NTNU
Norwegian University of
Science and Technology

CSjark

Automated Generation of Protocol Dissectors for Wireshark

Erik Bergersen
Jaroslav Fibichr
Sondre Johan Mannsverk
Terje Snarby
Even Wiik Thomassen
Lars Solvoll Tønder
Sigurd Wien

Group 9
Fall 2011

TDT4290
Customer Driven Project

Abstract

This paper addresses the problem of creating Lua dissectors for Wireshark, to analyze inter-process communication (IPC) with C structs. These dissectors are used to display the binary data in a readable format in Wireshark. Writing a Lua dissector manually is difficult and time consuming, therefore a solution for doing this automatically was necessary.

This problem was solved by parsing C structs defined in C header-files, then processing the abstract-syntax tree, and generating the Lua dissectors for the structs. Using configuration files ensures flexibility in the generation of dissectors.

The project resulted in CSjark, which is a stand-alone utility that acts as a supporting tool for Wireshark. Our utility is written in Python, and uses open source libraries pycparser and PLY to achieve this. For configuration, pyYAML was used to make the utility adaptable. The utility automates the process of generating dissectors for Wireshark from C header-files.

CSjark reduces the time it takes for developers to write dissectors, which will make it easier to utilize Wireshark for debugging of IPC traffic.

PREFACE

This report was written for a project in the course TDT4290 Customer Driven Project at Norwegian University of Technology and Science (NTNU). The project was executed on behalf of Thales Norway AS between the 30th of August and the 24th of November.

The project team consisted of seven students from the department of computer and information science at NTNU. Our task was to develop a tool for Wireshark that could automatically dissect C structs. The utility creates Lua scripts, which act as package dissectors in Wireshark.

The team would like to thank our main supervisor Daniela Soares Cruzes and her assistant Maria Carolina Passos for their continuous input and guidance throughout the project.

We would also like to thank our customer contacts from Thales, Christian Tellefsen and Stig Bjørlykke, for invaluable help and feedback during the development process.

Trondheim, November 24, 2011

Erik Bergersen

Sondre Johan Mannsverk

Jaroslav Fibichr

Even Wiik Thomassen

Lars Solvoll Tønder

Sigurd Wien

Terje Snarby

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Planning & Requirements

CHAPTER 1 INTRODUCTION

This chapter is a technical introduction to our project. It gives a concise explanation of the most important terms used in the report.

The first section briefly explains Wireshark, dissectors and how dissectors are used in Wireshark. The connection between Wireshark and the Lua structs protocol is also explained.

The second section describes how the Lua code works and how it is generated by our utility.

1.1 Wireshark and Dissectors

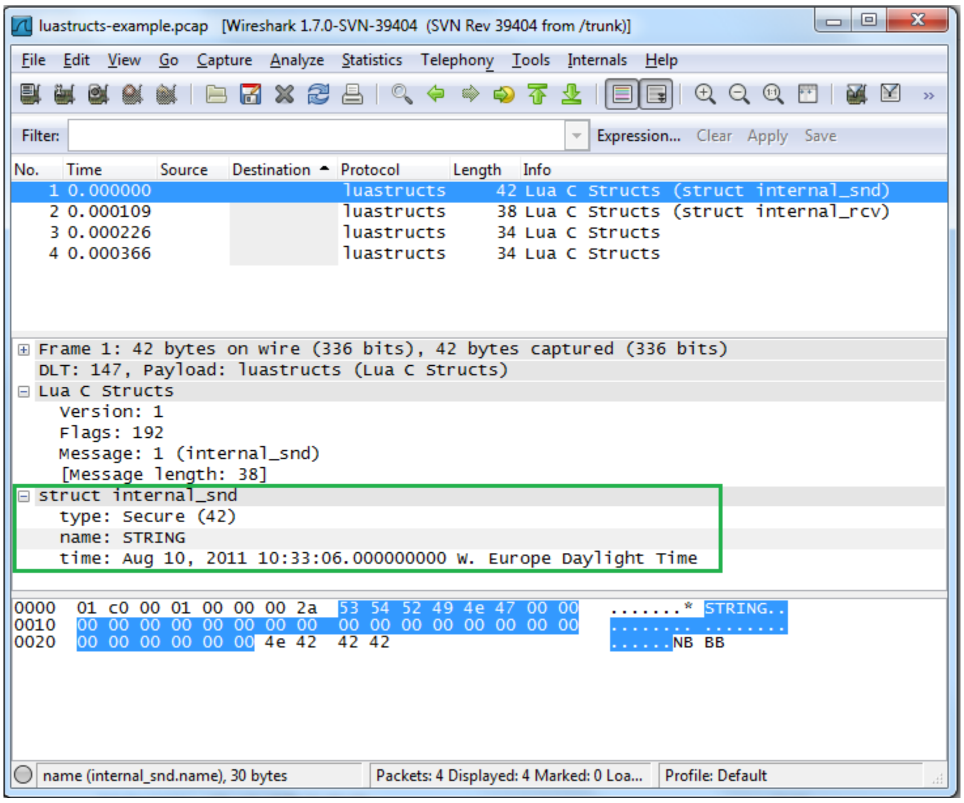
This section gives a brief introduction to Wireshark and dissectors. The first part describes what Wireshark is and what it can be used for. The second part explains exactly what a dissector is, and how a dissector can be used to extend Wireshark.

1.1.1 Wireshark

Wireshark is a program used to analyze network traffic. A common usage scenario is when a person wants to troubleshoot network problems or look at the internal workings of a network protocol. An important feature of Wireshark is the ability to capture and display a live stream of packets sent through the network. A user could, for example, see exactly what happens when he opens up a website. Wireshark will then display all the messages sent between his computer and the web server. It is also possible to filter and search on given packet attributes, which facilitates the debugging process.

In Figure 1.1, you can see a view of Wireshark. This specific example shows a capture file with four messages, or packets, sent between internal processes, in other words it is a view of messages sent by inter-process communication. Each of the packets contain one C struct. To be able to display the contents of the C struct, Wireshark must be extended. This can be accomplished by writing a dissector for the C struct.

Wireshark dissectors can be written in either C or Lua, and in our utility they are written in Lua. The difference between C and Lua dissectors, and the reason we used Lua is elaborated on in the preliminary study in chapter 4. Dissectors, in general, are explained more in detail below.



luastructs-example.pcap [Wireshark 1.7.0-SVN-39404 (SVN Rev 39404 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time      Source  Destination  Protocol     Length  Info
   1 0.000000                       luastructs       42 Lua C Structs (struct internal_snd)
   2 0.000109                       luastructs       38 Lua C Structs (struct internal_rcv)
   3 0.000226                       luastructs       34 Lua C Structs
   4 0.000366                       luastructs       34 Lua C Structs

⊞ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊟ Lua C Structs
    Version: 1
    Flags: 192
    Message: 1 (internal_snd)
    [Message length: 38]
⊟ struct internal_snd
    type: Secure (42)
    name: STRING
    time: Aug 10, 2011 10:33:06.000000000 W. Europe Daylight Time

0000  01 c0 00 01 00 00 00 2a  53 54 52 49 4e 47 00 00   .......* STRING..
0010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0020  00 00 00 00 00 00 4e 42  42 42                     ......NB BB

name (internal_snd.name), 30 bytes | Packets: 4 Displayed: 4 Marked: 0 Loa... | Profile: Default
```

Figure 1.1: Wireshark Screenshot

1.1.2 Dissectors

In short, a dissector is a piece of code run on a blob of data, which can dissect the data and display it in a readable format in Wireshark, instead of the binary representation.

Figure 1.1 displays four packets, with packet number 1 highlighted. The content of the packet is a C struct with three members, type, name and time, and is displayed inside the box in the figure. The C code for the struct is shown in Listing 1.1. The dissector takes the C struct, decodes its binary representation and makes it readable by humans. Without a dissector, Wireshark would just display the struct and struct members as a binary blob.

All the packets containing C structs belong to the protocol called luastructs. When opening a capture file in Wireshark , this protocol maps the id of the messages to the correct dissector, and calls them.

Listing 1.1: Example C Struct

```c
#include <time.h>

#define STRING_LEN 30

struct internal_snd {
    int     type;
    char    name[STRING_LEN];
    time_t  time;
};
```

1.2 From Struct Definition to Lua Dissector

This section explains what happens under the hood of a Lua dissector.

1.2.1 Lua Dissectors

Listing 1.2 shows what the code for the Lua dissector, used to display the content of packet 1 in Figure 1.1, looks like. The Proto variable defines a new protocol. In this example, a dissector for the internal_snd struct, called internal_snd, is created. The different fields of the struct are created as instances of ProtoField, and put into Protocol.fields. For example, the ”name” variable is a string in C, and as such it is created as a ProtoField.string with the name ”name”.

The protocol dissector function is the function that does the actual dissecting. A subtree for the dissector is created, and the description of the dissector is appended to the information column. All the ProtoFields are added to the subtree. Here you can see that the type, name and time fields are added for the internal_snd dissector. The content of the subtree is what is actually displayed when a struct is dissected in Wireshark. The buffer size allocated to the fields is the size of the members in C.

In the last line the dissector is added to the dissector table as a subdissector for the luastructs protocol. When running a capture file, where the internal_snd struct is being sent to another process, it is possible to see the exact contents of the struct. An example of this is shown in Figure 1.1.

Listing 1.2: Example Lua File

```lua
local PROTOCOL = Proto("internal_snd", "struct internal_snd")
local luastructs_dt = DissectorTable.get("luastructs.message")

local types = {[0]="None", [1]="Regular", [42]="Secure"}

local f = PROTOCOL.fields
f.type = ProtoField.uint32("internal_snd.type", "type", nil, ↩
    types)
f.time = ProtoField.absolute_time("internal_snd.time", "time")
f.name = ProtoField.string("internal_snd.name", "name")

function PROTOCOL.dissector(buffer, pinfo, tree)
   local subtree = tree:add(PROTOCOL, buffer())
   pinfo.cols.info:append(" (" .. PROTOCOL.description .. ")")

   subtree:add(f.type, buffer(0,4))
   subtree:add(f.name, buffer(4,30))
   subtree:add(f.time, buffer(34,4))
end

