start of data sequence, that is, a sync word with length ≥ 1 bit, usually after the preamble.
- A single message does not contain more than one preamble and sync word. This could be different in practice if the physical pause between messages is very short.[1]
- The number of captured messages will probably be limited due to legal transmission limits.
- Protocols are length-efficient binary protocols with no separators or redundancy and high entropy.
- Messages can be (partly) broken due to transmission and demodulation errors.
- We only have network traces and no additional data like source codes or program binaries.

3 Automatic Wireless Reverse Engineering

The proposed algorithm is divided into two main phases. First, a *preprocessing* is performed to align messages. Second, *field inference engines* work on these aligned messages to infer protocol fields and message types. Refer to fig. 1 for an overview.

The algorithm also works for protocols that include nibble sized fields. This is an important difference to related approaches because as we work on the physical layer we cannot be sure that information is byte aligned. An example for a protocol with nibble fields is the Oregon Scientific RF Protocol [2]. In the following sections we assume the protocol is completely unknown, that is, no prior knowledge is available. We show in section 3.8 how to consider prior knowledge.

[1] One solution is to add a *message splitting* functionality to our preprocessing from section 3.1. This works by *iteratively* applying our sync word finding and alignment procedures for full message and not stop after first occurrence. We could, however, not observe such short physical pauses for real devices so that this feature may not be needed in practice.

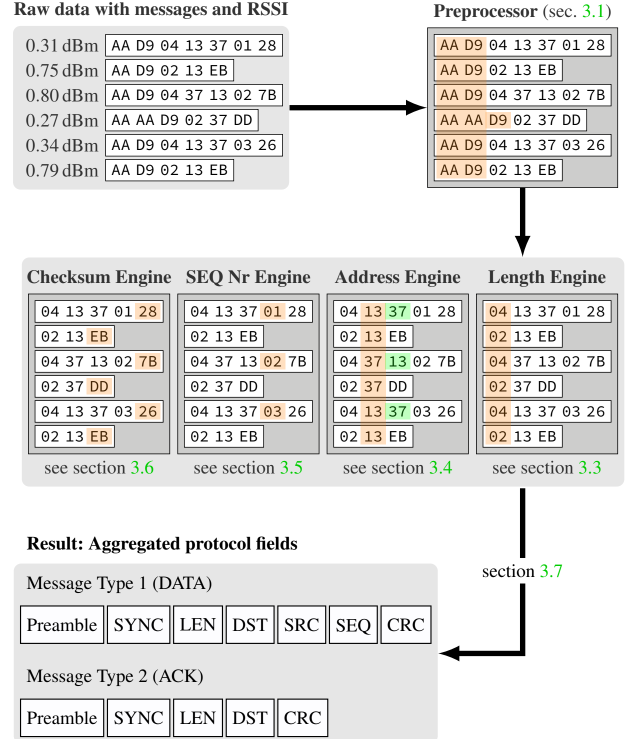


Figure 1: Overview of proposed algorithm

3.1 Preprocessing

The first step of the algorithm is to align messages based on possibly unknown sync words. We refer to this process as **preprocessing**. We split the preprocessing task into three steps. First, we identify the *preamble* of a message. Second, we *derive the sync word(s)* from the messages. Third, we create an *alignment vector* which stores the end of synchronization for each message. Splitting this task has an advantage: If sync words are known previously they can be entered by the user and our algorithm skips the first two steps.

In case sync words are not known beforehand, they need to be extracted from captured messages. The basic idea is to search for the first *differing bit* behind the preamble. This basic approach has three problems:

1. The preamble length can vary between messages.
2. Protocols may use more than one sync word.
3. Bits may be flipped or missing due to transmission errors in certain messages.

We tackle the first problem by determining the preamble length for each individual message, the second one by looking for a synchronization length instead of a specific sync word and the third one by creating histograms over all messages in order to be robust against errors in single messages. We explain how we do this in the following sections.

3.1.1 Finding the preamble

The purpose of a preamble is to synchronize the receiver. Therefore, it consists of a fixed pattern of alternating zeros and ones. Formally, we can define a preamble as

$$(a^n b^m)^k \tag{1}$$

with $a, b \in \{0, 1\}$, $a \neq b$ and $n, m, k \in \mathbb{N}^+$. For example, a common preamble is 10101010 with $n = m = 1$ and $k = 4$.

This formal representation of a preamble allows us to design a straight-forward algorithm for finding it. First, we get a by evaluating the first bit of the message. This also yield b because it is the inverse of a. Second, we look for the first occurrence of b to learn n and for the first occurrence of a behind the first b to learn m. Last, we count how often the pattern $a^n b^m$ repeats in order to get k.

The algorithm is very prone to errors. Especially in wireless communications we must expect bits to be missing, wrongly inserted or even flipped. From error characteristic experiments with Software Defined Radios, we know that such errors mostly occur at the beginning of a message. We take this into account by shifting the start value until we find a preamble with *sufficient* length, that is, we skip more and more bits at the beginning of a message. In experiments $k \geq 2$ turned out to be a good value for sufficient preamble length.

The preamble search is just a preparation for identifying sync words in the next step. Therefore, the preamble should not interfere with the sync word. This can, however, happen if the sync word starts in the same way as the preamble ends. Consider the example with $(10)^k$ in Figure 2:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	0	1	0	1	0	1	0	1	0	0	1		
2	1	0	1	0	1	0	1	0	0	1	1	0		
3	1	0	1	0	1	0	1	0	1	0	1	0	0	1

Figure 2: Messages with varying syncs and preambles

The first two messages share the eight bits preamble `10101010` while the third message has a ten bit preamble. The sync word for first and third message is `1001` while for second message it is `0110`. In the first message, the algorithm finds position 10 as preamble end. This is wrong, because positions 9 and 10 are part of the sync word, therefore, the preamble size should be decreased by two ($\rightarrow 8$). For the second message, the algorithm finds position 8 as preamble end and shall not decrease the preamble length by two ($= n + m$).

