TECHNISCHE UNIVERSITÄT DARMSTADT

SOFTWARE SUPPORT FOR MANUAL REVERSE ENGINEERING OF BINARY PROTOCOLS

MIRA SOPHIE WELLER

Master's Thesis

November 30, 2022

Secure Mobile Networking Lab
Department of Computer Science
Technische Universität Darmstadt

SECURE MOBILE NETWORKING

Mira Sophie Weller, *Software Support for Manual Reverse Engineering of Binary Protocols*, Master's Thesis, Technische Universität Darmstadt, 2022.

SEEMOO-MSC-0162
Date of submission: November 30, 2022

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Dr.-Ing. Jiska Classen

Secure Mobile Networking Lab
Department of Computer Science
Technische Universität Darmstadt

ABSTRACT

In binary protocol reverse engineering, researchers need to analyze and document the structure of packets. But there is no effective tool to annotate packet data and develop protocol dissectors. Existing software to analyze binary data usually either has no support for packet-based captures, or lacks annotation and on-the-fly parsing features. In this thesis, we develop *PRE Workbench*, a software that supports reverse engineers in analyzing proprietary binary protocols, using a custom-built viewer for binary data and a specialized description language for binary protocol structures. Our software supports verifying the documented structure with fast round-trip times as well as generating Wireshark dissectors as output. User studies and tests on known protocols show that our software simplifies reverse engineering of protocols and creating dissectors for them.

ZUSAMMENFASSUNG

Beim Reverse Engineering von Binärprotokollen müssen Forscher die Struktur der Pakete analysieren und dokumentieren. Es gibt jedoch kein effektives Werkzeug zur Annotierung von Paketdaten und zur Entwicklung von Protokoll-Dissektoren. Bestehende Software für die Analyse von Binärdaten bietet in der Regel entweder keine Unterstützung für paketbasierte Eingabedaten oder es fehlt an Annotations- und Parsing-Funktionen. In dieser Arbeit entwickeln wir *PRE Workbench*, eine Software, die Reverse Engineers bei der Analyse von proprietären Binärprotokollen unterstützt, indem sie einen maßgeschneiderten Viewer für Binärdaten und eine spezielle Beschreibungssprache für binäre Protokollstrukturen verwendet. Unsere Software unterstützt die schnelle Verifizierung der dokumentierten Struktur sowie die Generierung von Wireshark-Dissektoren daraus. Nutzerstudien und Tests mit bereits bekannten Protokollen zeigen, dass diese Software das Reverse-Engineering von Protokollen und die Erstellung von Dissektoren vereinfacht.

If you want to change the future,
start living as if you're already there.

— Lynn Conway

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my parents for supporting me in all the years of my studies, and to Sascha for his moral and emotional support while writing this thesis. I love you.

Special thanks for giving helpful advice while writing this thesis goes to Jiska Classen and Max Maass.

I especially thank Jiska Classen and Sascha Hoffmann for proofreading my thesis, and for many much-appreciated words of encouragement. Furthermore, thank you to all participants of my user studies for their time and input.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ADB	Android Debug Bridge
AES	Advanced Encryption Standard
ARI	Apple Remote Invocation
AST	Abstract Syntax Tree
BLE	Bluetooth Low Energy
CLI	Command-Line Interface
CSV	Comma-Separated Values
DPI	Deep Packet Inspection
GATT	Generic ATTribute Profile
GUI	Graphical User Interface
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IV	Initialization Vector
MDI	Multiple-Document Interface
PDML	Packet Description Markup Language
PGDL	Protocol Grammar Description Language
PRE	Protocol Reverse Engineering
PyPI	Python Package Index
RE	Reverse Engineering
RPC	Remote Procedure Call
SDR	Software Defined Radio
SPI	Serial Peripheral Interface
STA	Static Trace Analysis
TCP	Transmission Control Protocol

TDI	Tabbed-Document Interface
TLV	Type-Length-Value
Tvb	Testy, Virtual(-izable) Buffer of guint8*'s
UART	Universal Asynchronous Receiver / Transmitter
UI	User Interface

1 INTRODUCTION

This thesis revolves around the reverse engineering of proprietary transmission protocols based on binary, i.e., non-text-based data structures. We design, implement and evaluate the tool *PRE Workbench*, that interactively supports reverse engineers in this process. In the following sections, we give a brief overview of the motivation for why reverse engineering is done and why this tool is useful. We show what this tool contributes to research, and provide an outline of the structure of this document.

1.1 MOTIVATION

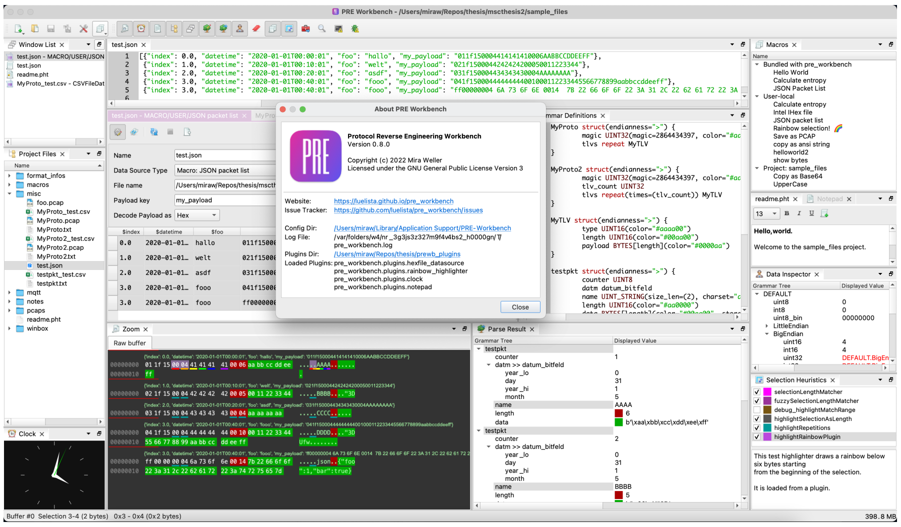
There are various reasons to analyze proprietary protocols. First of all, it can be useful as part of *fundamental research* to investigate the data structure and functionality of a protocol in order to publish this as a paper or use it as a starting point for further research.

Subsequently, this information can be used in the context of *security audits* to find security vulnerabilities in the protocol structure or in implementations. While manual reversing usually finds logic bugs, finding memory corruption bugs in protocol implementations can be automated with fuzzing. Fuzzing is more efficient with detailed protocol knowledge. Furthermore, one can establish *compatibility* on this basis, i.e. by developing free software compatible with proprietary devices. For example, one could develop a free alternative to an existing non-free app, or develop an open-source server for IoT devices that have been discontinued by the manufacturer or whose manufacturer cloud is not trustworthy. Finally, one can also perform *privacy audits* to find out what data is actually being transferred, whether it is transferred securely, and whether the device is compliant with privacy regulations.

From our own experience reversing protocols, as well as questions and projects online, we saw the need for a tool to support the manual protocol reverse engineering process. There are many existing tools for reversing binary file formats, as well as for inspecting well-known protocols. However, we could hardly find interactive software that supports the reverse engineering of unknown protocols. By interviewing researchers who reverse engineer on a regular basis, we were able to confirm the lack of and need for such tooling.

1.2 CONTRIBUTIONS

This inspired us to develop *PRE Workbench* (see Figure 1), a software to support researchers in reverse engineering protocols and documenting the results. Therefore, we conducted a user study to find out, which features researchers would like to see in an interactive tool to support their work. Our software supports various sources to import protocol traffic from, helps the discovery process by displaying different views and heuristic-based highlighting on data, and aids in documenting and sharing findings. *PRE Workbench* is published under an open-source license [69].



```
[{"index": 0.0, "datetime": "2020-01-01T00:00:01", "foo": "hallo", "my_payload": "011f150004414141410006AABBCCDDEEFF"},
 {"index": 1.0, "datetime": "2020-01-01T00:10:01", "foo": "welt", "my_payload": "021f15000442424242000500112233 44"},
 {"index": 2.0, "datetime": "2020-01-01T00:20:01", "foo": "asdf", "my_payload": "031f150004434343430004AAAAAAAA"},
 {"index": 3.0, "datetime": "2020-01-01T00:40:01", "foo": "fooo", "my_payload": "041f150004444444440010001122334455667788 99aabbccddeeff"},
 {"index": 3.0, "datetime": "2020-01-01T00:40:01", "foo": "fooo", "my_payload": "ff00000004 6A 73 6F 6E 0014  7B 22 66 6F 6F 22 3A 31 2C 22 62 61 72 22 3A
```

Name	test.json
Data Source Type	Macro: JSON packet list
File name	/Users/miraw/Repos/thesis/mscth
Payload key	my_payload
Decode Payload as	Hex

$index	$datetime	$foo	
0.0	2020-01-01…	hallo	011f15000
1.0	2020-01-01…	welt	021f15000
2.0	2020-01-01…	asdf	031f15000
3.0	2020-01-01…	fooo	041f15000
3.0	2020-01-01…	fooo	ff0000000

About PRE Workbench

Protocol Reverse Engineering Workbench
Version 0.8.0

Copyright (c) 2022 Mira Weller
Licensed under the GNU General Public License Version 3

Website:	https://luelista.github.io/pre_workbench
Issue Tracker:	https://github.com/luelista/pre_workbench/issues
Config Dir:	/Users/miraw/Library/Application Support/PRE-Workbench
Log File:	/var/folders/w4/nr_3g3js3z327m9f4v4bs2_h0000gn/T/ pre_workbench.log
Plugins Dir:	/Users/miraw/Repos/thesis/prewb_plugins
Loaded Plugins:	pre_workbench.plugins.hexfile_datasource pre_workbench.plugins.rainbow_highlighter pre_workbench.plugins.clock pre_workbench.plugins.notepad

Grammar Definitions

```
MyProto struct(endianness=">") {
        magic UINT32(magic=2864434397, color="#ac
        tlvs repeat MyTLV
}

MyProto2 struct(endianness=">") {
        magic UINT32(magic=2864434397, color="#ac
        tlv_count UINT32
        tlvs repeat(times=(tlv_count)) MyTLV
}

MyTLV struct(endianness=">") {
        type UINT16(color="#aaaa00")
        length UINT16(color="#00aa00")
        payload BYTES[length](color="#0000aa")
}

testpkt struct(endianness=">") {
        counter UINT8
        datm datum_bitfeld
        name UINT_STRING(size_len=(2), charset="c
        length UINT16(color="#aa0000")
```

```
{'index': 0.0, 'datetime': '2020-01-01T00:00:01', 'foo': 'hallo', 'my_payload': '011f150004414141410006AABBCCDDEEFF'}
00000000  01 1f 15 00 04 41 41 41  41 00 06 aa bb cc dd ee   .....AAAA.......
00000010  ff                                                 .
{'index': 1.0, 'datetime': '2020-01-01T00:10:01', 'foo': 'welt', 'my_payload': '021f15000442424242000500112233 44'}
00000000  02 1f 15 00 04 42 42 42  42 00 05 00 11 22 33 44   .....BBBB...."3D
{'index': 2.0, 'datetime': '2020-01-01T00:20:01', 'foo': 'asdf', 'my_payload': '031f150004434343430004AAAAAAAA'}
00000000  03 1f 15 00 04 43 43 43  43 00 04 aa aa aa aa      .....CCCC......
{'index': 3.0, 'datetime': '2020-01-01T00:40:01', 'foo': 'fooo', 'my_payload': '041f150004444444440010001122334455667788 99aabbccddeeff'}
00000000  04 1f 15 00 04 44 44 44  44 00 10 00 11 22 33 44   .....DDDD...."3D
00000010  55 66 77 88 99 aa bb cc  dd ee ff                  Ufw........
{'index': 3.0, 'datetime': '2020-01-01T00:40:01', 'foo': 'fooo', 'my_payload': 'ff00000004 6A 73 6F 6E 0014  7B 22 66 6F 6F 22 3A 31 2C 22 62 61 72 2
00000000  ff 00 00 00 04 6a 73 6f  6e 00 14 7b 22 66 6f 6f   .....json..{"foo
00000010  22 3a 31 2c 22 62 61 72  22 3a 74 72 75 65 7d      ":1,"bar":true}
```

Grammar Tree	Displayed Value
testpkt	
counter	1
datm >> datum_bitfeld	
year_lo	0
day	31
year_hi	1
month	5
name	AAAA
length	🟥 6
data	🟩 b'\xaa\xbb\xcc\xdd\xee\xff'
testpkt	
counter	2
datm >> datum_bitfeld	
year_lo	0
day	31
year_hi	1
month	5
name	BBBB
length	🟥 5

Hello, world.

Welcome to the sample_files project.

Grammar Tree	Displayed Value
DEFAULT	
uint8	0
int8	0
uint8_bin	00000000
LittleEndian	
BigEndian	
uint16	4
int16	4
uint32	DEFAULT.BigEn

- ✓ selectionLengthMatcher
- ✓ fuzzySelectionLengthMatcher
- debug_highlightMatchRange
- ✓ highlightSelectionAsLength
- ✓ highlightRepetitions
- ✓ highlightRainbowPlugin

This test highlighter draws a rainbow below six bytes starting
from the beginning of the selection.

It is loaded from a plugin.

Buffer #0 Selection 3-4 (2 bytes) 0x3 - 0x4 (0x2 bytes) 398.8 MB

Figure 1: Screenshot of PRE Workbench

1.3 STRUCTURE

In the following Chapter 2, we introduce theoretical concepts required to understand the later topics. After that, we conduct a review of related publications and software in Chapter 3. Chapter 4 explains the design process, core concepts and architecture of *PRE Workbench*, the tool we build. As a starting point for the design process, we conducted user interviews with several researchers, which we summarize here. In Chapter 5 we go into details of the software development, including the Graphical User Interface (GUI) components, parser, and Wireshark dissector generation. The evaluation in Chapter 6 is based on the one hand on a continuation of the user study from Chapter 4, on the other hand we examine, how well the description language is applicable to some exemplary protocols and how fast the parsers work. In Chapter 7 we discuss the results of our work and present possible future work. Finally, we conclude the thesis in Chapter 8, summarizing our contribution.

2 BACKGROUND

This chapter contains definitions and explanations of the concepts, methods and data structures used in the later chapters of this thesis. It can be referred to later on, as the following sections are referenced when the concepts are first used.

2.1 DEFINITIONS

In this section, we define different approaches to reverse engineering.

2.1.1 *Reverse Engineering*

In general, Reverse Engineering (RE) is "the process of studying another company's product to see how it is made, sometimes in order to be able to copy it" [49]. In the context of software, it usually means *binary reverse engineering*, so analyzing the compiled binary of a proprietary software, without having access to the source code. We can further divide this into *static* and *dynamic* RE. In static RE, a disassembler or decompiler is used to recreate human-readable code from the compiled binary executable, which is in turn studied by the reverse engineer. In the dynamic variant, the binary is executed, while closely monitoring its control flow and memory contents using a debugger, allowing the reverse engineer to study the runtime behaviour of the application [17].

2.1.2 *Protocol Reverse Engineering*

In contrast, *Protocol Reverse Engineering (PRE)* is the process of analyzing and documenting a communication protocol at the application or network layer, understanding the syntax and the semantics, sometimes in order to build a compatible protocol implementation. Based on the classifications by Kleber, Maile, and Kargl [28] and Li and Chen [30], we categorize PRE approaches in two main categories and a combination thereof:

- *Entity analysis* uses regular software RE methods, including static and dynamic binary RE, on implementations of the protocol (called entities). It is also called program-based PRE.
- *Trace or traffic analysis*, also termed network-based PRE, relies only on the traffic between entities. *Static Trace Analysis (STA)* requires only passive listening capabilities on the communication

medium, but no manipulation of an end entity. Offline analysis of previously recorded traces is possible, making it reproducible and allowing for usage of non-real-time algorithms. In *dynamic traffic analysis* the entity is stimulated by injecting packets or interacting with e.g. a device's user interface.

- *Hybrid* approaches combine entity and trace analysis and are commonly employed when performing manual analysis of complex protocols.

2.2 USER STUDIES

User studies can be conducted in all stages of a software development project [55]. In the product development phase they can influence the foundational goals of a product, ensuring the needs of future users will be considered. In addition to interviews, prototypes and click dummies can be used in this phase. Agile projects might replace the click dummies by runnable increments of the developed software. After each release, the success can be evaluated by further user studies.

In methods for user studies, generative and evaluative methods need to be distinguished [39]. The former will provide insights into user needs and possible UX optimizations before even starting development. They either start from a clean slate or from documents and prototypes and work with potential later users of a software, or people from the expected target demographic. Evaluative methods focus on a finished UX design or even a finished software, evaluating how well these are accepted by expected users or the established user base. They can be used to evaluate the success of a software project, or as the basis for a potential redesign or incremental improvement of the user interface.

2.3 USER INTERFACE CONCEPTS

In the following sections we describe some basic concepts of graphical user interfaces that are used in PRE Workbench or the prototypes for it.

2.3.1 *Window Handling*

For applications operating on any kind of document, there are several well-established styles to handle the use case of working on multiple documents at the same time [36]. For an overview, see Figure 2.

In a *single file application*, the software has no built-in support for working on multiple documents, instead, it relies on the operation system's multi-tasking support. Therefore, the user just opens multiple instances of the application to work on multiple documents. This

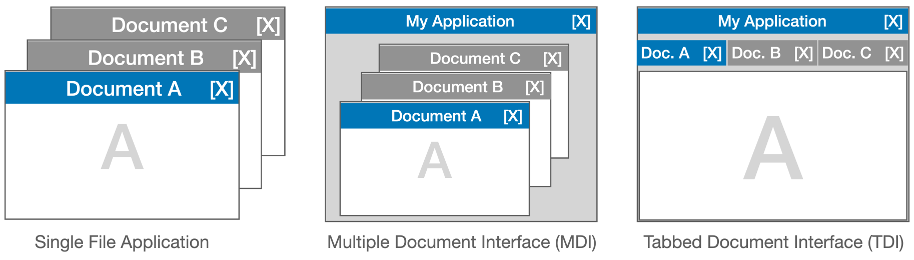


Figure 2: Schematic illustrations of different window-handling concepts

reduces implementation complexity, because no special handling for document switching needs to be implemented. It may come at the price of worse user experience, because of a cluttered desktop. Furthermore, sharing state between documents is harder with this approach.

Applications with the classic Multiple-Document Interface (MDI) provide a main window, in which multiple documents can be opened as overlapping child windows. The child windows are contained in the main window such that the user cannott move them outside the main window, and if the main window is moved, they are moved along. Classic MDI support is built into UI frameworks such as Windows Forms and Qt. However, it is not considered as user-friendly anymore, and many applications (e.g. Microsoft Office) moved to other interface styles. All documents are handled by the same application process, making implementation more complex, but simplifying shared state between documents.

More modern approaches provide a blend of better user experience and running in a single process. One common approach in current applications such as all web browsers is the Tabbed-Document Interface (TDI), and multi-windowed TDI. Here, only one document is shown at the same time, and a tab bar at the top of the window allows the user to switch between documents. This reduces clutter on the desktop, but is easier to handle by the user than classic MDI. Multi-windowed TDI allows the user to open several windows, all running in the same process, and move tabs between them. This is especially useful on systems with multiple monitors. A special case of TDI is used by most Integrated Development Environments (IDEs). They implement a tiled TDI, where users can place several tabs, containing documents or tool windows, next to each other. However, tiles are automatically arranged in a non-overlapping manner.

2.3.2 *List-Detail Pattern*

In a user interface following the list-detail pattern (Figure 3), a main list view and a detail view are displayed next to each other. The main list displays only the most important bits of information on each entity, commonly in the form of a table where each row represents an entity and each column a bit of information. When an entity from the

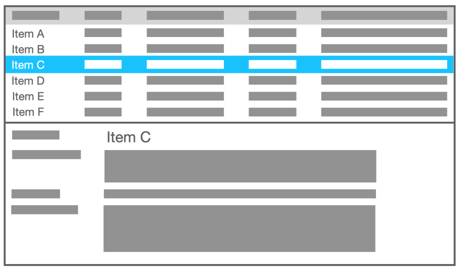


Figure 3: Schematic illustration of list-detail pattern

main list is selected, the detailed information on the selected entity is displayed in the detail view, often in a "Key: Value" format. If the entity has a graphical representation, e.g. if the entity is a picture, this can be displayed instead.

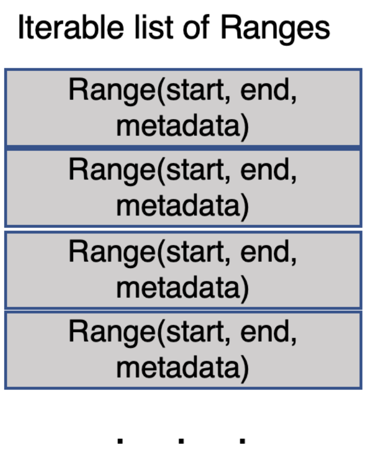
2.4 DATA STRUCTURES FOR INTERVAL INFORMATION

Our application needs to store metadata on ranges of bytes in a buffer. Metadata includes annotations added by the user (colors and text comments), parse results (name and value), and imported metadata from other applications. The byte ranges are allowed to overlap. There are many possible data structures to store this kind of data. The naive approach would be a simple list of `Range(start, end, metadata)` objects (see Figure 4a). We started using this approach. However, for painting the HexView display, we need to retrieve the relevant `Range` objects for each painted byte (all objects where `start <= byte index <= end`). If many `Range` objects are stored, retrieval is a performance bottleneck, because we need to scan the whole list (`O(n)` time complexity) for each painted byte.

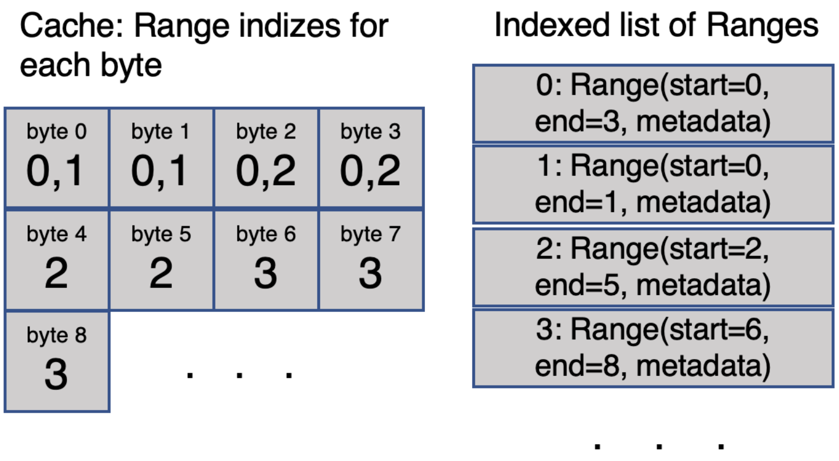
A simple optimization would be an array the length of the buffer, where each array element stores a list of references to the `Range` objects relevant to this byte (see Figure 4b). This is very time efficient (`O(1)` time complexity), but requires a lot of memory for larger files, and is especially memory inefficient for large files with few annotations.

We therefore decided on an approach that compromises between memory and access time. The buffer is divided into chunks of a fixed number of bytes [60]. For each chunk, a list of references to `Range` objects that have an overlap with the chunk is stored (see Figure 4c). Therefore, to search for `Range` objects containing a particular byte, only iterate over all Ranges in the chunk of that byte. Thus, the search time complexity is `O(c)`, where `c` is the chunk size.

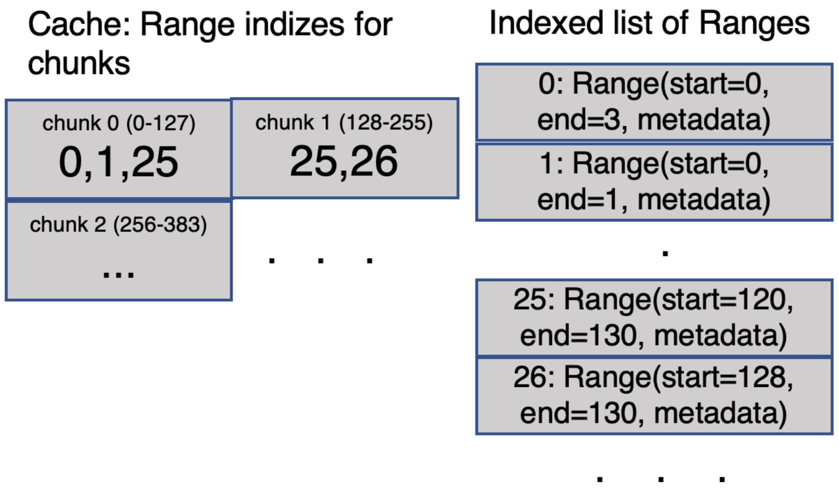
There are more complex and more efficient data structures to store intervals, for example interval trees [11] and nested containment lists [2]. However, we found our chunked approach fast enough for a lag-free GUI in regular use of the application. This approach is implemented in the `RangeList` class of our application.


Iterable list of Ranges
Range(start, end, metadata)
Range(start, end, metadata)
Range(start, end, metadata)
Range(start, end, metadata)
. . .

(a) Naive approach: Lowest memory usage, highest runtime


Cache: Range indizes for each byte
byte 0 0,1
byte 1 0,1
byte 2 0,2
byte 3 0,2
byte 4 2
byte 5 2
byte 6 3
byte 7 3
byte 8 3
. . .
Indexed list of Ranges
0: Range(start=0, end=3, metadata)
1: Range(start=0, end=1, metadata)
2: Range(start=2, end=5, metadata)
3: Range(start=6, end=8, metadata)
. . .

(b) Cached approach: Highest memory usage, lowest runtime


Cache: Range indizes for chunks
chunk 0 (0-127) 0,1,25
chunk 1 (128-255) 25,26
chunk 2 (256-383) ...
. . .
Indexed list of Ranges
0: Range(start=0, end=3, metadata)
1: Range(start=0, end=1, metadata)
.
25: Range(start=120, end=130, metadata)
26: Range(start=128, end=130, metadata)
. . .

(c) Chunked approach: Medium memory usage, medium runtime

Figure 4: Data structures for range information

3 RELATED WORK

In this chapter, we will discuss software tools for parsing well-known protocols, describing binary protocols and viewing and annotating arbitrary binary data. The last two categories are often found as features of hex editors. Research papers covered in the following sections include description languages for protocols which allow automatic parser generation and fuzzing. In distinction to the main topic of this thesis, we also give an overview of various automated reverse engineering approaches.

3.1 PROTOCOL ANALYSIS SOFTWARE

The first step to analyze network traffic is capturing and recording it. Often, the packet capture libraries libpcap [61] or Npcap [38] are used to record network packets at the interface layer with support of the operating system. If this is not possible due to missing permissions or encryption, we can use proxy software (e.g. mitmproxy [12]) to capture data streams at the transport, encryption or application layer, or import data from vendor debugging tools (e.g. Bluetooth logs from a phone). Another possibility are over-the-air sniffers for wireless protocols (e.g. BTLEmap [23]) and logic analyzers (e.g. Saleae logic analyzer [31]) for wired protocols. After capturing raw network traffic, we usually need to extract the payload data from the surrounding protocol stack. This problem is solved by multiple software tools which can automatically parse and analyze many known protocols.

3.1.1 *Wireshark*

The most popular network protocol analysis software is Wireshark [10], an open-source tool which combines raw network traffic capture capabilities with a large number of dissector modules. It can capture Ethernet, WLAN, USB, Bluetooth, and several other types of traffic.[37] Captured data is usually displayed live, and can be stored and loaded as PCAP files.

The main Graphical User Interface (GUI) is divided in three areas: packet list, packet details (a protocol tree) and packet bytes (a hex dump of the raw packet data). Upon selecting a row in the protocol tree, the corresponding bytes are highlighted, and vice versa.

The protocol tree is built by dissectors, which can be implemented in C or Lua in an imperative way. This gives maximum flexibility for complex protocols, but makes the parsing code very verbose, even for simple protocols. Each dissector receives as its input the payload of the underlying protocol. If the underlying protocol is a stream based protocol which needs segment reassembly (e.g. Transmission Control Protocol (TCP)), the dissector needs to handle incomplete packets and request more input, if necessary.

When developing a C dissector, Wireshark needs to be recompiled, leading to longer turn-around times. For Lua dissectors, and for the Generic and Python dissectors described below, this is not necessary, only a restart of Wireshark is required, making them easier and faster to develop. However, runtime parsing performance is better with C dissectors.

3.1.1.1 *Generic Dissector*

Generic Dissector [72] is a Wireshark plugin that allows the user to write dissectors in a declarative language, similar to C structures. They are parsed into internal data structures and can contain scripts, which are interpreted during the dissection. Predefined decoders can transform packet data before parsing, including Base64, Unicode and Advanced Encryption Standard (AES) decoding.

Unlike with imperative dissectors, no special code for segment reassembly is required, as this is handled seamlessly by the interpreter. However, it is not possible to implement protocols containing streams needing reassembly, because sub protocols nested in a Generic Dissector protocol have to occur in one continuous piece.

3.1.1.2 *Pyreshark*

Pyreshark [53] is a Wireshark plugin that enables the user to load Python code into Wireshark, to implement dissectors or other plugins in Python. Custom Python code can be executed to conditionally parse data. It includes no support for using Wireshark's reassembly algorithm, so correctly implementing TCP based protocols is not possible.

3.1.2 *ScaPy*

ScaPy [7] is a module for the Python programming language, which is geared towards interactive packet manipulation and building custom networking tools. Similar to Wireshark's dissectors, it contains Packet classes for many common network protocols. For simple protocols, a declarative syntax is used, which allows the library to parse and generate network packets from the same description. Special cases can be handled with custom Field types or pre- and post-processing Python code in the Packet class. It allows crafting and sending network packets and capturing the replies, so it can be used for network scanning, unit tests, fuzzing or other attacks.