luastructs_dt:add(1, PROTOCOL)
```

CHAPTER 2 PROJECT DIRECTIVE

This chapter will briefly introduce the project, its background and purpose.

2.1 Project Mandate

The purpose of this project was to develop a utility that automatically created Lua-dissectors for Wireshark, from C-header files. This report presents the planning, implementation and testing of the team's product, and documents the process from the initial requirement specification to the finished product.

The title of the project was "Wireshark - Automated generation of protocol dissectors" [10]. It was given to us by the customer and describes exactly what we were planning to accomplish. The name chosen for the utility was "CSjark". Sjark is the Norwegian name for an iconic type of fishing boat, most commonly used in Northern Norway. The reason why the team picked this name was because of the way the utility "fishes" for C structs in header files. The utility then creates dissectors for these structs, in order for Wireshark to display the struct information properly. This reminded the team of what fishermen do to prepare the fish for the market. The word Sjark is also pronounced in a similar way to "shark", which makes our utility name a play on words with "Wireshark", the program our utility was supposed to extend.

2.2 The Client

The client for this project is Thales Norway AS[1]. Thales is an international electronics and systems group, which focuses on defence, aerospace and secu-

[1] `http://www.thales.no/`

rity markets worldwide. The Norwegian branch primarily supplies military communication systems, used by the Norwegian Armed Forces and other members of North Atlantic Treaty Organization (NATO). Thales Norway AS consists of 180 highly skilled employees, which offers a wide range of technical competence [5].

2.3 Involved Parties

Three parties are involved in this project: a) the client, b) the project team, and c) the advisors.

The client, described in the section above, was represented by Christian Tellefsen and Stig Bjørlykke. See Table 3.3 for their contact information. The project team consisted of seven computer engineering students from NTNU, as listed in Table 3.4. For feedback and help during the project period our team was assigned a main advisor, Daniela Soares Cruzes. Daniela was also assisted by Maria Carolina Passos, who also helped us during the project. Their contact information can be found in Table 3.5.

2.4 Project Background

Thales currently uses Wireshark to analyze traffic data between different network nodes, for example, Internet Protocol (IP) packets sent between a client and a server. Thales' programs send C structs internally between processes, and when Thales debug their programs they want to look at the contents of these packets. To be able to use Wireshark for examining packets containing C structs, Wireshark had to be extended with protocol dissectors. Thales could write these manually, but as they have over 4000 C header files with structs, creating dissectors for all of them would take too much time for this type of debugging to be time efficient.

To make debugging inter-process communication in Wireshark viable, the customer wanted us to develop a utility that can generate dissectors automatically. The generated dissectors from this utility will be used by Wireshark's developers when they are solving problems in their programs. Thales therefore expects that our utility can save them valuable time and effort .

2.5 Project Objective

The objective from the customer was to design a utility that would be able to generate Lua code for dissecting the binary representation of C/C++ structs, allowing Wireshark to display, filter, and search through this data. The utility needed to support a flexible configuration, as this would make it more useable for debugging with Wireshark. The code and configuration also had to be well documented, making it easier for Thales to use and extend the tool as they see fit.

The objective from NTNU's point of view was that the team members would acquire practical experience in executing all phases of a bigger Information Technology (IT)-project and learn how to work together in a team.

Our team's goals for this project were to attain experience in working in a real development project, and to create a solution that the customer would be satisfied with.

2.6 Duration

Calculations done by the course staff suggested that each student should conduct 325 person-hours distributed over 13 weeks for the project. Our team, consisting of seven students, would therefore have a total of 2275 person-hours to spend on this project.

- Project start: August 30th
- Project end: November 24th

CHAPTER 3 PLANNING

This chapter explains the administrative part of the project. The project plan is presented in section 3.1. How the project is organized is covered in section 3.2, the quality assurance is described in section 3.3, and how we are supposed to handle possible risk can be found in section 3.4.

3.1 Project Plan

The project plan includes the specified plan on which tool we are using in this project, measurement of project effects, limitations and the concrete project work plan.

3.1.1 Measurement of Project Effects

Automatic generation of Lua scripts from C header files would bring considerable resource savings in the customer's usual work process. When the customer needs to know the contents of inter-process communication messages that include C structs, the use of our utility will save them time, and therefore financial resources.

The biggest impact on savings will be caused by enabling filtering of the messages by specific attributes in the C struct in Wireshark. Once this is made possible, searching for a specific struct or member will become easier.

Before creating the utility, C structs were investigated in two ways. The first, manual method, meant counting individual bytes of the binary file that includes data in C structs. This was possible only for small-sized messages. For bigger messages, this method was inapplicable, as a message can consist of several thousands bytes. The second method consisted of manually writing a dissector in Lua for the specific C header. Also, this method could not be used for more complex C structs, i.e. those using nested structs. At the start of this project there had been written around 10 Lua scripts manually.

According to the customer, they had approximately 3000 C structs in their code base at the beginning of the project. To debug all these structs, 3000 dissectors needs to be written. Time spent writing a dissector for a message manually depends on the struct's complexity. For simple structs, this takes around 15 minutes. It took about one hour for the most complicated dissectors that the customer had developed so far.

If one hour is the average time for creating a dissector for one message, our utility will be able to save around 3000 hours of work, not considering further changes to their code base. Due to everyday workload of the customer's development team, this amount of time could never be used to accomplish such a task.

None of the methods mentioned above are capable of processing messages with C structs that are big and complex, making it difficult to estimate time savings in these cases.

In some cases, a representative of Thales Norway AS has to physically move to a customer's site to solve a problem. With the delivered solution, this can sometimes be avoided, since the customer can use Wireshark with the dissectors, and then send the capture files to Thales. This means that the problem in some cases can be solved in-house. Savings in this case are not only time-based, as this will also directly cut the transportation costs, as well as being an environmental benefit. It is also possible it will increase the satisfaction for the client.

3.1.2 Limitations

As in all other projects, the project members had to deal with various limitations and constraints given either by the customer or personal limitations stemming from things such as lack of experience or conflicts with other personal responsibilities. The limitations as identified at the beginning of the project are listed below.

Technical Limitations

C preprocessor To fulfill all the requirements we might need to either modify an existing preprocessor or write our own, which can be a highly time consuming process.

Platforms Some of the platforms that are required for the utility are not available to the project team.

Non-technical Limitations

Experience No team members have experience with Lua-scripts, running a project with a larger team, or have planned a project before.

Time The project team has a limited time of 13 weeks with a project deadline that cannot be changed. Also, the team consists of seven members that have different schedules so finding a time when everyone is available for a meeting might be difficult. These limitations might lead to considerable delays in the project progress.

Language In this project the team will have to write and speak in English, which is a second language for all team members. This may lead to misunderstandings and will negatively affect the time it takes to write the report.

3.1.3 Tool Selection

To support collaboration and project management the team has considered and selected the listed tools for use in this project.

Git & GitHub

The team selected Git as the version control system, with git repository hosting provided by GitHub[1].

We had experience with Concurrent Versions System (CVS), Subversion (SVN), Git and Mercurial, and although everyone knew SVN and only two knew Git, we selected Git for this project. The main reason we didn't choose SVN was because of its lack of hosting capabilities, and the other reason was that, unlike Git and Mercurial, SVN does not have any of the advantageous features of a modern version control system like branches and a distributed repository model.

We evaluated free hosting sites of version control systems, which could also provide us with other collaborative features that we wanted. GitHub, Bitbucket[2] and SourceForge[3] all provided wiki and issue tracker in addition to free version control system hosting. We eliminated SourceForge because their focus is divided between software users and developers, while the other two sites are fully focused on developers. The two remaining sites provides almost identical features, where one focuses on Git and the other on Mercurial.

GitHub with Git version control system was selected because more team members had experience with Git than Mercurial. Since we use different

[1]`http://github.com/`
[2]`http://bitbucket.org/`
[3]`http://sourceforge.net/`

platforms, we will also use different git clients, but for Windows most of the team has selected tortoisegit.

Skype

Skype[4] is an application which allows the user to make video and voice chats over the Internet, including conference calls and chatting. The team used Skype to communicate and collaborate when we were not physically present at the same location at the same time.

Google Docs

Google Docs[5] is a free web site offering functionality for creating documents, spreadsheets and presentations. The benefits of using Google Docs are that it is easy to share documents with other users, and it is possible to collaborate in real-time. For this project we used Google Docs to collaborate on document drafts, and to share documents within the team and with the team's advisor.

LaTeX