How can the algorithm know whether to decrease the preamble length or not? The answer is, it cannot know at this point. Therefore, we save *both* lengths in two vectors that hold the lower $\mathbf{l} = (8, 6, 10)$ and upper $\mathbf{u} = (10, 8, 12)$ bounds for the preamble ends of each message, respectively.

3.1.2 Sync Word Identification

In the previous section we roughly estimated the *preamble ends* and now search for the specific sync word(s) in the protocol with the help of this information.

Calculating Difference Matrix We calculate a *difference matrix D* where d_{ij} denotes position of the first difference between message i and j. If there is no such difference, that is, messages are equal or one message is a prefix of the other we set $d_{ij} = 0$. For example, when calculating the difference matrix for the messages in fig. 2 we get

$$D = \begin{pmatrix} 0 & 9 & 11 \\ 0 & 0 & 9 \\ 0 & 0 & 0 \end{pmatrix}.$$

Note that we just calculate the upper triangle of the difference matrix because the comparison order does not matter.

Finding sync words With preamble end vectors $\mathbf{l}$ and $\mathbf{u}$ from section 3.1.1 and difference matrix D from section 3.1.2 we determine sync word candidates as follows: We iterate over all difference positions d_{ij} in D greater than zero, whereby each position represents a possible ending of the synchronization word. For each sync end $d_{ij} > 0$, the raw preamble ends $\{\mathbf{l}_i, \mathbf{l}_j, \mathbf{u}_i, \mathbf{u}_j\}$ for messages i and j indicate possible *sync starts* and we save all resulting sync word candidates. We perform these steps for every message pair and count the occurrence of sync word candidates yielding a histogram of possible sync words. After that the algorithm combines candidates with a **common prefix** in the histogram and determines the *most probable sync length*. [2] This way, the search for the sync word gets more robust against transmission errors and can even deal with different sync words. Finally, we take the most frequent sync length s_L from the histogram and return all found sync words with length s_L.

Message Alignment Once sync words are known, the algorithm aligns messages on them. If multiple sync words appear in a message, we align on the first occurring sync word. For further processing we remove the sync word and everything before from the messages and pass the aligned and cropped messages to the field inference engines. This way, the subsequent engines can work on reduced information and, more importantly, the messages are aligned on the sync word so that differential analyses are not biased by bit shifts.

[2] In theory, protocol sync words can vary in length. The algorithm can deal with this by choosing sync lengths whose frequencies are above a certain threshold. We think, however, that a varying sync word length is not common in practice, so we assume it to be constant for a protocol in this paper.

3.2 Field Inference Overview

Once messages are aligned properly, we can infer field semantics from captured messages. For this reason, our algorithm consists of dedicated *Field Inference Engines* that work with scoring functions to find the right label for a protocol field. In general, this process consists of three steps:

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**, that is, *identical sequences at same positions* inside and/or between clusters.
   (a) Score common ranges with an engine-specific *scoring function*.
   (b) Return common ranges with highest score if they surpass a minimum score $s_{\min}$.
   (c) If possible, merge the resulting ranges.
3. Add found labels to the current *message type*. If necessary, create new message types based on found labels.

Common Ranges The search for common ranges is a central part of every engine and must be **robust** against errors in single messages. We find common ranges by comparing the aligned messages column-wise and search for columns with equal values. In order to be tolerant against errors, we view a range as common if at least α percent of column values are equal for every message, whereby an α between 70% and 95% turned out to be a good compromise between accuracy and error-tolerance in practice. Having found a common range, we save its *start*, *length* and *value* together with the *message indices* this ranges applies to. An example is shown in fig. 3.

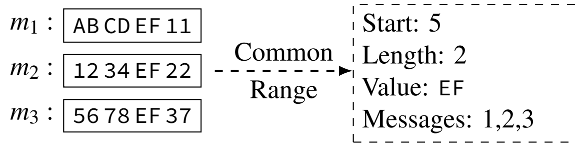


Figure 3: Example for common range of three messages

We describe the engine specific scoring functions and cluster features in the following sections.

3.3 Length Field Engine

A length field contains the size of following data, usually in bytes. In order to find length fields we cluster messages by their physical length in bytes. For each cluster we calculate the normalized histogram of common bits and identify ranges where at least eight bits are equal for α percent of the messages. In experiments $\alpha = 0.7$ produced good results.

Next, we rule out equal valued ranges across clusters because values must differ for messages with other physical lengths. All remaining ranges are scored with the function:

$$s = \frac{1}{1+k \cdot p} \cdot e^{-\frac{1}{2}\left(\frac{v-t}{\sigma}\right)^2} \tag{2}$$

whereby s is the score, t the target length, that is, the message length in bytes, p the start position of the common range and v the decimal interpreted value, whereby both *little endian* and *big endian* interpretations are tested. The constants σ and k are used to control the scoring function's decay, whereby in our experiments $\sigma = 2$ and $k = 0.25$ produce good results.

The idea of the function is to yield lower scores for high start positions because we expect length fields rather to appear at the beginning of messages. We evaluate the scoring function for a sliding window over common range values with standard integer window sizes between 8 and 64 bits, in order to find usual sized length fields.

3.4 Address Field Engine

We split the address field search into three steps as shown in fig. 4. First, the engine assigns a **participant** to every message. Second, the engine searches **possible addresses** of involved participants. Finally, the engine finds positions of address fields based on these possible addresses.