3.1.3 *CANAPE*

CANAPE [17] is an open-source network protocol analysis tool which provides an Integrated Development Environment (IDE) for binary network protocols. It is implemented as a C# GUI application and therefore only runs on Microsoft Windows. It intercepts and manipulates binary protocol data by providing a SOCKS proxy server, in a similar way as web Application security testing tools (like Burp [1] or Fiddler[2]) do for web requests by providing an HTTP proxy. This also allows replaying of captured network traffic. The goal is to provide a framework for developing parsers, fuzzers and proxies for complex protocols with a minimal amount of custom code. *Net graphs* (flowchart-like directed graphs) are used throughout the program: Incoming data flows through transforming and parsing nodes, describing the steps required to parse a complex protocol. In some protocols, a connection can be in different states (e.g. negotiation, authentication, data transfer). These states, and the transitions in between, are also represented as graphs. The parsers are either generated from a protocol description entered in a spreadsheet GUI, or implemented as imperative C# snippets.

3.1.4 *Universal Radio Hacker*

Universal Radio Hacker [43] is an open-source protocol analysis tool for low-level wireless protocols. It implements an interactive GUI for collecting raw signals via Software Defined Radio (SDR), demodulating them into a bitstream, decoding messages, labelling protocol fields and grouping messages by message type. Unlike the programs described above, URH operates at the radio signal or bitstream level, not at the byte level. It provides special features adapted to low-bandwidth radio protocols as often used in smart home and IoT devices. For example, it can auto-detect modulations like FSK and ASK, and decode differential encoding and data whitening.

Compared to GNUradio [20], a very powerful open-source tool for processing wireless signals, URH has fewer features but allows a much more user-friendly workflow for examining common, simpler radio protocols. In particular, the interactive configuration and visualization allows for exploration of unknown protocols on the physical and bitstream layer.

3.1.5 *Saleae Logic*

Saleae Logic 2 [31] is the accompanying software to the hardware Saleae logic analyzers. Logic analyzers are be used to record over-the-

1 https://portswigger.net/burp
2 https://www.telerik.com/fiddler

wire protocol data inside devices. After recording the raw electrical signals, Logic 2 allows the user to run so-called low-level and high-level protocol analyzers on the signals. Low-level analyzers transform the captured electrical signals into low-level binary data, e.g. into two byte streams (receive and transmit) in the case of UART communication. High-level analyzers further transform the results of a low-level analyzer, e.g. by parsing a higher level protocol layered on top of the UART byte stream, producing a list of protocol frames. This can be used to build a parser for a newly reverse engineered protocol, displaying it's results directly in the Logic 2 software. Analyzers are built in Python, on top of an SDK provided by Saleae.

3.2 ADVANCED HEX EDITORS

Investigating unknown proprietary protocols often includes looking at the raw bytes of which a packet consists, usually displayed as a hexadecimal and ASCII representation next to each other ("hex dump"). Therefore, we also researched the features of hex editors which are usually more advanced than the hex dump view of Wireshark. Interesting features include the possibility to interpret the selected data as various formats (integer, float, timestamp, color), calculate checksums and hashes, display histograms of occurrences or compare two files. Some hex editors can parse binary file formats using a format description in a proprietary syntax, or from a GUI. It can be edited on-the-fly by the user, allowing an interactive exploration of a file format. In Wireshark, the packet bytes hex viewer is linked to the packet tree, therefore the dissectors are providing format descriptions in a similar, yet less interactive way. In some editors, the user can interactively add annotations, comments or bookmarks to the hex dump, enabling them to directly record information gained about the file format. In Table 1, we provide an overview of the most common features of hex editors and some similar software.

3.2.1 *010 Editor*

010 Editor [59] is a commercial text and hex editor for Windows, Linux and macOS. Format descriptions can be provided in a proprietary scripting language called "Binary Templates". The syntax is similar to C structs, but supports loops and conditionals, as well as scripts to parse custom field formats. The *Inspector* interprets selected data as various integer, floating point and timestamp formats. Multiple hash and checksum algorithms can be calculated, and a histogram view can visualize the number of occurrences of values in the file.

Feature/Tool	010 Editor	Hexinator	hex-works.com	ImHex	Hobbits	Wireshark
Select Bytes	✓	✓	✓	✓	-	-
Data Inspector	✓	✓	✓	✓	-	-
Checksums	✓	✓	✓	✓	-	-
Hashes	✓	✓	-	✓	-	-
Histogram	✓	✓	-	✓	✓	-
Annotation	-	-	(✓) 4	(✓) 3	(✓) 3	-
Generic Parser	✓1	✓2	-	✓1	-	✓1,5
Packet Support	-	-	-	-	✓	✓

1 text-based, 2 GUI-based, 3 bookmarks only, 4 colors only, 5 requires restart

Table 1: Overview of hex editor features

3.2.2 *Hexinator*

Hexinator [42] is a commercial hex editor for Windows and Linux. The user can build format descriptions called "grammars" by selecting bytes in the hex editor, or from a GUI based around a tree structure. Grammars are stored in a proprietary XML format and can be applied to other files using their "Universal Parsing Engine". Selected data interpreted as various integer, floating point and color value formats is shown in the *Data Panel*, and checksums and hashes thereof can be displayed. It can also display a histogram of the entire file.

3.2.3 *Hex-Works*

Hex-Works [6] is an open-source online hex editor written in JavaScript, in which the user can manually colorize ranges of a file, and apply the color set to other files, allowing for easier comparison of similarly structured files. The colorizations are applied statically by byte offset, without any support for automatic repetitions or adaption to variable-length areas. An *Inspector* is provided which shows the selection interpreted as integer, and some simple checksums. The user can also enter an arithmetic expression with variables for the selected data, which is re-evaluated whenever the selection changes.

3.2.4 *ImHex*

ImHex [70] is an open-source hex editor targeted at reverse engineers and programmers. It is developed in C++ and uses a custom GUI framework, ImGui. Using a custom C++-like description language, the user can describe file formats to be parsed by the editor, displaying the results as a tree view and highlighting fields directly in the hex editor. It also provides features aimed at reverse engineering files, like a disassembler, data analyzer based on magic numbers, byte distributions and entropy, as well as search features for strings and hex values.

3.2.5 *Hobbits*

Hobbits is an open-source "software platform for analyzing, processing, and visualizing bits" [32]. It is not a classical hex editor, in that it is centered around bitwise instead of byte-wise data presentation, and does not allow direct editing of data. However, it can show hexadecimal and ASCII views of imported data. Bitstreams can be imported from files or over the network, and manipulated via so-called operators, keeping track of the intermediate steps of data manipulation in a tree structure. Data can also be analyzed using plugins, e.g. searching for bit strings or guessing possible frame lengths. In addition to the common hex and ascii display, Hobbits provides graphical representations, e.g. raster views, where each pixel represents a bit or byte.

3.3 DATA FORMAT DESCRIPTION AND PARSER GENERATION

In many network analysis tools (e.g. Wireshark [10]) and Intrusion Detection Systems (IDSs) (e.g. Snort [56], Zeek [63]), protocol analyzers (parsers) are handwritten, low-level C code. This is described as a lack of abstraction [40], which makes bugs hard to detect, and security vulnerabilities more likely. In other fields where parsers are required, like parsing of programming languages, it is very common to use parser generators (e.g. Yacc [62] or Bison [8]), which generate parsers from high-level grammars. Transferring this method to network protocol parsing, and binary parsing in general, is the topic of various research papers and open-source projects. Some of these we will present below, focusing on ones which are used in real-world products or have interesting properties.

3.3.1 *XDR and ASN.1*

XDR (External Data Representation) [14] is a standard for describing and serializing data, especially network protocols. It defines a common serialization format for primitive types like numbers, strings and lists, as well as a C-like description language for defining complex data types like structs and unions.

The **ASN.1 (Abstract Syntax Notation One)** standard provides similar features, but is more complex and more flexible. It provides a larger number of primitive types (e.g. Object IDs, multiple types of strings, sets), as well as multiple serialization formats, which are designed for different objectives, e.g. an extremely compact binary representation (Packed Encoding Rules, defined as a bitstream), fast and flexible parsing (Basic Encoding Rules, defined as a stream of bytes), or human-readability (XML Encoding Rules, a text-based format).

Both standards are relevant because they have been around since the 90s, and have therefore influenced many newer serialization standards (e.g. **Protobuf** [44], **Apache Thrift** [4], **FlatBuffers** [16]) in terms of the available primitive and complex data types, and the specification languages. These newer standards are structurally so similar that we do not further address them here.

The aforementioned standards are useful for defining new protocols, and are used as the basis of several standard protocols and file formats. From a description, serializers and deserializers (parsers) can be automatically generated for many programming languages. However, the serialization formats are not sufficiently adaptable to describe most existing binary data.

3.3.2 *Packet Types*

McCann and Chandra [33] mention another reason why, even for new protocols, bespoke serialization formats are created instead of using standardized ones, which is the requirement to keep overhead low by packing data "as tightly as possible". Using an excerpt from the Linux kernel firewall code as an example, they explain that traditional code for interpreting packets is verbose and error-prone. To facilitate the quick and correct implementation of such parsers, they present the *Packet Types* packet specification language. It supports some common idioms of network protocols, which cannot be implemented with simple C structures, namely length-prefixed fields, clean definition of protocol layers, endianness of numeric fields, and byte alignment of structure fields.

3.3.3 *BinPAC*

Pang et al. developed *BinPAC* [40], a framework to generate parsers for network protocols from a high-level declarative specification. They state that it is a problem that even for very complex protocols, analyzers are often written as low-level code with lacking abstraction, making programming errors likely and hard to notice. These handwritten parsers are often implemented in memory-unsafe languages like C, and directly interface with untrusted input from the network, errors are likely to result in security vulnerabilities. Furthermore, such parsers are often not reusable in other applications, because they are tightly coupled with their environment.

They compare this style of development to parsers for programming languages, which are usually automatically generated from declarative specifications of the language. Parser generators (e.g. yacc) from that domain are not easily applicable to network protocols, because some common patterns in network protocols are very unusual in programming languages. This includes length-prefixed data fields, and cases where parsing one field depends on data from another, non-adjacent field. These patterns are hard or impossible to describe with context-free grammars, the usual way to describe a programming language's syntax using recursive replacement rules.

The authors implement their own parser generator, which compiles a high-level description language to low-level parsers in C++, and demonstrate protocol parsers for multiple complex protocols (e.g. HTTP, DNS, SMB). They integrate these parsers into the Zeek IDS project, which now maintains the BinPAC code base, but the generated C++ code can also be used in other programs.

3.3.4 *Spicy*

Sommer et al. [58] implemented *HILTI*, a platform for network traffic analysis, consisting of an abstract machine model (essentially a programming language, which itself is a compilation target) with an instruction set optimized for networking applications, and a matching optimizing compiler toolchain. One of their evaluation applications is *BinPAC++*, a modification of BinPAC which generates code for their intermediate language.

Later on, they present the *Spicy* framework [57] as a successor, with a generalized specification language that also handles semantics, and an API for Deep Packet Inspection (DPI) applications. Unlike BinPAC, it contains a full programming language for handling protocol state and logic, making inline C++ code unnecessary. The authors implemented prototypes of integrations in Wireshark, Zeek and an HTTP proxy, which however were not added to the upstream projects. It also supports the reverse use case, assembling wire format from the specification. The implementations of HILTI and Spicy are published[3] under the BSD license.

3 https://github.com/zeek/spicy/

3.3.5 *KaiTai struct*

KaiTai struct [27] is a parser generation framework for binary data structures, consisting of a YAML-based specification language and parser generators for many target languages. It is mainly aimed at parsing files, but can be applied to PCAP files and network packets as well.

3.4 AUTOMATED PROTOCOL REVERSE ENGINEERING

Kleber, Maile, and Kargl [28] define two major ways of conducting Protocol Reverse Engineering (PRE): **Entity analysis** uses regular software reverse engineering methods on an implementation of the protocol. This is a useful approach if such software is accessible to the reverse engineer, which may not be the case for embedded devices, online services or heavily obfuscated software. Entity analysis or hybrid methods are the only way to reverse engineer a protocol if strong encryption is used to protect communications. In **trace analysis**, only the traffic between entities is analyzed to infer the protocol. **Static Trace Analysis (STA)**, on which the authors focus in their detailed survey, requires only passive listening capabilities on the communication medium, but no manipulation of an end entity. Offline analysis of previously recorded traces is possible, making it reproducible and allowing for usage of non-real-time algorithms. **Dynamic traffic analysis** additionally stimulates the entity by injecting packets or interacting with e.g. a device's user interface.

Li and Chen [30] categorize similarly in network-based (trace analysis), program-based (entity analysis) and hybrid (a combination of both) PRE methods. They present short overviews of select approaches of each category.

In this section, we also focus on trace analysis, as the focus of this thesis is on interpreting network traces.

3.4.1 *Protocol Informatics*

Beddoe [5] apply Bioinformatics algorithms to the protocol analysis domain. Sequence alignment is usually applied to DNA sequences, looking for similarities to find out which parts of DNA relate to shared properties of the organism. In this paper, captured network packets are aligned, using byte sequences instead of amino acid sequences. After aligning two packets, identical ranges are marked as keywords, changed ranges of same length as fixed-length fields, and gaps are interpreted as variable-length fields. The result can be represented as a consensus sequence: a new sequence where the identical ranges are copied from the aligned sequences and variables are replaced by placeholders.

The Needleman-Wunsch algorithm is used to find optimal alignments. Similarity matrices, which assign a similarity value to each possible pair of data values, are an important component to optimize alignment, and need to be adapted to the problem at hand. The authors present example matrices for text-based protocols, defining that ASCII chars are more likely to change to other ASCII chars in another packet, than to non-ascii bytes.

They analyze the detected columns (variables) using several techniques, for example offset comparison, to detect sequential numbers (where the number increases with each packet by an equal or similar offset), and mutation rate, to detect pseudo-random numbers like checksums (which have a high mutation rate).

The examples always use text protocols, but the authors claim the technique works on binary protocols as well.

The authors of the paper mention some possible next steps to explore: They propose to present the resulting data in an intuitive way to allow improved human estimation and understanding. This could be achieved by displaying data in matching colors, and by other interface design approaches. On the other hand, they acknowledge that the process can never be fully automated if accuracy is in mind.

4 DESIGN

For designing a software product, we first need to clarify the goals. To do this, we conducted user interviews, which are described in the first section. From that, we distilled conclusions and model scenarios in which the software could be used. We describe the generic workflow of a protocol reverse engineer shared by these scenarios. Furthermore, we study the characteristics of typical protocols encountered by potential users.

Derived from these we define basic object types we see throughout our application, a definition language which can describe the protocols we studied, and display and interaction modes used in the application.

4.1 USER INTERVIEWS

To kick of the design process, we used user interviews, a generative method of user studies as described in Section 2.2. We recruited reverse engineers with different levels of expertise, which could be potential users for our application, to answer our questions. The questionnaire used to survey these participants and the mock-ups shown to them are attached as Section A.1. In this section, we summarize the insights we gained from this first phase, the individual user interviews. In Chapter 6, we describe the following, evaluative phases of our user studies. We used these to further guide the development process and evaluate the project in the end.

4.1.1 *Interviewee 1*

The interviewee analyzed a proprietary binary protocol in their Bachelor's thesis, and is currently working as a student assistant on other protocol reverse engineering tasks.

ANALYZED PROTOCOLS The protocol is used for internal communications between components of a smartphone. As the protocol was not publicly described before, the goals were to find out the protocol structure, publish it as a paper and provide a Wireshark dissector to facilitate future research. It could be intercepted by enabling a debug mode of the operating system and reading the protocol frames from the system log. It was also possible to inject modified frames into the system for further analysis and fuzzing. The basic structure of the protocol was simple, consisting of a fixed-length header followed by Type-Length-Value (TLV) fields. However, there was a large number of TLV types containing individual data structures. No cryptography or deliberate obfuscation was used, however, the header was hard to analyze, because it used packed, intermingled bit fields spanning several bytes.

WORKFLOW To access protocol data, they implemented a script that reads the system log, parses the log lines containing hexadecimal representations of packets from it, and pipes them into Wireshark. This gave them a live view of the traffic. No custom code on the device is needed because the log can be accessed using developer tools on their computer.

Three main ways of interpreting the basic structure of the unknown protocol bytes were described by the interviewee. First, they tried to interpret packet contents based on known external context. One example is a user interaction, which is known to cause packet containing some known data, like sending an SMS where the SMS contents are known.

Second, differences and similarities between groups of packets were examined to identify fields with static (e.g. packet type) or dynamic (e.g. sequence number) contents. Third, they used context provided by the program library under test itself, in this case lines in the system log, that contained parsed field contents from the adjacent packets. They also injected packets with flipped single bits to determine field boundaries by watching which field contents in the log changes.

As the binary of the code handling the packets on the main processor side was accessible to the researcher, they used static analysis of the binary to gather further information of the data structure inside the individual TLV contents.

DIFFICULTIES They decided to implement a Lua dissector for Wireshark, because the protocol does not have high throughput and not performance sensitive, so the much higher complexity of implementing a C dissector is not justified.

While implementing the custom dissector, they noticed that parsed fields could only be annotated in Wireshark as full bytes, not individual bits – however, some fields consisted of single bits fragmented over multiple bytes (see example in Figure 5).

```
01000100 10111101
XX...... ..XXXXXX  Sequence No. = 01111101
....XXXX XX......  Frame Type = 010010
..XX.... ........  Reserved = 00

```

Figure 5: Example of fragmented bit field

Using fully automated protocol analysis tools was not successful, because they could not handle the fragmented bit fields.

USE CASES The interviewee reported various use cases, where support by a graphical tool would be beneficial to their workflow.

First, when correlating unknown packets to contextual information, it would be useful to integrate log outputs and manual annotations (e.g. of user interactions) into the packet list, or into a separate view which is linked to the packet list. Also, a way to record many short packet traces, each annotated with the information which user interaction lead to these packets was deemed helpful.

Second, they reported using a word processor to annotate hex dumps of packets, whereas a hex editor with good annotation and comment features would be a helpful tool.

Third, a graphical diff tool to view the differences between two packets selected from a list of packets would have been very useful for them when trying to find structures by comparing packets. Also, a way to directly annotate identical ranges as fields might be helpful. It would be necessary to manually select the packets to compare, because often, consecutive packets would be of completely different types (e.g. request and response), therefore, comparison would not yield useful results.

Fourth, piping data from a custom script is considered the most important way to import data. This provides the highest flexibility for unforeseen data sources.

A powerful packet filter, like the feature available in Wireshark, was seen as very important. Especially, filtering on the newly discovered and annotated could be helpful, for example to verify assumptions about possible field contents.

4.1.2 *Interviewee 2*

The respondent works as a PhD student and reverse engineered various protocols, including protocols of Bluetooth tracking devices, while working on their thesis. They also wrote their Master's thesis in the area. Most of their work is on device-specific over-the-air protocols, rarely on TCP/IP based protocols, but usually not on device-internal physical communication.

Their first goal of reverse engineering a protocol is usually to document its structure and publish it as a paper. Also, knowing the protocol structure is helpful for more directed fuzzing, by giving an idea which fields might be more vulnerable to incorrect input. This helps with the second goal of finding potential security vulnerabilities in the design or implementation of a protocol.

ANALYZED PROTOCOLS In their Master's thesis, they analyzed vendor-specific network and Bluetooth protocols used for locally communicating between a smartphone and notebook. One of the protocols was based on Bluetooth Low Energy (BLE) advertisements with proprietary manufacturer data. It contained some unencrypted fields, and a hashed app identifier. They needed to understand the binary data format to conduct further security research on this communication interface.

Common elements they often see in protocols are TLVs, as well as structured data formats like MessagePack [50] and ProtoBuf [44]. In formats like ProtoBuf, rudimentary structure information is provided in the data. They can still be hard to decode without the protocol definition, which can sometimes be retrieved from the application binary, if available.

Also very common is a packet type near the start, determining the following data structure. Often, request and response types are defined, sometimes, a fixed sequence of packet types is expected by the implementation. Other common elements include sequence numbers, and encryption meta data like Initialization Vectors (IVs).

They did not encounter deliberately obfuscated protocols or data formats yet. Encrypted protocols are quite common. To deal with these, they usually hook into the software to either extract the cryptographic key or the unencrypted raw data. This is only possible with access to the device, so a debugger can be attached to the process.

They once analyzed a TCP/IP based protocol for device-to-device communication with custom encryption, however, focus mostly on lower level protocols.

WORKFLOW Most of the analyzed protocols are not based on standard network protocols, where one can simply record a trace with Wireshark and have the lower layer protocols dissected automatically. Instead, they are e.g. radio protocols with standard physical layer, but custom MAC layers. Therefore, only the packet boundaries are known, and all bytes inside a packet must be individually reverse engineered. This makes it more difficult to load the data into Wireshark, because no predefined dissector can be used. They sometimes write custom dissectors in Lua, which is efficient to create due to its short and simple syntax. However, they found it difficult to correctly integrate it in Wireshark.

They reported using a hex editor with basic annotation features[3], however, it is unable to save these annotations over multiple sessions. The editor is mostly used for viewing, searching and annotation, and only sometimes for actually modifying it. They either export single packets from Wireshark into the editor, or sometimes even load the complete PCAP file into it.

For filtering and annotating log files, they wrote a custom GUI tool. It provides syntax highlighting of common log formats and highlights rows to which annotations were added.

USE CASES The most important use case would be capturing ideas and insights directly in the tool. This could take the form of attaching names and notes to byte ranges, highlighting with colors, and transferring these annotations to other packets of the same protocol.

Another use case is to try running various decoders on a packet or a byte range. This includes common encodings like MsgPack or ProtoBuf, as well as encryption methods like AES (with provided keys). One way to implement this in a general way would be a feature to run custom Python scripts on data.

They also proposed to generate Lua dissectors, even exporting just the annotations from a single packet as fields in the dissector might be very helpful.

In response to being shown this feature in the mockup, they also agreed that selection heuristics in the hex view (e.g. for detecting length fields close to the selected range) could be useful.

They stated that their preferred GUI layout is placing all elements in a single window, organized by a tabbed interface.

For importing data, they highlighted PCAP files as their single most common format. When recording packets, framing and timing information should usually be included, and PCAP is a common form to store this information. Raw hex dump or binary imports would only be useful for importing non packet based data, like custom data files.

4.1.3 *Interviewee 3*

The interviewee works as a student assistant, reverse engineering protocols and file formats mostly related to mobile and embedded devices. They often work on low-level hardware interfaces.

ANALYZED PROTOCOLS All protocols and file formats the interviewee reverse engineered so far are binary, neither encrypted nor obfuscated. They often contained list structures and length-prefixed data. The protocols were often device-internal communications, recorded from Universal Asynchronous Receiver / Transmitter (UART) or Serial Peripheral Interface (SPI) ports, or wireless communications. File formats were crash dumps and configurations extracted from flash memory or received over UART. They usually neither have access to the executable nor can they attach a debugger, because they mostly analyze closed embedded devices.

WORKFLOW They gained access to raw protocol data at the physical layer, e.g. connecting a logic analyzer to the device under test, or using a Software Defined Radio (SDR) to sniff over-the-air traffic. After that, they exported the data as binary, or an intermediate format produced by the sniffing software which was in turn converted to a binary. They reported using a very basic hex editor[34] to look at binary files, as well as a word processor to annotate hex dumps with colors, notes, indentations and line breaks.

For one protocol, they wrote a documentation of the format, so that another researcher was able to implement a Wireshark dissector.

For files, they usually write a simple parser script in Python which loads the binary and outputs parsed information as human-readable text. They did not implement Wireshark dissectors in that case because Wireshark is structured around network traffic.

USE CASES The interviewee explained various kinds of annotations they would like to use on a hex dump. First, they need to visually split up a file into sections, by introducing white space and a optional comment between them. They would also like to create a hierarchy of sections, e.g. by using different font sizes for the heading, or indentation. Second, simple annotations on byte ranges like colors and comments were seen as useful.

For UART data, it would be useful to import CSV files consisting of timestamps and bytes. They could be split into sections by placing section headers whenever a minimum time threshold is met between two bytes.

They recommended a feature to transfer a set of annotations to another file, optionally with a variable offset, to check whether it matches the same format.

It was suggested that for analyzing unknown structures, they might want to run external tools on a file, e.g. strings or binwalk. Another suggestion was automated scanning for timestamps. Also, they proposed a feature to automatically apply selected Wireshark dissectors on a file or selected byte range, to check if it matches a known protocol.

They are interested in an export function which transforms an annotation set into a Python script which can parse a file in the annotated format.

4.1.4 *Interviewee 4*

In their position as a postdoctoral researcher, the interviewee reverse engineers protocols and software as their own work or as a preliminary analysis before assigning them as thesis topics to supervised students. The goals of the preliminary analyses are to estimate the potential impact, the time required and to determine if there are starting points for analysis and modification.

Basic goals include determining how a protocol works, investigating its security, and developing compatible software. The description of how a protocol works, possibly combined with details about the data structures or a dissector implementation, may already be published as a paper in the case of proprietary protocols. For a structured security analysis, a threat model must first be developed in order to then identify critical points in the protocol process and critical data that must not be leaked or modified. Fuzzing can then be used on the one hand to find attack points, but also to develop a deeper understanding of the protocol.

ANALYZED PROTOCOLS The researcher usually examines internal protocols between components of a device, such as between the operating system and Bluetooth chip of a smartphone. Such protocols are binary encoded, usually not encrypted or obfuscated, but occasionally contain checksums. In some cases, the packet boundaries are unclear, which complicates converting dumps to a PCAP file.

Protocols are usually implemented in more than one place, often in different programming languages or environments, e.g. in a daemon process in the operating system on the main processor, and in the firmware of a peripheral chip. Therefore, it may be necessary to reverse engineer more than one implementation. For that purpose, they use dissassemblers like IDA Pro[25] and Ghidra[18].

WORKFLOW Common starting points for analysis are unencrypted parts of a protocol, strings found in binaries and firmware, and ways to inject modified packets.

For recording protocol traces, they usually have to develop their own ad-hoc tools, since ready-made tools like tcpdump or Wireshark do not work for these cases. Therefore, if the analysed software does not provide builtin debug features to log raw protocol data, either a hook has to be injected, e.g. with Frida[48], firmware has to be modified, or data needs to be intercepted at the physical layer.

These ad-hoc tools will either output raw byte streams or a list of raw packet bytes. Communication between the device under test and the researchers machine is usually already implemented in the utilized software like Frida or XCode. If raw byte streams are recorded, packet boundaries need to be determined afterwards. Because timing information can be obscured by buffering, it is often not possible to determine packet boundaries by transmission pauses. Therefore, the researcher often uses visual correlation by viewing a stream in hex format in a text editor, trying to align bytes by inserting line breaks manually or with search and replace.

To correlate similar packet contents, they insert hex dumps of multiple packets of the same type into a text editor, to use common text editing features like search, replace, insertion of line breaks and highlighting of identical text on them.

USE CASES Since the researcher often has to work with the output of ad hoc tools, they would welcome import functions for raw data. This includes both raw binary files that still need to be split into frames and directories with many raw binary files to be viewed as a list of packets.

If available, timing information can be used to correlate packets to e.g. syslog entries. To do this, you would have to save a recording with timestamps in order to then automatically jump to the appropriate places in the syslog.

Different tools use different formats to represent binary data as hexadecimal string, so it would be useful to be able to import and export data from files or clipboard in multiple formats (e.g. with or without `0x` prefixes).

Specialized search functions that automatically search for the same value in multiple encodings would be helpful, e.g. for a certain number as big endian and little endian integers, or a duration in minutes, seconds and nanoseconds.

Since it is often necessary to hand over the project, it is important that annotations can be easily passed on to other people, e.g. by exporting them to a more common format. They might also want to create a Wireshark dissector for a protocol, e.g. by automatically exporting annotations and protocol definitions as a basic Wireshark dissector which can then be manually refined.

As they already work with a large number of other programs at the same time, the software should not open unnecessarily many individual windows. A tabbed window interface is preferred.

4.1.5 *Conclusion*

From the user interviews, we can identify both common goals of the researchers, and common features of the protocols studied. In addition, we can summarize which software features are helpful according to the interviewees in order to achieve these goals more easily in the examined protocols.

Common to all interviewees is the goal of reverse engineering the structure of proprietary protocols and subsequently publishing them, usually in scientific publications and as open-source tooling.

The protocols studied use binary transmission formats, not text-based ones, often built in TLV format or compact, fixed data structures. Artificial obfuscation of the data is hardly to be found, in some cases, however, historically-grown structures can make them equally hard to understand. Some of the protocols are encrypted, forcing researchers to extract keys before being able to analyze the payloads.

For the graphical user interface, a tab-based interface was generally preferred over a single-file interface or a multi-window interface.

A feature all participants deemed very helpful is a hex editor with advanced annotation features, usable to view not only the contents of a single file, but also of one or more packets from a protocol dump. Requested annotation features include highlighting byte ranges with colors, placing short notes directly beneath a byte range or displayed as a tool tip, to mark potential data fields. There should be an easy way to interpret such selected bytes as different binary data types, e.g. numbers and timestamps. A further annotation feature is splitting the data visually into multiple sections to represent larger data structures, potentially with a hierarchical view for tree-like data. The annotations should also be able to be saved and transferred to other data sets.

Relevant data import formats are raw binary files, packet-based file formats (PCAP, reading a folder with raw data files), data transfer via standard output (pipe) from a script, and timestamped UART data.

Since protocols often have unpredictable peculiarities that cannot be foreseen in the development of this software, extensibility is a key feature. For this reason, it should be possible to integrate external tools and scripts. These should be able to run on single packets, multiple packets and selected byte ranges.

To infer the meaning of packets from context, features are requested to correlate log traces based on timing with log files or timestamped manual notes. This could be achieved by a log display with synchronized scrolling, or by interleaved display of log and packets.