LaTeX is a document markup language used to create reports, articles and books. It was chosen by the team for its high quality typesetting which produces professional looking documents, and because it is suitable for larger scientific reports [13]. Writing documents in LaTeX is very different from writing them in, for example, Microsoft Word, as most of the visual presentation is handled by the LaTeX system and not by the user itself. Because the writer does not have to spend time thinking about how the document looks, he can focus entirely on the content. LaTeX also provides automatic numbering of chapters and sections, automatic generation of table of contents, cross-referencing and bibliography management. Since LaTeX files are plain text files they are suitable for versioning with a version control system like Git. We will use LaTeX to write the final project report, and we have created a few templates for test plans and minutes.

Mailing List

NTNU – Trondheim
Norwegian University of
Science and Technology

For asynchronous communication the team used an electronic mailing list provided by Institute for Computer Science and Information Technology (IDI), NTNU.

[4] `http://www.skype.com`
[5] `http://docs.google.com/`

Google Calendar

Since all team members have Google accounts, we created a team calendar in Google Calendar[6] to help schedule and keep track of meetings. A single calendar that all members can include in their own prevents misunderstandings and duplication of work.

text2pcap

Since the customer could not provide us with capture files, we had to create them ourselves. The capture files are important for testing the generated Lua-scripts. In this project text2pcap[7] was used to generate capture files from ASCII[8] hex dumps. The text2pcap tool is included with Wireshark. The input to the tool is an ASCII hex dump as a text-file, and the output will be a pcap-file.

Hex Editor

The team used a hex editor to create input for text2pcap. To make it easier to write ASCII hex dumps, it was deemed necessary to write them in a hex editor. HxD[9] was the recommended hex editor for this project, as it is free and has all of the necessary functionality.

Violet

Violet[10] is a free and easy to use modeling software for making Unified Modeling Language (UML)-diagrams. It is also a cross platform solution, which means that all team members can use the same application. If we had to use different UML applications there could be a problem editing the diagrams due to incompatible file formats. The architectural and design team used Violet to create diagrams to illustrate the workings of the utility. As very advanced diagrams were not needed for this project, Violet seemed like a fitting tool.

3.1.4 Schedule of Results

This project had two deliveries,a pre-delivery and a final delivery. The milestones and sprints are listed below.

[6]`http://calendar.google.com/`
[7]`http://www.wireshark.org/docs/man-pages/text2pcap.html`
[8]`http://www.asciitable.com/`
[9]`http://mh-nexus.de/en/hxd/`
[10]`http://violet.sourceforge.net/`

Milestones

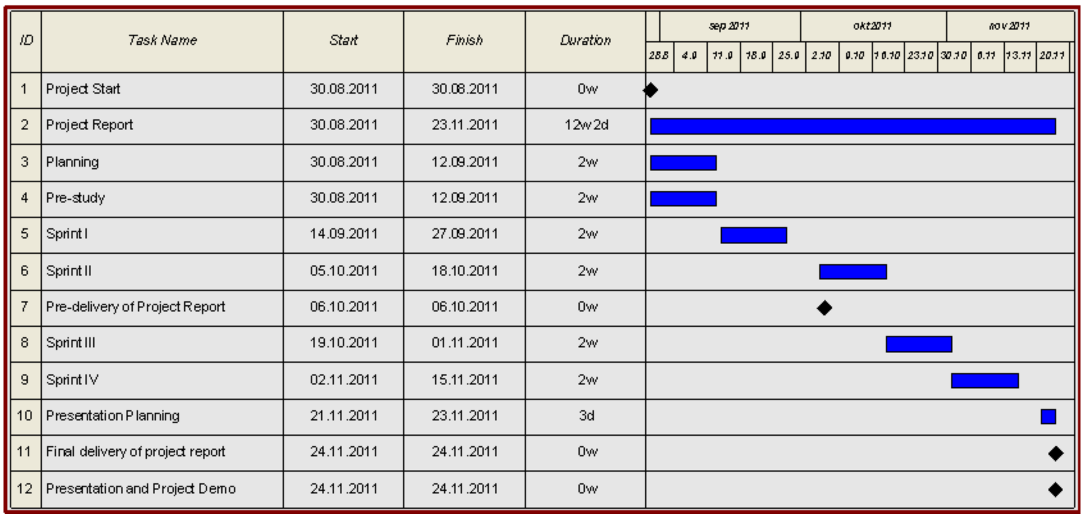
30. August	Project start
6. October	Pre-delivery of project report
24. November	Final delivery of project report
24. November	Presentation and project demo

Sprints

Sprint 1	14. September - 27. September
Sprint 2	5. October - 18. October
Sprint 3	19. October - 1. November
Sprint 4	2. November - 15. November

3.1.5 Concrete Project Work Plan

The first two weeks of the project was used on planning and pre study. The project was divided into four sprints that each lasted for two weeks. The first sprint had an estimated length of 200 person-hours, while the last three sprints had an estimate of 250 person-hours. The last one and a half weeks were used to finish the final report and prepare for the presentation. Table 3.1 shows the work breakdown structure, and the project timeline is in Figure 3.1.



ID	Task Name	Start	Finish	Duration
1	Project Start	30.08.2011	30.08.2011	0w
2	Project Report	30.08.2011	23.11.2011	12w 2d
3	Planning	30.08.2011	12.09.2011	2w
4	Pre-study	30.08.2011	12.09.2011	2w
5	Sprint I	14.09.2011	27.09.2011	2w
6	Sprint II	05.10.2011	18.10.2011	2w
7	Pre-delivery of Project Report	06.10.2011	06.10.2011	0w
8	Sprint III	19.10.2011	01.11.2011	2w
9	Sprint IV	02.11.2011	15.11.2011	2w
10	Presentation Planning	21.11.2011	23.11.2011	3d
11	Final delivery of project report	24.11.2011	24.11.2011	0w
12	Presentation and Project Demo	24.11.2011	24.11.2011	0w

Figure 3.1: Gantt Diagram

3.2 Project Organization

This section describes how the team was organized, which roles the developers were divided into and the partners of the project.

Table 3.1: Work Breakdown Structure

			Effort	
Task	From date	To date	Estimated	Actual
Misc	30.08.2011	24.11.2011	825	855
Project Management	30.08.2011	24.11.2011	275	454
Lectures	02.09.2011	18.10.2011	100	100
Self Study	30.08.2011	04.10.2011	100	71
Planning	05.09.2011	12.09.2011	150	122
Pre-study	05.09.2011	12.09.2011	100	49
Requirement Specification	05.09.2011	12.09.2011	100	59
Sprint 1	14.09.2011	27.09.2011	200	157
Sprint 1 Planning	14.09.2011	14.09.2011	30	29
Sprint 1 Work	15.09.2011	26.09.2011	150	98
Sprint 1 Review	27.09.2011	27.09.2011	20	30
Sprint 2	05.10.2011	18.10.2011	250	260
Sprint 2 Planning	05.10.2011	05.10.2011	30	36
Sprint 2 Work	06.10.2011	17.10.2011	200	206
Sprint 2 Review	18.10.2011	18.10.2011	20	18
Sprint 3	19.10.2011	01.11.2011	300	296
Sprint 3 Planning	19.10.2011	19.10.2011	30	48
Sprint 3 Work	20.10.2011	31.10.2011	250	234
Sprint 3 Review	01.11.2011	01.11.2011	20	14
Sprint 4	02.11.2011	15.11.2011	300	311
Sprint 4 Planning	02.11.2011	02.11.2011	30	34
Sprint 4 Work	03.11.2011	14.11.2011	250	263
Sprint 4 Review	15.11.2011	15.11.2011	20	14
Report & Presentation	30.08.2011	24.11.2011	400	452
Write Report	20.08.2011	24.11.2011	325	388
Presentation	22.11.2011	24.11.2011	75	64
Total	30.08.2011	24.11.2011	2275	2331

3.2.1 Project Organization

Our project organization had a flat structure, and the organization chart can be seen in figure 3.2. The roles listed in the organization chart are described in table 3.2.

3.2.2 Partners

This subsection lists the partners of this project. The customer of this project is Thales Norway AS, which is located at Lerkendal Stadium, Strindveien 1, 7030 Trondheim. The customer contacts are listed in Table 3.3. The development team consist of seven student from NTNU, and is listed in Table 3.4.

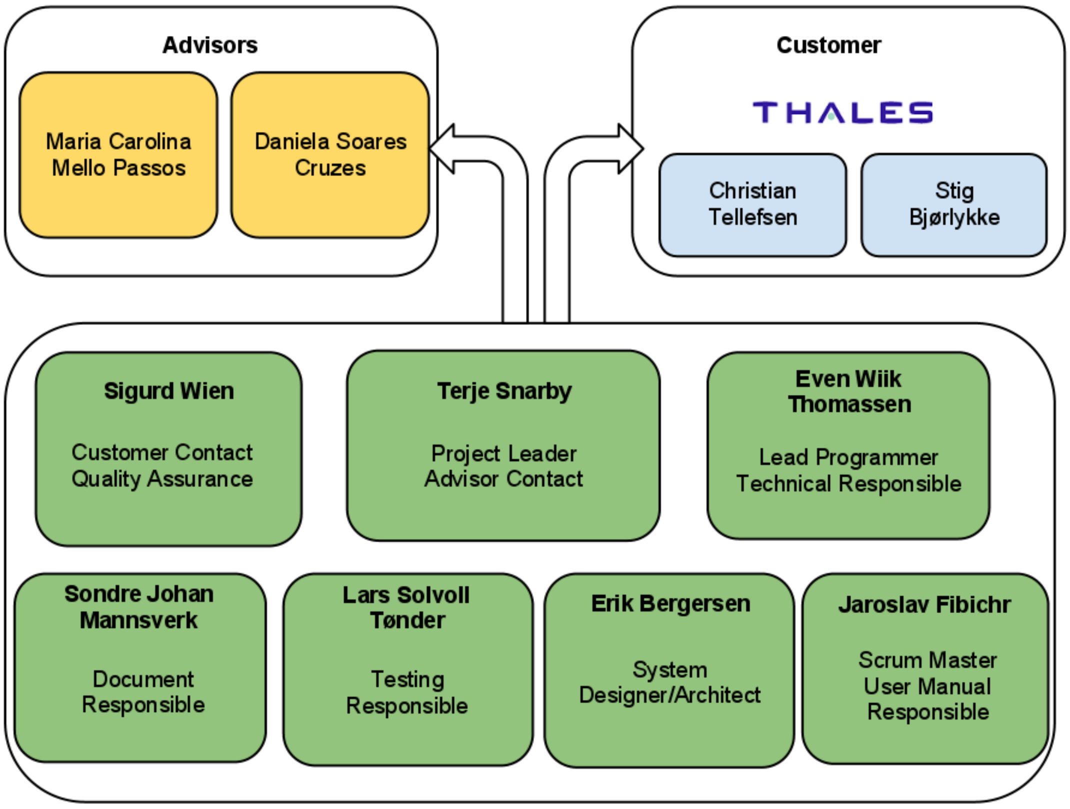


Figure 3.2: Project Organization

The team is assigned two advisors from the Department of Computer and Information Science at NTNU, listed in Table 3.5.

3.3 Quality Assurance

The following section contains internal processes and routines the team used in the project. This includes procedures for meetings, document templates and standards and internal reports.

3.3.1 Routines for Ensuring Quality Internally

We decided to organize in pairs when producing items, where the pair reviews each others work. This would be done in an effort to enhance the quality of the project, as we would be able to find more errors, and also get a broader perspective on style and solutions.

We also assigned quality assurance responsibilities for three articles: documents, code and tests. The respective team members tried to have a bird's

Table 3.2: Project Roles

Role name	Main responsibilities
Project manager	Responsible for having an overview of the project, delegating tasks and resolving conflicts.
Advisor contact	Responsible for distributing information between the team and the advisor.
Organizer	Responsible for setting up and informing the team about the meeting schedule.
Document master	Responsible for document quality and quantity.
System architect	The lead designer of the system.
Lead programmer	Makes sure everyone follows the agreed code standards, and ensures the quality of the code.
Customer contact	Responsible for distributing information between the team and the customer.
Technically responsible	Finds suitable technical solutions and makes sure that the essential tools are operative.
Technology evangelist	Brings in ideas about new technologies and tools.
Scrum master	Responsible for Scrum meetings.
Lead tester	Responsible for test coverage, both unit and end to end, and to ensure the quality of those tests.
Secretary	Takes note from each meetings and stores it in the cloud. Responsible for preparing minutes for advisor/customer.

Table 3.3: Customers

Name	Mobile	E-mail
Christian Tellefsen	959 98 765	christian.telefsen@thalesgroup.com
Stig Bjørlykke	982 29 806	stig.bjorlykke@thalesgroup.com

Table 3.4: Developers

Name	Mobile	E-mail
Terje Snarby	915 27 390	snarby@stud.ntnu.no
Even Wiik Thomassen	991 61 929	evenwiik@stud.ntnu.no
Sondre Johan Mannsverk	948 15 506	sondrejo@stud.ntnu.no
Erik Bergersen	917 48 305	eribe@stud.ntnu.no
Lars Solvoll Tønder	976 00 317	larssot@stud.ntnu.no
Sigrud Wien	472 54 625	sigurdw@stud.ntnu.no
Jaroslav Fibichr	451 26 314	jaroslaf@stud.ntnu.no

Table 3.5: Advisors

Name	Mobile	E-mail
Daniela Soares Cruzes	942 49 891	dcruzes@idi.ntnu.no
Maria Carolina Mello Passos	483 49 117	mariacm@idi.ntnu.no

eye overview in their area to catch further errors.

We agreed to have three weekly meetings to accommodate these routines.

- Monday 12-14
- Wednesday 12-17
- Friday 10-13

3.3.2 Phase Result Approval

To ensure the quality of the sprint deliverables, it was decided that at least one team member would go through the work of another before it is delivered.

We would also present the results to the customer and advisor. They would then have the opportunity to point out problems and misunderstandings, and suggest solutions. This was a result of the Scrum methodology: deliveries and deadlines throughout the project, making the progress very visible to the customer and advisors. We reckoned that we should be able to attain success at the end of the project because of the guidance and feedback received during the process of making the utility and report. If a problem appears, we will try to correct it and then reiterate the quality assurance.

3.3.3 Procedures for Customer Meetings

All customer meetings were to be scheduled with time, place, agenda specified. All background documents relevant to the meetings should also be supplied. This was to ensure efficient and effective meetings.

All customer meetings should be summarised in a document (minute). This document should include:

- Time of meeting
- Place
- Version
- Meeting responsible
- Names of the attendees
- Decisions
- Actions
- Clarifications
- The above should be in sequence according to time

This document was to be written and sent to the customer by 12:00 the day after the meeting. If the customer did not approve the minutes, the minutes would again be corrected and sent 12:00 the following day. The customer contact was responsible for the above tasks.

The customer committed to respond to our interactions within two working days.

3.3.4 Procedure for Advisor Meeting

The weekly advisor meeting will be at 10:30 every Friday unless otherwise stated.

Agenda for Meeting

A meeting with the advisor should be scheduled before 14:00 the day before the meeting, and this schedule should include:

- Time
- Place
- Agenda
- Status report
- Table of reported working hours
- Minutes for the last meeting
- Other relevant documents

Minutes from Meeting

The minutes should be written and sent to the advisor for approval before 12:00 the next work day after the meeting. If the advisor should reject the minutes, they should be corrected and re-sent 12:00 the day following the rejection. The minutes should include:

- Time of meeting
- Place
- Version
- Names of the attendees
- Decisions
- Actions

- Clarifications
- A rough timeline of the above

3.3.5 Document Templates and Standards

The team has specified procedures for templates and file organization.

Templates the Team Has Created

All templates were stored under docs/ on GitHub. The team had templates for:

- Meeting agenda
- Status report
- Meeting minutes

Standard for Organizing Files

We use GitHub and Google Docs to store the files included in this project. The location of a file is dependent on what type the file is.

- All source code is to be saved in the GitHub repository under CSjark/. This ensures that all the team members have the current version of the code. The structure for this folder:

```
CSjark/
    csjark/  -- today's source/, for source code
        test/  -- for unit tests
        etc/  -- for configuration files
        header/  -- header files used to test the program
    bin/  -- file for executing our program
    docs/  -- for CSjark-specific documentation
    utils/  -- for cpp.exe and fake header files
```

- All textual documents that are completed will be put in the docs/ folder.

- All LaTeX documents are stored in the GitHub repository under report/. The structure for this folder:

```
report/
    planning/  --   Planning & Requirements section
        img/  --  Images for this section
    sprints/  --  Sprint sections
        img/  --  Images for this section
    evaluation/  --  Evaluation section
    appendices/  --   Appendices section
```

- Examples of header-files and Lua-scripts, packet capture-files and information from customer is stored under Wireshark/.
- All documents or python code should compile before it is pushed to the repository

File Name Standard

The file name should consist of the name of the document (meeting minutes, agenda, phasedoc, e.g.) and the date, if applicable.

Coding Style Standard

The programming language used to implement the utility specified by the customer requirements was Python. The coding style the team agreed upon was the Python Standard Styling Guide as defined by PEP8[11]. In addition it was decided that the design should attempt to be pythonic, as detailed by PEP20[12].

3.3.6 Version Control Procedures

The team decided that every relevant digital item should be pushed to our repository at GitHub, and be checked out by other participants. Those who worked on a given item were supposed to commit and push their changes often, so that others could be as up to date as possible. All digital items were to be labeled with a version number, starting at version one. If an item went under review and was deemed insufficient by the customer, the version number was to also be incremented by one for each revision of the document

Relevant digital items includes source code, documents, picture files, binary blobs, etc.

NB: Google docs was not to be used for version control, so every document written there was also to be pushed to git hub.

[11]Style Guide for Python Code: http://www.python.org/dev/peps/pep-0008/
[12]The Zen of Python http://www.python.org/dev/peps/pep-0020/

3.3.7 Internal Reports

Some of the internal activities in the team should be documented. This includes:

- Activities, what is done, and what remains
- Minutes for internal meetings
- Milestones, complete/incomplete
- Effort registration shall be done daily by each team member.
- Sprint backlog should be updated daily by each team member.

These documents should follow the templates specified in Templates and Standards (A4) if applicable.

3.4 Risk Management

The following section lists the possible risk scenarios that could occur in the project, and how they were to be handled. Table 3.6 shows how to handle the possible risks in the team. Each risk has a consequence and a probability: High (H), Medium (M) or Low (L). Strategy and actions describes what we were supposed to do to reduce the consequences of the risk, or prevent the risk from happening altogether. Deadline states when we needed to handle the risk.

R1. Choosing an incompatible technical solution The team decides to use a technical solution that is not suited for the given problem, or decides on an implementation that is too time consuming.

R2. Too much focus on report We spend too much time working on the report and neglect the implementation.

R3. Too much focus on implementation We spend too much time working on the implementation and neglect writing all the needed documentation for the report.

R4. Illness/Absence Members of the team become ill or are otherwise unavailable.

R5. Key member is absent A member that has an important responsibility becomes ill.

R6. Conflicts within team Internal conflicts which hinders the team's ability to work together.

R7. Lack of technical competence The team lack the needed technical ability to solve the given problem.

R8. Miscommunication within team Team members don't know what to do, or misunderstands the task given to them.

R9. Miscommunication with customer The team misunderstands the requirements given by the customer.

R10. Lack of experience with Scrum The team does not have any experience in doing Scrum projects.

R11. Requirements added or modified late in the project The customer asks us to implement a new, and possibly time consuming requirement, or modifies a requirement in such a way that it needs to be reimplemented, late in the project.

Table 3.6: Handling Risks

Risk ID	R1
Risk factor	Choosing an incompatible technical solution
Consequences	H: The project will not be completed on time, or at all.
Probability	M
Strategy & actions	Do a good pre-study, consult the customer's technical expert.
Deadline	During the first sprint.
Responsible	Even and Erik
Risk ID	R2
Risk factor	Too much focus on report
Consequences	M: The product will not be of a satisfying quality.
Probability	M
Strategy & actions	Plan enough hours to use on the customer product.
Deadline	Continuous
Responsible	Sondre
Risk ID	R3
Risk factor	Too much focus on implementation
Consequences	H: The documentation will not be good enough, leads to a bad grade.
Probability	M
Strategy & actions	Plan enough hours to use on the report. Write documentation in parallel with implementation when it is possible. Write good requirements that limit the scope of the project.
Deadline	Continuous
Responsible	All
Risk ID	R4

Table 3.6: Handling Risks

Risk factor	Illness/Absence
Consequences	L/M/H: Consequences depend on how many members are absent, and how often they are absent. Absence may hinder the progress of the project in different ways.
Probability	M
Strategy & actions	Make sure several people are proficient in the technical parts of the project. Have backups for the most important roles.
Deadline	Continuous
Responsible	Terje
Risk ID	R5
Risk factor	Key member is absent
Consequences	H Absence of a key member may greatly hinder the team's process during the period of the absence.
Probability	L
Strategy & actions	The team should be updated on the work of key members, so that a team member they can step in for the key member on important tasks.
Deadline	Continuous
Responsible	All
Risk ID	R6
Risk factor	Conflicts within team
Consequences	M: May lead to bad morale, which could affect the work of the team. Could also be a waste of time.
Probability	M
Strategy & actions	Not all conflicts are bad. If the conflict is simply a disagreement over technical issues, or the planning of the project, it could benefit the team. All such conflicts should lead to a constructive discussion that the entire team should take part in. Other types of conflicts, that can not positively influence the project should be avoided if possible. The team should agree on specific ground rules.
Deadline	Continuous
Responsible	Terje
Risk ID	R7
Risk factor	Lack of technical competence
Consequences	H: The team is unable to solve the problem
Probability	H
Strategy & actions	Make sure the team is proficient in the programming languages and tools that are to be used. Decide on technical solutions that the team is already familiar with. Do a good pre-study of the parts the team is unfamiliar with. Consult with the customer's technical expert.
Deadline	Continuous
Responsible	All

Table 3.6: Handling Risks

Risk ID	R8
Risk factor	Miscommunication within team
Consequences	M: Team members waste time doing nothing, or doing something that is irrelevant.
Probability	M
Strategy & actions	Make sure that everyone knows what to do at all times. Ask questions if you are unsure about your specific task.
Deadline	Continuous
Responsible	Terje
Risk ID	R9
Risk factor	Miscommunication with customer
Consequences	H: The team waste time on functionality the customer did not want. The delivered product does not do what the customer asked for.
Probability	M
Strategy & actions	Make sure we and the customer have a common understanding of the requirements. Have frequent meetings with the customer, with a weekly demo of new features.
Deadline	Continuous
Responsible	Sigurd
Risk ID	R10
Risk factor	Lack of experience with Scrum
Consequences	M: The team does not provide the correct documents for the report, which could lead to a bad grade.
Probability	M
Strategy & actions	Learn how to properly do Scrum. Have Scrum meetings as often as possible. Get feedback on documents from advisor.
Deadline	Continuous
Responsible	Jaroslav
Risk ID	R11
Risk factor	Important requirements added or modified by customer late in the project
Consequences	M: The team may spend time on implementation when we instead should be finishing up the report, or prepare for the presentation.
Probability	H
Strategy & actions	Have a good dialog with the customer, and be prepared to say no to new requirements if we do not have the time to complete them.
Deadline	Continuous
Responsible	Jaroslav

CHAPTER 4 PRELIMINARY STUDY

This chapter presents the preliminary study for this project. In section 4.1 we have examined existing solutions, and in section 4.2 we provide a description of two popular software development methodologies.

Wireshark, which our utility should create dissectors for, are described in section 4.3. Section 4.4 contains the different programming languages we might use, while section 4.5 describes possible solutions for parsing C header files. Section 4.6 outlines possible configuration libraries, and section 4.7 discusses possible unit testing frameworks. In section 4.8 we describe tools for creating user documentation and section 4.9 describes three integrated development environments.

Section 4.10 provides the justifications for the choices we have made, and in subsection 4.10.6 we describe the framework our utility will require. At the end of the chapter, in section 4.11, we describe the license of our utility.

4.1 Similar Solutions

We started by searching for existing solutions in the problem space. This search turned up idl2wrs[1]. The other solution was suggested by our customer, Asn2wrs[2]. Both of these solutions are bundled with Wireshark.

4.1.1 idl2wrs

A tool for generating Wireshark dissectors from Interface Description Language (IDL) files. The tool is written in Python, and generates dissectors in C from IDL specifications. IDL is used as an interface to enable communication between software of different languages, in a language-neutral way. It is

[1]`http://wiki.wireshark.org/idl2wrs`
[2]`http://wiki.wireshark.org/Asn2wrs`

used for example in Sun RPC and Common Object Request Broker Architecture (CORBA). Since idl2wrs takes input in a different language than our utility will, and creates dissectors in a different language than our utility, we can not reuse any of its code. Instead we will look at its architecture and data structures, especially how it generates dissectors.

4.1.2 Asn2wrs

Is a tool for generating Wireshark dissectors from Abstract Syntax Notation One (ASN.1) protocols. Asn2wrs requires four input files: an ASN.1 protocol description, a configuration file and two template files. Advantages of using Asn2wrs are faster development because of easier recompilation, and plugins that are easy to distribute. The disadvantage is that code and makefiles are more complex [2].

Our customer cannot use this solution as it would require them to rewrite their C structs to ASN.1 descriptions, which would take a very long time. But the team can use the asn2wrs code as an example of how to create dissectors for Wireshark.

4.2 Software Development Methodology

In this section we describe two popular software development methodologies, while subsection 4.10.1 discusses which one we decided to use, and why.

4.2.1 Waterfall

Waterfall [6] is a software development methodology based on sequential phases. It consist of the following phases: requirement specification, design, implementation, integration, testing, deployment, and maintenance. In its pure form, these phases are non-overlapping and one way only, which means that each phase must be fully completed before the next can begin. Following the phases are listed in sequentially order.

Requirement specification Receiving requirements from a customer and then formalising these into concrete functional and non-functional requirements. These will again be further broken down into smaller work items that are easy to quantify in terms of time of use and importance. These metrics may help distinguish which features are to be prioritised.

Design The design is about planning how to implement the features from the requirement specification. The goal is to make a precise software architecture for the project that dictates most of the implementation phase. This may include (but not limited to) making class diagrams, data flow diagrams, state machines, user interface mock-ups, etc.

Implementation Implementing and coding the design made in the design phase on a component level.

Integration Integrating the different components that results from the implementation phase.

Testing Thoroughly test the result of the implementation and integration. The goal is to find and fix bugs introduced in these phases.

Deployment Delivering the resulting software to the customer. This may include installing the software on their systems. This is also the phase where the customer either accepts or rejects the resulting software.

Maintenance Large software projects are almost impossible to make completely bug free, and therefore a certain amount of maintenance may be required. The obvious tasks are to either fix or provide viable workarounds for problems that appear during normal use. Maintenance may also include developing new features that the customer finds the need for.

4.2.2 Scrum