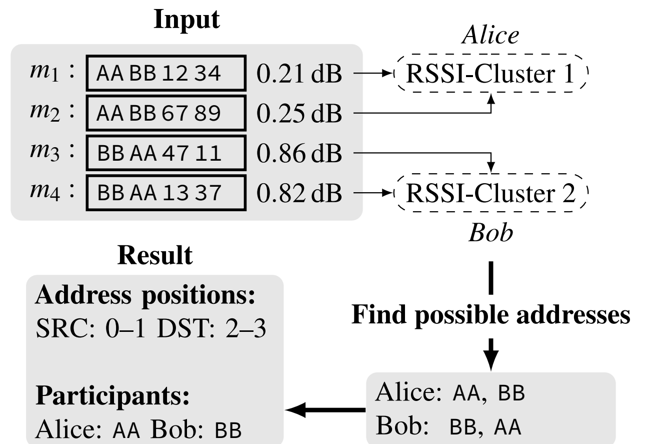


Figure 4: Address Field Engine example for messages between Alice (address: AA) and Bob (address: BB)

3.4.1 Assign Participants

The engine begins with ensuring that every message has a participant (= sending device) assigned, whereby we assume the number of participants $|P|$ to be known. The automatic assignment has two strategies: First, if prior knowledge is available and, more specifically, a source address label is present in the message while a participant with this address is known, this participant is assigned to the message.

The second strategy is more generic and relies on the Received Signal Strength Indicator (RSSI) of a message. During recording, devices are placed at different distances from SDR and/or have different transmitting powers. Therefore, it can be expected that messages of a certain participant have a similar RSSI. In order to assign participants, the RSSIs of **all** messages are **clustered** with $|P|$ centers so that the highest RSSIs are in first cluster, the second highest in second cluster and so on. With this clustering we can assign a participant to a message by looking up the cluster index of the message's RSSI. Users can control to which cluster a participant gets assigned with a **relative RSSI** as shown in fig. 5.

	Name	Shortname	Color	Relative RSSI
1	Alice	A		0 (low)
2	Bob	B		1
3	Carl	C		2 (high)

Figure 5: Configuration of Relative RSSI

Note that only messages without an assigned participant are considered in this routine. A message can already have a participant assigned either through a prior run of this engine or through manual assignment by the user. The routine skips messages that already have a participant assigned.

3.4.2 Find Address Candidates

Once each message has a participant, the next task of the engine is to find **possible addresses** for each participant. In order to find address candidates, we start by clustering the messages by participant and find common ranges within each participant cluster. As a result, we have a set of common ranges for each participant cluster. Then we compare the values of the individual common ranges pairwise for each pair of participants and search for *longest common substrings* (LCSs) in these values, for example, LCSs(`CAFE1337`, `1337CAFE`) = [`CAFE`, `1337`]. If found, we add these substrings to address candidates of both participants. If no common substring is found we add both values as address candidates for both participants to cover, e.g., ACK messages with only one address.

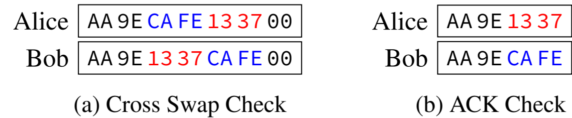
The address for a participant can also be previously known either through a prior run of this engine or manual configuration by the user. The search of address candidate for participants with known address is skipped by the algorithm. Moreover, we can filter address candidates when at least one address is known beforehand: Since it can be assumed that addresses in a protocol have a common length, all possible addresses with a different length can be ruled out.

3.4.3 Find Address Fields

The final step of this engine is to find **address positions** and infer **address types**, that is SRC or DST, based on the address candidates determined in the previous step.

Find possible ranges We start by iterating through the messages of each participant. For each message the system finds occurrences of address candidates and, if found, creates common range objects from their position in the message. If the same position was already found in another message, the index of the current message is added to the existing indices of the according common range object.

Scoring Subsequently, the common ranges are scored. The scoring is based on two observations: First, addresses are mostly located next to each other. Second, in acknowledgement messages (ACKs) there may be only one address present. This leads to two checks for a common range: **Cross Swap Check** and **ACK Check**. The Cross Swap Check is successful if two common ranges of different participants have swapped values and one range starts right after the other range (fig. 6a). The ACK check verifies whether two common ranges of two participants have same position but different values (fig. 6b).



(a) Cross Swap Check (b) ACK Check

Figure 6: Cross Swap check and ACK check example for Alice (CA FE) and Bob (13 37)

The idea is to increase the score for a range if one of these two checks is successful. Therefore, we increase the score by $\frac{N_{rng}}{N_P}$ whereby N_{rng} is the number of messages in this common range and N_P the number of messages of the current participant. This way, common ranges that apply for more messages get a higher score. If both checks should be successful, the score is only increased *once*. Ranges will be eliminated when they do not surpass a minimum score. In our experiments a minimum score of 0.1 produces good results.

Assign participant addresses The next step is to assign addresses to all participants without a known address based on the previously scored ranges and the address candidates from section 3.4.2. We score an address candidate in the following way: If the algorithm finds the address candidate in a message together with another address candidate, we increase the score. The underlying idea is, that the source address (=participant address), is likely to be included in a message that contains multiple addresses. On the other hand, when the message only contains one address, this is probably rather a destination address. Finally, we assign for each participant the address candidate with the highest score.

Field type inference Having assigned the most likely address to each participant, the next step is to infer address fields from the found common ranges. In order to do this, we need to distinguish between source address (SRC) and destination address (DST). The rule is simple: If the common range value is equal to the participant address we infer a SRC, otherwise a DST label. Note, we only consider the highest scored ranges to prevent multiple SRC and DST labels being inferred. As a final step, we check for messages that were sent to **broadcast**: First, we find all messages M_{SRC} that have a SRC but no DST label assigned. Second, we take the positions of DST labels from messages with SRC label. Finally, we evaluate the value of messages in M_{SRC} at DST position and look if the address appears multiple times but not as SRC address. If this is the case, the algorithm takes this address as broadcast address.