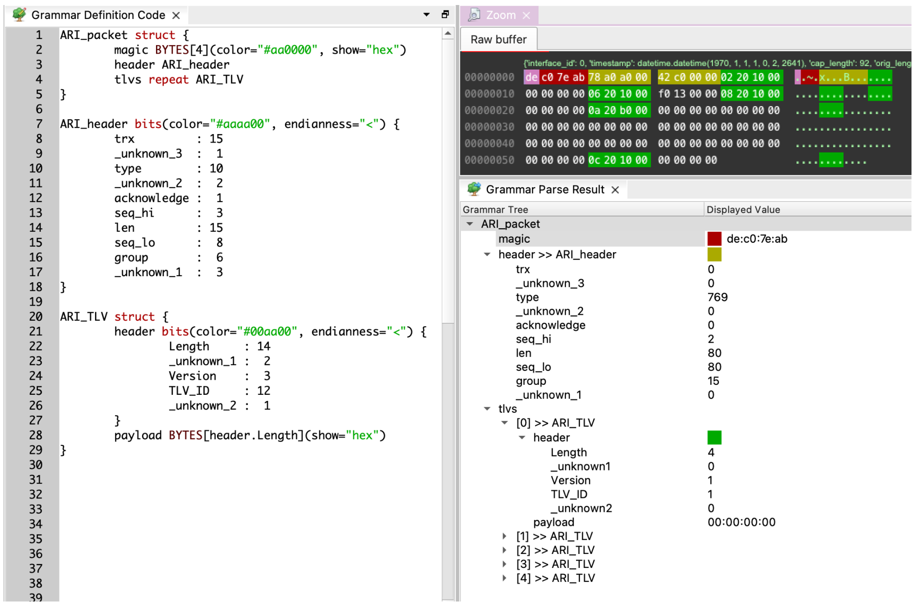
Export functions are required to further use the results outside the software. Of particular interest are the export of annotations as parser code, e.g. as Wireshark dissector or as Python script, which prints a human-readable version of parsed data.

4.2 MODEL SCENARIOS

In this section, we present selected model scenarios of reverse engineering protocols. One the one hand, we describe the real-world protocols serving as basis of the scenarios. On the other hand, we pick out some facets of these protocols to explain why they are well-suited for demonstrating features or justifying design decisions of the application. After that, in Section 4.3, we will use them to explain the workflow from data acquisition, over preparation and import, to annotating the packets and finally creating and validating a grammar.

4.2.1 *ARIstoteles*

The Apple Remote Invocation (ARI) protocol is used for device-internal communication in iOS devices between main processor and baseband chip. It was reverse engineered and publicly described by



```
ARI_packet struct {
        magic BYTES[4](color="#aa0000", show="hex")
        header ARI_header
        tlvs repeat ARI_TLV
}

ARI_header bits(color="#aaaa00", endianness="<") {
        trx         : 15
        _unknown_3  :  1
        type        : 10
        _unknown_2  :  2
        acknowledge :  1
        seq_hi      :  3
        len         : 15
        seq_lo      :  8
        group       :  6
        _unknown_1  :  3
}

ARI_TLV struct {
        header bits(color="#00aa00", endianness="<") {
                Length     : 14
                _unknown_1 :  2
                Version    :  3
                TLV_ID     : 12
                _unknown_2 :  1
        }
        payload BYTES[header.Length](show="hex")
}
```

```
            {'interface_id': 0, 'timestamp': datetime.datetime(1970, 1, 1, 1, 0, 2, 2641), 'cap_length': 92, 'orig_leng
00000000  de c0 7e ab 78 a0 a0 00  42 c0 00 00 02 20 10 00  ..~.x...B.......
00000010  00 00 00 00 06 20 10 00  f0 13 00 00 08 20 10 00  ................
00000020  00 00 00 00 0a 20 b0 00  00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 0c 20 10 00  00 00 00 00              ............
```

Grammar Tree	Displayed Value
ARI_packet	
magic	🟥 de:c0:7e:ab
header >> ARI_header	🟨
trx	0
_unknown_3	0
type	769
_unknown_2	0
acknowledge	0
seq_hi	2
len	80
seq_lo	80
group	15
_unknown_1	0
tlvs	
[0] >> ARI_TLV	
header	🟩
Length	4
_unknown1	0
Version	1
TLV_ID	1
_unknown2	0
payload	00:00:00:00
[1] >> ARI_TLV	
[2] >> ARI_TLV	
[3] >> ARI_TLV	
[4] >> ARI_TLV	

Figure 6: Parsed ARI frame in PRE Workbench

Kröll et al. [29]. They captured traces by reading out system logfiles, and injected packets by hooking into a daemon process and calling methods there. No lower networking layer is present in this case, so analysis is performed directly on the extracted protocol frames.

The protocol itself has a simple base structure, each frame starts with a fixed-length header, followed by a varying number of TLV blocks (see Figure 6). The frame header as well as the TLV headers have a more intricate structure, because they contain packed fields of varying lengths in bits, which are not aligned to byte boundaries, and which are then converted to little-endian byte order. A large number of different types are defined, therefore Kröll et al. [29] partially automated the process to generate dissectors for the TLV values from the disassembled driver software.

We will focus on ingesting the sample protocol traces and frames the authors published, and parsing the header structures. We skip the actual TLV values, because due to their large number, they are better suited to an automated approach.

The sample data is present in two different formats: as PCAP files with a custom link type, and as raw binary files, each directly containing the ARI frames without any lower layer.

4.2.2 *Bluetooth Smart Lock*

The eqiva Smartlock is a battery-powered device which can be installed on the inside of a front door. It allows the user to lock and unlock the door over Bluetooth via a smartphone app. We analyzed its security and communication protocols previously [67]. To this end, we sniffed

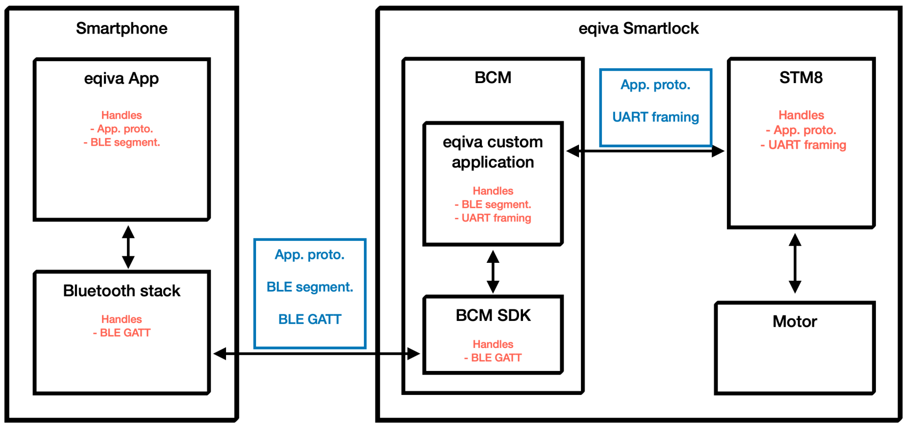


Figure 7: eqiva Smartlock architecture

data on multiple transports, and we had multiple custom protocol layers to reverse engineer. An overview of the architecture is visualized in Figure 7.

The lock hardware consists of two different microcontrollers, a Bluetooth SOC by Broadcom and a 8-bit microcontroller by ST. In the Bluetooth SOC, an SDK provided by Broadcom implements the BLE protocol, and a custom application developed by eqiva implements a custom segmentation protocol layer, which allows to transport longer data frames in the short payload fields of BLE Generic ATTribute Profile (GATT) messages. This Bluetooth SOC in turn communicates over UART to the 8-bit microcontroller, which handles the actual application-level protocol, and also controls the actual motor which locks and unlocks the door. The UART communication use a simple, custom UART framing protocol layer.

Therefore, the three custom protocols we have to deal with are UART framing, BLE segmentation and the application protocol. Over-the-air, we see application protocol frames, wrapped in BLE segmentation, transported over BLE GATT messages. On UART, we see application protocol frames, wrapped in UART framing.

4.3 WORKFLOW

Based on the user interviews and the model scenarios, we describe below the components of a reverse engineering workflow that can be implemented with PRE Workbench.

4.3.1 *Raw Data Acquisition*

As seen with the ARI protocol, data could be present in raw binary files, and in PCAP files. In the case of the Smart Lock protocol, we also need to import Bluetooth traces, and Comma-Separated Values (CSV) files containing data from a serial port dump. We also want to handle network data, either as live captures using the Wireshark Command-Line Interface (CLI), or by pasting TCP stream data from Wireshark from the clipboard.

We need to handle a variety of data acquisition and import methods, producing either single `ByteBuffers` or lists thereof. Therefore, we propose the concept of a `DataSource`, which the user can parameterize and run. All data sources we implemented, and the ways to implement custom data sources are described in Section 5.2.

4.3.2 *Data Extraction and Preparation*

In some cases users need to further prepare imported data, for example splitting data into frames, decrypting or decoding data according to some protocol-specific rules. As these are hard to foresee and implement comprehensively, we allow users to run macros on data objects, so they can implement these rules in short Python programs.

4.3.3 *Data Display*

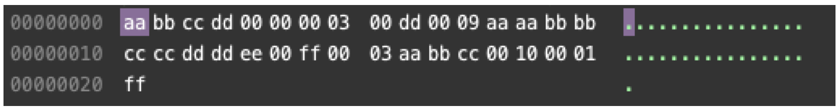
Once the data is successfully imported, researchers often want to explore them interactively. In the case of protocol traces, a list view is initially provided for this purpose, which displays the raw data and any metadata of the packets. One or more packets can then be selected from this list and get displayed in a detailed view as a HexView. In the case of individual binary files, the HexView display is loaded directly.

PACKET LIST The list view allows not only to select packets for the detailed view, but also to run macros on the packets, filter based on expressions, mark packets for future reference, as well as to customize the columns displayed.

HEXVIEW The HexView component is the key component of the application. It displays the hexadecimal and ASCII representation of the binary payload of packets or files, in a format commonly described as a *hexdump* (see Figure 8a). Furthermore, the HexView allows the researcher to explore the data in a variety of ways:

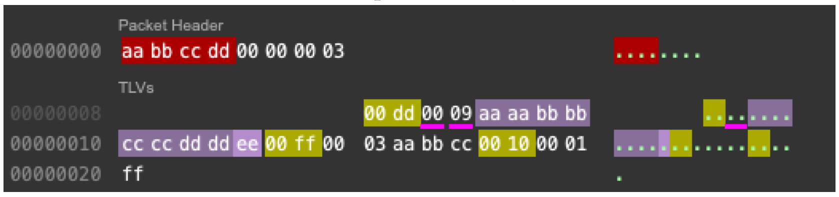
Upon selection of a byte range, several heuristic detection methods are run in the background, to highlight matching information in the vicinity of the selected data. For example, if a likely length field is found next to the selection, it will be highlighted. New heuristics can be implemented as simple Python functions, either directly in the application code, or in a plugin.

A data inspector panel shows interpretations of selected bytes in various formats such as integers, floating point numbers, timestamps, colors and IP addresses.



```
00000000  aa bb cc dd 00 00 00 03  00 dd 00 09 aa aa bb bb  ................
00000010  cc cc dd dd ee 00 ff 00  03 aa bb cc 00 10 00 01  ................
00000020  ff                                               .
```

(a) A plain *hexdump*



```
          Packet Header
00000000  aa bb cc dd 00 00 00 03                           ........
          TLVs
00000008                           00 dd 00 09 aa aa bb bb          ........
00000010  cc cc dd dd ee 00 ff 00  03 aa bb cc 00 10 00 01  ................
00000020  ff                                               .
```

(b) An annotated *hexdump*

Figure 8: Screenshots of the HexView component

INTERACTIVE ANNOTATION The HexView also aids in documenting insights the researcher gained into the protocol already. Bytes and byte ranges can be annotated in different colors, and with textual comments (see Figure 8b). These annotations can be displayed on multiple packets simultanously, allowing the researcher to immediately detect pattern matches or mismatches.

4.3.4 *Interactive Dissector Development*

We want to give the user the possibility to specify the protocol grammar by editing it as a textual representation as well as by highlighting bytes in a Graphical User Interface (GUI). To this end, we need a textual representation, which should be easy to edit by the user, an internal structure, which the GUI can work on, as well as the possibility to convert between them in both directions. A textual representation based on a general-purpose programming language is to expressive, allowing the user to type in constructs which the GUI could not handle. Basing it on a general-purpose structured language like XML or JSON was deemed not comfortable enough to edit by hand. For that reason, we designed a custom language for grammar specification in a roughly C-like syntax.

USER INTERFACE The user interface should give the user multiple ways to work with protocol traces and grammar definitions. We provide a code editor to directly work on the textual representation. This is especially useful for quickly typing in known elements of the protocol, as well as for more complex structural changes which are easy by copy-and-pasting code snippets, but more difficult to represent in the other views. Furthermore, the user can highlight bytes in the HexView and use the *ClickGrammar* feature to assign them a name and data type, adding them as structure fields in the grammar definition. Finally, a tree view displays the parse result of the grammar definitions in a tree structure. In the tree view, the user can also edit the structure fields using the GUI, to change their visual representation (e.g. background color). All these views are synchronized in all directions, e.g. changes made in the tree view automatically propagate to the code.

VALIDATION To validate the produced grammar definitions against the protocol traces, we provide multiple features. First, a whole trace file can be parsed using a grammar definition. If any parsing errors occur, they are printed in a log output, alerting the researcher to possible mistakes in the grammar definition. Second, assertions of field values can be placed in the definition, so that a parse error is raised if it does not match. Finally, a packet list can be searched for packets matching a specific expression, allowing to find packets which contain unexpected values.

DISSECTOR EXPORT As researchers explained in the interviews, they want to publish their results in a format useful to other researchers. While PRE Workbench grammar definitions are a concise form to describe a protocol, formats compatible with more popular applications are better suited for reuse. Therefore, we provide a feature to convert said definitions into dissector code compatible with the Wireshark protocol analyzer [10].

4.3.5 *Macros*

Because protocols and data formats can be arbitrarily complex, researchers often need to write custom code to pre-process data. Macros allow them to do this without having to leave PRE Workbench. They can be used to perform actions on the object types explained in Section 4.4, either consuming or producing one of these objects, or both, to transform data. They can also perform completely custom actions without interacting with any core application objects.

4.4 MAIN DATA STRUCTURES

In this section, we describe the main representations of data in the application, which are represented as classes in the implementation (more details in Section 5.1). These are also found as essential elements in the user interface.

One of the core objects of PRE Workbench is the **byte buffer**. A byte buffer contains a sequence of zero or more bytes (numeric values between zero and 255), and accompanying metadata. Metadata can either apply to the buffer as a whole, or to byte ranges inside the buffer. Annotations, section titles and parsed field values are stored as range-based metadata in the byte buffer object. Whole-buffer metadata can include a grammar definition name, annotation set name, as well as information provided by the data source, e.g. a packet timestamp or direction. Byte buffers are either handled individually or as part of a byte buffer list. When a binary file is opened in the application, it is loaded into a byte buffer, and displayed individually.

The **byte buffer list** is another core object. It holds an ordered list of byte buffers, accompanied by metadata belonging to the whole list. When a PCAP file is loaded, the packets are loaded into individual byte buffers, which are bundled in a byte buffer list. In this case, individual byte buffers store packet-level metadata like the capture timestamp, while the list stores file-level metadata like interface configuration. A byte buffer list is displayed using a list-detail interface (see Section 2.3.2), with a list of packets, where the columns can display packet metadata, fields and payload, and a detail view which displays the payload of the selected packet in a HexView.

A **data source** produces one of the objects mentioned above, given some input parameters. For example, the *PCAP file data source* requires a file name as input parameter, and generates a byte buffer list containing all packets from the PCAP file. Data sources can be defined in the application source code, in plugins or in user-defined macros. This allows flexible import of the wide variety of input data that researchers work with.

Range objects represent a consecutive range of bytes in a specific byte buffer and are used in many places throughout the application. For example, the current selection in a HexView is represented by a range, with its *buffer*, *start offset* and *end offset* properties. Furthermore, range objects can store additional metadata like a field name or a background color, and are used to annotate byte ranges in a byte buffer.

4.5 GRAPHICAL USER INTERFACE

As we want to develop a GUI application, we have to decide on the general environment to develop in, which frameworks to use, and which paradigms to use for window and file handling.

4.5.1 *Environment and Frameworks*

The environment includes which operating systems the application should work on, whether it is a desktop application, a web application running a local or online server, or a mobile app. Due to the nature of the tasks performed using our software, a mobile app was out of the question, and based on the fact that we want to support local network sniffing, some out-of-browser component on the user's machine is required. Most users we interviewed work on Linux or macOS operating systems, so a cross-platform application seemed appropriate.

The available frameworks depend on the environment. For cross-platform native desktop applications, comprehensive frameworks like

Qt [47] or wxWidgets [73] are available. Another possibility for cross-platform desktop applications are frameworks based on JavaScript and web browser technology, like Electron [15]. Finally, a local web application could be developed, by creating a server software running directly on the user's machine, and a frontend running in the web browser. The frontend could use any common web framework, like Sencha Ext JS [52], or a custom combination of web components.

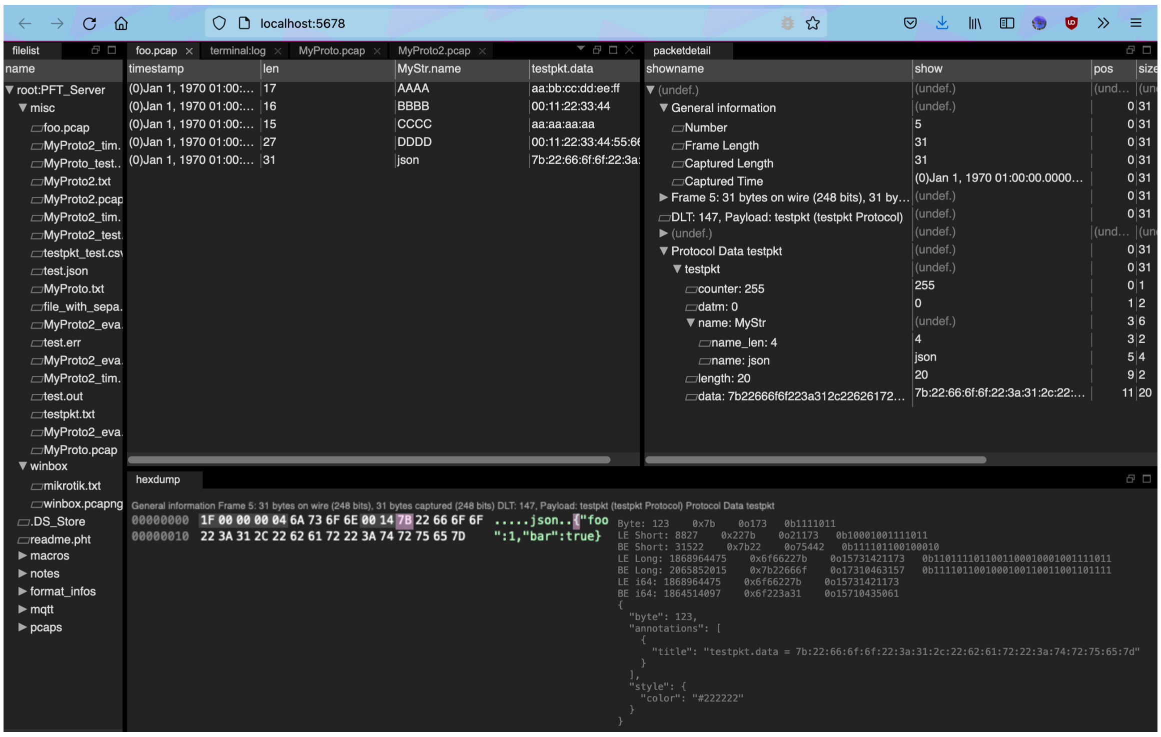
During the design phase, we evaluated multiple GUI environments and frameworks. We developed prototypes of different extents in each case. We evaluated developing an Electron application, however, we wanted to use a Python backend because we wanted to interface with ScaPy at that point. Therefore, we evaluated using a locally running backend implemented in Python or Node.js and a web application running in the browser as frontend. To this end, we tried out the Sencha Ext JS framework and the GoldenLayout dock panel toolkit.

The first more extensive prototype (Figure 9) was implemented as a web application in TypeScript, using the GoldenLayout layout manager [21] and various other web components [74][35], connecting over WebSockets to a local Python backend. Downsides were the missing portability of code and the complexity of shared state between frontend and backend implementation, as well as missing mature GUI components. The former led us to implement an overly complex Remote Procedure Call (RPC) solution, the latter to implement a custom tree/grid component. Both turned out to be too complex to maintain, and leading away from the actual scope of this thesis.

Therefore, we discarded this approach and implemented another extensive frontend prototype in Python with the PyQt5 module (Figure 10). This allowed us to use the Qt framework's mature library of GUI components, and also give up the strict separation between frontend and backend. The necessity to work with local files and local network sniffing made a local desktop application seem like a better fit anyway. Furthermore, we gained the option of integrating some graphical features we develop into Wireshark in future work, because it also uses Qt as its GUI framework. Finally, the vast number of Python modules is available for developing the application itself and for plugins our users might want to build.

4.5.2 *Window Handling*

As described in Section 2.3.1, there are several common approaches to handle the need of working with multiple documents at the same time. Based on the expected expertise of potential users, i.e. people from a software development and reverse engineering background, we assumed that users would prefer a similar document handling as in Integrated Development Environments (IDEs) and reverse engineering tools. User feedback in the interviews confirmed this, leading us to



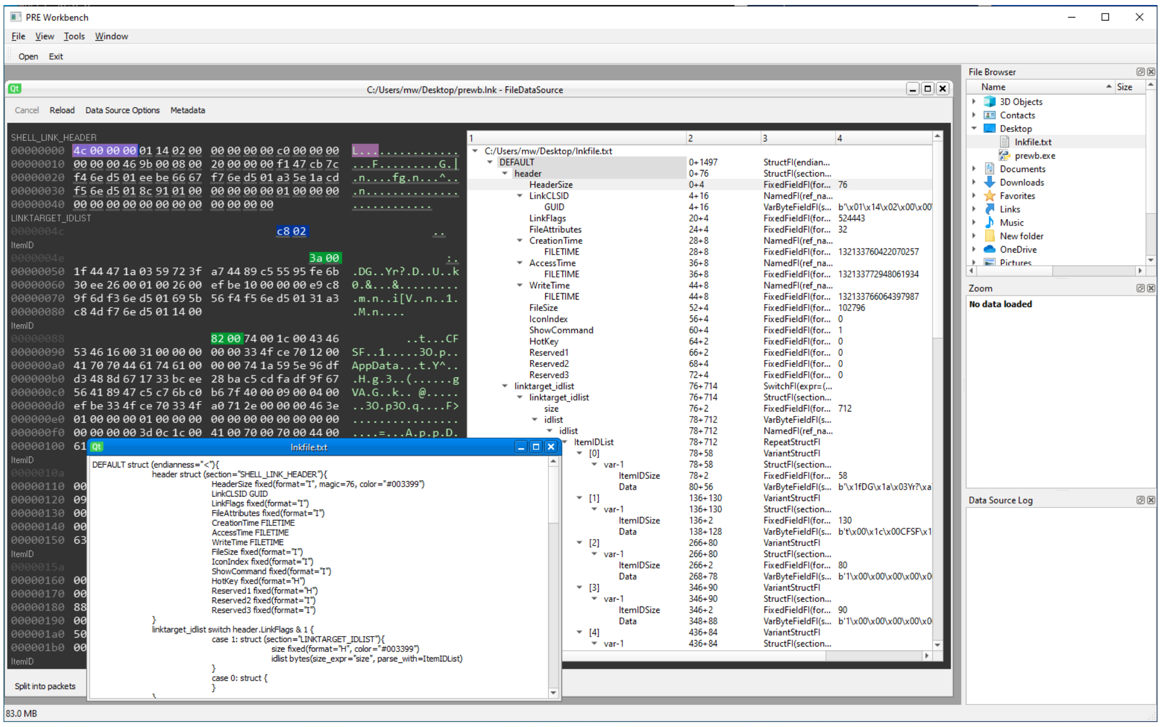
timestamp	len	MyStr.name	testpkt.data
(0)Jan 1, 1970 01:00:…	17	AAAA	aa:bb:cc:dd:ee:ff
(0)Jan 1, 1970 01:00:…	16	BBBB	00:11:22:33:44
(0)Jan 1, 1970 01:00:…	15	CCCC	aa:aa:aa:aa
(0)Jan 1, 1970 01:00:…	27	DDDD	00:11:22:33:44:55:66
(0)Jan 1, 1970 01:00:…	31	json	7b:22:66:6f:6f:22:3a:

showname	show	pos	size
▼ (undef.)	(undef.)	(und…	(un
▼ General information	(undef.)	0	31
Number	5	0	31
Frame Length	31	0	31
Captured Length	31	0	31
Captured Time	(0)Jan 1, 1970 01:00:00.0000…	0	31
▶ Frame 5: 31 bytes on wire (248 bits), 31 by…	(undef.)	0	31
DLT: 147, Payload: testpkt (testpkt Protocol)	(undef.)	0	31
▶ (undef.)	(undef.)	(und…	(un
▼ Protocol Data testpkt	(undef.)	0	31
▼ testpkt	(undef.)	0	31
counter: 255	255	0	1
datm: 0	0	1	2
▼ name: MyStr	(undef.)	3	6
name_len: 4	4	3	2
name: json	json	5	4
length: 20	20	9	2
data: 7b22666f6f223a312c22626172…	7b:22:66:6f:6f:22:3a:31:2c:22:…	11	20

```
General information Frame 5: 31 bytes on wire (248 bits), 31 bytes captured (248 bits) DLT: 147, Payload: testpkt (testpkt Protocol) Protocol Data testpkt
00000000  1F 00 00 00 04 6A 73 6F 6E 00 14 7B 22 66 6F 6F  .....json..{"foo
00000010  22 3A 31 2C 22 62 61 72 22 3A 74 72 75 65 7D     ":1,"bar":true}
```

```
Byte: 123    0x7b    0o173    0b1111011
LE Short: 8827    0x227b    0o21173    0b10001001111011
BE Short: 31522    0x7b22    0o75442    0b111101100100010
LE Long: 1868964475    0x6f66227b    0o15731421173    0b1101111011001100010001001111011
BE Long: 2065852015    0x7b22666f    0o17310463157    0b1111011001000100110011001101111
LE i64: 1868964475    0x6f66227b    0o15731421173
BE i64: 1864514097    0x6f223a31    0o15710435061
{
  "byte": 123,
  "annotations": [
    {
      "title": "testpkt.data = 7b:22:66:6f:6f:22:3a:31:2c:22:62:61:72:22:3a:74:72:75:65:7d"
    }
  ],
  "style": {
    "color": "#222222"
  }
}
```

Figure 9: Frontend prototype developed as a local web application in TypeScript, using the Golden-Layout dock panel suite, and a custom grid component



```
SHELL_LINK_HEADER
00000000  4c 00 00 00 01 14 02 00  00 00 00 00 c0 00 00 00  L...............
00000010  00 00 00 46 9b 00 08 00  20 00 00 00 f1 47 cb 7c  ...F........G.|
00000020  f4 6e d5 01 ee be 66 67  f7 6e d5 01 a3 5e 1a cd  .n....fg.n...^..
00000030  f5 6e d5 01 8c 91 01 00  00 00 00 00 01 00 00 00  .n..............
00000040  00 00 00 00 00 00 00 00  00 00 00 00              ............
LINKTARGET_IDLIST
0000004c                                       c8 02                    ..
ItemID
0000004e                                             3a 00              :.
00000050  1f 44 47 1a 03 59 72 3f  a7 44 89 c5 55 95 fe 6b  .DG..Yr?.D..U..k
00000060  30 ee 26 00 01 00 26 00  ef be 10 00 00 00 e9 c8  0.&...&.........
00000070  9f 6d f3 6e d5 01 69 5b  56 f4 f5 6e d5 01 31 a3  .m.n..i[V..n..1.
00000080  c8 4d f7 6e d5 01 14 00                          .M.n....
ItemID
00000088                           82 00 74 00 1c 00 43 46          ..t...CF
00000090  53 46 16 00 31 00 00 00  00 00 33 4f ce 70 12 00  SF..1.....3O.p..
000000a0  41 70 70 44 61 74 61 00  00 00 74 1a 59 5e 96 df  AppData...t.Y^..
000000b0  d3 48 8d 67 17 33 bc ee  28 ba c5 cd fa df 9f 67  .H.g.3..(......g
000000c0  56 41 89 47 c5 c7 6b c0  b6 7f 40 00 09 00 04 00  VA.G..k.. @.....
000000d0  ef be 33 4f ce 70 33 4f  a0 71 2e 00 00 00 46 3e  ..3O.p3O.q....F>
000000e0  01 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  ................
000000f0  00 00 00 00 3d 0c 1c 00  41 00 70 00 70 00 44 00  ....=...A.p.p.D.
00000100  61
ItemID
0000010a
00000110  00
00000120  09
00000130  00
00000140  00
00000150  63
ItemID
0000015a
00000160  00
00000170  00
00000180  88
00000190  00
000001a0  50
000001b0  00
ItemID
```

1	2	3	4
C:/Users/mw/Desktop/lnkfile.txt			
DEFAULT	0+1497	StructFI(endian…	
header	0+76	StructFI(section…	
HeaderSize	0+4	FixedFieldFI(for…	76
LinkCLSID	4+16	NamedFI(ref_na…	
GUID	4+16	VarByteFieldFI(s…	b'\x01\x14\x02\x00\x00
LinkFlags	20+4	FixedFieldFI(for…	524443
FileAttributes	24+4	FixedFieldFI(for…	32
CreationTime	28+8	NamedFI(ref_na…	
FILETIME	28+8	FixedFieldFI(for…	132133760422070257
AccessTime	36+8	NamedFI(ref_na…	
FILETIME	36+8	FixedFieldFI(for…	132133772948061934
WriteTime	44+8	NamedFI(ref_na…	
FILETIME	44+8	FixedFieldFI(for…	132133766064397987
FileSize	52+4	FixedFieldFI(for…	102796
IconIndex	56+4	FixedFieldFI(for…	0
ShowCommand	60+4	FixedFieldFI(for…	1
HotKey	64+2	FixedFieldFI(for…	0
Reserved1	66+2	FixedFieldFI(for…	0
Reserved2	68+4	FixedFieldFI(for…	0
Reserved3	72+4	FixedFieldFI(for…	0
linktarget_idlist	76+714	SwitchFI(expr=(…	
linktarget_idlist	76+714	StructFI(section…	
size	76+2	FixedFieldFI(for…	712
idlist	78+712	VarByteFieldFI(s…	
idlist	78+712	NamedFI(ref_na…	
ItemIDList	78+712	RepeatStructFI	
[0]	78+58	VariantStructFI	
var-1	78+58	StructFI(section…	
ItemIDSize	78+2	FixedFieldFI(for…	58
Data	80+56	VarByteFieldFI(s…	b'\x1fDG\x1a\x03Yr?\xa
[1]	136+130	VariantStructFI	
var-1	136+130	StructFI(section…	
ItemIDSize	136+2	FixedFieldFI(for…	130
Data	138+128	VarByteFieldFI(s…	b't\x00\x1c\x00CFSF\x1
[2]	266+80	VariantStructFI	
var-1	266+80	StructFI(section…	
ItemIDSize	266+2	FixedFieldFI(for…	80
Data	268+78	VarByteFieldFI(s…	b'1\x00\x00\x00\x00\x0
[3]	346+90	VariantStructFI	
var-1	346+90	StructFI(section…	
ItemIDSize	346+2	FixedFieldFI(for…	90
Data	348+88	VarByteFieldFI(s…	b'1\x00\x00\x00\x00\x0
[4]	436+84	VariantStructFI	
var-1	436+84	StructFI(section…	

```
DEFAULT struct (endianness="<"){
        header struct (section="SHELL_LINK_HEADER"){
                HeaderSize fixed(format="I", magic=76, color="#003399")
                LinkCLSID GUID
                LinkFlags fixed(format="I")
                FileAttributes fixed(format="I")
                CreationTime FILETIME
                AccessTime FILETIME
                WriteTime FILETIME
                FileSize fixed(format="I")
                IconIndex fixed(format="I")
                ShowCommand fixed(format="I")
                HotKey fixed(format="H")
                Reserved1 fixed(format="H")
                Reserved2 fixed(format="I")
                Reserved3 fixed(format="I")
        }
        linktarget_idlist switch header.LinkFlags & 1 {
                case 1: struct (section="LINKTARGET_IDLIST"){
                        size fixed(format="H", color="#003399")
                        idlist bytes(size_expr="size", parse_with=ItemIDList)
                }
                case 0: struct {
                }
```

Figure 10: Frontend prototype developed as a Qt framework desktop application in Python

implement a Tabbed-Document Interface (TDI) with dockable and floatable windows.