Scrum [16] is an agile development methodology based on the philosophy that it is impossible to completely and accurately plan everything in a software project before you begin. It is therefore more or less based on iterations of the waterfall phases described in subsection 4.2.1, but instead of having these phases being strictly sequential, they are run in a more 'as needed' basis. Each iteration in Scrum is called a sprint and typically lasts between two and four weeks. This time period is fixed for each project, so the sprint will always end on time. To make this possible, features that are not completed on time is deferred to a later sprint. Each sprint should result in a runnable product that potentially could deliver some value to the customer, even if this requires some redundant work.

Main Scrum Roles

Scrum Master has the responsibility of maintaining the process and for removing obstacles for other team members. In short, the Scrum master tries to keep the other team members focused on their tasks.

Product Owner represents and speaks for the customer. Not necessarily a part of the customer's organization, but must have the stated authorities.

Team members are responsible for creating and delivering the product. Should consist of a self organizing team of five to nine persons with a cross functional skill set.

Scrum Artifacts

Product backlog contains a high level description of all the desired features for the project. These should be prioritised based on their business value and evolve along with the project.

Sprint backlog contains what the team is committed to complete over the next sprint. These commitments are features broken down into work items. These items should not be larger than 16 hours of work, and they should be described so that everyone in the team could contribute to implementing them.

Burn down chart A daily updated chart consisting of what work remains in the sprint. Its purpose is both to show what work to do next and to give a visual representation of the work progress.

A sprint begins with the sprint planning meeting, which consists of two stages. In the first, the team and the product owner prioritizes the product backlog. In the second, the team discusses what features they can commit to, based on priority, and break these down into work items, which are added to the sprint backlog. This should include giving each item an estimated completion time.

The sprint itself consists of producing what is required for completing work items, updating the burn down chart, and daily Scrum meetings. In these daily meetings each team member provides a short update of what they did the day before, what they plan to do today and what problems might be in their way. These problems should not be discussed in this meeting, but rather dealt with separately after the meeting, which is the Scrum master's responsibility.

At the end of the sprint cycle, the team should hold a Scrum review meeting. In this meeting the team should discuss what was completed and what was not, and demonstrate the completed features for the customer.

After the review meeting, a separate retrospect meeting should be held with all the team members, where all members share their reflections of how the sprint went and on how we could improve for the next sprint. This is important for improving the process.

4.3 Wireshark

Wireshark[3] is a free, open source network protocol analyzer. It lets you capture and browse traffic running through a computer network. Wireshark is currently being developed by the Wireshark team, a group of networking experts spanning the globe [7]. Because of its rich set of features and ease of use, Wireshark is the de facto standard in many different industries and the educational community. Wireshark is able to dissect and display data from a plethora of different protocols. One of its strengths lies in the ease of which developers can add their own dissectors, post-dissector and taps.

Dissectors can be written in either C or Lua. Most dissectors are written in C for increased speed. Lua-scripts are mostly used as prototypes or to process non time crucial data as they don't need compilation to be used. Our customer uses Wireshark not only to browse through and filter regular networking traffic, but also for monitoring inter-process communication where it is important to have a tool that can easily be extended to dissect and display structures and data types unique to the organization.

Our utility should read C header files and create Wireshark dissectors written in Lua for structs found in the header files.

4.4 Programming Languages

The dissectors we have to generate are written in Lua, and we have looked at both Java and Python programming language for our utility. In this section we describe these different languages. In subsection 4.10.2 we describe which language we selected and why.

4.4.1 Lua

Lua[4] is a multi-paradigm, dynamically typed programming language that is designed to be lightweight, so it can easily be embedded into applications. Lua has only a few basic data structures: boolean, numbers, strings and table. Still Lua implements advanced features such as first-class functions, garbage collection, closures, coroutines and dynamic module loading. Lua was created in 1993 at the Pontifical Catholic University of Rio de Janeiro, in Brazil [8].

The output of our utility will be Wireshark dissectors written in Lua. While Wireshark supports dissectors written in both C and Lua, Lua is

[3] `http://www.wireshark.org/`

[4] `http://www.lua.org/`

preferred because they can be added without recompiling Wireshark. This is important since some of Thales customers do not allow recompiled versions of Wireshark. Lua dissectors interface with Wireshark through a simple Application Programming Interface (API).

4.4.2 Java

Java[5] is an object-oriented, structured, imperative, statically typed programming language. It was originally developed by Sun Microsystems, which is now a subsidiary of Oracle Corporation. Java was released in 1995, and it derived much of its syntax from C and C++, but with fewer low-level facilities. Java's strength are portability, automatic memory management, security, good documentation and an extensive standard library [3]. Java has several tools and libraries of varying quality for creating parsers, for example ANother Tool for Language Recognition (ANTLR) and Java Compiler Compiler (JavaCC). A detailed description of ANTLR can be found in subsection 4.5.1.

4.4.3 Python

Python[6] is a general-purpose, multi-paradigm, object-oriented, imperative, dynamically typed programming language. It was created by Guido van Rossum, and is today developed by Python Software Foundation and the Python community. Python's strength include automatic memory management, large and comprehensive standard library, portability, powerful but very clear, concise and simple syntax [1]. There exists several pure Python libraries for creating lexers and parsers, like Python Lex-Yacc (PLY), pycparser and cppheaderparser. These are described further in section 4.5.

4.5 Parsers Libraries & Tools

This section contains various tools and libraries we have looked at for solving the challenge of parsing C header files. They range from language-independent tools like GNU Compiler Collection (GCC) and Clang to Python-only libraries like PLY and pycparser. The justification for the libraries we selected can be found in subsection 4.10.3.

[5]`http://java.com/`
[6]`http://www.python.org/`

4.5.1 ANTLR

ANTLR[7], ANother Tool for Language Recognition, is a compiler toolkit for creating lexers and parsers from grammar files. It can create these compilers for several different target languages, including Java and Python. There exists ANTLR grammar files for the challenges we are facing: parsing C, C preprocessor step and parsing ASN.1. These grammars configure ANTLR to create Java lexers and parsers that reads and validates inputted source code files.

4.5.2 PLY

PLY[8] is a Python alternative to the popular lexer and parser compilers lex and yacc. It also comes with a 95% completed C preprocessor in case we are required to modify the preprocessor for our utility. Other special purpose parsers like pycparser and cppheaderparser depends on PLY. These are described later in this section.

4.5.3 pycparser

There are two Python libraries for parsing C with the same name, but different capitalization, pycparser[9] and PyCParser[10]. While they both aim to solve almost the same problem, the first one appears to have better documentation, is a more mature project and support more of the C99 specification. pycparser requires PLY to work.

4.5.4 cppheaderparser

Cppheaderparser[11] is a parser for C++ header files written in Python. It is an alternative for pycparser in case we need to parse C++ files instead of simple C header files. It also depends on PLY.

4.5.5 GCC

GNU Compiler Collection[12] (GCC) is a compiler system, which has front ends that parse C and C++ code, and is written in C and C++. It can be used in our utility as an external tool that does the parsing and then outputs an intermediate language representation, which we can parse/search to find the C struct definitions.

[7] `http://www.antlr.org/`
[8] `http://www.dabeaz.com/ply/`
[9] `http://code.google.com/p/pycparser/`
[10] `https://github.com/albertz/PyCParser`
[11] `http://sourceforge.net/projects/cppheaderparser/`
[12] `http://gcc.gnu.org/`

Its drawbacks are a lack of flexibility if we need to change its behaviour, and we will still need to write a custom parser or use something like GCC-XML and an Extensible Markup Language (XML) parser.

4.5.6 Clang

Clang[13] is a compiler front end for C, C++, Objective-C and Objective-C++, written in C++. Clang differ from GCC as it behaves as a library rather than an external tool, but for Java we will have to use it like GCC because there are no Java-Clang bindings. It supports outputting the abstract syntax tree as XML, which our utility then will need to parse. Clang provides bindings for Python so it can be used as a library, but its main drawback is, like GCC, a lack of flexibility. Clang is a part of the LLVM toolkit.

4.6 Configuration Frameworks

This section looks at different configuration frameworks for Python. Which we selected and why is explained in subsection 4.10.4.

Our utility needs a flexible configuration, as some of the information we shall display does not exist in the files we parse. For example there are no clear relationship between enumerated values in messages and their names. These must be provided through a configuration.

4.6.1 YAML Ain't Markup Language

YAML Ain't Markup Language (YAML)[14] (YAML Ain't Markup Language) is a data serialization format. It is designed to be easy to read and write for humans. YAML syntax was designed to be easily mapped to data types common to most high-level languages. While most programming languages can use YAML for data serialization, YAML excels in working with those languages that are fundamentally built around the three basic primitives. These include the new wave of agile languages such as Perl, Python, PHP, Ruby, and Javascript.

PyYAML[15] is a YAML parser for the Python programming language, and it is available for both the 2.x and 3.x branch of Python. It is licensed under the Massachusetts Institute of Technology (MIT) license.

[13]http://clang.llvm.org/
[14]http://yaml.org/
[15]http://pyyaml.org/

4.6.2 configparser

configparser[16] is a Python module used for managing user-editable configuration files. The files are organized into sections, and each section can contain name-value pairs for configuration data. Value interpolation using Python formatting strings is also supported, to build values that depend on one another.

configparser module is a part of Python standard library, and therefore does not require installation or configuration to use.

4.6.3 ConfigObj

ConfigObj[17] is a simple but powerful config file reader and writer (originally based on ConfigParser). Its main feature is that it is very easy to use, with a straightforward programmer's interface and a simple syntax for config files. Among others, it has these additional features:

- Nested sections (subsections), to any level
- List values
- Multiple line values
- String interpolation (substitution)
- Integrated with a powerful validation system

Currently, ConfigObj module only exists for Python up to version 2.7. It is under the Berkeley Software Distribution (BSD) license.

4.7 Unit Testing Frameworks

There are many different unit testing frameworks for Python. We have evaluated three of them to see which best suits our utility, which we describe in this section. In subsection 4.10.5, we describe which one we selected and why.

4.7.1 py.test

py.test[18] is a mature, full-featured testing tool. It runs on Python 2.4-3.2, PyPy and Jython-2.5.1 interpreters on both Windows and Posix platforms. It is well documented and popular in the Python community. The best known project that uses it is PyPy, which has over 16,000 unit tests. py.test

[16]`http://docs.python.org/py3k/library/configparser.html`
[17]`http://www.voidspace.org.uk/python/configobj.html`
[18]`http://pytest.org/latest/`

discovers tests automatically by searching for modules, classes, functions and methods that starts with "test_". It uses the assert statement to test variables and values. These implicit behaviours make tests easier and faster to write, but harder to learn and understand.

4.7.2 nose

nose[19] testing framework extends Python's unittest library to make testing easier. It provides an alternative test discovery and running process for unittest, which is intended to mimic the behavior of py.test as much as reasonably possible without resorting to too much magic. nose support easy-to-write plugins, and it comes bundled with the most popular ones. It supports both Python 2.x and 3.x branches.

4.7.3 Attest

Attest[20] is a test automation framework for Python, emphasising modern idioms and conventions. It supports test collecting using Python decorators, introspection of the assert statement, treating tests as Python modules rather than scripts. Attest is a rather young framework, with limited features and documentation. Attest is a sub-level project of the Pocoo project.