3.5 Sequence Number Field Engine

Sequence numbers are increasing counters used for flow control and freshness in a protocol. We find sequence numbers with a *matrix of decimal differences E* between adjacent messages. For example, if we take four messages with five bytes as shown on the left in fig. 7 the matrix E means, for example, that m_1 and m_2 have a decimal difference of 1 at the first byte. Moreover, all first bytes of the four messages differ by 1 so the first byte is a good candidate for the sequence number.

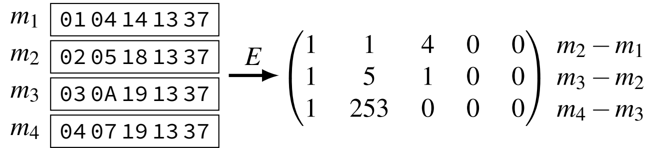


Figure 7: Matrix of decimal differences E (mod 256)

In general, we look at columns of E that only contain constants or zeros because sequence numbers may remain constant between some messages. In order to deal with overflows such as $255 \rightarrow 0$ we take the decimal difference modulo 256 or, more generally, 2^n whereby n is the considered n gram length and, e.g., is 8 for byte and 4 for nibble protocols.

The score s for a column of E is set to

$$s = \begin{cases} 0, & \text{if only zeros in column} \\ \frac{\#(\text{most_common_value} \neq 0) - \#\text{zeros}}{\#\text{values}}, & \text{else} \end{cases}$$

Next, we create common range objects (section 3.2) from all columns surpassing minimum score $s_{\min}$ resulting in sequence number candidates. For longer sequence numbers, these ranges must be **merged**. We merge two adjacent ranges if they apply to the same messages (rows). Note, this applies to both little and big endian sequence numbers.

3.6 Checksum Field Engine

Manufacturers of IoT devices can choose from a large pool of checksum algorithms and variants whereby most of them can be fine-tuned with additional parameters. Hence, it can be difficult to detect the right algorithm with correct parameters for a checksum and, additionally, the correct range of data bits the checksum was applied to. Fortunately, many manufacturers use variants of the Cylic Redundancy Check (CRC) algorithm and most of them put the checksum field at the end of all protocol fields. For this engine, we assume that the protocol uses only one variant and parameter set for all messages.

We apply a checksum search algorithm that differentiates between CRC and other checksums. The CRC engine aims to test all CRC parameters for each the datarange from the beginning of the data up to the possible checksum field. For efficiency reasons we contribute an algorithm that relies on the homomorphic property of CRC that works as follows. In a **caching phase** (fig. 8) we calculate all checksum values of the bit sequence $1(0)^{n-1}$ whereby n is the length of data up to the CRC field candidate.

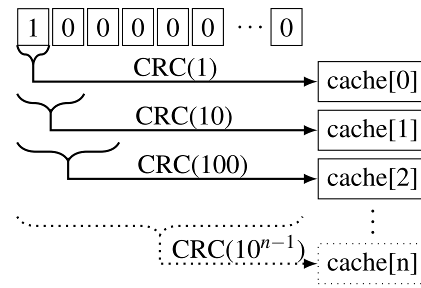


Figure 8: Filling the CRC cache

Then we apply (fig. 9) all these cached CRC values stepwise and in reverse order to the initial CRC value over the complete datarange via XOR. This enables us to shift the start position of the datarange window without having to calculate the CRC over and over again. In this way we loop over the most common CRC variants and parameters used in Internet of Things communication, for example CRC16-CCITT.

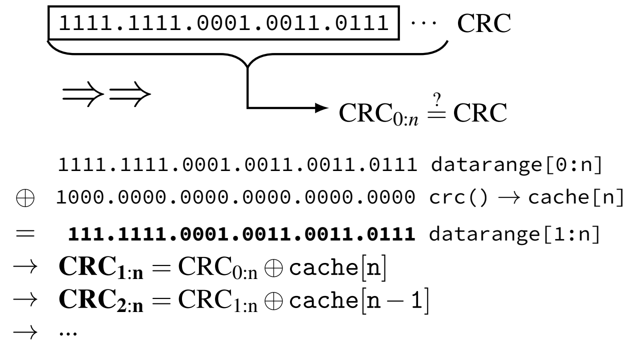


Figure 9: Checking datarange using the cache

Sometimes there are several possible results for checksum algorithm, parameters and datarange. If this is the case, our algorithm outputs the result that works for the majority of tested messages. The Checksum Field Engine returns the position of the checksum field, the determined checksum variant and it's parameters and, additionally, the associated datarange the checksum covers. Note, the engine is not limited to CRCs but can also detect more specialized checksums, for example, the checksum algorithms used in the Wireless Short Packet Protocol [13].

3.7 Message Type Assignment

The engines from previous sections return a set of labels. In the next step, we group these labels into *message types* based on their **messages indices**. If the indices match and the fields do not *overlap* a new message type is created from these fields and messages as shown in the example in fig. 10. If two ranges with shared message indices overlap, we choose the range with the highest score. For multiple overlapping ranges, we choose the ranges that maximize the total score.

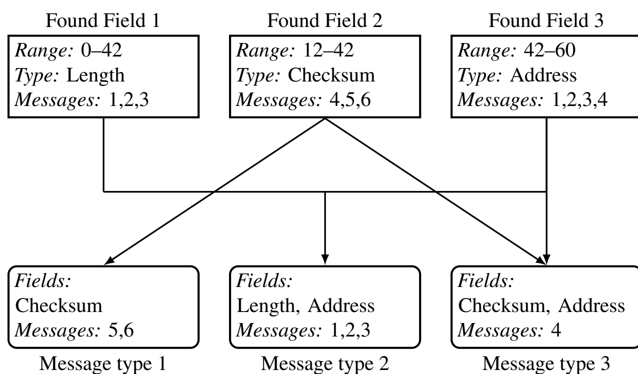


Figure 10: Merging of found fields to infer message types

Having created the message types, the start and end indices of the fields are updated so that they are absolute and not relative to the sync word anymore. In other words, we reverse the alignment transformation we made in section 3.1.2. Moreover, we add `Preamble` and `Sync` labels to the message types in this stage based the information gathered in section 3.1.