4.5.3 *File and Project Handling*

We examined the UX concepts of several applications with regards to file handling and grouped them in four broad categories.

Category A is the *classic multi document* application, e.g. Microsoft Word, most modern text editors like VS Code, Sublime Text, Notepad++, and image editors like Gimp and Photoshop. In these applications, each open window can be either "untitled", or associated with a file, and it can be either *dirty* (modified, unsaved) or *not dirty* (unmodified, saved).

As category B, *file-oriented multi document* application, we characterized IDEs like IntelliJ, and modern Web Browsers. Each document must be associated with a file (or an URL in case of browsers), cannot be dirty (either files are auto-saved, or the application has no editing support).

In category C, we placed *project-oriented* applications, where all data is stored in a project file, which in turn can contain abstract data containers which are not associated to a file. Files can be imported into the application, but the resulting container is detached from the source file. Saving data back to a file is a distinct "export" process. We found this concept in the Hobbits application (see Section 3.2.5) as well as some audio/video software, where media is imported into a media library.

The last category D is *pipeline-oriented* applications, like GNU Radio. These applications are centered around data pipelines with various data sources, filters and outputs. Files can be the start or end of a pipeline, but are just one data source and sink of many possible. The data shown in the GUI is the result of a user-defined pipeline (e.g. "load file → split according to rules → display as list of binary packets").

4.5.3.1 *Conclusion*

Based on user input of frequently working on multiple independent projects, we devised a project-based workflow instead of one based on completely independent files. An entire folder can be loaded as a project, with all related settings, window layouts, macros, annotations, and grammars stored in a self-contained project database file. Regarding these data, our application is designed as described in category C. This allows users to, on one hand, share grammars and annotations easily between files of the same project, while on the other hand, keeping different projects cleanly separated. Multiple instances of the application can be opened at the same time to work on different projects.

On the other hand, users reported to often work on files generated by external tools, so we included some concepts from category A, allowing users to directly load the actual data files from the project folder, and keep the association to the file. Finally, users also want to work on live data, e.g. ingesting the results of network sniffing directly into the application, leading us to introduce some lightweight pipeline-oriented concepts into the application. Therefore, our application is based on category C, but contains some concepts from A and D.

5 IMPLEMENTATION

The first prototype was implemented as a web application in TypeScript, connecting over WebSockets to a local Python backend. Missing portability of code between frontend and backend implementation led us to re-implement the frontend in Python with the PyQt5 module. This also allowed us to use Qt's mature library of Graphical User Interface (GUI) components. Furthermore, we gain the possibility of integrating some graphical features into Wireshark, which also uses Qt as its GUI framework.

5.1 IMPLEMENTING MAIN DATA STRUCTURES

In the Section 4.4, we defined the core data structures of the application. In this section, we will go into more details on their implementation.

5.1.1 *Byte Buffer*

`ByteBuffers` are the basic data container used throughout the application. When a binary file is opened in the application, it is loaded into a byte buffer, and displayed in the HexView component. When a PCAP file is loaded, the packets are loaded into individual byte buffers, which are bundled in a byte buffer list.

METADATA Byte buffers store five kinds of metadata:

- A dictionary of metadata provided by the data source, e.g. the packet timestamp from a PCAP file.
- Whether it is marked in the list or not.
- A list of annotations, stored internally in a `RangeList` as described in Section 2.4.
- The name of the annotation set which is currently applied to the buffer (`annotation_set_name`).
- If the buffer was parsed using a grammar description: the grammar description name (`fi_root_name`), and the parsed fields, as a dictionary accessible by their name (`fields`) and in a tree structure according to the grammar (`fi_tree`).

PACKET VS. STREAM A `ByteBuffer` can represent any sequence of bytes, e.g. a packet, a file or a data stream (e.g. a reassembled

TCP stream). In the latter case it is often required to split the stream into individual packets. There are multiple ways to do this in PRE Workbench:

- A grammar description can be created, and the `store_into` parameter can be set on the type instance which represents a packet. Once a buffer is parsed with this description, the packets are stored for further use in a new `ByteBufferList`.
- A macro can use the `ByteBuffer` as input data and produce a `ByteBufferList` from it. This allows the user to use arbitrary Python code to split the data.
- Instead of loading the whole stream into a `ByteBuffer`, a custom `DataSource` can be created which splits the data into packets directly while loading it.

5.1.2 *Byte Buffer List*

A ByteBufferList stores an ordered list of `ByteBuffers` and a dictionary containing metadata, which is usually provided by the data source (e.g. headers of a PCAP file).

It supports live updates, so data sources can add more packets in the background. This is useful for live network traces, or "tailing" of a file generated by an external tool. The list is usually displayed by a `PacketListWidget`, which also supports updating the display on the fly.

STREAM REASSEMBLY Stream reassembly is supported by allowing each `ByteBuffer` in the list to have a reassembly key, which can be used to add more bytes to an existing buffer. For example, the four-tuple of a TCP stream (source address, source port, destination address, destination port) could be used as the reassembly key. Then, each time a new TCP segment is parsed by a grammar with the appropriate `reassemble_into` param, it is appended to the ByteBuffer. For TCP, one usually would not implement this in PRE Workbench, but use the well-tested TCP stream following feature of Wireshark, but this is useful when reverse engineering proprietary data segmentation protocols (see e.g. Section 4.2.2).

5.1.3 *Format Infos*

A `FormatInfo` object is the internal representation of a specific type instance in a grammar definition, e.g. a specific `struct`, or a byte array with specific length and display styles. It is instantiated from a subtree of the Abstract Syntax Tree (AST) of a grammar file. These objects implement the actual parsing from a ByteBuffer, and can convert themselves back to their text representation.

The `FormatInfoContainer` stores a dictionary mapping definition names to `FormatInfo` objects. It allows serializing and deserializing the contained `FormatInfos`. Several subclasses exist, for example an interactive implementation, which can ask the user to create a new definition if a non-existant name is referenced. Also, the default implementation reads and writes the definitions from and to a text file, while the `ProjectFormatInfoContainer` store the definitions in a project database.

5.2 DATA SOURCES

In the application source code and in plugins, data sources are implemented as Python classes inheriting from either `DataSource` or `SyncDataSource`. The former can provide data asynchronously, allowing for live capture of data, for example by calling a third-party process or by reading from the network. The latter are easier to implement and only load data synchronously, which is most useful to implement data importers for local files of different types. User-defined macros always work synchronously, they are just a Python code snippet which is given all input parameters as predefined variables and needs to store the generated `ByteBuffer` or `ByteBufferList` object in the `output` variable.

5.2.1 *PCAP Files*

The predefined *PCAP file data source* simply loads all packets from a local PCAP file by directly parsing the file contents. It supports the PCAP and PCAPNG file types in big-endian and little-endian byte order.

5.2.2 *CSV Files*

The *CSV file data source* loads a file in the Comma-Separated Values (CSV) format into a `ByteBufferList`, so that each row is converted to a `ByteBuffer`. One column of the CSV file has to be selected to provide the payload, which can be decoded from a hexadecimal or Base64 string. All remaining columns are stored as packet metadata in the ByteBuffer. The user can configure the exact format of the file, specifying the column delimiters, quote characters, and whether a header row is present or not.

5.2.3 *Binary Files*

Raw binary files can be imported using two different predefined data sources, the binary file and directory of binary files data sources.

The *binary file data source* simply loads binary data from a file into a `ByteBuffer` without any modification. No metadata is generated. The *directory of binary files data source* scans a local directory using a *glob*-style search pattern, and loads all matching files into a `ByteBufferList`. The file names and modification timestamps are stored as packet metadata.

5.2.4 *Macro*

As a demonstration for the macro capabilities, and as a template for users who want to implement their own, we implemented three different macro data sources: One which imports a JSON file as a `ByteBufferList`, another that imports an Intel HEX file as `ByteBuffer`, as well as one that imports a binary file splitted into packets using static delimiters as a `ByteBufferList`.

The *JSON file data source* works similar to the CSV one. Instead of rows in the CSV file, there needs to be a JSON array at the root level. Each array item must be a dictionary, each of which is converted into a `ByteBuffer`, where one entry is used as the payload and all others are stored as metadata. If the user needs to import a different JSON structure, they can copy the macro to their project and adapt the code accordingly.

5.2.5 *Data Import From Wireshark*

As mentioned in Section 4.3.1, we want to harness the multitude of existing Wireshark dissectors. To this end, we allow the user to import packets via Wireshark's command-line utility, `tshark`. It provides a feature to export traces with full dissector output in the Packet Description Markup Language (PDML), an XML-based file format [41]. It allows live captures, as well as importing existing PCAP files, so we implemented the *Live capture via Tshark* and *PCAP file via Tshark* data sources.

The PDML format is structured in the same way as the tree view in Wireshark's GUI. As we also support a tree structure for the protocol fields in our `ByteBuffers`, we can store the dissector output almost verbatim in our internal data structures. PDML also contains byte range information on the fields, so we can also store and display them as annotations.

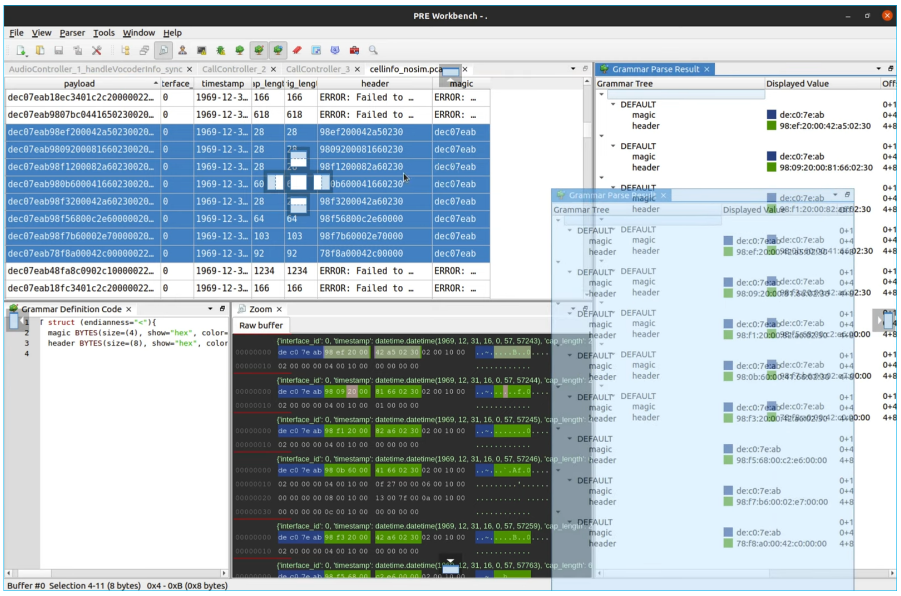
This allows the user to fully use our GUI with data imported from Wireshark, seeing the existing annotations from the dissector and adding their own ones later on.

5.3 DATA EXPLORATION GUI

The software provides multiple interconnected views on the gathered data. The main element is the HexView, which is either displayed immediately after loading a single binary file, or once a packet is selected after loading a packet list.

5.3.1 *Dock Panel Layout*

Qt natively provides a basic implementation of dockable tool windows and tabbed documents. However, we found this not powerful enough, especially since no split screen view of documents is possible. Therefore, we use the open source *Qt Advanced Docking System* library, which "lets you create customizable layouts using a full featured window docking system similar to what is found in many popular integrated development environments (IDEs) such as Visual Studio" [19]. Especially, it supports arbitrarily nested tiled and tabbed layouts, allowing the user to view multiple documents at once. While implementing the library in our application, we contributed to it, by updating the PyQt5 bindings and build process, and fixing some bugs. The screenshot in Figure 11 shows how a tool window is moved to a different dock area.



payload	terface_	timestamp	ap_lengt	ig_leng	header	magic
dec07eab18ec3401c2c20000022…	0	1969-12-3…	166	166	ERROR: Failed to …	ERROR: …
dec07eab9807bc0441650230020…	0	1969-12-3…	618	618	ERROR: Failed to …	ERROR: …
dec07eab98ef200042a50230020…	0	1969-12-3…	28	28	98ef200042a50230	dec07eab
dec07eab9809200081660230020…	0	1969-12-3…	28	28	9809200081660230	dec07eab
dec07eab98f1200082a60230020…	0	1969-12-3…	28	28	98f1200082a60230	dec07eab
dec07eab980b600041660230020…	0	1969-12-3…	60	6	0b600041660230	dec07eab
dec07eab98f3200042a60230020…	0	1969-12-3…	28	2	98f3200042a60230	dec07eab
dec07eab98f56800c2e60000020…	0	1969-12-3…	64	64	98f56800c2e60000	dec07eab
dec07eab98f7b60002e70000020…	0	1969-12-3…	103	103	98f7b60002e70000	dec07eab
dec07eab78f8a00042c00000022…	0	1969-12-3…	92	92	78f8a00042c00000	dec07eab
dec07eab48fa8c0902c10000022…	0	1969-12-3…	1234	1234	ERROR: Failed to …	ERROR: …
dec07eab18fc3401c2c20000022…	0	1969-12-3…	166	166	ERROR: Failed to …	ERROR: …

```
struct (endianness="<"){
  magic BYTES(size=(4), show="hex", color=
  header BYTES(size=(8), show="hex", color
```

```
{'interface_id': 0, 'timestamp': datetime.datetime(1969, 12, 31, 16, 0, 57, 57243), 'cap_length': 2
00000000  de c0 7e ab 98 ef 20 00 42 a5 02 30 02 00 10 00  ..~.....B..0....
00000010  02 00 00 00 04 00 10 00 00 00 00 00              ............
{'interface_id': 0, 'timestamp': datetime.datetime(1969, 12, 31, 16, 0, 57, 57244), 'cap_length': 2
00000000  de c0 7e ab 98 09 20 00 81 66 02 30 02 00 10 00  ..~.......f.0....
00000010  02 00 00 00 04 00 10 00 01 00 00 00              ............
{'interface_id': 0, 'timestamp': datetime.datetime(1969, 12, 31, 16, 0, 57, 57245), 'cap_length':
00000000  de c0 7e ab 98 f1 20 00 82 a6 02 30 02 00 10 00
00000010  02 00 00 00 04 00 10 00 00 00 00 00
{'interface_id': 0, 'timestamp': datetime.datetime(1969, 12, 31, 16, 0, 57, 57246), 'cap_length':
00000000  de c0 7e ab 98 0b 60 00 41 66 02 30 02 00 10 00
00000010  02 00 00 00 04 00 10 00 0f 27 00 00 06 00 10 00
00000020  00 00 00 00 08 00 10 00 13 00 7f 00 0a 00 10 00
00000030  00 00 00 00 0c 00 10 00 00 00 00 00
{'interface_id': 0, 'timestamp': datetime.datetime(1969, 12, 31, 16, 0, 57, 57259), 'cap_length':
00000000  de c0 7e ab 98 f3 20 00 42 a6 02 30 02 00 10 00
00000010  02 00 00 00 04 00 10 00 00 00 00 00
{'interface_id': 0, 'timestamp': datetime.datetime(1969, 12, 31, 16, 0, 57, 57763), 'cap_length':
```

Grammar Tree	Displayed Value	Off
DEFAULT		0+1
magic	🟦 de:c0:7e:ab	0+4
header	🟩 98:ef:20:00:42:a5:02:30	4+8
DEFAULT		0+1
magic	🟦 de:c0:7e:ab	0+4
header	🟩 98:09:20:00:81:66:02:30	4+8
DEFAULT		0+1
magic	🟦 de:c0:7e:ab	0+4
header	🟩 98:f1:20:00:82:a6:02:30	4+8
DEFAULT		0+1
magic	🟦 de:c0:7e:ab	0+4
header	🟩 98:f5:68:00:c2:e6:00:00	4+8
DEFAULT		0+1
magic	🟦 de:c0:7e:ab	0+4
header	🟩 98:f7:b6:00:02:e7:00:00	4+8
DEFAULT		0+1
magic	🟦 de:c0:7e:ab	0+4
header	🟩 78:f8:a0:00:42:c0:00:00	4+8

Buffer #0 Selection 4-11 (8 bytes) 0x4 - 0xB (0x8 bytes)

Figure 11: Screenshot of Qt Advanced Docking System in use in PRE Workbench

5.3.2 *Packet List*

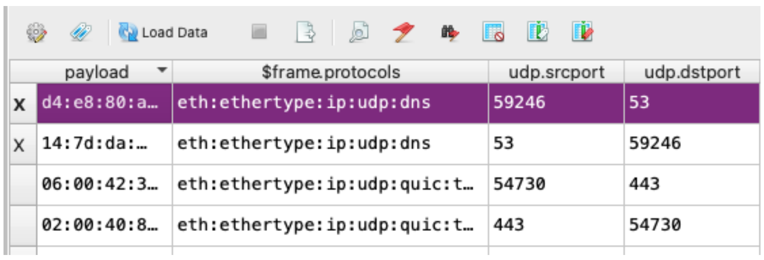
The packet list widget displays a `ByteBufferList` in a classical table layout (see Figure 12), each row representing a ByteBuffer. Packets selected in the list are displayed in the `Zoom` tool window in a HexView component, where they can be further examined.

The column contents are rendered based on user-defined expressions, which have access to the `ByteBuffer` and all its fields and other metadata. Expressions can be entered manually, to perform calculations on packet contents, or generated automatically from the available fields using a *Quick Add Column* feature. The user can transfer custom column layouts to other packet lists via the clipboard.

A marker feature allows the user to mark interesting of the packets for future reference. They can also use the search feature to mark all packets for which a provided expression evaluates to `true`.

Furthermore, macros with input type ByteBuffer or ByteBufferList can be executed on selected rows or on the whole list, allowing the user to process packets using custom code.

Internally, the widget is based on the Qt-provided `QTableWidget` with a custom `QAbstractItemModel`, which implements an adapter between a `ByteBufferList` and the Qt framework.



	payload	$frame.protocols	udp.srcport	udp.dstport
X	d4:e8:80:a…	eth:ethertype:ip:udp:dns	59246	53
X	14:7d:da:…	eth:ethertype:ip:udp:dns	53	59246
	06:00:42:3…	eth:ethertype:ip:udp:quic:t…	54730	443
	02:00:40:8…	eth:ethertype:ip:udp:quic:t…	443	54730

Figure 12: Screenshot of packet list

5.3.3 *HexView*

The *HexView* component is a viewer for binary data and is the main visual component of the application. It displays one or multiple ByteBuffers in a hexadecimal and ASCII representation ("hex dump") next to each other. Ranges of bytes can be annotated with different foreground and background colors. Section headings can be inserted to visually split a buffer.

It is implemented as a completely custom Qt Widget, including rendering the display, handling selections, scrolling, and keyboard shortcuts.

ANNOTATIONS In the HexView `ByteBuffers` can be interactively annotated with color via context menu or keyboard shortcut. In addition, a new section can be started and provided with a corresponding heading. These annotations can be saved in an Annotation Set and applied to further `ByteBuffers`.

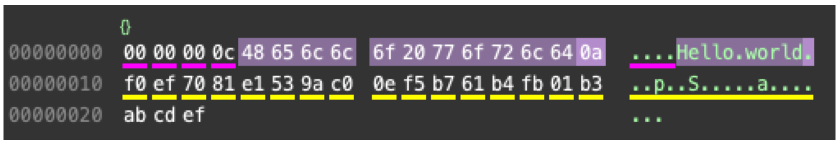
In addition, the HexView can display annotations that were added by parsing with a grammar definition (see Section 5.5.1).

HEURISTIC HIGHLIGHTER Upon selection of a byte range, several heuristic detection methods are run in the background, to highlight matching information in the vicinity of the selected data.

For example, if a range of bytes is selected, the *selection length matcher* will convert the number of bytes into various formats, e.g. big and little endian integer, decimal and hexadecimal string. It will then search for these representations of the selection length in and around the selection. The *hash matcher* will calculate digests according to different hash algorithms and search for them before and after the selection (Figure 13a).

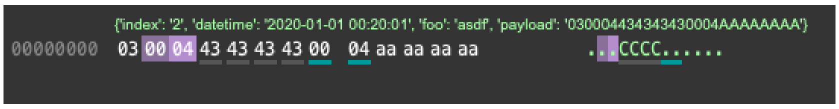
The other way around, if a short byte range is selected, its integer value will be interpreted as a number of bytes, and this many bytes after the selection will be highlighted in a light grey. Repetitions of the same bytes as selected are highlighted in teal (Figure 13b).

New heuristics can be implemented as simple Python functions, either directly in the application code, or in a plugin. For example, a plugin could match for checksums of the selected bytes.



```
             {}
00000000   00 00 00 0c 48 65 6c 6c  6f 20 77 6f 72 6c 64 0a   ....Hello.world.
00000010   f0 ef 70 81 e1 53 9a c0  0e f5 b7 61 b4 fb 01 b3   ..p..S.....a....
00000020   ab cd ef                                           ...
```

(a) The *selection length* (magenta) and *hash* (yellow) matchers



```
           {'index': '2', 'datetime': '2020-01-01 00:20:01', 'foo': 'asdf', 'payload': '030004434343430004AAAAAAAA'}
00000000   03 00 04 43 43 43 43 00  04 aa aa aa aa             ...CCCC......
```

(b) The *repetition* matcher (teal) and the *selection as length* highlighter (grey)

Figure 13: Screenshots of heuristic highlighter results

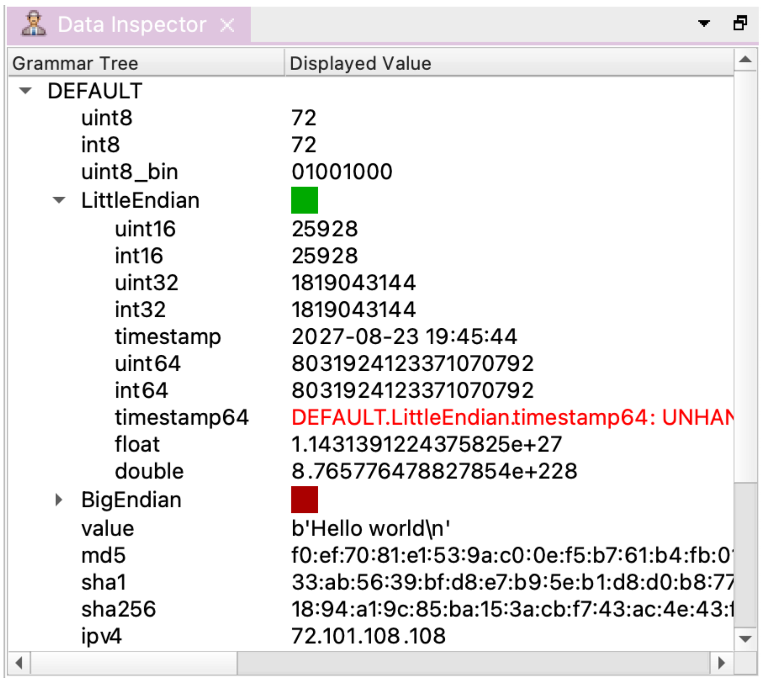
5.3.4 *Helpers*

We implemented several features which aid the reverse engineer in working with binary data.

DATA INSPECTOR A data inspector panel (Figure 14) shows interpretations of the selected byte range in various formats such as integers, floating point numbers, timestamps, colors and IP addresses. It also displays hashes calculated from the selection.

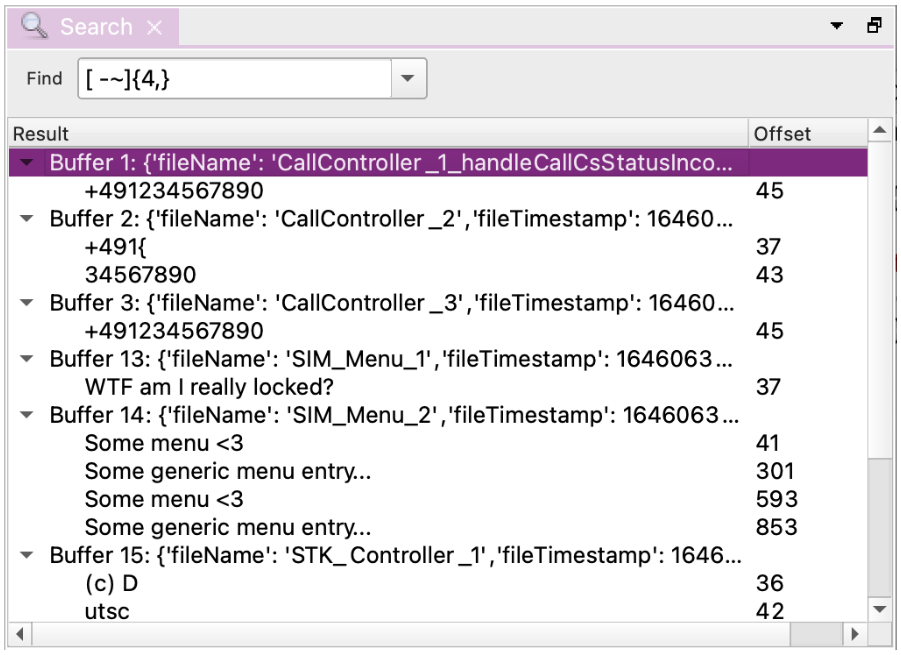
REGEX SEARCH The user can use regular expressions to search for patterns in one or multiple buffers. The results are displayed in a tree view (Figure 15), clicking a result highlights it in the HexView.

RUNNING EXTERNAL TOOLS Using the external tools feature, the user can use any external command line tool and apply it to the se-



Grammar Tree	Displayed Value
DEFAULT	
uint8	72
int8	72
uint8_bin	01001000
LittleEndian	🟩
uint16	25928
int16	25928
uint32	1819043144
int32	1819043144
timestamp	2027-08-23 19:45:44
uint64	8031924123371070792
int64	8031924123371070792
timestamp64	DEFAULT.LittleEndian.timestamp64: UNHAN
float	1.1431391224375825e+27
double	8.765776478827854e+228
BigEndian	🟥
value	b'Hello world\n'
md5	f0:ef:70:81:e1:53:9a:c0:0e:f5:b7:61:b4:fb:0
sha1	33:ab:56:39:bf:d8:e7:b9:5e:b1:d8:d0:b8:77
sha256	18:94:a1:9c:85:ba:15:3a:cb:f7:43:ac:4e:43:1
ipv4	72.101.108.108

Figure 14: Screenshot of the data inspector



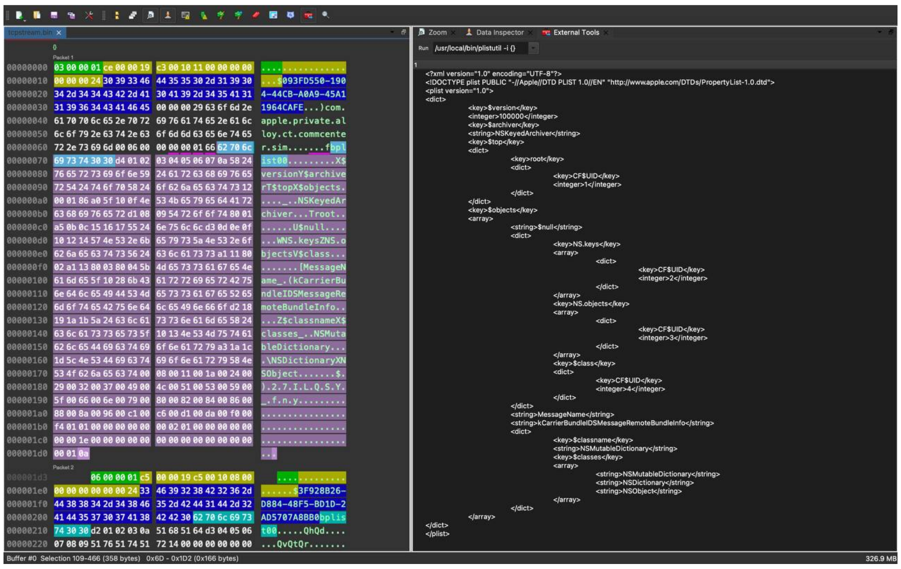
[-~]{4,}

Result	Offset
Buffer 1: {'fileName': 'CallController_1_handleCallCsStatusInco...	
+491234567890	45
Buffer 2: {'fileName': 'CallController_2','fileTimestamp': 16460...	
+491{	37
34567890	43
Buffer 3: {'fileName': 'CallController_3','fileTimestamp': 16460...	
+491234567890	45
Buffer 13: {'fileName': 'SIM_Menu_1','fileTimestamp': 1646063...	
WTF am I really locked?	37
Buffer 14: {'fileName': 'SIM_Menu_2','fileTimestamp': 1646063...	
Some menu <3	41
Some generic menu entry...	301
Some menu <3	593
Some generic menu entry...	853
Buffer 15: {'fileName': 'STK_Controller_1','fileTimestamp': 1646...	
(c) D	36
utsc	42

Figure 15: Screenshot of the regular expression search

lected data. The currently selected byte range is written to a temporary file, whose name is inserted into the command line at the location marked with `{}`.