4.7.4 coverage.py

coverage.py[21] is a tool for measuring code coverage of Python programs. It is typically used to measure the effectiveness of unit tests, by showing which parts of the code are exercised by tests. coverage.py support Python 2.3 to 3.2. It can output results in plain text, HyperText Markup Language (HTML) and XML.

4.8 User Documentation Tools

Some of the non-functional requirements for our utility is user documentation. In this section we describe a tool for writing such documentation, and a free hosting site for our user documentation.

[19] `http://readthedocs.org/docs/nose/en/latest/`
[20] `http://packages.python.org/Attest/`
[21] `http://nedbatchelder.com/code/coverage/`

4.8.1 Sphinx

Sphinx[22] is a Python tool for writing documentation, that makes it easy to create intelligent and beautiful documentation. It is used for the standard Python documentation, and it is popular in the Python community. Sphinx uses reStructuredText as its markup language, which is a easy-to-read, what-you-see-is-what-you-get plain text markup syntax and parser system. Our use case for sphinx is writing documentation for our utility, how to use it and configure it. Sphinx can generate output in several different formats, including HTML and latex/pdf.

4.8.2 Read the Docs

Read the Docs[23] is a free hosting of documentation for the open source community. It supports Sphinx docs written with reStructuredText, and it can automatically pull from Git, Subversion, Bazaar, and Mercurial repositories. We can configure it so it automatically pulls and compiles our user documentation from our GitHub repository whenever we push any changes.

4.9 Integrated Development Environment

4.9.1 PyCharm

PyCharm[24] is a cross platform, proprietary Integrated Development Environment (IDE) for Python. It has good support for text editing, syntax highlighting, auto indentation, code navigation, code completion and automatic error checking. There is also a decent debugger and unit test support that can help finding errors and it has integrated version control support, including git, which makes it easy to synchronize with a remote repository. The most used functions are also paired with keyboard shortcuts.

The downside with PyCharm is that it requires a relative expensive license. It is, however, possible to apply for classroom licenses that are free of charge. The latter is a requirement to make this IDE a viable option.

4.9.2 PyScripter

PyScripter[25] is a Windows only, open source IDE for Python. It has support for basic text editing functions relevant to programming, like syntax highlighting, auto

[22] `http://sphinx.pocoo.org/`
[23] `http://readthedocs.org/docs/read-the-docs/`
[24] `http://www.jetbrains.com/pycharm/`
[25] `http://code.google.com/p/pyscripter/`

indentation, code completion, debugger and file management. It also has some support for navigating the code, for example by offering to find the next point in the code that references a certain variable or function. The mentioned function mostly has keyboard shortcuts.

It does not have support for automatic error checking in the program, so it will not alert the user of spelling and syntax errors. It also lacks integration with any version control systems like git or svn. The code completion and code navigation is a little lacking. It will, for example, not suggest importing files if you reference a class from another module, and it cannot give a complete list of usages of a function.

4.9.3 Vi IMproved (VIM)

VIM[26] is cross-platform, open source text editor originally created for the Amiga. It is not regarded as an IDE, but it provides all the regular features of text editors, including syntax highlighting, auto-completion, auto-indentation, searching, multiple undo and redo. It can be configured to support almost everything modern IDE's support, and its extensive customizability is considered parts of its strength. But it is also parts of its weakness, it is very difficult for new VIM users to learn how to use it effectively. Therefore we do not suggest any team member that is not already experienced with VIM to use it.

4.9.4 Summary

PyCharm is by far the best IDE evaluated in terms of functionality, and it is the one that mirrors Eclipse the most, which is an advantage, since most team members are best acquainted with Eclipse. It will be the recommended IDE for this project, given that we can acquire classroom licenses.

On the other hand, there is no real reason to dictate the use of IDE, since what determines the productivity of a team member is more how well you know the specific tool you are using. It will therefore be up to each team member to choose what IDE/text editor they want to use.

4.10 Evaluation and Conclusion

In this section we provide a justification for the choices we have made in regards to process, programming language, and libraries we will use. Then in subsection 4.10.6 we give a brief description of the framework we will construct for our utility.

[26]`http://www.vim.org/`

4.10.1 Development Process Choice

We have chosen Scrum for our development strategy. We do not have a lot of experience with software development either individually or as a team, so we have little personal knowledge of how much we are able to produce, and the task may present challenges that we are not prepared for when the project starts. For these reasons we believe that we need to take an agile approach to this project. This way, we may both learn as we go, and adjust later iterations by the result of the previous. We may also have something to deliver even if we do not have time to implement all the desired features.

The Scrum methodology fits these goals perfectly, and is therefore a natural choice. The risk factor here is that all team members are mostly unfamiliar with Scrum, while we have at least a little knowledge of waterfall. We do, however, think that the time and risk of learning will not outweigh the benefit it will give us over waterfall.

4.10.2 Programming Language Choice

We originally selected Java as our programming language because it would run on all the platforms required, it offered automatic memory management so it would be easier to debug, and it was the only language everyone on the team had experience with.

We looked at ANTLR for generating a C lexer and parser in Java, which looked very promising. It also provided grammar files for creating a C preprocessor in Java. Closer evaluation revealed that the C preprocessor grammar was written in 2006, and had stopped working in 2008 as newer versions of ANTLR was not backwards compatible. Also the generated C parser only validated C code, it did not create an abstract syntax tree that we could traverse. This meant that using Java and ANTLR would require us to modify these grammars to suit our needs, and ANTLR's lack of documentation became a significant risk for our project.

These issues and feedback from our customer made us evaluate Python for developing our utility. We found several libraries for parsing C files, and even one for parsing C++ header files. These are described in section 4.5.

We decided to use Python for this project because the parsing libraries for Python came in working condition with decent documentation, and because we were able to create a small working prototype in Python in just a few hours. We estimate that it would take at least a week to achieve the same result in Java.

A challenge with our decision is the fact that not all team members have sufficient experience with Python. Most team members must therefore do some self study before we start the first sprint.

4.10.3 Parsers Libraries & Tools Choice

We outlined three different approaches for parsing C header files. The first approach is to write a custom parser ourself, the second is to use a C parsing library, and the third is to use a toolkit parser like GCC and Clang.

We felt that writing our own C parser with C preprocessor would possibly take up a lot, if not all, of the available project time. The third option would add a large dependency that our customer want to avoid if possible. GCC and Clang can be challenging to install and use on Windows.

Therefore using a C parser library would be the best solution, and as mentioned above, Java with ANTLR proved challenging. So we evaluated Python parser libraries.

We decided to use pycparser. We favored pycparser over PyCParser and cppheaderparser because it has better documentation, it seemed to be a more mature project, and it supports the most of the C99 specification. pycparser depends on PLY, so our utility will also depend on it.

For C preprocessor we have selected to use a tool for Windows that comes with pycparser, on Mac we will use the one that comes with XCode, and on other platforms we will either use GCC or tools that comes with the platform. If we need to modify a C preprocessor, we might use PLY's incomplete C preprocessor.

4.10.4 Configuration Framework Choice

We have listed a summary of some of the advantages and drawbacks of the different configuration frameworks we looked at in Table 4.1.

Table 4.1: Configuration Summary

	YAML	configparser	ConfigObj
Advantages	+Simplicity +Flexibility	+Easy to use	+Easy to use +Flexibility +Nesting +Type validation
Drawbacks	-External library -No type validation	-Lacks nesting -Lacks lists -No type validation	-External library -Lacks lists
Latest version	3.10	3.2	4.7.2
Python branch	2.7 and 3.3	2.7 and 3.3	2.7
License	MIT	PSF L	BSD-new

We decided to use YAML for handling configuration files, as it covered most of our requirements. Because we decided to use the latest version of Python, version 3.2.2, the range of possible configuration frameworks was reduced. Therefore, although ConfigObj are very suitable for our task, it was eliminated as it is only available up to version 2.7. This left us with two main possibilities: YAML and configparser. configparser turned out to be insufficient for us, mainly because it lacked lists. Lists are needed for description of hierarchical structures of the C headers. YAML has only two minor disadvantages we should be aware of. Firstly, there is no type validation mechanism, so we will have to create our validation manually, and secondly, it is an external library. We find this drawback minor for now, but it can turn out to be a problem in the future. Except for these issues, YAML, more specifically pyYAML, seems to have a good potential for creating flexible configuration support for our utility.

4.10.5 Unit Testing Framework Choice

The three frameworks we looked at are very similar, being modern Python testing frameworks. They differ in maturity and what is often called magic in the Python community.

py.test is the most mature but also the most magic, it uses a lot of introspection to discover tests and it has no API. nose is heavily influenced by py.test, but it tries to be more explicit, and provides an API. Attest is the youngest testing framework, and like nose, has less magic and focuses on providing a very pythonic API. Being the youngest also means it has the least documentation, functionality and plugins. Therefore Attest might be the easiest testing framework to learn. Therefor we decided to use Attest for unit testing of our utility.

4.10.6 Our Framework

Our utility will need to take C header files as input, search through them to find struct definitions, and create Lua scripts that dissects the structs in Wireshark.

To find the structs we will use pycparser to parse the input files, create an abstract syntax tree, and to find the struct definitions. We will use pyYAML to read configuration from file, which together with the struct definitions will be placed in some suitable data structures for generating dissectors.

The versions of the different tools and libraries we are using can be found in Table 4.2.

4.11 IP Rights & License

The customer have explained that they do not intend to distribute our utility, and that we are free to license it as open source if we want to, under whichever license we feel is most suited. They suggested GNU[27] General Public License

[27] `http://www.gnu.org/`

Table 4.2: Versions of Tools and Libraries

Library/Tool	Version	Why
Python	CPython 3.2.2	Latest stable standard Python implementation
pycparser	2.06-dev	Development version, for _Bool support
pyYAML	3.10	Latest stable version
PLY	3.4	Latest stable version
Attest	0.6-dev	Development version, for Python 3.2 support
Sphinx	1.1	Latest stable version
WireShark	1.7.0-SVN	Latest nightly build, for Lua support

(GPL) as Wireshark is released under it.

When we decided which license to use, we had to consider the licenses of the libraries and tools we depend upon. This is summarized by Table 4.3.

Table 4.3: Licenses

Wireshark	GNU GPL v3
PLY	BSD-new
pycparser	BSD-new
pyYAML	MIT
Our utility	GNU GPL v3

Some of the requirements for our utility might require us to modify the C preprocessor in PLY and the pycparser library, which made us consider the new 2-clause BSD license the most suited for us. Since it also gives us the option to later move to a more restrictive license, like GPL, we selected it.

During sprint 3 we discovered that Lua dissectors which interfaces with the Wireshark API must be under GPL license. We therefor decided during Sprint 4 to change the license of our utility, as well as the license on any generated Lua dissectors, to GPL version 3.

CHAPTER 5 REQUIREMENTS

This chapter describes a utility that creates Wireshark dissectors from C header files. The dissectors must interpret binary representations of C structs. In section 5.1 we give a high level overview of the utility and lists all the functional and non-functional requirements, while section 5.4 provides use cases for the utility, and section 5.6 contains the complete product backlog.

5.1 List of Requirements