3.8 Considering a priori knowledge

In previous sections we assumed that protocol specifications are completely unknown. There are various potential sources for (partial) protocol knowledge such as leaked documents or stickers with addresses printed on investigated devices. In this section, we show how to consider prior knowledge, that is, partially known message types and fields to improve accuracy. Note, this prior knowledge can also come from a previous run of our system, so that even totally unknown protocols profit.

We assume a priori knowledge to be **correct** but not necessarily **complete**, that is, messages may have a message type assigned where not all fields are labeled. This leads to the following **rules** when dealing with *a priori* 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.

The general idea of our solution is to run the previously mentioned engines **for each message type**. For a totally unknown protocol we assume all messages have a common message type which entails no labels, so that the previous description of our algorithm remains valid as a special case of this more general algorithm. Our system considers non-empty message types in the following way:

- Engines for fields that are already present in the message type are skipped.
- All previously labeled ranges are ignored by the engines.
- If a new message type needs to be created (section 3.7) the original message type is **split**, that is, it's labels are copied over to all newly created types.

Splitting message types may seem like a contradiction of rule 3, but after splitting the new message types include the same labels as the original message type so that only new knowledge is added and prior labels remain unchanged.

Since our algorithm profits from prior knowledge, we run it until no new fields are found in an iteration. The general procedure is shown in listing 1.

```
1  i = 0
2  while new_fields_found and i < max_iterations:
3    i = i + 1
4    for mt in existing_message_types:
5      new_fields = []
6      for engine in engines:
7        if field_of_engine not in mt:
8          new_fields.extend(engine.run(mt))
9
10     # Split message type if necessary
11     add_to_message_type(mt, new_fields)
```

Listing 1: Iterative algorithm

4 Experimental Validation

In this section we evaluate the accuracy and performance of our algorithm. For accuracy evaluation, we use 8 generated protocols with different fields and number of involved participants. An overview of the protocols is given in table 1.

The protocols vary strongly in field lengths and cover, for example, messages with long preamble (72 bits), short synchronization (4 bits), non-default preamble patterns (`0x8888`)

or even no preamble at all. Moreover, the **byte order** of length and sequence number fields varies. The payload data for all protocols is randomly generated. A detailed description of these protocols is given in appendix A.

Given a set of messages M with expected fields E_i for message i, we define the **accuracy** a as

$$a = \frac{1}{|M|} \cdot \sum_{i=1}^{|M|} \frac{|E_i \cap F_i|}{|E_i|} \tag{3}$$

whereby F_i are the found fields for message i. In the following experiments we assume there is no prior knowledge available, that is, only the captured messages and the number of involved participants are presented to the algorithm.

4.1 Test against number of messages

Wireless communication tends to be efficient and, therefore, not use many messages for a process such as switching on a light bulb. Furthermore, legal transmission windows limit the amount of messages a researcher can capture at a time. In this experiment, we investigate the effect of the number of captured messages on the accuracy of our algorithm. It can be expected that the accuracy increases when more messages are available to the algorithm. Results are shown in fig. 11.

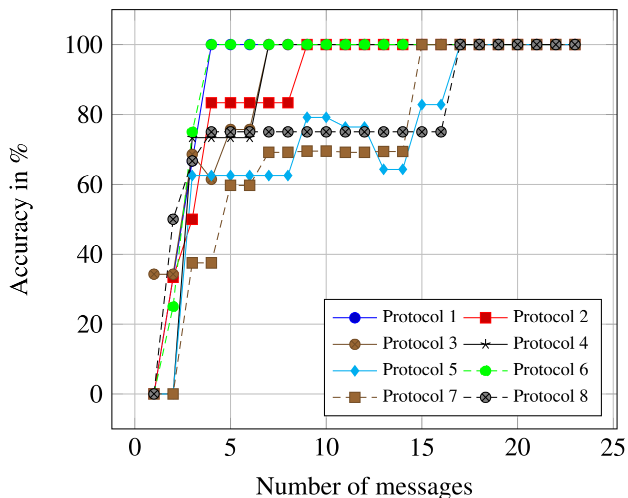


Figure 11: Accuracy for increasing number of messages

The accuracy reaches 100% for all protocols when more than 17 messages are available to the algorithm. For six of the protocols the algorithm infers fields with 100% accuracy when at least 7 messages are available. Protocol 5 and 7 need more messages than other protocols to reach full accuracy because three or four participants are involved in their communication so more messages are needed to infer participant addresses and find position of address fields.

4.2 Test against errors

We investigate how messages with errors affect the accuracy of our algorithm because broken messages are common in wireless captures. In this experiment, we create broken messages by setting the bits to random values beginning at a random message position. This is the worst case for the algorithm because some data remains valid in broken messages.

To calculate the accuracy for this experiment, we only consider non-broken messages in eq. (3) because the algorithm will not label broken parts of messages and the goal of this experiment is to find out how resistant the algorithm is against errors, that is, how many intact messages are labeled correctly while having increasing number of broken messages.

We start for all protocols with a total of 30 messages, to ensure they are labeled with 100% accuracy when no message is broken. Then we start to increase the number of broken messages. The result of this experiment is shown in fig. 12.