The screenshot in Figure 16 shows the external tools feature in use as a researcher decodes a binary plist (property list) file embedded in a protocol frame.



```
/usr/local/bin/plistutil -i {}
```

```
00000000 03 00 00 01 ce 00 00 19  c3 00 10 11 00 00 00 00  ................
00000010 00 00 00 24 30 39 33 46  44 35 35 30 2d 31 39 30  ...$093FD550-190
00000020 34 2d 34 34 43 42 2d 41  30 41 39 2d 34 35 41 31  4-44CB-A0A9-45A1
00000030 31 39 36 34 43 41 46 45  00 00 00 29 63 6f 6d 2e  1964CAFE...)com.
00000040 61 70 70 6c 65 2e 70 72  69 76 61 74 65 2e 61 6c  apple.private.al
00000050 6c 6f 79 2e 63 74 2e 63  6f 6d 6d 63 65 6e 74 65  loy.ct.commcente
00000060 72 2e 73 69 6d 00 06 00  00 00 00 01 66 62 70 6c  r.sim.......fbpl
00000070 69 73 74 30 30 d4 01 02  03 04 05 06 07 0a 58 24  ist00.........X$
00000080 76 65 72 73 69 6f 6e 59  24 61 72 63 68 69 76 65  versionY$archive
00000090 72 54 24 74 6f 70 58 24  6f 62 6a 65 63 74 73 12  rT$topX$objects.
000000a0 00 01 86 a0 5f 10 0f 4e  53 4b 65 79 65 64 41 72  ...._..NSKeyedAr
000000b0 63 68 69 76 65 72 d1 08  09 54 72 6f 6f 74 80 01  chiver...Troot..
000000c0 a5 0b 0c 15 16 17 55 24  6e 75 6c 6c d3 0d 0e 0f  ......U$null....
000000d0 10 12 14 57 4e 53 2e 6b  65 79 73 5a 4e 53 2e 6f  ...WNS.keysZNS.o
000000e0 62 6a 65 63 74 73 56 24  63 6c 61 73 73 a1 11 80  bjectsV$class...
000000f0 02 a1 13 80 03 80 04 5b  4d 65 73 73 61 67 65 4e  .......[MessageN
00000100 61 6d 65 5f 10 28 6b 43  61 72 72 69 65 72 42 75  ame_.(kCarrierBu
00000110 6e 64 6c 65 49 44 53 4d  65 73 73 61 67 65 52 65  ndleIDSMessageRe
00000120 6d 6f 74 65 42 75 6e 64  6c 65 49 6e 66 6f d2 18  moteBundleInfo..
00000130 19 1a 1b 5a 24 63 6c 61  73 73 6e 61 6d 65 58 24  ...Z$classnameX$
00000140 63 6c 61 73 73 65 73 5f  10 13 4e 53 4d 75 74 61  classes_..NSMuta
00000150 62 6c 65 44 69 63 74 69  6f 6e 61 72 79 a3 1a 1c  bleDictionary...
00000160 1d 5c 4e 53 44 69 63 74  69 6f 6e 61 72 79 58 4e  .\NSDictionaryXN
00000170 53 4f 62 6a 65 63 74 00  08 00 11 00 1a 00 24 00  SObject.......$.
00000180 29 00 32 00 37 00 49 00  4c 00 51 00 53 00 59 00  ).2.7.I.L.Q.S.Y.
00000190 5f 00 66 00 6e 00 79 00  80 00 82 00 84 00 86 00  _.f.n.y.........
000001a0 88 00 8a 00 96 00 c1 00  c6 00 d1 00 da 00 f0 00  ................
000001b0 f4 01 01 00 00 00 00 00  00 02 01 00 00 00 00 00  ................
000001c0 00 00 1e 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000001d0 00 01 0a                                          ...
000001d3          06 00 00 01 c5  00 00 19 c5 00 10 08 00       .............
000001e0 00 00 00 00 00 00 24 33  46 39 32 38 42 32 36 2d  .......$3F928B26-
000001f0 44 38 38 34 2d 34 38 46  35 2d 42 44 31 44 2d 32  D884-48F5-BD1D-2
00000200 41 44 35 37 30 37 41 38  42 42 30 62 70 6c 69 73  AD5707A8BB0bplis
00000210 74 30 30 d2 01 02 03 0a  51 68 51 64 d3 04 05 06  t00.....QhQd....
00000220 07 08 09 51 76 51 74 51  72 14 00 00 00 00 00 00  ...QvQtQr.......
```

```
Buffer #0  Selection 109-466 (358 bytes)   0x6D - 0x1D2 (0x166 bytes)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>$version</key>
        <integer>100000</integer>
        <key>$archiver</key>
        <string>NSKeyedArchiver</string>
        <key>$top</key>
        <dict>
                <key>root</key>
                <dict>
                        <key>CF$UID</key>
                        <integer>1</integer>
                </dict>
        </dict>
        <key>$objects</key>
        <array>
                <string>$null</string>
                <dict>
                        <key>NS.keys</key>
                        <array>
                                <dict>
                                        <key>CF$UID</key>
                                        <integer>2</integer>
                                </dict>
                        </array>
                        <key>NS.objects</key>
                        <array>
                                <dict>
                                        <key>CF$UID</key>
                                        <integer>3</integer>
                                </dict>
                        </array>
                        <key>$class</key>
                        <dict>
                                <key>CF$UID</key>
                                <integer>4</integer>
                        </dict>
                </dict>
                <string>MessageName</string>
                <string>kCarrierBundleIDSMessageRemoteBundleInfo</string>
                <dict>
                        <key>$classname</key>
                        <string>NSMutableDictionary</string>
                        <key>$classes</key>
                        <array>
                                <string>NSMutableDictionary</string>
                                <string>NSDictionary</string>
                                <string>NSObject</string>
                        </array>
                </dict>
        </array>
</dict>
</plist>
```

```
326.9 MB
```

Figure 16: Screenshot of the external tools panel

ENTROPY MEASUREMENT The Shannon entropy [54] is a measure for the informational value of a message. In a reverse engineering context, it is useful to find packets which contain encrypted or compressed data. These will have a high entropy, because they have a similar distribution of byte values as random data, which means the frequency of each byte value is asymptotic equal, leading to the maximum entropy value. The built-in *Calculate Entropy* macro calculates the Shannon entropy of a buffer according to Equation 1. It can be applied to a list of `ByteBuffers`, for example all packets of a trace, and adds a new `$entropy` meta data field to each.

$$H(X) := -\frac{1}{8} \sum_{x \in \mathcal{X}} p(x) \log p(x) \tag{1}$$

5.4 EXTENSIBILITY

PRE Workbench can be extended by the user using macros and plugins. Macros are small code snippets, which are edited and run directly in the application, for predefined purposes, usually without a custom user interface. Plugins are developed in an external editor, and can extend the application more widely. For example, users can add custom tool windows using a plugin.

5.4.1 *Macros*

As explained in Section 4.3.5, researchers often need to write custom code to process protocol data. To integrate that as smooth as possible into the workflow, we implemented macro support. Macros are small user-defined Python code snippets with a defined input data type and parameter set. There are various places throughout the application, where macros can be run on data. Based on the input data type, only compatible macros are displayed. For example, macros with input types `BYTE_ARRAY` or `STRING` are only displayed in the context menu of the HexView or the text editor component, respectively. The input object to the macro is provided in a variable named `input`, which is directly usable in the macro code.

INPUT TYPES The following macro input types are currently implemented:

NONE
: The macro has no input. It can be executed by double-clicking it in the *Macros* tool window.

BYTE_BUFFER
: The macro expects a single `ByteBuffer` as input. It can be executed by right-clicking a packet in a *PacketListWidget* or in the context menu of a *HexView*. If multiple packets are selected, the macro is called repeatedly.

BYTE_BUFFER_LIST
: The macro expects a `ByteBufferList` as input. It can be executed in the same ways as a BYTE_BUFFER macro, but is only called once.

BYTE_ARRAY
: The macro expects a bytes type (sequence of bytes without metadata). It can be executed in the *HexView* context menu after selecting a byte range.

STRING
: The macro expects a character string as input. It can be executed from the selection context menu of all text editor components in the application.

DATA_SOURCE
: The macro shows up in the *Data Source Type* select box in the Data Source window. Its output (to be placed in the output variable

by the macro) will be displayed in the Data Source window's output widget.

SECURITY MODEL Macros run in the application process, without any additional sandboxing, therefore they have the same permissions as the app itself, usually full user permissions. We want to allow users to store macros per project. Finally, researchers are expected to share their project files. Therefore we implemented a *TOFE* (Trust On First Edit) security model based on a hash of the macro code, meaning macros are trusted once a user opened them once in the code editor, giving them a chance to review the code.

On the one hand, this means the security is completely invisible to a user working only locally on a project, because they will have edited the code of every macro they use, therefore all macros are trusted. On the other hand, if a project file from another machine or a colleague is loaded, all new macros are considered untrusted, meaning the user needs to first open each macro in the editor, carefully review it, and save it again. A hash of the code is then stored in their local configuration, marking it as trusted. It can then be run as usual.

5.4.2 *Plugins*

Plugins allow for a variety of extensions to the application by the user, including custom tool windows, file types, data sources, selection highlighters, and Protocol Grammar Description Language (PGDL) functions.

In terms of technical implementation, a plugin is a Python code file that is placed in the plugin directory by the user. Once it is marked as enabled in the *Manage Plugins* dialog, it is imported and executed when the program is started. Plugins are imported directly after the project file is loaded, but before the main window is initialized, allowing plugin developers to access project-specific data, and to still register custom tool windows, file types and data sources before the main window is opened.

We provide a number of example plugins [68] to aid as a starting point in developing custom plugins. The recommended way to develop plugins is to open the plugin directory as a PyCharm [45] project and configure PyCharm to use the Python interpreter which is used to run PRE Workbench. This way, full autocomplete support on internal objects is available.

5.5 INTERACTIVE DISSECTOR DEVELOPMENT

Interactive dissector development is powered by three core components. First, the textual representation of a dissector, in a user-friendly syntax named PGDL. This language is described in the first subsection.

This is has a directly corresponding UI element, the *Grammar Definition Code* tool window. Second, there is an internal tree representation of the dissector (internally called `FormatInfo`), which can be converted back-and-forth with the textual representation. And third, the parse result, an internal representation of parsed protocol data in a tree structure. This contains references to the `FormatInfo` structure, allowing the user to directly modify the dissector from the GUI elements in which the parse result is displayed. These are *HexView*, providing the *ClickGrammar* feature, and the *Parse Result* tool window, which are described in the second and third subsections.

5.5.1 *Protocol Grammar Definition Language*

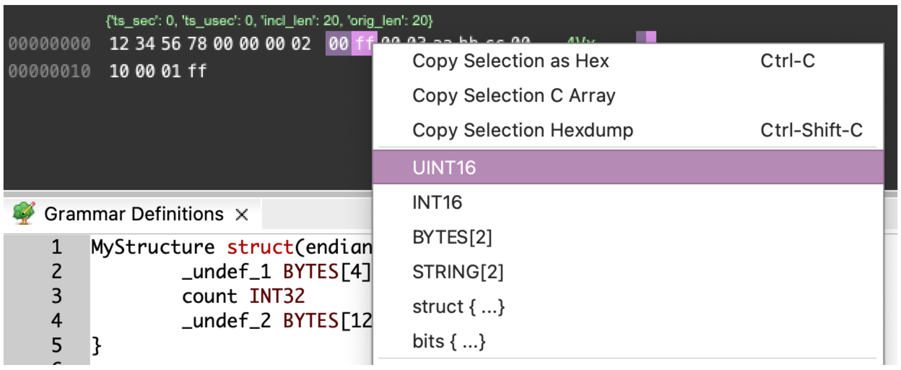
For designing the PGDL, we were led by the model protocols (Section 4.2) as well as some common protocols like the TCP/IP stack. We analyzed these protocols to find out which elements we need to describe these protocols' grammars. For example, many protocols consist of sequences of differently typed fields, so a basic struct-like element is required, which allows us to describe a fixed sequence of named fields of different types.

More complex protocols include repeated elements with variable repetition counts, requiring a loop-like element to describe the repetition of another type for either a number of times specified by some earlier header field, or until the parsed bytes end. Another common element is the Type-Length-Value (TLV) structure, where the *type* field defines with which definition the value needs to be parsed. This causes the need for a switch-case-like element. We also included a union element, which can be useful during the analysis phase, if the actual definition is still unclear and we want to see the parsing results of several definitions at the same time. Furthermore, a variant element is provided, which tries parsing the provided bytes with several different definitions, returning the results of the first successful parse.

For the actual contents of the above-mentioned data structures, we implemented the data types which are available in Wireshark dissectors, among them general data types like signed and unsigned integers, byte arrays, strings and bit fields, as well as more specialized types like IP addresses and UUIDs. The grammar definition language, including a full list of all supported data types, is described in detail in Appendix B.

5.5.2 *ClickGrammar*

The *ClickGrammar* feature is enabled by creating a new grammar definition from the *HexView* context menu. It starts with a `struct` containing only a single byte array, named `_undef_1`. The current buffer is then parsed using this definition. After that, a byte range can be selected, and the context menu will list all data types which can have the selected length (see Figure 17). By selecting a data type, and typing in a field name in the displayed dialogue box, a new field is inserted in the `struct`, while the `_undef_1` field is splitted and resized as needed to make room for the new field. All remaining undefined byte ranges can be assigned types in the same way.



```
{'ts_sec': 0, 'ts_usec': 0, 'incl_len': 20, 'orig_len': 20}
00000000  12 34 56 78 00 00 00 02  00 ff
00000010  10 00 01 ff
```

```
MyStructure struct(endian
        _undef_1 BYTES[4]
        count INT32
        _undef_2 BYTES[12
}
```

Figure 17: Screenshot of the ClickGrammar feature in use, displaying all possible types for a two byte range

5.5.3 *Grammar Parse Result Tree*

The *Parse Result* tool window (see Figure 18) displays the parse results associated with the currently selected byte buffers. Each field is displayed with its name, value, byte offset, and, if configured, highlight color. Selecting a field in the tree view highlights it in the *HexView*, and vice versa. The user can directly open a code editor to view and modify the definition of this field in PGDL syntax. They can also edit the visualization of fields, including the background and foreground color, section titles, and visibility in the tree view. If a parsing error occurs, it displays the error message in place of the field value.

5.5.4 *Automated Testing*

To verify that the implementation stays correct in the light of code refactorings or addition of new features, we used automated unit tests. In total, there are 34 unit tests verifying the correct function of the PGDL-based parser. Five of these check the correct parsing and generation of the PGDL syntax itself, another three the correct parsing of PCAPnG files based on test fixtures from [22]. The others verify the proper parsing of arbitrary data using PGDL descriptions. Tests were either written during the implementation of a new feature to immediately verify it, or as bugs were fixed to prevent regressions. We used the `pytest` framework [24] to run the tests and summarize the results.

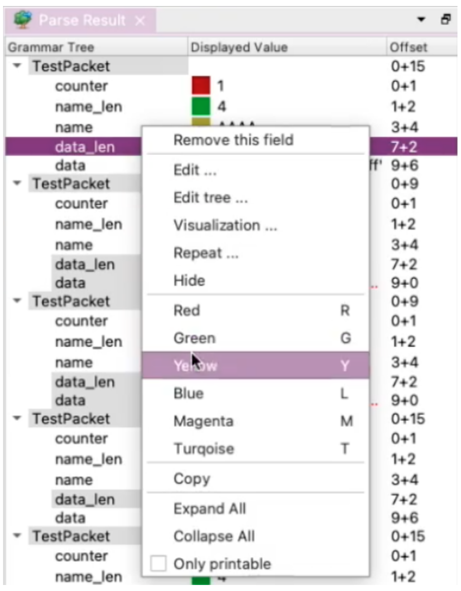


Figure 18: Screenshot of the Parse Result tool window

5.6 GENERATING WIRESHARK DISSECTORS

As a powerful way to further use the protocol grammars developed in PRE Workbench, we planned to generate Wireshark dissectors from them. In this thesis, we implemented a proof-of-concept code generator for Wireshark Lua dissectors, which supports a subset of the protocol grammar language. In this section, we first introduce some basic internal concepts of *EPAN*, Wireshark's packet parsing engine, and their usage from the Dissector API [9], before we explain the code generation process itself. We also discuss the current limitations of our proof of concept, and show an example.

5.6.1 *Wireshark Concepts*

When the results of a packet capture is displayed in Wireshark, it automatically runs appropriate dissectors on the captured data. Dissectors can register themselves in so-called dissector tables, announcing that they are able to parse certain protocols. The user can also manually choose which dissector to use.

Dissectors can either be written in C and in Lua. C dissectors can be integrated into the Wireshark codebase. Lua dissectors as well as additional C dissectors are loaded at runtime from the user's plugin directory.

For each packet it should parse, the dissector's parse function is called. It is passed a *Tvb* holding the payload to parse, a *TreeNode* into which the results should be stored, and a *PacketInfo* structure.

The Dissector API has been extended over many Wireshark versions and therefore contains some historical oddities and obsolete methods.

Especially the handling of the many supported data types for protocol fields is inconsistent, requiring some workarounds in the generated Lua code.

5.6.1.1 *Data Buffers*

Captured data is provided to the dissector in a data structure called a Testy, Virtual(-izable) Buffer of guint8*'s (Tvb). *Testy* is a joking reference to the fact that the buffer performs bounds checking[1]. *Virtual* means that the buffer does not need to hold its own backing data, but can reference a subset or a composition of other Tvbs' data. This allows us in the dissector code to create Tvbs of sub-ranges of our packet without much memory overhead.

5.6.1.2 *Packet Details Tree View*

The dissector writes its results into a data structure which directly corresponds to the representation in the *Packet Details* tree view in the Wireshark GUI.

In its entry point, it is provided with a *TreeNode* representing the root node of the tree view. Usually, it creates a single child node, representing the parse results of the current protocol layer, in which further children are created to represent the individual data fields. A TreeNode provides various methods to add child nodes, of which we use `add` and `add_packet_field` in the generated code.

Each data field occuring in the protocol should be registered as a *ProtoField* method, making them usable in packet filter expressions. In the parameters, we also provide details on how to parse the packet from the Tvb, and how to render it in the tree.

5.6.2 *Code Generation*

Dissector generation is implemented as a separate sub-package of PRE Workbench, which can be executed as a command-line utility as well as from the GUI. The Lua-specific parts of code generation are implemented as another sub-package, to prepare for implementation of C dissector generation in the future.

The code generator is provided with a `FormatInfoContainer` holding the actual grammar definitions, the name of the main definition from which dissection should start, and an optional list to filter which definitions from the container should be exported.

It generates some boilerplate code to initialize a new protocol and register it in the `DissectorTable`, and an entry function from which dissection starts and which creates the root `TreeNode`. It also generates

1 According to a code comment, "the buffer gets mad when an attempt is made to access data beyond the bounds of the buffer" [66].

the code that defines the `ProtoFields` from a list produced by the code which generates the actual parser functions.

5.6.2.1 *Building the Tree*

The actual parser code, which reads data from the `Tvb`, parses the binary data, and generates the `TreeNodes` from it, is generated from a visitor class. The visitor is initially called with all `FormatInfos` defined on the root level of the grammar file, and generates parser code for each of them. This parser code is then wrapped in a Lua function named after the grammar definition. The visitor then descends into the child types of the composite types. For example, the method for a `repeat` generates a loop statement, and calls the visitor on the child type to generate the code inside the loop.

Once the visitor reaches a `FieldFI` (FormatInfo for a pre-defined field type), it generates code to add a `TreeNode` to the subtree, store the parsed value into the `fval` (field value) table for future reference, and to check `magic` values if specified. It also stores information about the field definition into a list which is later used to register the `ProtoField` definitions on the protocol, allowing the user to use the fields in Wireshark's filter and column expressions.

5.6.2.2 *Expressions*

Expressions are handled by running a *transformer* on the abstract syntax tree of the expression. A transformer is essentially a visitor, but each function has a return value. The same pattern is used internally in PRE Workbench for evaluating expressions and for converting them to their string representations.

In this case, the transformer generates a corresponding Lua expression. For example, references to a field from the same struct or an ancestor are represented as the `anyfield_expr` in the AST, and as a plain identifier in the PRE Workbench expression syntax. In the Lua code, we store the values in a table named `fval`, therefore we transform an expression like `(length - 2)` to `fval['length'] - 2`. Transformations for more complex expressions, like bitwise operations or access to nested fields, are not implemented yet in the prototype. However, because Lua is a turing-complete language, there are no theoretical limitations preventing full expression compatibility.

5.6.2.3 *Registration*

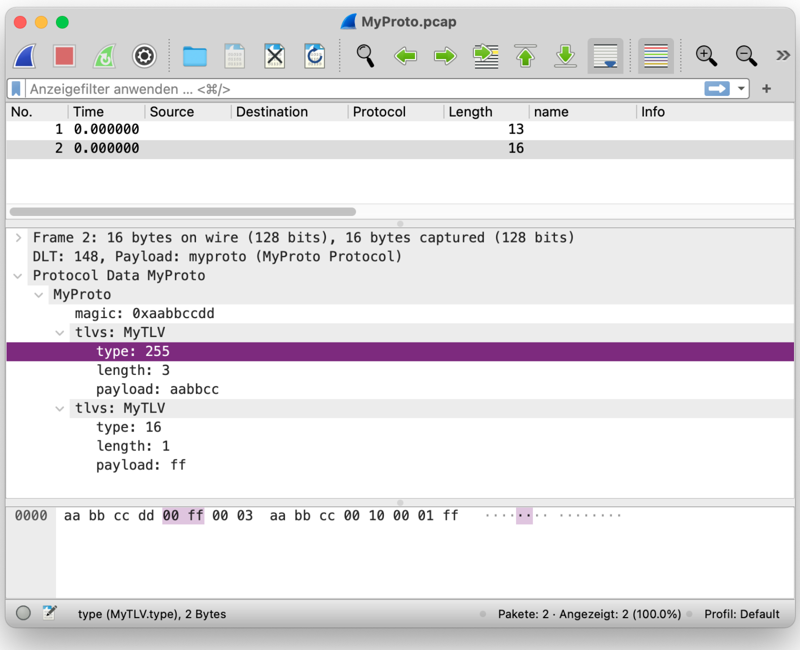
When calling the code generator, the user can also specify the dissector table in which the generated dissector should register itself, and the pattern to match (e.g. `udp.port` and the port number in the case of UDP-based protocol). Multiple sets of table and pattern can be provided in case the protocol is transmitted over several different ports.

If the protocol does not have an underlying layer like IP or UDP, the user can register the dissector in the special `wtap_encap` table, allowing them to parse raw bytes from a PCAP file.

5.6.3 *Limitations*

Our current code generator implementation is limited to a subset of possible PRE Workbench protocol grammars. Only `structs`, `repeats`, named references to other types and a subset of the built-in types are supported, other types like bitfield, variant, union and switch are not implemented in the prototype. In the expression syntax, only simple expressions consisting of references to fields in the same structure, as well as basic maths, are supported. Even this prototype implementation is already useful for users: for simple protocols, they can generate complete Wireshark dissectors, and for more complex ones, they can generate a stub to extend manually in the Lua code.

5.6.4 *Example*



```
No.   Time      Source   Destination   Protocol   Length   name   Info
    1 0.000000                                    13
    2 0.000000                                    16

> Frame 2: 16 bytes on wire (128 bits), 16 bytes captured (128 bits)
  DLT: 148, Payload: myproto (MyProto Protocol)
v Protocol Data MyProto
  v MyProto
      magic: 0xaabbccdd
    v tlvs: MyTLV
        type: 255
        length: 3
        payload: aabbcc
    v tlvs: MyTLV
        type: 16
        length: 1
        payload: ff

0000  aa bb cc dd 00 ff 00 03  aa bb cc 00 10 00 01 ff   ········ ········

type (MyTLV.type), 2 Bytes          Pakete: 2 · Angezeigt: 2 (100.0%)    Profil: Default
```

Figure 19: Wireshark parsing *MyProto* using the automatically generated dissector

To show the capabilites of the code generator, we use a dummy protocol called *MyProto*. It consists of packets, which start with a magic number, followed by any number of TLV sequences. It can be described by the grammar definitions in Listing 1. Generating a

```
MyProto struct(endianness=">") {
  magic UINT32(magic=0xAABBCCDD, show="hex")
  tlvs repeat MyTLV
}

MyTLV struct(endianness=">") {
  type UINT16
  length UINT16
  payload BYTES[length]
}
```

Listing 1: Definition of the *MyProto* sample protocol

Lua dissector using `prewb_codegen` from these definitions results in the Wireshark dissector shown in Section C.2, which can directly be stored into the Wireshark plugin directory. The screenshot in Figure 19 shows Wireshark using this dissector on a sample PCAP file containing packets of the MyProto protocol.

5.7 APPLICATION DEPLOYMENT

Publishing a Python GUI application in a way that it is easily installable by end-users on different platforms is a non-trivial task. For the first iteration, we only built a Python module to be published to the Python Package Index (PyPI). This allowed users to install the software using the `pip` command-line tool on their computers and worked on all machines PyQt supplies a binary distribution for (this includes x86_64 machines running macOS, Linux and Windows).

However, on M1 macOS computers, special workarounds[2] were required, as no M1 build of PyQt is available. To make the installation on M1 Macs more user-friendly, we packaged the application into a *macOS App Bundle* using PyInstaller, which is automatically executed using the Rosetta compatibility layer. We also provide a Windows distribution packaged using PyInstaller and InnoSetup.

The Python module distribution path is still used for installation on Linux systems, by users of the command-line utils distributed with *PRE Workbench*, and by users wanting to install additional Python packages to import them into custom macros or plugins.

2 Instructions from the manual at that time: "To run the application on the Rosetta compatibility layer, create a copy of Terminal.app (call it something like Terminal (Rosetta).app), click Get Info in its context menu, and check the Open using Rosetta checkbox under General. Afterwards, follow the instructions for Intel Macs."

6 EVALUATION

We evaluated *PRE Workbench* in two different ways: First, we conducted a user study with some of the researchers we interviewed in Section 4.1 and some other researchers. Second, we used the software itself to annotate some protocols, evaluating how well the software was suited for importing the traces and how well the description language was suited for describing the protocol data structure. Finally, during the development, we conducted some benchmarks for optimizing the parsing speed and GUI responsiveness, which we also present in this chapter.

6.1 USER STUDY

In the first phase of the user study, we conducted user interviews to establish goals and guide the design of our software. This phase is described in Section 4.1.

In the second phase, we evaluated the basic usability of our GUI concept. We did this in an early phase of development, with a very simplified proof of concept, and used the results to influence design decisions.

In the third phase, we gave working increments of the software to beta testers. The results influenced the implementation, in terms of prioritizing features, and optimizing the user experience.

In the fourth and final phase, we conducted a usability test on the final version of the software. We documented the findings, and fixed some remaining bugs which surfaced during the test.

In all phases, we solicited feedback in multiple ways: First, by noting down questions testers asked during use, second, by observing them during use (via screen share or in person), third, by using the automated error reporting implemented after the first user tests, and finally, by directly asking for feedback.

6.1.1 *Results Second Phase*

One main result was that a tab- and dock-panel-based GUI is better accepted than the classic Multiple-Document Interface (MDI) with freely movable child windows. The freely movable windows often led to confusion, e.g. when they were moved out of the visible area. Therefore, we decided to hard-code the GUI concept to dock-panel, and not to leave it adjustable between dock-panel and classic MDI as before.

This simplifies the development and was considered appropriate, as the classic MDI was not considered user friendly.

In addition, it turned out that pre-compiled application packages for common operating systems should be made available, since the installation via the Python package manager *pip* led to problems for some users.

Moreover, by observing the users, we found numerous possibilities for detail improvements of the usability. For example, users requested that more elements of application state should be persistent over restarts, like column layouts of lists, or last used folder for file dialogs. Also, some context menus proved to be not discoverable enough, therefore we decided to add more menu and toolbar items. We have also decided to integrate automatic error reporting for future user studies, so that unhandled exceptions and application crashes can be reported to us automatically.

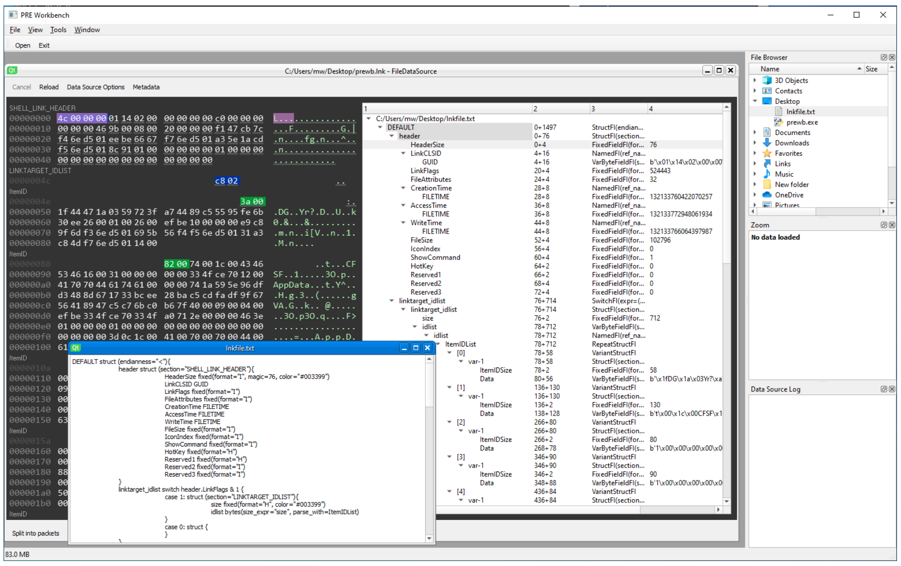
6.1.2 *Results Third Phase*

We had two beta testers who directly used the software in a reverse engineering project they were currently working on. One of them was the researcher from the user interview in Section 4.1.4. The other was a student working on a practical course at SEEMOO.