We are to create a utility that allows Wireshark to interpret the binary representations of C-language structs. While C structs seldom are exchanged across networks, they are sometimes used in inter-process communication. The purpose of the utility described here is to provide Wireshark with the capability of automatically dissecting the binary representation of a C struct, as long as its definition is known.

The expected work flow for the utility is to read one or more C header files, which contain struct definitions, and output Wireshark dissectors, implemented in Lua scripts. A configuration file or source code annotations in the header files may be used when additional configuration is required.

Table 5.1 and Table 5.2 lists the functional requirements, while Table 5.3 lists non-functional requirements. Each requirement have a priority (Pri) and a complexity (Cmp): H, M or L. This is explained in subsection 5.1.1 and subsection 5.1.2.

5.1.1 Prioritization

The team has, in cooperation with the customer, prioritized the requirements in four categories: *a*) High, *b*) Medium, *c*) Low or *d*) Optional.

High Core functionality of the utility that must be implemented.

Medium Requirements that will improve the value of the utility.

Low Requirements that will not add much value to the utility.

Optional Requirement that may be implemented depending on available time.

5.1.2 Complexity

The team has estimated the complexity for each requirement. We use the following categories: *a*) High, *b*) Medium or *c*) Low.

High Functionality that seems difficult and non-trivial to create.

Medium Functionality that seems time consuming but straight forward.

Low Requirements that are trivial to implement.

5.1.3 Final Requirements

The functional requirements are listed in Table 5.1, optional requirements in Table 5.2, and non-function requirements are listed in Table 5.3.

5.2 Requirements Evolution

The customer provided an initial requirements specification for the utility at the start of the project, which can be seen in Appendix E.

We made some initial changes to the format, created some non-functional requirements and added priority and complexity to each requirement. This resulted in the initial requirements listed in Table E.1 and Table E.2.

Based on feedback provided by the customer during the sprints, we added several new requirements or rewrote already existing requirements, which are described in this section. The final requirements are described in subsection 5.1.3.

5.2.1 Sprint 1

The following new requirements were added during this sprint based on feedback from customer.

FR2-D The dissector shall be able to recognize invalid values for a struct member.

FR4-D Configuration must support specifying the ID of dissectors.

FR4-E Configuration must support custom Lua files for specific protocols.

Table 5.1: Functional Requirements

ID	Description	Pri.	Cmp.
FR1	The utility must be able to read basic C language struct definitions from C header files	H	
FR1-A	The utility must support the following basic data types: int, float, char and boolean	H	L
FR1-B	The utility must support members of type enum	H	L
FR1-C	The utility must support members of type struct	H	M
FR1-D	The utility must support members of type union	M	M
FR1-E	The utility must support members of type array	H	M
FR1-F	The utility should detect structs with the same name, and report it as an error	M	L
FR2	The utility must be able to generate Lua dissectors for Wireshark for the binary representation of C struct	H	
FR2-A	The dissector shall be able to display simple structs	H	L
FR2-B	The dissector shall be able to support structs within structs	M	M
FR2-C	The dissector must support Wireshark's built-in filter and search on attributes	H	L
FR2-D	The dissector shall be able to recognize invalid values for a struct member	L	L
FR2-E	The dissector shall be able to guess dissector from packets size	L	L
FR3	The utility must support C preprocessor directives and macros	H	
FR3-A	The utility shall support #include	H	L
FR3-B	The utility shall support #define and #if	H	L
FR3-C	The utility shall support `_WIN32`, `_WIN64`, `__sparc__`, `__sparc` and `sun`	M	H
FR4	The utility must support user configuration	M	
FR4-A	Configuration must support valid ranges for struct members	L	L
FR4-B	Configuration must support custom Lua files for specific protocols	H	H
FR4-C	Configuration must support custom handling of specific data types	L	M
FR4-D	Configuration must support specifying the ID of dissectors	H	L
FR4-E	Configuration must support various trailers (other registered protocol)	L	H
FR4-F	Configuration must support integer members which represent enumerated named value	M	L
FR4-G	Configuration must support members which are bit string	M	L
FR4-H	The utility shall support automatic generation of placeholder configuration	L	L
FR4-I	Configuration must support specifying the size of a struct members	M	L
FR5	The dissectors must be able to handle binary input which size and endian depends on originating platform	M	
FR5-A	Flags must be specified in configuration for each platform	M	M
FR5-B	Generate dissectors with correct alignment depending on platform	M	M
FR5-C	Generate dissectors which support both little and big endian platforms	H	M
FR5-D	Generate dissectors which support different sizes depending on platforms	M	H
FR6	The utility shall support parameters from command line	H	
FR6-A	Command line shall support parameter for C header file	H	L
FR6-B	Command line shall support parameter for configuration file	H	L
FR6-C	Command line shall support batch processing of C header and configuration files	L	M
FR6-E	Command line shall support #define and –Include directives	M	L
FR6-F	The utility shall only generate dissectors from structs with valid id and theirs' dependencies	L	M

Table 5.2: Optional Requirements

FR2-F	Dissectors shall display a warning if a struct member contains uninitialized memory	L	M
FR6-D	When running batch mode, dissectors that already are generated, shall not be regenerated, if the source are not modified since last run	L	H
FR7	The utility shall be able to fetch configuration directly from source code	L	
FR7-A	The utility shall find struct descriptions from Doxygen comments	L	H
FR7-B	The utility shall find configuration of #define enums from header files	L	H

Table 5.3: Non-Functional Requirements

ID	Description	Pri.	Cmp.
NR1	The utility shall be able to run on latest Windows and Solaris operating system	M	L
NR2	The dissector shall be able to run on Windows x86, Windows x86-64, Solaris x86, Solaris x86-64 and Solaris SPARC	M	M
NR3	The utility shall only have a command line user interface.	H	L
NR4	The utility must have sufficient documentation to allow a person, with no prior knowledge of the system or Wireshark, to be able to use it to generate Lua dissectors after five hours of reading	M	M
NR5	The utility must have sufficient documentation to allow a person, with prior knowledge of Wireshark, to be able to use it to generate Lua dissectors after one hour of reading	M	M
NR6	The utility must have sufficient documentation to allow a person, already proficient with the system, to be able to extend its functionality after four hours of reading	M	M
NR7	The utility code should follow standard python coding convention as specified by PEP8 and try to follow python style guidelines defined by PEP20	H	L
NR8	All Python modules, classes, functions and methods in the utility should have docstrings which explains their code	L	L

FR6-C Generate dissectors which support both little and big endian platforms.

FR6-D Generate dissectors which support different sizes depending on platforms.

5.2.2 Sprint 2

Based on feedback from the customer we added four new requirements in sprint 2, in addition to other small requirements changes.

Requirement FR4-B was split into two new requirements, FR4-F and FR4-G. Requirement FR6-B was completely rewritten, and the following requirements changed id during sprint 2.

- FR4-E -> FR4-B
- FR5 -> FR4-E
- FR6 -> FR5
- FR7 -> FR6

The following new requirements were added in sprint 2.

FR1-F The utility should detect structs with the same name, and report it as an error.

FR4-F Configuration must support integer members which represent enumerated named value.

FR4-G Configuration must support members which are bit string.

FR5-B Generate dissectors with correct alignment depending on platform.

The customer also provided the following requirement descriptions and feedback.

FR1-E: Support member of type array Arrays should be displayed as a sub-level. Multidimensional arrays should have one sub-level per dimension. Dissectors should also display the type of the array and show indexes for sublevels.

FR2-B: Struct within structs Inner structs should be displayed as a sub-level of the outer struct. When an external dissector is called, it should be called with a name and not an id, in order to not assign an id to structs that are never used as a base.

FR4-E: Headers/trailers The customer specified this mean that a given struct is a header, and that it can have various trailers. A configuration file should specify the kind of trailer, and what variable inside the struct which specify how many trailer items to expect.

FR5-C: Endian handling The header part of the packet, which include the platform flag, will always be in big endian (network order).

FR6-C: Batch mode The customer clarified this to mean that the utility should be able to run completely unattended given a set of command line arguments. For example it should not ask the users any questions under this mode. This is to be able to run it as a cron job at night.

Data Alignment The customer said that we could have some problems with alignment with the current offsets, because the different platforms may pad the data members of a struct to match an integer number of words on that platform.

5.2.3 Sprint 3

Customer feedback during sprint 3 resulted in one new non-functional requirement.

NR5 The utility must have sufficient documentation to allow a person, with prior knowledge of Wireshark, to be able to use it to generate Lua dissectors after one hour of reading.

Non-functional requirements NR5 to NR7 had their id increased by one.
The customer also provided the following requirement descriptions and feedback.

FR5: Support multiple platforms We should not generate separate dissector files for the different platforms. It is much better if we have one protocol with different functions for the different platforms, as this would not lead to such a performance hit in Wireshark.

FR2-F: Uninitialized memory It would be nice if our utility was able to detect uninitialized memory for debugging purposes. Different compilers use default patterns ins members that are uninitialized. If our utility could detect these patterns, we could display the data in Wireshark with a warning that lets the user know that the data might have been uninitialized.

FR4-D: Dissector id's The customer informed us that a struct could belong to several dissector id's. We should implement this by specifying a list of id's instead of just a single one.

Include dependencies The customer described a scenario where a struct was included in another struct, which is resident in a different header file that the first struct has not included. In the scenario both files are included in a third header file. This creates a dependency that we must take into consideration and implement in the utility.

5.2.4 Sprint 4

As it became clear in sprint 3 that we would complete all the given requirements before the end of sprint 4, we requested new features which we would consider to implement. The following new requirements were suggest by the customer.

FR2-E The dissector shall be able to guess dissector from packets size.

FR2-F The dissector shall display an warning if a struct member contains uninitialized memory.

FR4-H The utility shall support automatic generation of configuration files.

FR4-I Configuration must support specifying the size of a struct members.

FR6-E Command line shall support #define and –Include directives.

FR6-F The utility shall only generate dissectors from structs with valid id and theirs' dependencies.

FR7 The utility shall be able to fetch configuration directly from source code.

FR7-A The utility shall support generation of struct member description from Doxygen comments.

FR7-B The utility shall support reading configuration for #define enums from the header files.

The customer also provided the following requirement descriptions and feedback.

FR4-B: Custom Lua files The customer wanted a way to be able to fetch the buffer offset value to be able to both use it to pick specific parts of the message and modify it.

FR3: Support for the #pragma directive The customer uses the #pragma directive to identify the version of the file. This will not affect the dissectors generated in any way. #pragma will, however, crash CSjark if they are not removed.

FR4-H: Auto-generation of configuration files The customer asked if we could make the utility auto-generate template configuration for each struct it can generate a dissector for. This would make it easier for the user to configure structs, as the user only needs to fill in an id number in most of the cases.

FR6-F: Only generate useful dissectors The customer wants a mode where the utility only generates dissectors for structs with an corresponding configuration file containing an id, and for structs and unions that are inside these structs.

FR6-E: Support specific C preprocessor directives The customer wants the utility to be able to take certain C preprocessor directive arguments from command line. For example –Include or –Define.

FR7-A: Find struct description from Doxygen comments Make our utility be able to read comments corresponding to the struct and use these for the description field in dissectors. This feature would make generating dissectors require less manual configuration.