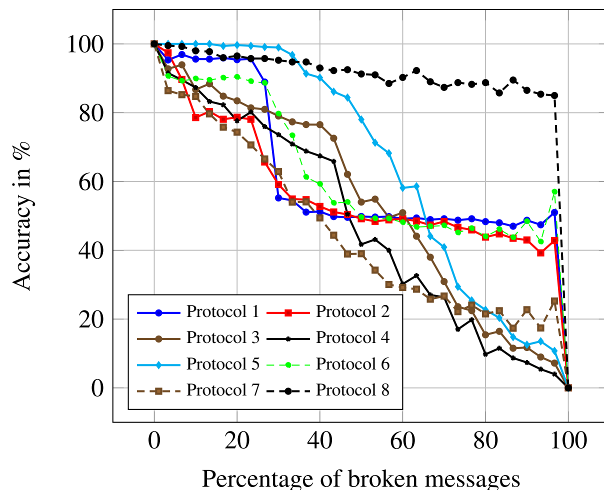


Figure 12: Accuracy for increasing number of messages with error for a total of 30 messages with 100 runs per measurement

Results suggest, that the algorithm is quite robust against errors. The majority of protocols are labeled with more than 80% accuracy when 20% of messages are broken.

4.3 Performance measurement

We evaluate the performance of our algorithm for an increasing number of messages. In contrast to the previous section we use real-world smart-home protocols to measure the performance for practical protocols. We use three real world captures for evaluation. First, the communication of two EnOcean devices using the Wireless Short Packet Protocol [13]. Second, a challenge-response procedure between a wireless door lock and a remote control from german manufacturer Homematic using the proprietary BidCoS protocol. Third, a communication of two RWE Smarthome devices using a unnamed proprietary protocol developed by eQ-3.

Table 1: 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	2	data	10/10 byte	16	16	8	16	16	8	**8**
	CRC8 CCITT		ack	×	16	16	8	×	16	×	**8**
4	contains ack and	2	data	8/64 byte	16	16	8	16	16	×	**16**
	CRC16 CCITT		ack	×	16	16	8	×	16	×	**16**
5	three participants	**3**	data	8/64 byte	16	16	8	16	16	8	×
	with ack frame		ack	×	16	16	8	×	16	×	×
6	short address	2	data	8/64 byte	×	16	8	**8**	×	8	×
7	four participants,	**4**	data	8/8 byte	**16**	**16**	8	**24**	**24**	×	16
	varying preamble		ack	×	**8**	**16**	×	×	**24**	×	16
	size, varying sync		kex	64/64 byte	**24**	**16**	×	**24**	**24**	×	16
8	nibble fields + LE	1	data	542/260 byte	**4**	**4**	16 (**LE**)	×	×	16 (**LE**)	×

We start with a number of eight messages because our algorithm finds all physical fields in these real-world protocols when using at least eight messages. For every performance measurement, the accuracy of field inference was 100% throughout these experiments. The measurements were taken on a computer with i7-6700K CPU@4.00GHz and 16GB RAM. For every number of messages we take the mean of 100 performance measurements. Results are shown in fig. 13.

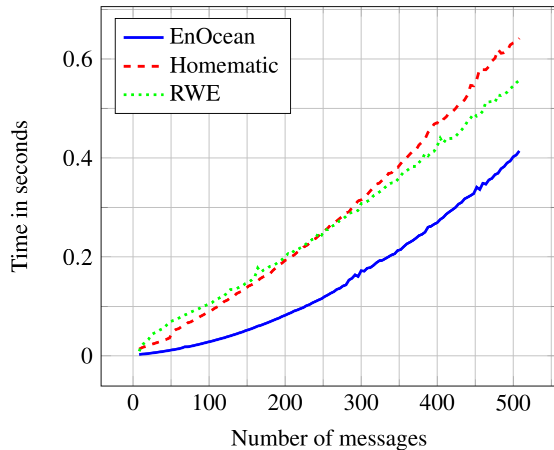


Figure 13: Algorithm performance for real-world smart-home protocols with increasing number of messages

Results show that our algorithm performs in under one second even for 500 messages. While the algorithm is designed to work on few messages, it can also deal with a larger number of messages. Therefore, a typical protocol capture with around 10 messages can be bootstrapped by the algorithm without a visible delay for the end-user. Since engines of our algorithm work independently, the performance could be further improved by *parallelization*. The results, however, indicate this optimization is not required in practice.

5 Related Work

In their survey Narayan et al. [15] describe several methods for automated reverse engineering of network protocols. Most of these algorithms are designed for application level protocols like HTTP. In fact, we are not aware of a single approach that focuses on the physical layer like our contribution. This is a harder problem because we cannot use information from lower OSI layers such as IP addresses.

Protocol Informatics (PI) [3] is a pioneer work in this field. It uses the Needleman-Wunsch sequence alignment algorithm [16] from bioinformatics to handle variable length fields like *USER alice* and *USER bob* in FTP whereby the username has a variable length. We do not use Needleman-Wunsch algorithm as we do not expect variable length fields or optional fields. We focus on physical layer binary protocols and use a histogram to measure similarity between messages because sequence alignment would be prone to transmission errors that are common in wireless communications especially if a Software Defined Radio is used for recording.

RolePlayer [12] and ScriptGen [14] are based on PI's sequence alignment algorithm and infer the protocol state machine to the level necessary for adaptive replay. While they are not designed purely for protocol reverse engineering they can mimic both sides of a communication by learning the protocol from network traces, for example, in order to emulate the command and control messages of a botnet.

Discoverer [11] improved this initial work. The authors showed that Needleman-Wunsch algorithm is not suited for messages with the same format as it would not consider, e.g., length fields. Their method is based on token classes (text or binary) and Format Distinguisher (FD) fields such as `\n` or `\t`. A FD separates encapsulated protocols from each other, e.g., in CIFS/SMB protocol a NetBIOS header encapsulates an SMB header, which in turn may encapsulate a RPC message.