Both users had difficulties with basic usability flaws, but were able to use the software successfully after some guidance by the author. Most problems were features that were not discoverable in the User Interface (UI), e.g. features only accessible using shortcut keys, which were documented in the manual but had no visible button or menu entry in the UI. Furthermore, error messages resulting from incorrect use were often not descriptive enough for the user to find out what they did wrong. We noted these obstacles and used them as a basis for further improvements to the user experience. We also added *Help* buttons throughout the application to make the existing documentation more accessible.

Other problems were simple bugs, where some feature just did not work in certain circumstances or with certain types of input data. These were fixed in further increments of the application.

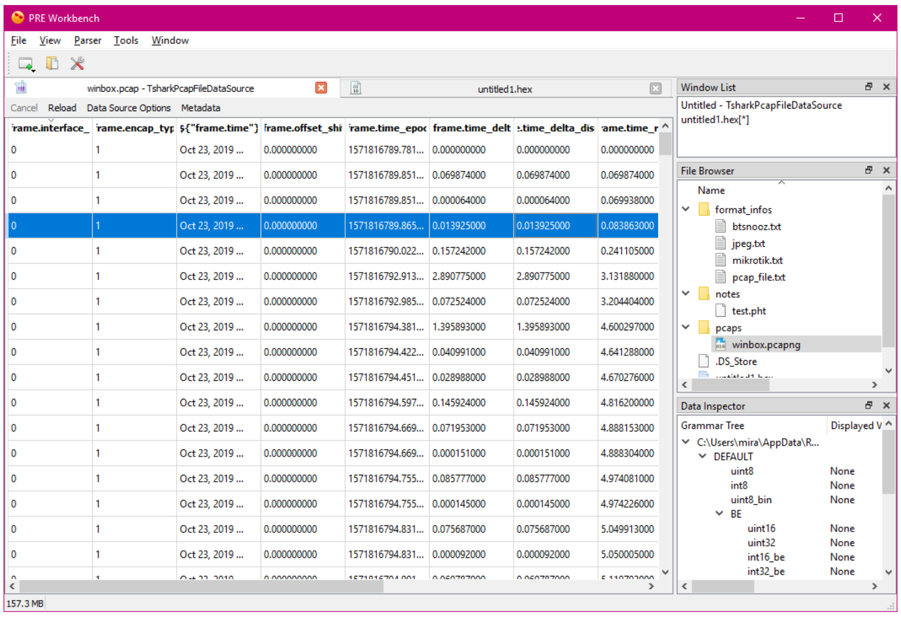
Finally, there were missing features for which the users had to use workarounds, like having to convert input data into a format supported by the application first, or using external tools to fully parse a protocol structure. We implemented some of these features, especially more supported input formats, but have to acknowledge that there are too many possible data formats out there to support all of them out of the box. Therefore, we additionally made the application more extensible, allowing users to program their own data sources for the application as Python-based macros. We also decided to make the definition language more powerful, so users need to refer to external



```
SHELL_LINK_HEADER
00000000  4c 00 00 00 01 14 02 00  00 00 00 00 c0 00 00 00  L...............
00000010  00 00 00 46 9b 00 08 00  20 00 00 00 f1 47 cb 7c  ...F........G.|
00000020  f4 6e d5 01 ee be 66 67  f7 6e d5 01 a3 5e 1a cd  .n....fg.n...^..
00000030  f5 6e d5 01 8c 91 01 00  00 00 00 00 01 00 00 00  .n..............
00000040  00 00 00 00 00 00 00 00  00 00 00 00              ............
LINKTARGET_IDLIST
0000004c                                       c8 02                      ..
ItemID
0000004e                                             3a 00                :.
00000050  1f 44 47 1a 03 59 72 3f  a7 44 89 c5 55 95 fe 6b  .DG..Yr?.D..U..k
00000060  30 ee 26 00 01 00 26 00  ef be 10 00 00 00 e9 c8  0.&...&.........
00000070  9f 6d f3 6e d5 01 69 5b  56 f4 f5 6e d5 01 31 a3  .m.n..i[V..n..1.
00000080  c8 4d f7 6e d5 01 14 00                           .M.n....
ItemID
00000088                           82 00 74 00 1c 00 43 46          ..t...CF
00000090  53 46 16 00 31 00 00 00  00 00 33 4f ce 70 12 00  SF..1.....3O.p..
000000a0  41 70 70 44 61 74 61 00  00 00 74 1a 59 5e 96 df  AppData...t.Y^..
000000b0  d3 48 8d 67 17 33 bc ee  28 ba c5 cd fa df 9f 67  .H.g.3..(......g
000000c0  56 41 89 47 c5 c7 6b c0  b6 7f 40 00 09 00 04 00  VA.G..k.. @.....
000000d0  ef be 33 4f ce 70 33 4f  a0 71 2e 00 00 00 46 3e  ..3O.p3O.q....F>
000000e0  01 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  ................
000000f0  00 00 00 00 3d 0c 1c 00  41 00 70 00 70 00 44 00  ....=...A.p.p.D.
```

```
DEFAULT struct (endianness="<"){
        header struct (section="SHELL_LINK_HEADER"){
                HeaderSize fixed(format="I", magic=76, color="#003399")
                LinkCLSID GUID
                LinkFlags fixed(format="I")
                FileAttributes fixed(format="I")
                CreationTime FILETIME
                AccessTime FILETIME
                WriteTime FILETIME
                FileSize fixed(format="I")
                IconIndex fixed(format="I")
                ShowCommand fixed(format="I")
                HotKey fixed(format="H")
                Reserved1 fixed(format="H")
                Reserved2 fixed(format="I")
                Reserved3 fixed(format="I")
        }
        linktarget_idlist switch header.LinkFlags & 1 {
                case 1: struct (section="LINKTARGET_IDLIST"){
                        size fixed(format="H", color="#003399")
                        idlist bytes(size_expr="size", parse_with=ItemIDList)
                }
                case 0: struct {
                }
```

(a) Classic MDI



frame.interface_	frame.encap_typ	${"frame.time"}	frame.offset_shi	frame.time_epoc	frame.time_delt	.time_delta_dis	ame.time_r
0	1	Oct 23, 2019 ...	0.000000000	1571816789.781...	0.000000000	0.000000000	0.000000000
0	1	Oct 23, 2019 ...	0.000000000	1571816789.851...	0.069874000	0.069874000	0.069874000
0	1	Oct 23, 2019 ...	0.000000000	1571816789.851...	0.000064000	0.000064000	0.069938000
0	1	Oct 23, 2019 ...	0.000000000	1571816789.865...	0.013925000	0.013925000	0.083863000
0	1	Oct 23, 2019 ...	0.000000000	1571816790.022...	0.157242000	0.157242000	0.241105000
0	1	Oct 23, 2019 ...	0.000000000	1571816792.913...	2.890775000	2.890775000	3.131880000
0	1	Oct 23, 2019 ...	0.000000000	1571816792.985...	0.072524000	0.072524000	3.204404000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.381...	1.395893000	1.395893000	4.600297000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.422...	0.040991000	0.040991000	4.641288000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.451...	0.028988000	0.028988000	4.670276000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.597...	0.145924000	0.145924000	4.816200000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.669...	0.071953000	0.071953000	4.888153000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.669...	0.000151000	0.000151000	4.888304000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.755...	0.085777000	0.085777000	4.974081000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.755...	0.000145000	0.000145000	4.974226000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.831...	0.075687000	0.075687000	5.049913000
0	1	Oct 23, 2019 ...	0.000000000	1571816794.831...	0.000092000	0.000092000	5.050005000

(b) Tabbed interface

Figure 20: Early proof-of-concept implementations for GUI feedback

tools in fewer cases. To this end, we added a new syntax element for bit fields, as well as a new data type to display calculated values in the parse tree.

In general, users were able to load the data they recorded using other tools into PRE Workbench, they were able to use the annotation features of the HexView, and could also describe the basic syntax of their binary protocols using the definition language.

6.1.3 *Results Fourth Phase*

After we had largely completed the implementation, we performed final user tests to confirm usability and applicability in practice. To do this, we asked four people to install the newest version of PRE Workbench and apply it to a real-world protocol that they had analyzed in the past. Three of them were also part of earlier phases (interviewees one, three and four from Section 4.1). We observed and guided the participants during the tests via screen sharing over BigBlueButton.

For all participants in this phase, we noted the used operating system, the protocol they test the application on, and their previous experience. For the user test, we went through the same basic tasks with everyone:

- Loading a trace of their chosen protocol from a file, splitting into packets if necessary.
- Viewing some packets in the HexView and annotating them using colors and section headers, and applying the annotation set to multiple packets.
- Creating a (simplified) grammar definition for their protocol, using the ClickGrammar feature and/or manual input.
- Parsing multiple packets using the definition.
- Adding fields from the definition to the packet list as columns.

All users were able to perform these tasks on their protocols. However, in all cases some guidance was needed. This is due to the fact that we wanted to complete the tests within an hour and therefore gave hints instead of letting the users find information in the documentation.

For loading trace data, performance varied based on the format of the traces. PCAP files were very easy to load. For CSV files, a user noted that the import dialog could be more user friendly, e.g. marking which fields are mandatory. Most difficult was a binary file containing multiple packets, separated by protocol-specific start and end marker sequences. For this file a custom data source macro was necessary.

In the annotation step, users appreciated the selection heuristics feature, especially detection of length fields and repeated bytes, as well as the binary (bitwise) view. However, they missed the possibility to select individual bits.

When creating grammar definitions, the *ClickGrammar* feature was praised as simplifying the workflow by eliminating the need to manually calculate field offset and length.

Feedback for the overall UI was positive, the Integrated Development Environment (IDE) style dock panel interface was well received. The reasons given were the well-organized display of many files open at the same time due to the tab interface, as well as the high flexibility due to the docking functionality. Criticism included the not very expressive icons, the still missing keyboard usability in some places, as well as the fact that some buttons are not clearly recognizable as such. It was noted that the software requires some training, but this was considered reasonable for a tool of this feature set.

6.1.4 *Overall Results*

The continued user tests over the development cycle showed a clear progression in usability and feature set. They also allowed to prioritize our development efforts based on user's needs.

The overall UI paradigm evolved towards the IDE style dock panel interface, providing a feature-rich, customizable interface for expert users. The *ClickGrammar* feature (Section 4.3.4) we implemented for the fourth phase of user testing provided a leap in the usability of the software, allowing users to define fields using a point-and-click approach, neither needing to know field types by heart nor calculating offsets and length manually.

The software proved to be very stable in the sense that it rarely crashed in a way leading to data loss. This is mainly due to the fact that with Python we use a memory-safe programming language and with Qt a very mature UI framework.

In summary, there remain many more features that could be implemented in a future version of the application, but testing has shown that it can be used well for its intended purpose.

6.2 USABILITY ON KNOWN PROTOCOLS

We evaluate the application by using it on some protocols where the structure is already documented, either because they are open protocols, or because we or other researchers reverse engineered them in the past. We focused on the ability to import and parse traces of these protocols. In this section, we describe the difficulties of parsing these protocols and how we used the Protocol Grammar Description Language (PGDL) and custom macros to parse them.

6.2.1 *Internet Protocol Stack*

First, we evaluated PRE Workbench on some base protocols of the Internet protocol stack, namely the Ethernet MAC [1], IPv4 [26] and TCP [65] headers.

DATA IMPORT Internet protocol dumps are commonly stored in the PCAP file format generated by tcpdump and Wireshark. We used a PGDL definition (Section C.4) to build our internal parser for importing PCAP and PCAPnG files. These file formats contain complex elements like endianness determined by a byte order mark, multiple block types and padding on 32-bit boundaries. All these elements can also be expressed in PGDL definitions.

PARSING We specified the syntax of some common protocols of the Internet protocol stack using PGDL to show that it is powerful enough for most applications. For example, the grammar definition of Ethernet, IPv4 and TCP headers including variable-length option fields and integer fields not aligned to byte boundaries can be expressed fully in PGDL as shown in Section C.3.

6.2.2 *Bluetooth Smart Lock*

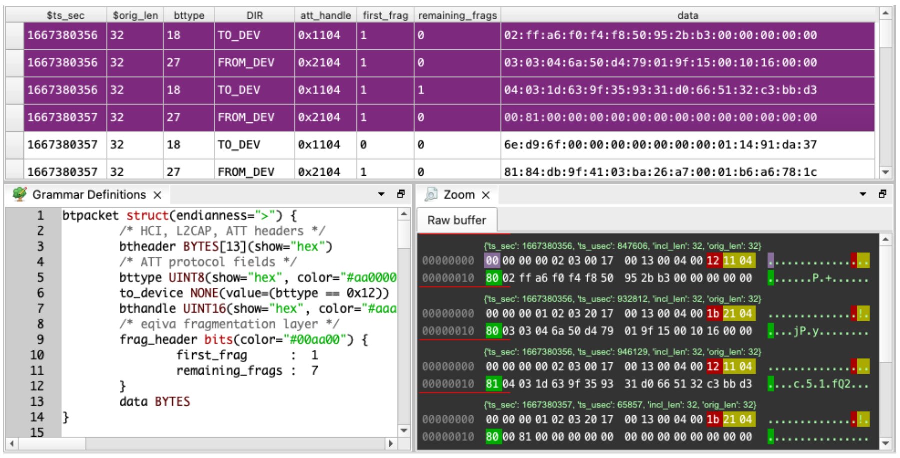
Second, we evaluated the software on the proprietary binary protocol of a Bluetooth Smart Lock developed by a German vendor [67]. The architecture and protocols of this IoT system are summarized in Section 4.2.2. We focused on the connection between the Android app and the IoT device via Bluetooth Low Energy.

We did not capture data on the other channels: The app does not have a relevant Internet connection. The serial connection on the device has the same data as Bluetooth.

DATA IMPORT Using the *Bluetooth snoop log* feature on Android, we sniffed the connection between phone and device. We downloaded the data from the Android device via Android Debug Bridge (ADB). Then, we used Wireshark to filter and convert it to the PCAP file format, which we could then to import into PRE Workbench.

PARSING To parse the multi-layered protocol, we first specified a minimal PGDL description for the Bluetooth packet, including Bluetooth lower layers HCI, L2CAP and ATT, as well as the proprietary fragmentation layer. The description as well as a parse result is shown in Figure 21.

To reassemble the fragmented packets, we needed to implement a custom macro iterating over the parsed ByteBufferList. The macro also decrypts the encrypted packets. The decryption key, extracted from

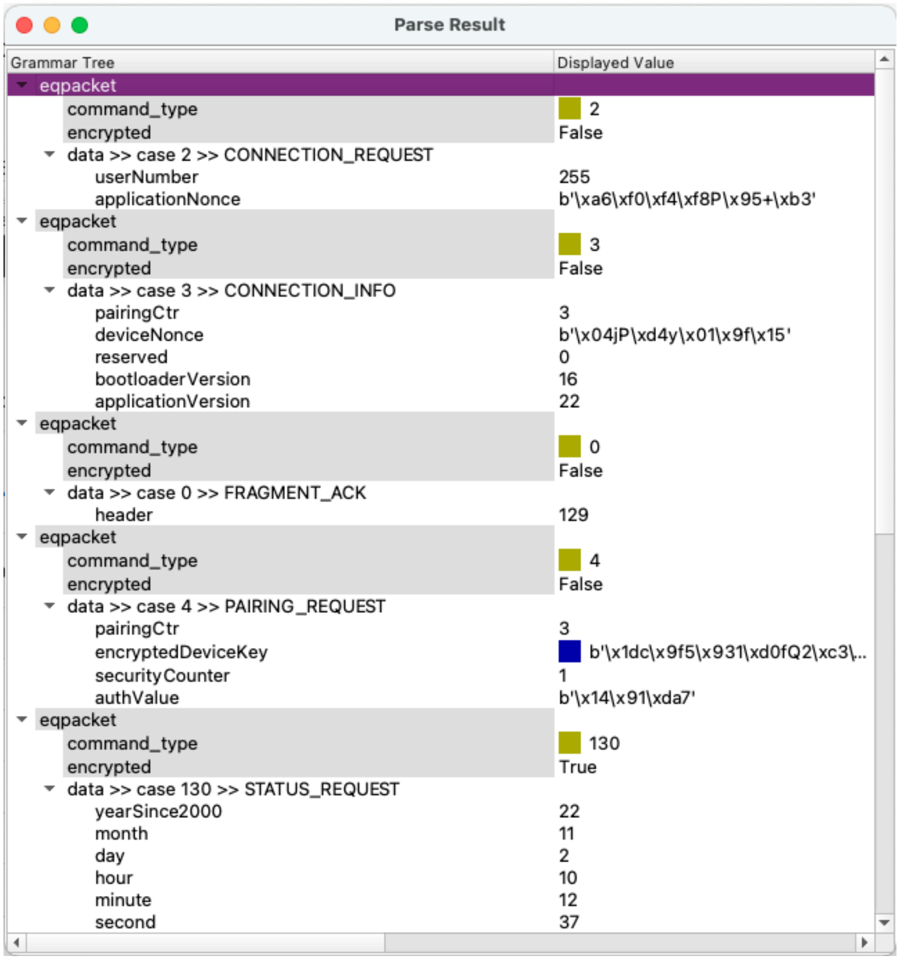


$ts_sec	$orig_len	bttype	DIR	att_handle	first_frag	remaining_frags	data
1667380356	32	18	TO_DEV	0x1104	1	0	02:ff:a6:f0:f4:f8:50:95:2b:b3:00:00:00:00:00
1667380356	32	27	FROM_DEV	0x2104	1	0	03:03:04:6a:50:d4:79:01:9f:15:00:10:16:00:00
1667380356	32	18	TO_DEV	0x1104	1	1	04:03:1d:63:9f:35:93:31:d0:66:51:32:c3:bb:d3
1667380357	32	27	FROM_DEV	0x2104	1	0	00:81:00:00:00:00:00:00:00:00:00:00:00:00:00
1667380357	32	18	TO_DEV	0x1104	0	0	6e:d9:6f:00:00:00:00:00:00:00:01:14:91:da:37
1667380357	32	27	FROM_DEV	0x2104	1	0	81:84:db:9f:41:03:ba:26:a7:00:01:b6:a6:78:1c

```
btpacket struct(endianness=">") {
        /* HCI, L2CAP, ATT headers */
        btheader BYTES[13](show="hex")
        /* ATT protocol fields */
        bttype UINT8(show="hex", color="#aa0000
        to_device NONE(value=(bttype == 0x12))
        bthandle UINT16(show="hex", color="#aaa
        /* eqiva fragmentation layer */
        frag_header bits(color="#00aa00") {
                first_frag      :  1
                remaining_frags :  7
        }
        data BYTES
}
```

```
           {'ts_sec': 1667380356, 'ts_usec': 847606, 'incl_len': 32, 'orig_len': 32}
00000000   00 00 00 00 02 03 00 17  00 13 00 04 00 12 11 04   ................
00000010   80 02 ff a6 f0 f4 f8 50  95 2b b3 00 00 00 00 00   .......P.+......
           {'ts_sec': 1667380356, 'ts_usec': 932812, 'incl_len': 32, 'orig_len': 32}
00000000   00 00 00 01 02 03 20 17  00 13 00 04 00 1b 21 04   .............!.
00000010   80 03 03 04 6a 50 d4 79  01 9f 15 00 10 16 00 00   ....jP.y........
           {'ts_sec': 1667380356, 'ts_usec': 946129, 'incl_len': 32, 'orig_len': 32}
00000000   00 00 00 00 02 03 00 17  00 13 00 04 00 12 11 04   ................
00000010   81 04 03 1d 63 9f 35 93  31 d0 66 51 32 c3 bb d3   ....c.5.1.fQ2...
           {'ts_sec': 1667380357, 'ts_usec': 65857, 'incl_len': 32, 'orig_len': 32}
00000000   00 00 00 01 02 03 20 17  00 13 00 04 00 1b 21 04   .............!.
00000010   80 00 81 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
```

Figure 21: Smart Lock raw Bluetooth packets



```
Grammar Tree                                   Displayed Value
▼ eqpacket
    command_type                               ▨ 2
    encrypted                                  False
  ▼ data >> case 2 >> CONNECTION_REQUEST
      userNumber                               255
      applicationNonce                         b'\xa6\xf0\xf4\xf8P\x95+\xb3'
▼ eqpacket
    command_type                               ▨ 3
    encrypted                                  False
  ▼ data >> case 3 >> CONNECTION_INFO
      pairingCtr                               3
      deviceNonce                              b'\x04jP\xd4y\x01\x9f\x15'
      reserved                                 0
      bootloaderVersion                        16
      applicationVersion                       22
▼ eqpacket
    command_type                               ▨ 0
    encrypted                                  False
  ▼ data >> case 0 >> FRAGMENT_ACK
      header                                   129
▼ eqpacket
    command_type                               ▨ 4
    encrypted                                  False
  ▼ data >> case 4 >> PAIRING_REQUEST
      pairingCtr                               3
      encryptedDeviceKey                       ■ b'\x1dc\x9f5\x931\xd0fQ2\xc3\...
      securityCounter                          1
      authValue                                b'\x14\x91\xda7'
▼ eqpacket
    command_type                               ▨ 130
    encrypted                                  True
  ▼ data >> case 130 >> STATUS_REQUEST
      yearSince2000                            22
      month                                    11
      day                                      2
      hour                                     10
      minute                                   12
      second                                   37
```

Figure 22: Smart Lock reassembled, decrypted and parsed application packets

a QR code packaged with the device, is given as an input parameter to the macro. We were able to reuse the decryption code for the custom AES-based encryption scheme from our open-source Python implementation of the protocol, *pysmartlock* [46]. Finally, the macro creates a new ByteBufferList with the resulting reassembled, decrypted packets. These are in turn parsed using another PGDL description. The results of this process are visualized in Figure 22.

The application protocol has a rather simple structure. Our PGDL description consists of a main `switch` type, which chooses depending on a type field from about 30 packet types. Some of the packet types contain bit-wise fields, which could be succinctly described using the `bits` type.

6.2.3 *Apple Remote Invocation*

Finally, we used the Apple Remote Invocation (ARI) protocol for evaluation, reverse engineered by Kröll et al. [29]. We created grammar definitions for the packet and TLV header structures of the ARI protocol. We used these definitions to parse the sample files from the *ARIstoteles* Git repository [51].

DATA IMPORT We imported data from the sample files in the *ARIstoteles* Git repository. This caused no problems because the sample files were in two common formats: First, PCAPnG files containing directly the ARI data, without any lower layer headers. Second, raw binary files containing a single ARI packet each. In *PRE Workbench*, these can be imported either individually using the *Binary File Data Source* or as a packet list using *Directory Of Binary Files Data Source*.

PARSING We parsed the headers of ARI packet and ARI TLVs.

Here, the tricky part was a convoluted header structure containing integers not aligned to byte boundaries, in little-endian encoding. However, the `bits` type as described in Section B.1.8 is able to describe this structure.

6.3 DISSECTION SPEED

We expect our application to be used mostly with small files of up to a few hundred packets. This assumption is based on the fact that our application is intended to manually annotate binary packet data, which a typical researcher will not do on more than a few hundred packets. Occasionally, users might want to annotate or parse big trace files, in which case performance of loading and parsing files matters.

Therefore we evaluated the speed of PRE Workbench's grammar-based parser in Graphical User Interface (GUI) and Command-Line Interface (CLI) mode, and compared it to a Wireshark Lua dissector

-	300 packets	3000 packets	30000 packets
PRE Workbench GUI	0.14 sec	1.25 sec	12.6 sec
PRE Workbench CLI	0.35 sec	0.91 sec	6.0 sec
Tshark CLI	0.34 sec	0.67 sec	4.6 sec

Table 2: Dissection speed evaluation

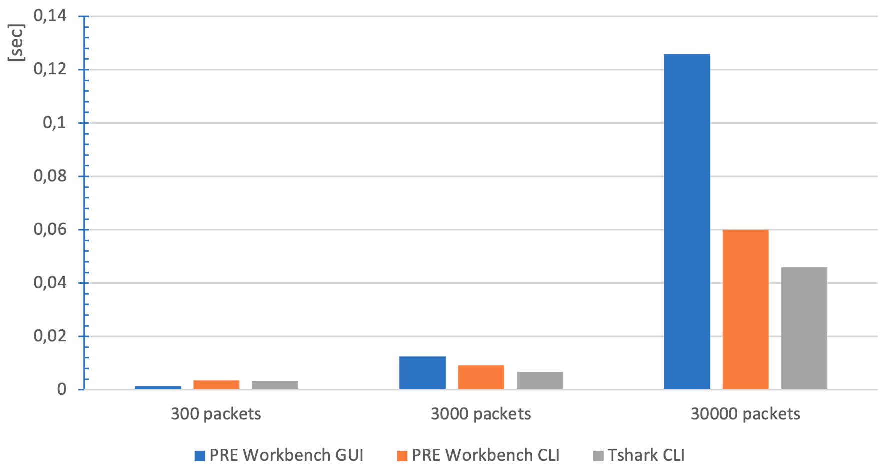


Figure 23: Dissection speed evaluation

generated by PRE Workbench. We used an example protocol similar to the one described in Section 5.6.4, and generated test files with three different packets repeated 100, 1000 and 10000 times. One of them violates the protocol definition to also include exception handling in the evaluation.

The evaluation was performed by manually running the respective tools on the author's computer, a MacBook Pro with a 2.6 GHz Intel Core i7 processor. The command line tools were timed using the `time` command line utility, the PRE Workbench GUI displays the elapsed time for loading data in the status bar. Each combination of tool and input file was repeated three times. Only the fastest result of each case was noted, because "in a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values [...] are typically not caused by variability in [...] speed, but by other processes interfering with your timing accuracy." [64]

The results are presented in Table 2. We confirmed that for small traces of a few hundred packets, the performance differences are irrelevant. However, very large files of tens of thousands of packets take a long time to load in the GUI, because `Range` objects are created for each field, annotating individual bytes in the buffer. This is not performed in the CLI, where only the actual parse result objects are generated and printed as JSON.

More research showed that approximately two seconds of the load time for the 30000 packets file are spent parsing the PCAP file. Our PCAP parser is currently very inefficient, it uses the protocol parser internally to parse the PCAP file format. This can easily be made more efficient by switching to an external PCAP reader library. We verified this by replacing the PCAP parser in the PRE Workbench CLI with the `dpkt` library [13], which reduced the PCAP parsing time from two seconds to 0.05 seconds.

The Wireshark GUI was excluded from performance tests, because it is difficult to measure load times exactly there. However, it feels fast even for large files, because packet parsing is performed on-the-fly. When features are used that require the whole file to be parsed, like filtering by an expression, times are roughly similar to Tshark CLI times. A similar feature could be implemented in PRE Workbench in a future version, if the need for handling such files should arise more often, either by parsing packets in a background thread, or by only parsing displayed rows.

As a conclusion, the parser itself is similar in speed to Wireshark Lua dissectors, however, our GUI is slow when loading large files, and our PCAP parser can be replaced by a faster library. For small files up to a few hundred packets, which we assume are more common in our use case, all components are sufficiently fast.

7 DISCUSSION

In this chapter, we discuss the relationship of PRE Workbench to other tools in the field of protocol reverse engineering, as well as possible future work on the software.

7.1 RELATIONSHIP TO OTHER TOOLS

To clarify our contribution, we would like to distinguish our software from other programs in the following.

WIRESHARK At the first glance, PRE Workbench is very similar to Wireshark, both can load a packet trace, display it as a list and hex dump and dissect packets.

The difference lies in the nature of the dissectors. In Wireshark, packet traces are the data it operates on, whereas dissectors are a fairly static component of the software itself, mostly present in the form of compiled C code. In PRE Workbench, dissectors are data which the user works on in the application. They can be modified on-the-fly, and exported into a Wireshark compatible format.

We also considered implementing our application as a plugin to Wireshark, however, we decided against that: Wireshark does not provide an API for plugins with complex GUI features [71], therefore we would have needed to work directly on the Wireshark core code, which is implemented in the C programming language. We wanted to quickly iterate on the GUI, which is a lot more efficient in Python.

CANAPE The CANAPE application also has similarities to PRE Workbench. Apart from the fact that CANAPE is out of support and only works on Microsoft Windows, PRE Workbench provides additional features specifically aimed at dissector development:

- Point-and-click creation of dissectors directly from the hex dump view (ClickGrammar)
- Live update of parse results
- Generation of Wireshark dissector code
- Support for more input file types, e.g. PCAPnG

7.2 FUTURE WORK

We identified Future Work in three categories: improvement of existing implementation, new features, and evaluation.

UX IMPROVEMENTS We were able to identify the following major potentials for improvement of the user experience:

- Extend point-and-click grammar generation, by supporting some options like color, endianness and display style directly in the name entry dialog, and support creation of complex types (e.g. repeat) in the tree view.
- Support hotkeys in more places, especially for grammar creation.
- Clean up and improve the menu system. Currently, many actions are only available through context menus, but they would be better discoverable if also visible in the main menu and in tool bars.
- Customize dialogs. Currently, many dialogs are programmatically generated, but some would profit from a more specialized dialog or combining multiple dialogs into one (e.g. the macro editor, which is currently spread over three different dialogs).
- Expressions for custom columns in PacketList would benefit from an easier-to-use expression syntax and an autocomplete feature in the editor.