Based on these FD fields they perform a recursive clustering to the encapsulated protocols and finally merge results together. Note, for wireless IoT protocols a distinction between text or binary tokens is generally superfluous because these protocols use a binary format for efficiency. Furthermore, IoT protocols seldom rely on FD fields but rather preambles like `10101010`.

Antunes et al. [1] boil down the problem of protocol reverse engineering to building two finite automata: One automaton describes the protocol format and the other one the protocol state machine. The difficulty is to build the automata from a limited sample set of the language represented by the automaton. Their approach is designed for text protocols with a FIELD value structure such as `USER bob`. The space behind `USER` is an easy detectable. Physical layer protocols, however, do not consist of such a FIELD value structure.

The aforementioned approaches work on network traces. In contrast to that several authors [6–8, 10, 18] investigate the execution trace of the binary that implements the network protocol. This is referred to as *dynamic analysis* and yields additional information. For example, four consecutive bytes can be identified as a DWORD, that is, a four-byte integer if the execution trace reveals that they are processed as a DWORD. We assume the program binary is not accessible for analysis and therefore can not build on this idea.

The most comparable works to our approach are *FieldHunter* [4] and *WASp* [9]. *FieldHunter* is designed to infer field types from network traces. Authors use statistical methods to identify interesting ranges in messages and propose heuristics to derive field types. *FieldHunter* is designed to work on both text-based and binary protocols but also operates on a higher OSI layer as it, for example, uses IP addresses to infer host identifiers. *WASp* is a tool to automatically analyze proprietary wireless byte protocols built on top of IEEE 802.15.4. Additionally, *WASp* automatically generates spoofed packets based on found rules. Compared to our work *WASp* relies on known field positions such as Preamble or Length from the IEEE 802.15.4 standard.

Compared to previous work our contribution is novel in two ways. First, our algorithm works on the *physical layer* without making any assumption about the underlying protocol. Second, it is designed for *wireless protocols*, especially those used in Internet of Things, and is optimized for smaller datasets, that is, 10-100 messages compared to previous approaches that work on 1000-10000 messages. The reason for this is that IoT communication cannot be as easily triggered as, say, DNS because there are duty cycles on certain frequencies and, moreover, you have to perform physical actions like pressing a button in order to trigger messages.

6 Conclusion

We propose a framework for automatic protocol reverse engineering especially aimed at proprietary wireless protocols under consideration of their specifics such as RSSI and preambles. Our solution can perform a bootstrap of an unknown protocol but is also able to take prior knowledge into account. We believe this will speed up and simplify security investigations of Internet of Things devices. The framework consists of dedicated engines for inferring protocol fields such as `Length` or `Address` and automatically creates message types based on newly found and previously known fields (section 3.7).

In experiments, all fields of test protocols (appendix A) are found with 100% accuracy if enough messages are captured. A major design goal of the algorithm is to be robust against errors. In experiments, field inference accuracy mostly stays above 80% when 20% of messages contain random errors.

We verified our algorithm with three real-world protocols (section 4.3) and it infers all physical fields when at least eighth messages are available. While this is a limited test set, we expect our solution to work on a variety of IoT wireless protocols as they share common structures, for example, preamble and synchronization determined by hardware chips.

Finding fields in the payload such as an on/off bit of a wireless socket or the temperature of a radiator valve is not in the scope of this paper so it has to be manually performed by the researcher. We believe, however, that finding preamble, synchronization, addresses, sequence number and checksum automatically will help security researchers because payloads can be quickly identified when other fields are labeled.

Future work is the suggestion of attacks based on the found fields and detection of cryptography in data payload as well. Ultimately, an automated security score could be calculated based on found cryptography and protocol complexity and, thereby, giving the researcher an initial idea about the protocol security right from the captured messages.

Availability

Our algorithm is available as open source software as part of the Universal Radio Hacker [17] and integrated into its GUI as shown in fig. 14. All shown labels and message types in this screenshot were automatically found by the algorithm after hitting the `Analyze` button.

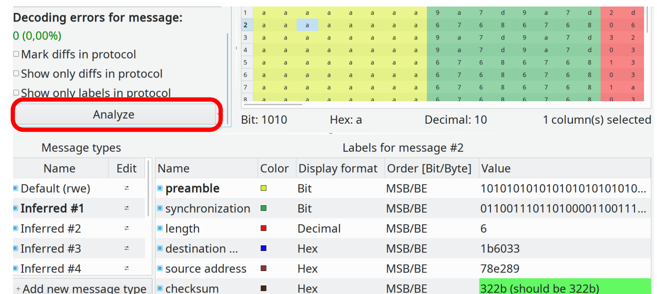


Figure 14: Integration into URH

References

[1] João Antunes, Nuno Neves, and Paulo Verissimo. Reverse engineering of protocols from network traces. pages 169–178.

[2] aweatherguy. Oregon Scientific RF Protocol Description.

[3] Ma Beddoe. Network protocol analysis using bioinformatics algorithms.

[4] Ignacio Bermudez, Alok Tongaonkar, Marios Iliofotou, Marco Mellia, and Maurizio M. Munafò. Towards automatic protocol field inference. 84:40–51.

[5] Eric Blossom. GNU radio: Tools for exploring the radio frequency spectrum. 2004:4.

[6] J. Caballero, P. Poosankam, C. Kreibich, and Song D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. pages 621–634.

[7] Juan Caballero and Dawn Song. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. 57(2):451–474.

[8] Juan Caballero and Dawn Song. Polyglot : Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis Polyglot : Automatic Extraction of Protocol Message Format using. pages 317–329.

[9] Kibum Choi, Yunmok Son, Juhwan Noh, Hocheol Shin, Jaeyeong Choi, and Yongdae Kim. Dissecting Customized Protocols: Automatic Analysis for Customized Protocols based on IEEE 802.15.4. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks - WiSec '16*, pages 183–193. ACM Press.

[10] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. pages 110–125.

[11] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. (2):199–212.

[12] Weidong Cui, Vern Paxson, Nicholas C Weaver, and Randy H Katz. Protocol-Independent Adaptive Replay of Application Dialog. 4(4):279–293.

[13] ISO. Information technology – Home electronic system (HES) architecture – Part 3-11: Frequency modulated wireless short-packet (FMWSP) protocol optimised for energy harvesting – Architecture and lower layer protocols.

[14] Corrado Leita, Ken Mermoud, and Marc Dacier. ScriptGen: An automated script generation tool for honeyd. 2005:203–214.

[15] John Narayan, Sandeep K. Shukla, and T. Charles Clancy. A Survey of Automatic Protocol Reverse Engineering Tools. 48(3):1–26.

[16] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. 48(3):443–453.

[17] Johannes Pohl and Andreas Noack. Universal Radio Hacker: A Suite for Analyzing and Attacking Stateful Wireless Protocols. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association.

[18] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S. Anna. Automatic Network Protocol Analysis. pages 1–18.

A Protocols

In this section we describe the protocols used for experiments in section 4. The payload for each protocol was, if not stated otherwise, 8 byte for every even and 64 byte for every odd message. The payload data was randomly generated. The sending participant was changed cyclical after each message to ensure every participant sends a message. The only exception is for protocols that contain an ACK message type: for such protocols an ACK is sent by the receiver for every incoming message.

A.1 Protocol 1

There were 2 participants involved in communication: Alice (`0xdead`) and Bob (`0xbeef`). The protocol has one message type with the following fields:

- preamble: `0xaa`
- synchronization: `0x1337`
- length: 8 bit
- source address: 16 bit
- destination address: 16 bit
- sequence number: 8 bit

A.2 Protocol 2

There were 2 participants involved in communication: Alice (`0xdead01`) and Bob (`0xbeef24`). The protocol has one message type with the following fields:

- preamble: `0xaaaaaaaaaaaaaaaaaa`
- synchronization: `0x1337`
- length: 8 bit
- source address: 24 bit
- destination address: 24 bit
- sequence number: 16 bit (**big endian**)

A.3 Protocol 3

There were 2 participants involved in communication: Alice (`0x1337`) and Bob (`0xbeef`). The protocol has 2 message types with the following fields:

- **data**
  - preamble: `0xaaaa`
  - synchronization: `0x9a7d`
  - length: 8 bit
  - source address: 16 bit
  - destination address: 16 bit
  - sequence number: 8 bit
  - payload: 10 byte
  - checksum: CRC8 CCITT
- **ack**
  - preamble: `0xaaaa`
  - synchronization: `0x9a7d`
  - length: 8 bit
  - destination address: 16 bit
  - checksum: CRC8 CCITT

A.4 Protocol 4

There were 2 participants involved in communication: Alice (`0x1337`) and Bob (`0xbeef`). The protocol has 3 message types with the following fields:

- **data**
  - preamble: `0x8888`
  - synchronization: `0x9a7d`
  - length: 8 bit
  - source address: 16 bit
  - destination address: 16 bit
  - payload: 8 byte
  - checksum: CRC16 CCITT
- **data2**
  - preamble: `0x8888`
  - synchronization: `0x9a7d`
  - length: 8 bit
  - source address: 16 bit
  - destination address: 16 bit
  - payload: 64 byte
  - checksum: CRC16 CCITT
- **ack**
  - preamble: `0x8888`
  - synchronization: `0x9a7d`
  - length: 8 bit
  - destination address: 16 bit
  - checksum: CRC16 CCITT

A.5 Protocol 5

There were 3 participants involved in communication: Alice (`0x1337`), Bob (`0xbeef`) and Carl (`0xcafe`). The protocol has 2 message types with the following fields:

- **data**
  - preamble: `0xaaaa`
  - synchronization: `0x9a7d`
  - length: 8 bit
  - source address: 16 bit
  - destination address: 16 bit
  - sequence number: 8 bit
- **ack**
  - preamble: `0xaaaa`
  - synchronization: `0x9a7d`
  - length: 8 bit
  - destination address: 16 bit

A.6 Protocol 6

There were 2 participants involved in communication: Alice (`0x24`) and Bob (`0xff`). The protocol has one message type with the following fields:

- synchronization: `0x8e88`
- length: 8 bit
- source address: 8 bit
- sequence number: 8 bit

A.7 Protocol 7

There were 4 participants involved in communication: Alice (`0x313370`), Bob (`0x031337`), Charly (`0x110000`) and Daniel (`0x001100`). The protocol has 3 message types with the following fields:

- **data**
  - preamble: `0xaaaa`
  - synchronization: `0x0420`
  - length: 8 bit
  - destination address: 24 bit
  - source address: 24 bit
  - payload: 8 byte
  - checksum: CRC16 CC1101
- **ack**
  - preamble: `0xaa`
  - synchronization: `0x2222`
  - destination address: 24 bit
  - checksum: CRC16 CC1101
- **kex**
  - preamble: `0xaaaaaa`
  - synchronization: `0x6767`
  - destination address: 24 bit
  - source address: 24 bit
  - payload: 64 byte
  - checksum: CRC16 CC1101

A.8 Protocol 8

The protocol has 2 message types with the following fields:

- **data1**
  - preamble: `0xa`
  - synchronization: `0x9`
  - length: 16 bit (**little endian**)
  - sequence number: 16 bit (**little endian**)
  - payload: 542 byte
- **data2**
  - preamble: `0xa`
  - synchronization: `0x9`
  - length: 16 bit (**little endian**)
  - sequence number: 16 bit (**little endian**)
  - payload: 260 byte