PGDL IMPROVEMENTS The Protocol Grammar Description Language (PGDL) can mainly be improved in the comment syntax, and its error handling capabilities.

Currently, comments are only supported in specific syntactic places, e.g. before structure fields. Usability of the language would increase if comments were allowed in all places where common programming languages allow them.

The error messages for grammar and data parsing errors are currently quite verbose, but and can be confusing to users. This could be improved by handling more error cases explicitly, instead of relaying the internal Python exceptions to the user. Also, in the current implementation, many errors arise only at data parse time, but could be already detected at grammar parse time, e.g. invalid function names in expressions.

COMPLETING LUA GENERATION The Lua generator is currently a proof-of-concept, which does not implement all types and expression operators supported by PGDL. The necessary work to complete this is described in Section 5.6.3.

7.2.1 *New Features*

The Wireshark dissector generator is developed with the premise of being extensible to other target languages. As C dissectors are more performant in runtime, it might be promising to implement C code generation.

Several new features to support the reverse engineer during static analysis could be implemented: More heuristic detection capabilities could be implemented, either as part of the heuristic highlighter, or as a macro working on packet lists, comparing them for similarities. In a similar approach, external tools for automatic Protocol Reverse Engineering (PRE) could be integrated, for example, to be applied to a packet list and search for field boundaries. Also, according to interviewed researchers, a binary diff feature displaying the differences between two or more packets would be a useful addition.

Another new avenue to make PGDL descriptions more useful would be to generate fuzzing inputs based on protocol grammars. The field values could either specified directly, e.g. from a JSON file, or by placing data generation directives in the PGDL syntax, e.g. a field could be annotated `UINT16(generate=(rand(0,5000)))` to generate random values between 0 and 5000 in this place, or `UINT16(generate=(len(payload)))` to calculate a correct value from another field in the packet. Another possibility is to make field values in the parse result tree editable, so that packets could be modified directly in the GUI.

7.2.2 *Evaluation*

The software can be further evaluated as more researchers use it on more protocols.

First, the forth user study phase could be continued with additional subjects. Furthermore, after the application gains more users, these users could be asked for feedback. More participants in the user study or feedback round would allow usage of standardized questionnaires.

Second, bug and crash reports users submit through the app can be evaluated continuously. Users of the application are expected to be technically knowledgeable and therefore able to provide usable bug reports or even patches.

And third, it would be possible to introduce a telemetry feature which reports usage data, e.g. which features are used most. However, much consideration would have to be given to protecting the privacy and intellectual property of users.

8 CONCLUSIONS

In this thesis, we studied the workflow of researchers performing manual protocol reverse engineering of binary protocols. We found a need for a software to support this task. Therefore, we developed *Protocol Reverse Engineering Workbench*, a software to support researchers in analyzing and documenting protocols, as well as in verifying their findings.

As part of this, we created a desktop application to import, annotate and parse protocol data. We designed the Protocol Grammar Description Language (PGDL), a declarative language to describe binary protocols. We implemented a parser that parses packets according to a PGDL description, as well as a code generator to generate Wireshark dissectors from a PGDL source.

Our goals are to design a software that is user-friendly for expert users, extensible enough to handle unexpected input data, and allows users to iterate quickly on protocol structures by providing fast turn-around times. To reach these goals, we based our design and implementation process on interviews and user studies conducted with researchers and students who are potential users of the software. We evaluated our success using a final stage of user studies, accompanied by automated unit tests and performance evaluation of the parser. We also manually tested the application's import features and the description language on several toy and real-world protocols.

User evaluation showed that the application can be used for the dissection of protocols the researchers last worked on, and that its IDE-style user interface is sufficiently flexible while still intuitive. It also worked well in our own manual tests.

A fundamental limitation of the PGDL lies in its declarative nature, making it impossible to handle certain complex protocol features directly in the grammar, e.g. fragmentation or encryption. This is mitigated by extending the application using macros.

In conclusion, we developed a software that proved to be a useful addition to a protocol reverse engineers toolbox, simplifying the discovery and documentation of a binary protocol's structure. The software is published under an open-source license [69].

A USER STUDIES

In this appendix we provide the questions asked and the mock-ups shown to participants during the first phase of our user studies. As the interviews were conducted in German, the questions are provided as-is in German language. Responses and conclusions from the interviews can be found in Section 4.1.

A.1 INTERVIEW QUESTIONS (GERMAN)

Nutzerinterviews zur Untersuchung der Anforderungen an eine Software, die Forschende beim Reverse Engineering von Binärprotokollen unterstützt, indem in einem GUI Protokollmitschnitte annotiert und daraus Dissectoren generiert werden können.

A.1.1 *Rahmenbedingungen deiner Arbeit*

- In welchem Kontext analysierst du Protokolle? (Thesis, Pentest, ...)
- Was sind Ziele und Zwischenziele der Untersuchung? (Finden von Sicherheitslücken, Entwickeln kompatibler Software, Auslesen von Daten, schriftliche Dokumentation, Entwickeln eines Wireshark-Dissector, ...)
- Was sind charakteristische Eigenschaften der untersuchten Protokolle?
  - Binary/Textbasiert
  - Verschlüsselung
  - Obfuscation
  - ...
- Wie ist dein Workflow? Wie gehst du vor?
- Welche Tools nutzt du dabei aktuell?

A.1.2 *Technische Rahmenbedingungen*

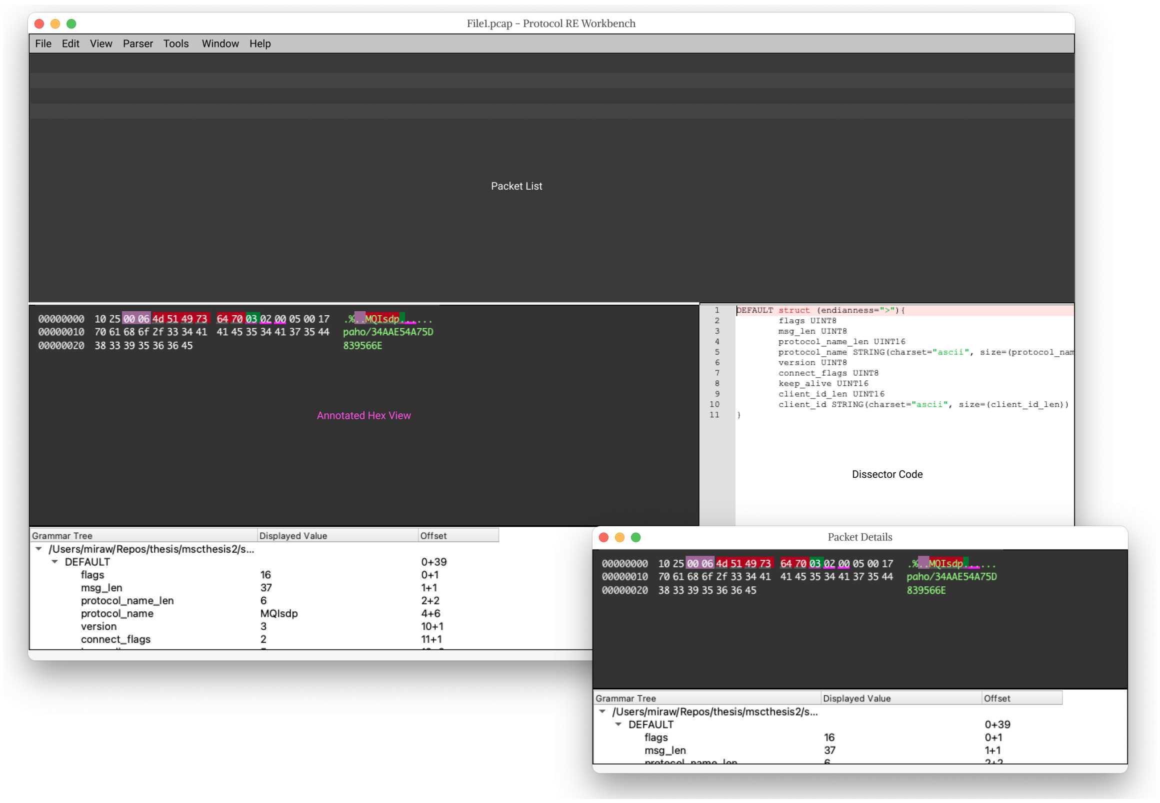
- White box / Black box - Hast du Zugang zum Source Code / Binaries einer oder beider Kommunikationsseiten?
- Auf welchen Plattformen laufen diese (z.B. PC, Mobile, Embedded, eigene/fremde Cloud), hast du dazu Zugang (z.B. auf

Applikationsebene, Debugger, System/Root-Zugang, Encryption keys)?

- Wo werden Protokollmitschnitte angefertigt (auf dem untersuchten Device, Sniffer dazwischen) und mit welchen Tools? In welchen Formaten liegen sie dann vor?

A.1.3 *Annotation Tool*

- Was fehlt dir aktuell, was ist unnötig umständlich? Was wäre nützlich?
- Würde dir eine Software helfen, in der du Protokollmitschnitte interaktiv annotieren, und Dissektoren entwickeln kannst?
- Wie sollte die Benutzerführung aufgebaut sein?
  - Eigenes GUI/Einbettung in existierende Software (z.B. Jupyter, Wireshark)
  - Mehrere Dateien pro Instanz? (Tabbed, MDI?)
    - Eine Datei (z.B. Wireshark - siehe Figure 24)
    - Mehrere Dateien in einzelnen Fenstern (z.B. GIMP - siehe Figure 25)
    - ... oder Tabs (z.B. Webbrowser - siehe Figure 26)
  - Anordnung und Anzahl der Bedienelemente (z.B. Paketliste, Hex dump, Dissection tree, Dissector code)
  - Maus/Tastaturbedienung
- Welche Importfunktionen wären wichtig um Mitschnitte einzulesen? (z.B. pcap, CSV, logic analyzer traces, raw serial data)
- Ist es relevant, Daten in Echtzeit während dem Mitschnitt anzuzeigen? (Live capture)
- Ist es relevant, Mitschnitte zeitlich zu korrelieren (z.B. mit anderen Mitschnitten, mit Notizen mit Timestamp, mit Video, etc)


File1.pcap – Protocol RE Workbench
File Edit View Parser Tools Window Help
Packet List
Annotated Hex View
Dissector Code
Packet Details

```
00000000  10 25 00 06 4d 51 49 73  64 70 03 02 00 05 00 17   .%..MQIsdp......
00000010  70 61 68 6f 2f 33 34 41  41 45 35 34 41 37 35 44   paho/34AAE54A75D
00000020  38 33 39 35 36 36 45                               839566E
```

```
DEFAULT struct (endianness=">"){
        flags UINT8
        msg_len UINT8
        protocol_name_len UINT16
        protocol_name STRING(charset="ascii", size=(protocol_nam
        version UINT8
        connect_flags UINT8
        keep_alive UINT16
        client_id_len UINT16
        client_id STRING(charset="ascii", size=(client_id_len))
}
```

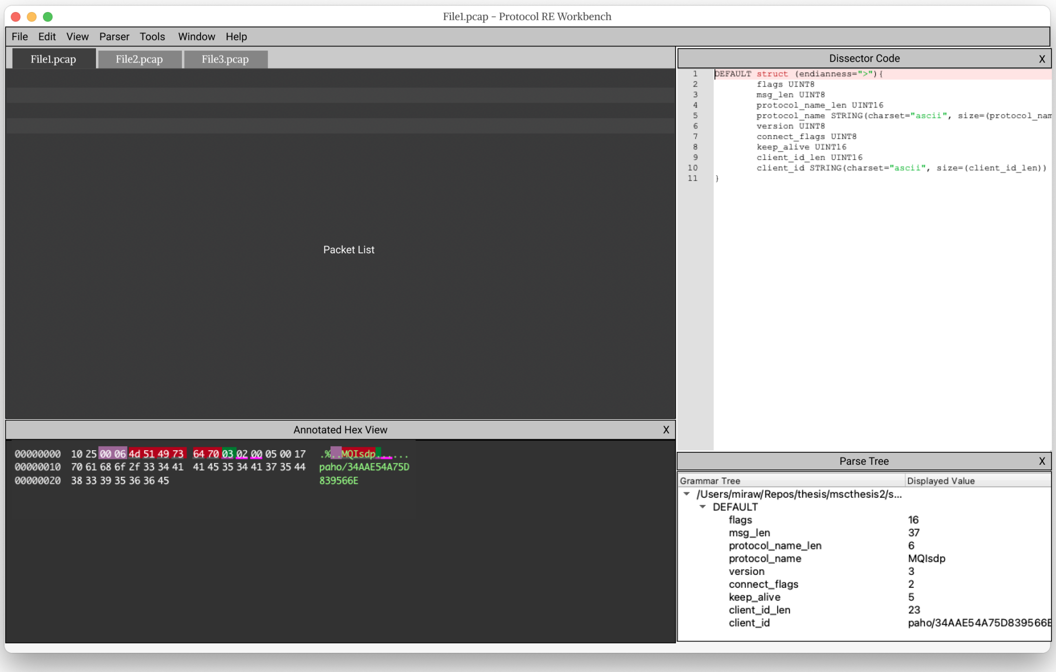
Grammar Tree	Displayed Value	Offset
/Users/miraw/Repos/thesis/mscthesis2/s...		
DEFAULT		0+39
flags	16	0+1
msg_len	37	1+1
protocol_name_len	6	2+2
protocol_name	MQIsdp	4+6
version	3	10+1
connect_flags	2	11+1

Packet Details:

```
00000000  10 25 00 06 4d 51 49 73  64 70 03 02 00 05 00 17   .%..MQIsdp......
00000010  70 61 68 6f 2f 33 34 41  41 45 35 34 41 37 35 44   paho/34AAE54A75D
00000020  38 33 39 35 36 36 45                               839566E
```

Grammar Tree	Displayed Value	Offset
/Users/miraw/Repos/thesis/mscthesis2/s...		
DEFAULT		0+39
flags	16	0+1
msg_len	37	1+1

Figure 24: Mock-up: Single Trace File per Instance, Detail Windows for Comparing Packet Contents (like e.g. Wireshark)

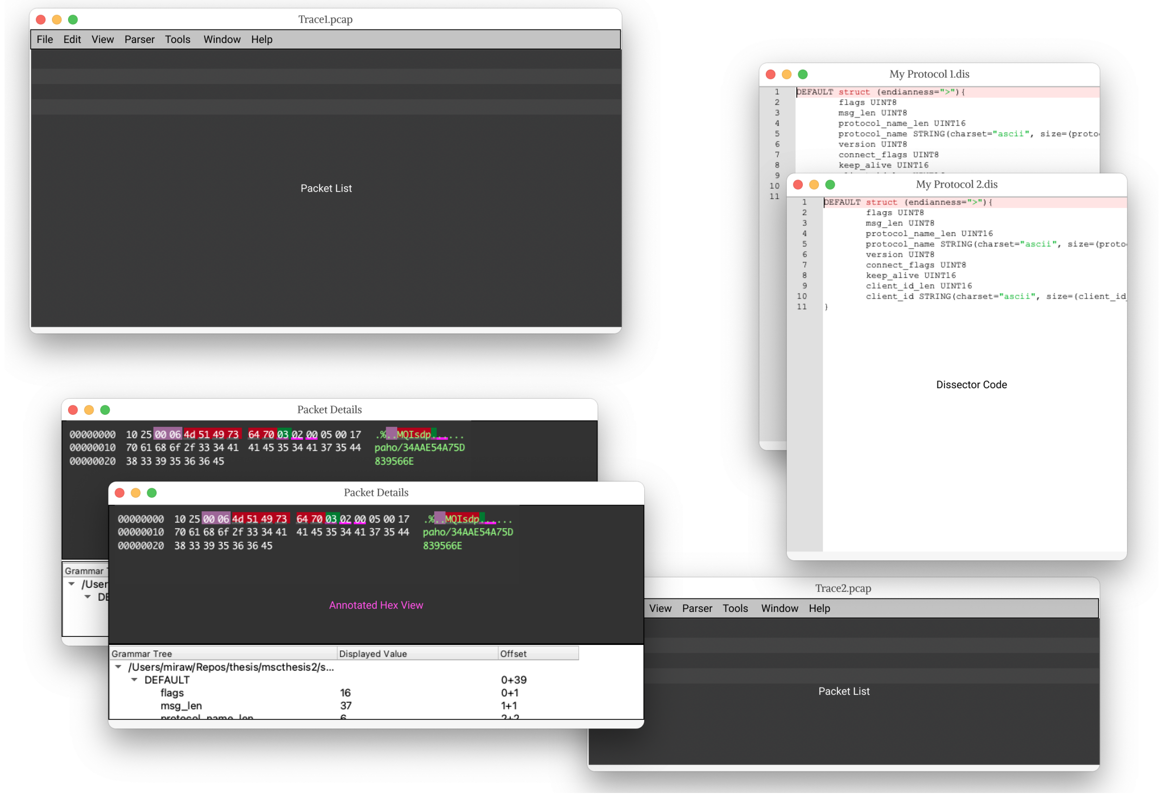


```
DEFAULT struct (endianness=">"){
        flags UINT8
        msg_len UINT8
        protocol_name_len UINT16
        protocol_name STRING(charset="ascii", size=(protocol_nam
        version UINT8
        connect_flags UINT8
        keep_alive UINT16
        client_id_len UINT16
        client_id STRING(charset="ascii", size=(client_id_len))
}
```

```
00000000  10 25 00 06 4d 51 49 73  64 70 03 02 00 05 00 17   .%..MQIsdp......
00000010  70 61 68 6f 2f 33 34 41  41 45 35 34 41 37 35 44   paho/34AAE54A75D
00000020  38 33 39 35 36 36 45                               839566E
```

Grammar Tree	Displayed Value
/Users/miraw/Repos/thesis/mscthesis2/s...	
DEFAULT	
flags	16
msg_len	37
protocol_name_len	6
protocol_name	MQIsdp
version	3
connect_flags	2
keep_alive	5
client_id_len	23
client_id	paho/34AAE54A75D839566E

Figure 25: Mock-up: Tabbed MDI Interface with Dock Panels



```
DEFAULT struct (endianness=">"){
        flags UINT8
        msg_len UINT8
        protocol_name_len UINT16
        protocol_name STRING(charset="ascii", size=(proto
        version UINT8
        connect_flags UINT8
        keep_alive UINT16
        client_id_len UINT16
        client_id STRING(charset="ascii", size=(client_id
}
```

```
00000000  10 25 00 06 4d 51 49 73  64 70 03 02 00 05 00 17   .%..MQIsdp......
00000010  70 61 68 6f 2f 33 34 41  41 45 35 34 41 37 35 44   paho/34AAE54A75D
00000020  38 33 39 35 36 36 45                               839566E
```

Grammar Tree	Displayed Value	Offset
/Users/miraw/Repos/thesis/mscthesis2/s...		
DEFAULT		0+39
flags	16	0+1
msg_len	37	1+1
protocol_name_len	6	2+2

Figure 26: Mock-up: Multi-Window / Inspector Style (like e.g. GIMP)

B PROTOCOL GRAMMAR DESCRIPTION LANGUAGE

A grammar file consists of a list of type definitions, in the format `name1 type_instance1 name2 type_instance2 ...`, each element separated by white-space. The following sections explain the main syntax elements of the language, namely types, parameters (including values) and expressions. For a formal specification of the language, see Section C.1.

B.1 TYPES

A *type* (or *type instance*) is a specific data type as specified in a grammar file, e.g. a `struct` containing specific fields. Type instances can occur at the root level of the file, preceded by an identifier, forming a *type definition*, or inline, for example as the type of a `struct` field or as the child type of a `repeat`.

Each type instance belongs to a *type class*, which determines the definition syntax as well as the parser behavior. Most type classes specify composite types, meaning they are constructed from one or more other types (e.g. the `struct` type class). Exceptions are named type references, predefined types, and the `bits` type class. The named type reference is a special type class in that it does not define a new type, but refers to another type definition, which was defined at the root level. All type classes are introduced in the rest of this section.

B.1.1 *Named Type Reference*

In any place where a type instance is expected, a name can be used to reference another type definition in the same file. Definitions can be placed in the file in any order. An example for a named type reference is shown in Listing 2.

This allows for generalization, because the same type instance can be referenced in multiple places (e.g. to define a common header shared by many different packet types). It can also make the grammar easier to read, because special cases can be put away at the end of the file, and the nesting depth can be reduced. After the name, parameters configuring parsing or visualization details can be provided in parentheses. This makes more generic type definitions possible, where e.g. the endianness is left open until the usage.

```
MyPacket struct {
  /* references to user-defined types */
  header MyHeader
  payload MyPayload
}
/* here are the type definitions referenced above */
MyHeader struct { ... }
MyPayload struct { ... }
```

Listing 2: Example of named type references

B.1.2 *Predefined Types*

Many common types of integers, strings, floating-point numbers and network addresses are predefined. A complete list is provided in Table 3. For easier adaption, they are named the same as in Wireshark dissectors.

As with all type instances, parameters can be provided in parentheses after the type name. The parameters can be preceded by a size expression in square brackets. Example code using predefined types is shown in Listing 3.

LENGTHS For variable-length types (marked with length **E** in Table 3), a size expression needs to be provided in square brackets after the type name.

For dynamic-length types (marked with length **D**), the actual length of the field is determined during the parsing process. Currently this only applies to `STRINGZ`, a null-terminated string.

Prefix-length types (marked with length **P**) are composite types consisting of an unsigned integer determining the length of the directly following payload field. The size in bytes of the unsigned integer needs to be provided in the mandatory parameter `size_len`.

PARAMETERS All integer-based types with more than one byte, as well as the floating-point and the `GUID` types, require the `endianness` parameter, containing a < or > sign denoting the byte order. The string types require the `charset` parameter. These parameters can also be specified on a parent type, e.g. the enclosing struct, and cascade down to all child elements.

The `ABSOLUTE_TIME` type supports an optional `unit` parameter to specify the time unit as `us`, `ms` or `s`. If it is omitted, the parser tries to guess the unit. Timestamp are always interpreted as relative to the UNIX epoch, and in the current system timezone.

The `NONE` type is used to calculate values from an expression specified in the `value` parameter, without parsing additional data from the buffer.

Type Name	Length	Comments
`NONE`	0	Used for calculated fields
`BOOLEAN`	1	0 = False, 1-255 = True
`CHAR`	1	Unsigned Integer
`E_UINT`	**E**	Unsigned Integer
`UINT8`	1	Unsigned Integer
`UINT16`	2	Unsigned Integer
`UINT24`	3	Unsigned Integer
`UINT32`	4	Unsigned Integer
`UINT40`	5	Unsigned Integer
`UINT48`	6	Unsigned Integer
`UINT56`	7	Unsigned Integer
`UINT64`	8	Unsigned Integer
`E_INT`	**E**	Signed Integer
`INT8`	1	Signed Integer
...	...	(according for INT16 to INT56)
`INT64`	8	Signed Integer
`FLOAT`	4	IEEE 754 Single-Precision Float
`DOUBLE`	8	IEEE 754 Double-Precision Float
`ABSOLUTE_TIME`	**E**	Unsigned Integer, Decoded as Timestamp
`BYTES`	**E**	Bytes, Variable Length, Requires Size Expression
`UINT_BYTES`	**P**	Length-prefixed Bytes, Requires Parameter `size_len`
`STRING`	**E**	String, Requires Parameter `charset`
`UINT_STRING`	**P**	Length-Prefixed String, Requires Parameters `charset` and `size_len`
`STRINGZ`	**D**	Null-Terminated String, Requires Parameter `charset`
`ETHER`	6	Ethernet MAC Address, Formatted as xx:xx:xx:xx:xx:xx
`IPv4`	4	IPv4 Address, Network Byte Order, Formatted String
`IPv6`	16	IPv6 Address, Network Byte Order, Formatted String
`GUID`	16	UUID, Formatted as UUID String
`EUI64`	8	EUI64, Formatted as xx:xx:xx:xx:xx:xx:xx:xx

The length of fixed-length fields is specified in bytes.
The following types are supported by Wireshark, and are reserved but not implemented yet in PRE Workbench: RELATIVE_TIME, PROTOCOL, IEEE_11073_SFLOAT, IEEE_11073_FLOAT, IPXNET, FRAMENUM, PCRE, STRINGZPAD, FCWWN, AX25, VINES, OID, REL_OID, SYSTEM_ID

Table 3: Predefined types

All types support the optional `show` parameter, which contains either an internal function name or a format string, either of which is applied to the parsed value before displaying it. For example, `UINT8(show="hex")` or `UINT8(show="0x%02x")` to display the value 15 as `0x0f`.

```
/* unsigned integer, 4 byte, little endian. */
name_length UINT32(endianness="<")

/* character string in UTF-8 encoding, referencing a previous
    field for its size. */
name STRING[name_length](charset="utf-8")

/* IP version 4 address, in binary, in network byte order. */
src_addr IPv4

/* calculates a value using an expression, without parsing data
    */
length_in_bits NONE(value=(name_length * 8))
```

Listing 3: Examples of using predefined types with parameters

B.1.3 *struct*

A `struct` is defined as an ordered list of named field definitions, where each field has a type. The size of a struct equals the sum of the sizes of all contained fields, no padding is applied automatically. The value resulting from parsing a struct is an ordered dictionary, with the field names as keys and the parse results of the fields as values. Multiple examples of `struct` type instances are shown in Listing 4.

B.1.4 *repeat*

Instances of the `repeat` type class can be used to repeatedly parse a specified child type. The number of repetitions can be specified in several ways: The `times` parameter can be used to specify the number of repetitions up front as the result of an expression (similar to a for loop in common programming languages). For sample code using this parameter, see example A in Listing 4. Alternatively, an expression can be passed in the `until` parameter which will be evaluated after each parsing of the child type, terminating the loop if it returns true (like a do-until-loop, see example B). Another possibility is repetition until a parse error occurs, which is possible by setting `until_invalid=true` (see example C). A typical case where this can be used is if each repetition starts with a magic value, enforced via the `magic` parameter. Finally, if no parameter is specified at all, the child type is repeated until the data runs out (see example D).

```
/* A) Number of repetitions specified up-front */
Int32Array struct(endianness=">") {
  count UINT16
  items repeat(times=(count)) INT32
}

/* B) Conditional repetition, until parsed value is zero */
Int32UntilZero repeat(until=(this[-1] == 0))
  INT32(endianness=">")

/* C) Repetition until parsing fails, here: until magic != 255 */
MagicArray repeat(until_invalid=true) struct {
  magic_byte UINT8(magic=255)
  value_byte UINT8
}

/* D) Repetition until end of buffer, here: buffer length must be
     divisible by 3, otherwise parsing fails */
RepeatTillEnd repeat struct {
  a UINT8
  b UINT8
  c UINT8
}
```

Listing 4: Examples with `struct` and `repeat` types

In all cases, the parse result is an ordered list of the parse results of the child type.

B.1.5 *variant*

The `variant` type class can be used to try different child types in sequence until the buffer can be successfully parsed. A typical use case is different `struct` child types, each starting with a different magic number enforced with the `magic` parameter. A mismatching magic number causes the parsing to fail, so the next child type is tried. For example, the internal PCAP file parser uses a construct similar to Listing 5, trying out different file formats and byte orders until a match is found. If none of the variants match, the whole composite type returns a parse error.

```
CaptureFile variant {
  PcapngFile(endianness=">")
  PcapngFile(endianness="<")
  PcapFile(endianness=">")
  PcapFile(endianness="<")
}
```

Listing 5: Example of a `variant` type

B.1.6 *switch*

The switch type class serves a similar purpose as variant, namely to parse a buffer with different child types depending on the contents. However, unlike the variant, the switch type allows the selection of the child type based on a preceding value, e.g. in a header structure, while the variant selects based on the parsed contents.

Analogous to switch-case statements in common programming languages, an expression can be specified after the switch keyword whose value is compared to the values after the case keywords. The child type specified at the first matching case is used to parse the buffer. An example is shown in Listing 6.

The current implementation does not support a *default* case yet, however, this could be implemented either by extending the syntax with a default keyword, or by defining a sentinel value that matches everything.

```
MyPacket struct {
  header struct {
    type UINT8
  }
  payload switch (header.type) {
    case 1: Payload1
    case 2: Payload2
  }
}
Payload1 struct { ... }
Payload2 struct { ... }
```

Listing 6: Example of a switch type

B.1.7 *union*

The union type is very similar to the variant type. It also parses the same byte range with multiple different types. The main difference is that parsing is not stopped after the first successful parse with one child type. Instead, the buffer is parsed with all child types, and a dictionary containing all parse results is returned. If parsing should also continue if parse errors occur in some child types, the parameter ignore_errors=true is used. Example code is shown in Listing 7.

```
UnsignedOrSigned union {
  unsigned UINT16
  signed INT16
}
```

Listing 7: Example of a union type

B.1.8 *bits*

Using the `bits` type, a packed bit field is described. It can be used for integer values containing *bitwise or*-ed flags, as well as for numeric fields which do not align to byte boundaries. Example code is shown in Listing 8.

A `bits` definition consists of an ordered list of fields, each described by a field name and the length in bits. By specifying a length of one bit, a single-bit flag can be defined. Otherwise, the field is parsed as an unsigned integer. Signed integers are currently unsupported in bit fields. Implementation-wise, the `bits` type is a special case in that it does not consist of child types, but still returns a composite result. The parse result is an ordered dictionary mapping the field name to the numeric value (see Figure 27).

If little-endian encoding is configured, the bytes making up the bit field are reversed byte-wise first, before the individual bits are extracted.

```
TCP_flags bits(endianness=">") {
  hdr_len   : 4
  _reserved : 3
  NS        : 1
  CWR       : 1
  ECE       : 1
  URG       : 1
  ACK       : 1
  PSH       : 1
  RST       : 1
  SYN       : 1
  FIN       : 1
}
```

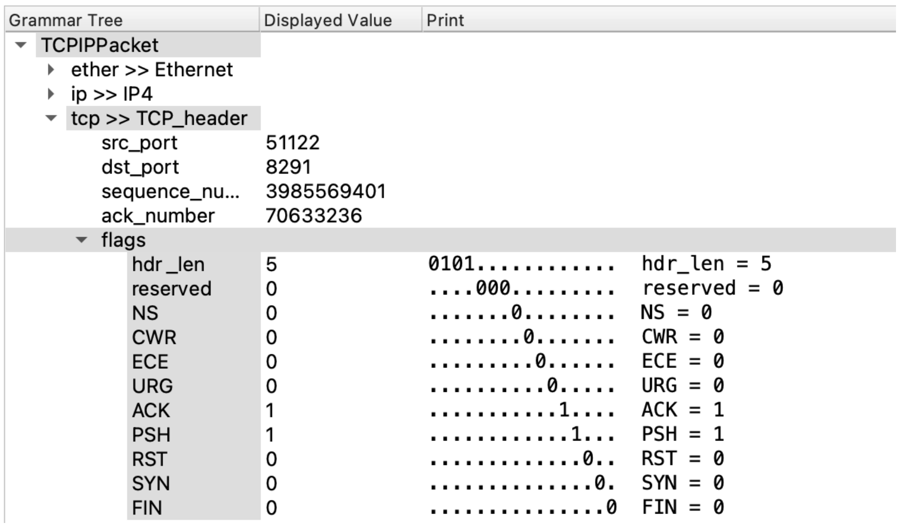
Listing 8: Example of `bits` type describing the flags in the TCP header

B.2 PARAMETERS

All types can be parameterized to specify additional information on how to parse or display this field. Some types have required parameters, which lead to a parse error if omitted. The parameter `size` has a special abbreviated syntax, allowing to use `BYTES[42]` instead of `BYTES(size=(42))`.

The language differentiates between (constant) values and (dynamic) expressions. Syntactically, each parameter is assigned a value. The value is interpreted at compile time of the expression, i.e. only once even if a `repeat` type is repeated multiple times.

Special possible values are expressions and named type references. Expressions are wrapped in parenthesis, marking them to be interpreted at data parse time. They are described in detail in the next



```
Grammar Tree                  Displayed Value   Print
▾ TCPIPPacket
  ▸ ether >> Ethernet
  ▸ ip >> IP4
  ▾ tcp >> TCP_header
      src_port                51122
      dst_port                8291
      sequence_nu...          3985569401
      ack_number              70633236
    ▾ flags
        hdr_len               5                 0101............  hdr_len = 5
        reserved              0                 ....000.........  reserved = 0
        NS                    0                 .......0........  NS = 0
        CWR                   0                 ........0.......  CWR = 0
        ECE                   0                 .........0......  ECE = 0
        URG                   0                 ..........0.....  URG = 0
        ACK                   1                 ...........1....  ACK = 1
        PSH                   1                 ............1...  PSH = 1
        RST                   0                 .............0..  RST = 0
        SYN                   0                 ..............0.  SYN = 0
        FIN                   0                 ...............0  FIN = 0
```

Figure 27: Parse tree of the flags in the TCP header

section. Named type references are used with the `parse_with` parameter, which allows a byte array with predefined length to be parsed with another type. Examples for these special values are provided in Listing 9.

```
MyFrame struct {
  length UINT8
  /* short for BYTES(size=(length - 1), parse_with=MyPayloadType)
     (length - 1) is an expression, evaluated at data parse time.
     MyPayloadType references another type. */
  payload BYTES[length - 1](parse_with=MyPayloadType)
}
MyPayloadType struct { ... }
```

Listing 9: Example of parameters with expression and type reference

B.3 EXPRESSIONS

As opposed to other value types like strings or numbers, expressions are evaluated at data parse time, i.e. each time a field of that type is parsed, so in a `repeat` type, at each repetition. The general syntax is borrowed from C and Java.

Fields that were parsed earlier on can be referenced in an expression, making it possible to calculate field or array lengths from any previous field value. All fields from the current structure, or from a parent structure as seen in the parsing stack, can be directly accessed by their name (see example in Listing 10). This means the language uses dynamic scope, as opposed to lexical scope, where only elements from parent structures as seen in the Abstract Syntax Tree (AST) would be accessible (see example in Listing 11). Fields from child constructed types are accessible using the member operator (e.g. `header.length`)

Precedence	Type	Operators
1	Conjunctions	`||` `&&`
2	Equality	`==` `!=`
3	Comparison	`<` `>` `<=` `>=`
4	Math (lower)	`+` `-` `&` `|` `^`
5	Math (higher)	`*` `/` `<<` `>>`
6	Primary	`(...)` `f(x)` `a[i]` `a.b`

Table 4: Operators in increasing order of precendence

Name	Description
`hex`	Returns a hex string representation of the parameter.
`dec`	Returns a decimal string representation of the parameter.
`dotted_quad`	Returns a dot-separated decimal representation of the bytes passed as a parameter, like in an IPv4 address.
`ip6`	Formats a bytes object of length 16 as an IPv6 address.
`len`	Returns the length of the parameter.
`snip`	Truncates the parameter to 32 bytes, if longer.
`pad`	Returns the number of bytes required to pad the current buffer to a multiple of N bytes.

Table 5: Built-in functions

and the array index operator (e.g. `values[0]`), which can also be combined.

Math and bitwise operations are supported as well. A full list of supported operators is provided in Table 4. A number of built-in functions is available. For a complete list, see Table 5. In the current implementation, functions are always pure (they do not have side effects) and always have exactly one parameter. It is not possible to define custom functions directly in the language, but PRE Workbench plugins (see Section 5.4.2) can register additional functions by using the `@ExprFunctions.register()` annotation.

```
MyHeader1 struct {
  length UINT8
  payload MyPayload
}
MyHeader2 struct(endianness="<") {
  length UINT32
  payload MyPayload
}
MyPayload struct {
  /* Dynamic scope rules allow 'length' in the following
     expression to access the length field of whichever struct
     MyPayload was referenced from. With lexical scope, none
     of the length fields would be accessible here. */
  content BYTES[length]
}
```

Listing 10: Example of dynamic scope

```
MyPacket struct {
  length UINT8
  payload struct {
    /* Lexical scope rules allow 'length' in the following
       expression to access the length field in MyPacket, because
       the containing struct is a parent in the AST. */
    content BYTES[length]
  }
}
```

Listing 11: Example of lexical scope

C CODE

C.1 FORMAL SPECIFICATION OF PROTOCOL GRAMMAR DEFINITION LANGUAGE

```
grammar_file: type_definition*
type_definition: IDENTIFIER type_instance

// Type instances
type_instance: named | struct | repeat | variant | switch |
               union | bits

named: IDENTIFIER params

struct: "struct" params "{" field* "}"
field: IDENTIFIER type_instance

repeat: "repeat" params type_instance

variant: "variant" params "{" type_instance* "}"

switch: "switch" expression params "{"
      ("case" expression ":" type_instance)*
    "}"

union: "union" params "{" (IDENTIFIER type_instance)* "}"

bits: "bits" params "{" (IDENTIFIER ":" number)* "}"

// Parameters
params: ("[" size_expr "]")? ("(" (IDENTIFIER value)* ")")?
value: string | number | dict | list | "true" | "false" | "null"
       | "(" expr_value ")"
       | named


// Static value types
list : "[" [value ("," value)*] "]"
dict : "{" [pair ("," pair)*] "}"
pair : string ":" value
number: NUMBER
string: ESCAPED_STRING

// Definitions for expressions and comments are omitted for
    brevity.
```

Listing 12: Formal specification of Protocol Grammar Definition Language

C.2 GENERATED WIRESHARK DISSECTOR

```lua
--------------------------------------------
-- init
MyProto_proto = Proto("MyProto", "MyProto Protocol")

--------------------------------------------
-- ws_field_defs
local f_MyProto_magic = ProtoField.uint32("MyProto.magic", "magic", base.DEC)
local f_MyTLV_type = ProtoField.uint16("MyTLV.type", "type", base.DEC)
local f_MyTLV_length = ProtoField.uint16("MyTLV.length", "length", base.DEC)
local f_MyTLV_payload = ProtoField.bytes("MyTLV.payload", "payload", base.NONE)
MyProto_proto.fields = {f_MyProto_magic,f_MyTLV_type,f_MyTLV_length,f_MyTLV_payload}

--------------------------------------------
-- parser functions
function MyProto_proto.dissector(buffer, pinfo, tree)
  local subtree = tree:add(MyProto_proto, buffer(), "Protocol Data MyProto")
  local fieldValues = {}
  end_offset = parse_MyProto(buffer, pinfo, subtree, "", fieldValues)
  subtree:set_len(end_offset)
end

-- definition MyProto
function parse_MyProto(buffer, pinfo, treenode, title_prefix, fval)
  local subtree = treenode:add(MyProto_proto, buffer(), title_prefix .. "MyProto")
  local offset = 0
  local field_item
  -- struct MyProto
  -- field MyProto_magic UINT32
  len = 4  -- static length
  field_item = subtree:add(f_MyProto_magic, buffer(offset, len))
  fval[’magic’] = buffer(offset, len):uint()
  if fval[’magic’] ~= 2864434397 then
    field_item:add_expert_info(PI_MALFORMED, PI_ERROR, "magic value mismatch, expected 2864434397, got " .. fval["magic"])
  end
  offset = offset + len

  -- repeat MyProto_tlvs
  while offset < buffer:len() do
  -- named MyProto_tlvs_
  offset = offset + parse_MyTLV(buffer(offset), pinfo, subtree, '
      tlvs: ', fval)
  if false then break end
  end
  subtree:set_len(offset)
  return offset
end

-- definition MyTLV
function parse_MyTLV(buffer, pinfo, treenode, title_prefix, fval)
  local subtree = treenode:add(MyProto_proto, buffer(),
      title_prefix .. "MyTLV")
  local offset = 0
  local field_item
  -- struct MyTLV
  -- field MyTLV_type UINT16
  len = 2  -- static length
  field_item = subtree:add(f_MyTLV_type, buffer(offset, len))
  fval['type'] = buffer(offset, len):uint()
  offset = offset + len

  -- field MyTLV_length UINT16
  len = 2  -- static length
  field_item = subtree:add(f_MyTLV_length, buffer(offset, len))
  fval['length'] = buffer(offset, len):uint()
  offset = offset + len

  -- field MyTLV_payload BYTES
  len = fval['length']  -- expression-based length
  field_item = subtree:add(f_MyTLV_payload, buffer(offset, len))
  offset = offset + len

  subtree:set_len(offset)
  return offset
end



--------------------------------------------
-- registration
local dissector_table = DissectorTable.get("wtap_encap")
dissector_table:add(148, MyProto_proto)
```

Listing 13: MyProto.lua

C.3 PGDL DEFINITION OF ETHERNET, IPV4 AND TCP HEADER

```
  ETHER_header struct(endianness=">") {
    dst_mac ETHER
    src_mac ETHER
    ethertype UINT16
```

```
}
IP4_header struct(endianness=">") {
  fields bits {
    version : 4
    header_len : 4
    tos : 8
    packet_len : 16
    ident : 16
    reserved_flag : 1
    dont_fragment : 1
    more_fragments : 1
    fragment_offset : 13
  }
  ttl UINT8
  protocol UINT8
  header_checksum UINT16
  src_ip IPv4
  dst_ip IPv4
}
TCP_header struct(endianness=">") {
  src_port UINT16
  dst_port UINT16
  sequence_number UINT32
  ack_number UINT32
  flags bits {
    hdr_len : 4
    reserved : 3
    NS : 1
    CWR: 1
    ECE:1
    URG:1
    ACK:1
    PSH:1
    RST:1
    SYN:1
    FIN:1
  }
  window_size_value UINT16
  checksum UINT16
  urgent_pointer UINT16
  options BYTES[b.hdr_len * 4 - 20](parse_with=TCP_options)
}
TCP_options repeat variant {
  UINT8(description="End of options", magic=0)
  UINT8(description="No operation (padding)", magic=1)
  struct (description="Regular option") {
    option_kind UINT8
    option_length UINT8
    option_data BYTES[option_length]
  }
}
```

Listing 14: Definition of Ethernet, IPv4 and TCP header in PGDL

C.4 PCAP AND PCAPNG FILE FORMAT DEFINITION

```
pcap_file variant {
  struct (endianness="<", section="pcap file, little endian"){
    header pcap_header
    packets repeat pcap_packet
  }
  struct (endianness=">", section="pcap file, big endian"){
    header pcap_header
    packets repeat pcap_packet
  }
  struct (endianness="<", section="pcapNG file, little endian")
      {
    first_block pcapng_first_block
    rest_blocks repeat pcapng_block
  }
  struct (endianness=">", section="pcapNG file, big endian"){
    first_block pcapng_first_block
    rest_blocks repeat pcapng_block
  }
}

pcap_header struct (section="pcap file header"){
  magic_number UINT32(description="'A1B2C3D4' means the
      endianness is correct", magic=2712847316)
  version_major UINT16(description="major number of the file
      format")
  version_minor UINT16(description="minor number of the file
      format")
  thiszone INT32(description="correction time in seconds from
      UTC to local time (0)")
  sigfigs UINT32(description="accuracy of time stamps in the
      capture (0)")
  snaplen UINT32(description="max length of captured packed
      (65535)")
  encap_proto UINT32(description="type of data link (1 =
      ethernet)")
}

pcap_packet struct {
  pheader struct (section="pcap packet header"){
    ts_sec UINT32(description="timestamp seconds")
    ts_usec UINT32(description="timestamp microseconds")
    incl_len UINT32(description="number of octets of packet
        saved in file")
    orig_len UINT32(description="actual length of packet")
  }
  payload BYTES[pheader.incl_len]
}

pcapng_first_block struct (section="pcapNG first block"){
  block_type UINT32(magic=0x0A0D0D0A, color="#999900", show="0x%08X")
  block_length UINT32(color="#666600")
  block_payload struct {
    byte_order_magic UINT32(magic=439041101, color="green", show="0x%08X")
    version_major UINT16
    version_minor UINT16
    section_length INT64
    options BYTES[block_length-28](parse_with=pcapng_options)
  }
  block_length2 UINT32(color="#666600")
}

pcapng_block struct (section="pcapNG block"){
  block_type UINT32(color="#999900", show="0x%08X")
  block_length UINT32(color="#666600")
  block_payload BYTES[block_length - 12](parse_with=pcapng_block_payload)
  block_length2 UINT32(color="#666600")
}

pcapng_block_payload switch block_type {
  case 0x0A0D0D0A: pcapng_SHB
  case 1: pcapng_IDB
  case 3: pcapng_SPB
  case 5: BYTES
  case 6: pcapng_EPB
}

pcapng_SHB struct {
  byte_order_magic UINT32(magic=439041101, color="green", show="0x%08X")
  version_major UINT16
  version_minor UINT16
  section_length INT64
  options pcapng_options
}

pcapng_IDB struct {
  linktype UINT16
  reserved UINT16
  snaplen UINT32
  options pcapng_options
}

pcapng_EPB struct {
  interface_id UINT32
  timestamp_hi UINT32
  timestamp_lo UINT32
  cap_length UINT32
  orig_length UINT32
  payload BYTES[cap_length]
  payload_padding BYTES[pad(4)](textcolor="#888888")
}

pcapng_SPB struct {
  orig_length UINT32
  payload BYTES[block_length - 16]
  payload_padding BYTES[pad(4)](textcolor="#888888")
}

pcapng_options repeat struct {
  code UINT16(color="#660666")
  length UINT16
  value BYTES[length](textcolor="#d3ebff")
  padding BYTES[pad(4)](textcolor="#666")
}
```

Listing 15: Definition of the PCAP and PCAPNG file formats in PGDL

BIBLIOGRAPHY

[1] *802.3-2018 - IEEE Standard for Ethernet*. IEEE. DOI: 10.1109/IEEESTD.2018.8457469. URL: https://ieeexplore.ieee.org/document/8457469/ (visited on 11/06/2022).

[2] A. V. Alekseyenko and C. J. Lee. "Nested Containment List (NCList): A New Algorithm for Accelerating Interval Query of Genome Alignment and Interval Databases." In: *Bioinformatics* 23.11 (June 1, 2007), pp. 1386–1393. ISSN: 1367-4803, 1460-2059. DOI: 10.1093/bioinformatics/btl647.

[3] Peter Ammon and Kevin Wojniak. *Hex Fiend, a Fast and Clever Hex Editor for macOS*. URL: https://hexfiend.com/ (visited on 01/29/2022).

[4] *Apache Thrift - Home*. URL: https://thrift.apache.org/ (visited on 07/11/2022).

[5] Marshall A Beddoe. "Network Protocol Analysis Using Bioinformatics Algorithms." In: *Toorcon* (2004).

[6] Michael Bilenko. *Hex-Works: Online Hex Editor Tool*. URL: https://hex-works.com/eng (visited on 01/29/2022).

[7] Philippe Biondi. *Scapy*. URL: https://scapy.net/ (visited on 11/24/2022).

[8] *Bison - GNU Project - Free Software Foundation*. URL: https://www.gnu.org/software/bison/ (visited on 07/11/2022).

[9] *Chapter 11. Wireshark's Lua API Reference Manual*. URL: https://www.wireshark.org/docs/wsdg_html_chunked/wsluarm_modules.html (visited on 07/27/2022).

[10] Gerald Combs. *Wireshark Network Protocol Analyzer*. Version 3.6.3. Mar. 23, 2022. URL: https://www.wireshark.org/ (visited on 04/04/2022).

[11] Thomas H. Cormen, ed. *Introduction to Algorithms*. 3rd ed. Cambridge, Mass: MIT Press, 2009. 1292 pp. ISBN: 978-0-262-03384-8 978-0-262-53305-8.

[12] Aldo Cortesi, Mayimilian Hils, and Thomas Kriechbaumer. *Mitmproxy - an Interactive HTTPS Proxy*. URL: https://mitmproxy.org/ (visited on 11/24/2022).

[13] Dug Song. *Dpkt — Dpkt 1.9.2 Documentation*. URL: https://dpkt.readthedocs.io/en/latest/ (visited on 08/21/2022).

[14] M. Eisler. *XDR: External Data Representation Standard*. STD 67. RFC Editor / RFC Editor, May 2006. URL: http://www.rfc-editor.org/rfc/rfc4506.txt.

[15] *Electron | Plattformübergreifende Desktop-Anwendungen mit JavaScript, HTML und CSS entwickeln*. URL: https://www.electronjs.org/ (visited on 08/17/2022).

[16] *FlatBuffers: FlatBuffers*. URL: https://google.github.io/flatbuffers/ (visited on 07/11/2022).

[17] James Forshaw. *Attacking Network Protocols: A Hacker's Guide to Capture, Analysis, and Exploitation*. 1 edition. San Francisco: No Starch Press, Dec. 8, 2017. 336 pp. ISBN: 978-1-59327-750-5.

[18] *Ghidra*. National Security Agency. URL: https://ghidra-sre.org/ (visited on 01/29/2022).

[19] githubuseroxFFFF and Manuel Freiholz. *githubuseroxFFFF/Qt-Advanced-Docking-System*. Aug. 25, 2022. URL: https://github.com/githubuser0xFFFF/Qt-Advanced-Docking-System (visited on 08/26/2022).

[20] *GNU Radio - The Free & Open Source Radio Ecosystem · GNU Radio*. GNU Radio. URL: https://www.gnuradio.org/ (visited on 07/13/2022).

[21] *GoldenLayout- a Multi-Window Javascript Layout Manager for Webapps*. URL: https://golden-layout.com/ (visited on 08/17/2022).

[22] Hadriel Kaplan. *Pcapng-Test-Generator*. URL: https://github.com/hadrielk/pcapng-test-generator.

[23] Alexander Heinrich, Milan Stute, and Matthias Hollick. "DEMO: BTLEmap: Nmap for Bluetooth Low Energy." In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. July 8, 2020, pp. 331–333. DOI: 10.1145/3395351.3401796. arXiv: 2007.00349 [cs].

[24] Holger Krekel. *Pytest*. Version 7.2.x. URL: https://docs.pytest.org/en/7.2.x/.

[25] *IDA Pro*. Hex Rays. URL: https://hex-rays.com/ida-pro/ (visited on 01/29/2022).

[26] *Internet Protocol*. Request for Comments RFC 791. Internet Engineering Task Force, Sept. 1981. 51 pp. DOI: 10.17487/RFC0791. URL: https://datatracker.ietf.org/doc/rfc791 (visited on 11/06/2022).

[27] *Kaitai Struct: Declarative Binary Format Parsing Language*. URL: http://kaitai.io/ (visited on 11/24/2022).

[28] Stephan Kleber, Lisa Maile, and Frank Kargl. "Survey of Protocol Reverse Engineering Algorithms: Decomposition of Tools for Static Traffic Analysis." In: *IEEE Communications Surveys Tutorials* 21.1 (Firstquarter 2019), pp. 526–561. DOI: 10.1109/COMST.2018.2867544.

[29] Tobias Kröll, Stephan Kleber, Frank Kargl, Matthias Hollick, and Jiska Classen. "ARIstoteles – Dissecting Apple's Baseband Interface." In: Sept. 30, 2021, pp. 133–151. ISBN: 978-3-030-88417-8. DOI: 10.1007/978-3-030-88418-5_7.

[30] XiangDong Li and Li Chen. "A Survey on Methods of Automatic Protocol Reverse Engineering." In: *2011 Seventh International Conference on Computational Intelligence and Security*. 2011 Seventh International Conference on Computational Intelligence and Security (CIS). Sanya, Hainan, China: IEEE, Dec. 2011, pp. 685–689. ISBN: 978-1-4577-2008-6 978-0-7695-4584-4. DOI: 10.1109/CIS.2011.156.

[31] *Logic Analyzers from Saleae - #1 with Professional Engineers*. URL: https://www.saleae.com/ (visited on 07/11/2022).

[32] *Mahlet-Inc/Hobbits*. Mahlet, Apr. 3, 2022. URL: https://github.com/Mahlet-Inc/hobbits (visited on 04/04/2022).

[33] Peter J McCann and Satish Chandra. "Packet Types: Abstract Specification of Network Protocol Messages." In: *ACM SIGCOMM Computer Communication Review* 30.4 (2000), pp. 321–333. ISSN: 1581132239.

[34] Jaka Mocnik and Chema Celorio. *GHex*. URL: https://wiki.gnome.org/Apps/Ghex (visited on 01/20/2022).

[35] *Monaco Editor*. URL: https://microsoft.github.io/monaco-editor/ (visited on 08/17/2022).

[36] "Multiple and Single Document Interfaces." In: Matthew MacDonald. *Pro .NET 2.0 Windows Forms and Custom Controls in C#*. Berkeley, CA: Apress, 2006, pp. 655–691. ISBN: 978-1-4302-0110-6. DOI: 10.1007/978-1-4302-0110-6_19.

[37] *Network Media Specific Capturing - Wireshark*. URL: https://wiki.wireshark.org/CaptureSetup/NetworkMedia (visited on 04/04/2022).

[38] *Npcap: Windows Packet Capture Library & Driver*. URL: https://nmap.org/npcap/ (visited on 11/24/2022).

[39] *Nutzerstudien in der Softwareentwicklung einsetzen*. URL: https://www.jambit.com/aktuelles/news/nutzerstudien-in-der-softwareentwicklung/ (visited on 08/09/2022).

[40] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. "Binpac: A Yacc for Writing Application Protocol Parsers." In: *Proceedings of the 6th ACM SIGCOMM on Internet Measurement - IMC '06*. Rio de Janeriro, Brazil: ACM Press, 2006, p. 289. ISBN: 978-1-59593-561-8. DOI: 10.1145/1177080.1177119.

[41] *PDML*. URL: https://wiki.wireshark.org/PDML (visited on 08/26/2022).

[42] Andreas Pehnack. *Hexinator*. URL: https://hexinator.com/ (visited on 11/24/2022).

[43] 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). 2018. URL: https://www.usenix.org/conference/woot18/presentation/pohl (visited on 05/25/2022).

[44] *Protocol Buffers*. Google Developers. URL: https://developers.google.com/protocol-buffers (visited on 02/27/2022).

[45] *PyCharm: The Python IDE for Professional Developers by JetBrains*. JetBrains. URL: https://www.jetbrains.com/pycharm/ (visited on 09/06/2022).

[46] *Pysmartlock Source Code*. GitHub. URL: https://github.com/luelista/hwreverse (visited on 11/02/2022).

[47] *Qt Framework - One Framework to Rule All!* URL: https://www.qt.io/product/framework (visited on 08/17/2022).

[48] Ole André V. Ravnås. *Frida*. URL: https://frida.re/ (visited on 01/27/2022).

[49] *Reverse Engineer*. In: *Merriam Webster*. URL: https://www.merriam-webster.com/dictionary/reverse+engineer (visited on 05/25/2022).

[50] Sadayuki Furuhashi. *MessagePack: It's like JSON. but Fast and Small.* URL: https://msgpack.org/ (visited on 02/27/2022).

[51] *Seemoo-Lab/Aristoteles*. Secure Mobile Networking Lab, Oct. 18, 2022. URL: https://github.com/seemoo-lab/aristoteles (visited on 10/25/2022).

[52] *Sencha Ext JS - Comprehensive JavaScript Framework, UI Components*. Sencha.com. URL: https://www.sencha.com/products/extjs/ (visited on 08/17/2022).

[53] Eshed Shaham. *Pyreshark*. Feb. 27, 2022. URL: https://github.com/ashdnazg/pyreshark (visited on 02/27/2022).

[54] Claude Elwood Shannon. "A Mathematical Theory of Communication." In: *The Bell system technical journal* 27.3 (1948), pp. 379–423. ISSN: 0005-8580.

[55] David Sloan, Catriona Macaulay, Paula Forbes, and Scott Loynton. "User Research in a Scientific Software Development Project." In: Jan. 1, 2009, pp. 423–429. DOI: 10.1145/1671011.1671066.

[56] *Snort - Network Intrusion Detection & Prevention System*. URL: https://www.snort.org/ (visited on 11/19/2019).

[57] Robin Sommer, Johanna Amann, and Seth Hall. "Spicy: A Unified Deep Packet Inspection Framework for Safely Dissecting All Your Data." In: *Proceedings of the 32nd Annual Conference on Computer Security Applications - ACSAC '16*. The 32nd Annual Conference. Los Angeles, California: ACM Press, 2016, pp. 558–569. ISBN: 978-1-4503-4771-6. DOI: 10.1145/2991079.2991100.

[58] Robin Sommer, Matthias Vallentin, Lorenzo De Carli, and Vern Paxson. "HILTI: An Abstract Execution Environment for Deep, Stateful Network Traffic Analysis." In: *Proceedings of the 2014 Conference on Internet Measurement Conference - IMC '14*. The 2014 Conference. Vancouver, BC, Canada: ACM Press, 2014, pp. 461–474. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663735.

[59] SweetScape. *010 Editor - Professional Text/Hex Editor with Binary Templates*. 010 Editor - Professional Text/Hex Editor with Binary Templates. URL: https://www.sweetscape.com/010editor/ (visited on 01/28/2022).

[60] Daniel Szelogowski. *Chunk List: Concurrent Data Structures*. Feb. 17, 2022. arXiv: 2101.00172 [cs]. URL: http://arxiv.org/abs/2101.00172 (visited on 08/26/2022).

[61] *Tcpdump/Libpcap*. URL: https://www.tcpdump.org (visited on 11/24/2022).

[62] *The LEX & YACC Page*. URL: http://dinosaur.compilertools.net/ (visited on 07/11/2022).

[63] *The Zeek Network Security Monitor*. URL: https://www.zeek.org/ (visited on 11/24/2022).

[64] *Timeit — Measure Execution Time of Small Code Snippets — Python 3.9.13 Documentation*. URL: https://docs.python.org/3.9/library/timeit.html#timeit.Timer.repeat (visited on 08/18/2022).

[65] *Transmission Control Protocol*. Request for Comments RFC 793. Internet Engineering Task Force, Sept. 1981. 91 pp. DOI: 10.17487/RFC0793. URL: https://datatracker.ietf.org/doc/rfc793 (visited on 11/06/2022).

[66] *Tvbuff.h - Wireshark Source Code*. July 21, 2022. URL: https://github.com/boundary/wireshark/blob/07eade8124fd1d5386161591b52e177ee6ea849f/epan/tvbuff.h (visited on 07/27/2022).

[67] Mira Weller. *Analysis of eQ-3 BLUETOOTH® Smart Lock*. Sept. 8, 2018. URL: https://github.com/luelista/hwreverse/blob/978d173537c76822de4abc8c07c7efb4dcbd4bc2/eqiva-smartlock/report.pdf (visited on 07/22/2022).

[68] Mira Weller. *PRE Workbench Example Plugins*. July 27, 2022. URL: https://github.com/luelista/prewb_example_plugins (visited on 09/07/2022).

[69] Mira Weller. *Protocol Reverse Engineering Workbench*. Nov. 21, 2022. URL: https://github.com/luelista/pre_workbench (visited on 11/29/2022).

[70] WerWolv. *ImHex*. Apr. 4, 2022. URL: https://github.com/WerWolv/ImHex (visited on 04/04/2022).

[71] *Wireshark · Wireshark-dev: Re: [Wireshark-dev] GUI Functionality from Plugins*. URL: https://www.wireshark.org/lists/wireshark-dev/201209/msg00118.html (visited on 10/09/2022).

[72] *Wireshark Generic Dissector (Wsgd)*. Version 3.6.X. Nov. 22, 2021. URL: http://wsgd.free.fr/index.html (visited on 04/04/2022).

[73] *wxWidgets: Cross-Platform GUI Library*. URL: https://www.wxwidgets.org/ (visited on 08/17/2022).

[74] *Xterm.Js*. URL: http://xtermjs.org/ (visited on 08/17/2022).

ERKLÄRUNG ZUR ABSCHLUSSARBEIT

gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Mira Sophie Weller, die vorliegende Master's Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

THESIS STATEMENT

pursuant to § 22 paragraph 7 and § 23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Mira Sophie Weller, have written the submitted Master's Thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§ 38 paragraph 2 APB), the thesis would be graded with 5.0 and counted as one failed examination attempt. The thesis may only be repeated once. In the submitted thesis the written copies and the electronic version for archiving are pursuant to § 23 paragraph 7 of APB identical in content.

Darmstadt, November 30, 2022

Mira Sophie Weller