FR7-B: Read int-enum configuration from header files Some of the integer members may really be intended to be used as an enum, with the numerical values corresponding to strings. Some of the headers contains #define directives that specifies these string and integer pairs. The customer would like the utility to be able to find these and encode it in the configuration file. The benefit is that the customer then does not have to configure this manually.

FR2-E: Guess the dissector from packet size The customer wants Wireshark to be able to guess what dissectors to use from the size of the packet. It could compare this size to the size dissectors expects. If multiple dissectors fit, it could try to display the message with all suitable dissectors. This makes the customer able to use some dissectors without manually configuring the message ID for the struct it represents.

FR6-E: A way to ignore headers in a specified folder The customer wants a way to specify what folder to ignore in batch mode. This is to avoid complex headers that are irrelevant to the customer's goal (does not include structs), but which our utility fails to parse.

5.3 Requirement Description

This section gives a short description of the requirements, to give the reader of the paper a better understanding of the requirements. The description for each group of requirement are described below:

FR1 To be able to parse the header-files, the utility need to have support for different C data types and definitions. This requirement list the different members that the utility shall support.

FR2 The requirement specify what the utility shall create dissector for, and what they shall support to be able to be display the packet correctly in Wireshark.

FR3 To be able to parse the header-files, the utility will need to support some C preprocessor directives and macros. This requirement covers what the utility need to support.

FR4 To make the utility flexible, there is a need to support configuration of how the utility should handle different data types, custom code and configuration how to display members in Wireshark. This requirement specify what the utility should support configuration of.

FR5 To be able to support different platforms, the utility will need functionality that can be different between the platforms. The requirement lists what the utility must support, to handle different platforms.

FR6 These requirement tells what kind of command-line parameters the utility should support.

FR7 The requirement in this category, is for automatic genereation from the header-files. With automatic generation there will be faster the configure the system.

The relationship between the requirements can be seen in Figure 5.1.

5.4 Use Cases

This sections contains use case diagrams for our two actors, and detailed textual use cases for these diagrams.

5.4.1 Actors

An actor specifies a role played by an external person or entity that interact with our utility. We have three types of actors to consider. First is the primary actor that uses the utility to generate dissectors from C header-files. A secondary actor is a user who configures the utility to change the

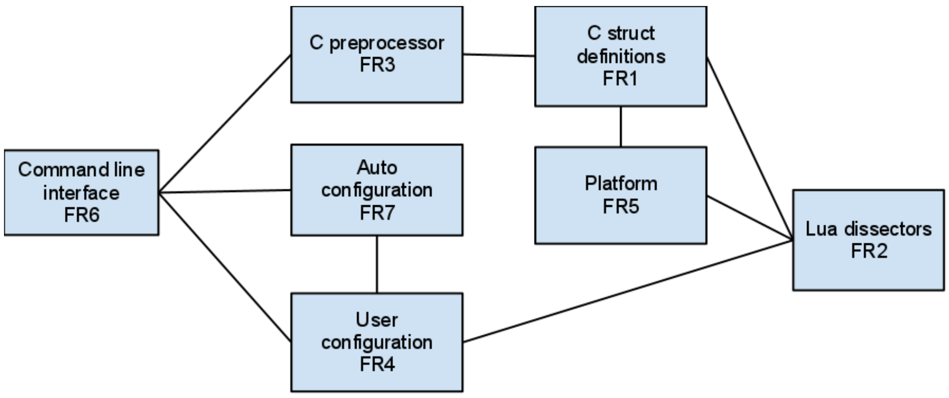


Figure 5.1: Relationship Between Requirements

output of it. Finally, we have an offstage actor, which does not use our utility himself, but uses the outputted dissectors in Wireshark.

We have defined two use case actors for our utility. The customer has specified that the offstage actor, called developer, is the most important actor.

Developer User of the generated Wireshark dissectors, offstage actor

Administrator User and configurer of utility, primary and secondary actor

5.4.2 Use Case Diagrams

Figure 5.2 shows the use case diagram for the administrator, and Figure 5.3 is the use case diagram for the developer.

5.4.3 Textual Use Cases

Each of the use cases is described textually below, to give a better understanding of the use cases diagrams. The textual use cases can be seen in Table 5.4-5.11.

5.5 User Stories

To make it easier to implement the requirements, user stories were written. The user stories describes how the requirements should be implemented. The user stories that was written can be found in the sprint design for each of the sprints. Table 5.12 shows a template of a user story.

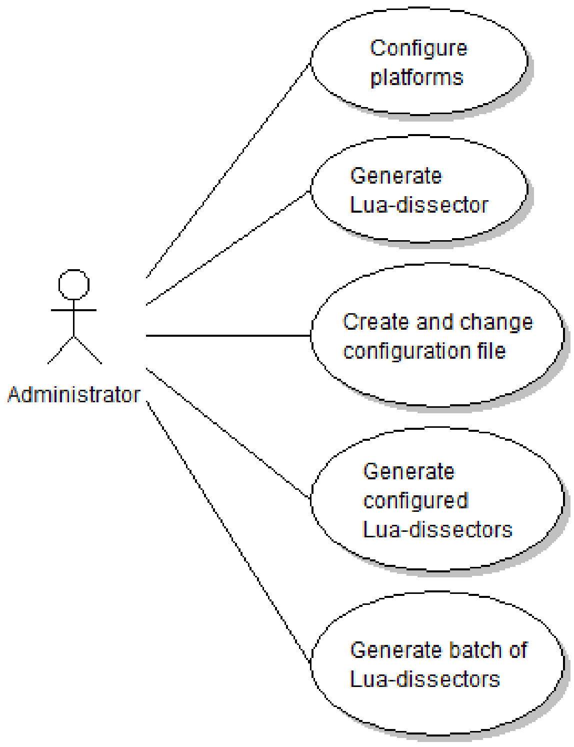

Configure platforms
Generate Lua-dissector
Create and change configuration file
Generate configured Lua-dissectors
Generate batch of Lua-dissectors
Administrator

Figure 5.2: Use Case Diagram: Administrator

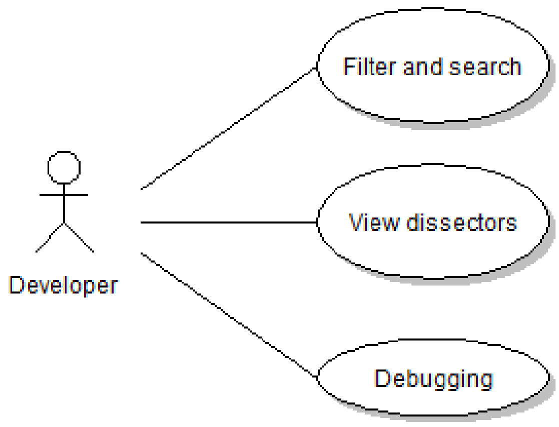
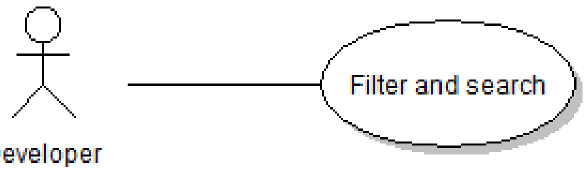


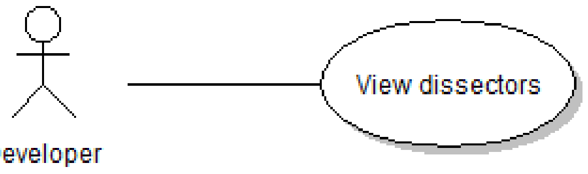
Figure 5.3: Use Case Diagram: Developer

Table 5.4: Filter and Search Textual Use Case



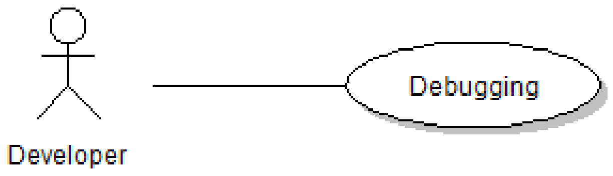
Element	Description
Use case name	Filter and search on attributes
Goal	The developer wants the correct set of results based on the search phrase
Summary	The developer would like to filter and search on attributes in the packets displayed in Wireshark
Preconditions	Wireshark needs to be running with dissectors.
Postconditions	Wireshark displays the results.
Flow of Events	1. The developer selects the search field in Wireshark's GUI. 2. The user types in a search phrase. 3. Wireshark will present the search results that match the query.
Exceptions	None

Table 5.5: View Dissector Textual Use Case



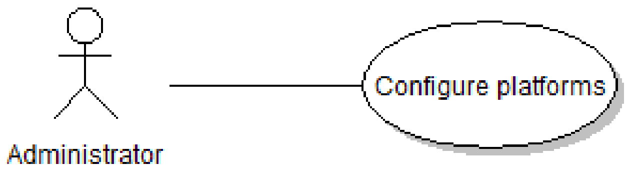
Element	Description
Use case name	View the dissectors in Wireshark
Goal	View structs correctly dissected in Wireshark
Summary	The developer would like to dissect a structs and have the members and values displayed in Wireshark by using the dissectors in Wireshark's plugin folder.
Preconditions	1. The developer have Wireshark running with dissectors. 2. The dissector for a struct will dissect it correctly, according to the initial internal structure of the struct.
Postconditions	Wireshark displays the struct with the correct structure and values.
Flow of Events	1. The developer selects a struct message in Wireshark. 2. Wireshark calls the correct dissector and dissects the selected message. 3. Wireshark displays the members and values of the selected message.
Exceptions	1. The correct dissector for a struct might not exist in Wireshark's plugin folder, making it impossible to dissect the message.

Table 5.6: Debugging Textual Use Case



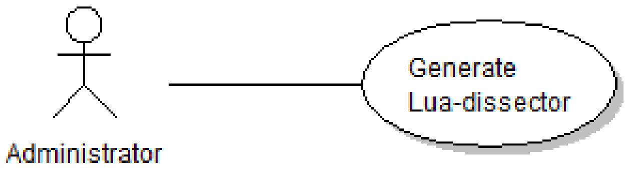
Element	Description
Use case name	Debugging
Goal	The developer wants to debug inter-process communication.
Summary	The developer wants to debug inter-process communication by using Wireshark extended by dissectors.
Preconditions	1. The developer have Wireshark running with dissectors. 2. Wireshark have access to the packets sent between the processes that the developer wants to debug.
Postconditions	Wireshark displays the communication dissected.
Flow of Events	1. The developer selects the inter-process communication to debug. 2. Wireshark calls the correct dissector and dissects the selected messages. 3. The developer is able to debug the process communication by looking at the dissected messages.
Exceptions	1. The correct dissectors might not exist in Wireshark's plugin folder, making it impossible to dissect the messages. 2. The inter-process communication might not only consist of structs, but also data structures that Wireshark are unable to display.

Table 5.7: Configure Platforms Textual Use Case



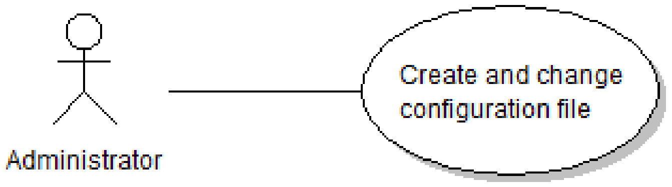
Element	Description
Use case name	Configure platforms
Goal	Successfully configure the supported platforms.
Summary	The administrator wants to be able to configure the platforms that the utility supports. This includes adding, removing and editing supported platforms.
Preconditions	The administrator must have access to the utility's platform module (the source code).
Postconditions	The changes must be saved to the platform module.
Flow of Events	1. The administrator locates the platform module in the utility. 2. The administrator makes changes to the module to achieve the wanted support. 3. The administrator saves the changes and exits.
Exceptions	The changes made to the platform module were not of correct syntax. Leaving the utility defected.

Table 5.8: Generate Lua Dissector Textual Use Case



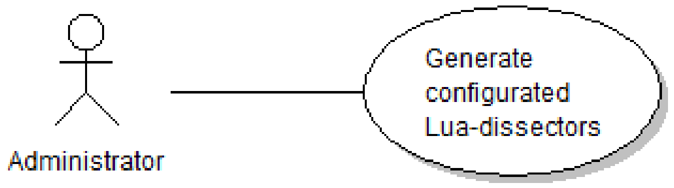
Element	Description
Use case name	Generate Lua-dissector
Goal	Successfully generate a Lua-dissector.
Summary	The administrator wants to generate a Lua-dissector based on a header file.
Preconditions	1. The administrator has the utility and its dependent libraries installed. 2. The administrator has a header file, which a Lua-dissector shall be made from.
Postconditions	The utility outputs the generated Lua-dissector to a default- or given output location.
Flow of Events	1. The administrator feeds the utility with the header- and configuration file 2. The utility generates a Lua-dissector based on the input. 3. The utility outputs the Lua-dissector to a default- or given output location.
Exceptions	1. The header file might not be of correct syntax. 2. The utility's dependencies might not be covered, resulting in a crash of the utility.

Table 5.9: Create and Change Configuration File Textual Use Case



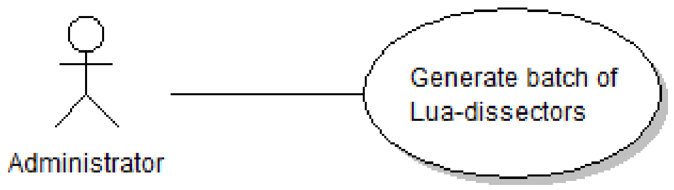
Element	Description
Use case name	Create and change configuration file
Goal	Successfully create and/or change a configuration file.
Summary	The administrator wants to create and/or change a configuration file.
Preconditions	Create or find an existing configuration file
Postconditions	Save the file.
Flow of Events	The administrator changes the located file to get the wanted configuration.
Exceptions	The created or changed configuration file might not be of correct syntax.

Table 5.10: Generate Configured Lua Dissectors Textual Use Case



Element	Description
Use case name	Generate configured Lua-dissectors.
Goal	Successfully generate a configured Lua-dissector.
Summary	The administrator wants to generate a Lua-dissector from a header file with an associated configuration file.
Preconditions	1. The administrator has the utility and its dependent libraries installed. 2. The administrator has the header and configuration pair, which the Lua-dissector shall be made from.
Postconditions	The utility outputs the generated Lua-dissector to a default- or given output location.
Flow of Events	1. The Administrator feeds the utility with the header- and configuration-file 2. The utility generates a Lua-dissector based on the input. 3. The utility outputs the Lua-dissector to a default- or given output location.
Exceptions	1. The header and/or configuration file might not be of correct syntax. 2. The utility's dependencies might not be covered, resulting in a crash of the utility.

Table 5.11: Generate Batch of Lua Dissectors Textual Use Case



Element	Description
Use case name	Generate batch of Lua-dissectors.
Goal	Successfully create multiple Lua-dissectors.
Summary	The administrator wants to process multiple header- and configuration-file pairs in one run of the utility.
Preconditions	The administrator knows the locations of all the files to parse.
Postconditions	The utility outputs the generated Lua-dissectors to a default- or given output location.
Flow of Events	1. The administrator feeds the utility with the header- and configuration-files. 2. The utility generates Lua-dissectors for all the input headers. 3. The utility outputs the Lua-dissectors to a default- or given output location.
Exceptions	1. A header or configuration file might not be of correct syntax, which will make the utility skip that actual file and proceed with the rest of the files in the batch. 2. The utility's dependencies might not be covered, resulting in a crash of the utility.

Table 5.12: User Story Template

Header	Value
ID	ID for the user stories, written like USxx.
Requirements	The requirement that the user story describes.
What	Description of what the user want to achieve.
How	Description of how the requirement should be implemented.
Result	What the result is after the implementation.

5.6 Product Backlog

The complete product backlog can be seen in Table 5.13. The listed actual work hours does not reflect the total hours used on each requirement. Any fixes, improvements or refactoring done in a later sprint is not included. Testing and user documentation is also not considered.

Optional requirements which we did not implement are listed in Table 5.14. These optional requirements are described in section 12.2.

Table 5.13: Product Backlog

			Hours	
Req.	Description	Sprint	Est.	Act.
FR1	Read basic C struct definitions		**52**	**51**
FR1-A	Support data types: int, float, char and boolean	SP1	24	21
FR1-B	Support members of type enum	SP2	6	5
FR1-C	Support members of type struct	SP2	7	3.5
FR1-D	Support members of type union	SP3	5	6
FR1-E	Support members of type array	SP2	7	12
FR1-F	Detect structs with same name	SP2	3	3.5
FR2	Generate Wireshark dissectors in Lua		**69**	**59.5**
FR2-A	Display simple structs	SP1	28	25
FR2-B	Support display of structs within structs	SP2	11	15
FR2-C	Support Wireshark filter and search on attributes	SP3	3	1.5
FR2-D	Recognize invalid values for a struct member	SP1	22	15
FR2-E	Guess dissectors from packet size	SP4	5	3
FR3	Support C preprocessor directives and macros		**24**	**7.5**
FR3-A	Support #include	SP1	8	2
FR3-B	Support #define and #if	SP1	11	3
FR3-C	Support `_WIN32`, `_WIN64`, `__sparc` etc	SP3	5	2.5
FR4	Support user configuration		**91**	**71**
FR4-A	Support valid ranges for struct members	SP1	30	15
FR4-B	Support custom Lua files for specific protocols	SP3	10	7.5
FR4-C	Support custom handling of specific data types	SP2	6	5
FR4-D	Support specifying the ID of dissectors	SP2	7	9
FR4-E	Support various trailers (other registered protocols)	SP2	18	15
FR4-F	Support enumerated named values	SP2	5	6.5
FR4-G	Support bit strings	SP2	10	11.5
FR4-H	Automatic generation of placeholder configuration	SP4	1	0.5
FR4-I	Support specifying the size of unknown struct members	SP4	4	1
FR5	Handle input which size and endian depends on platform		**40**	**23.5**
FR5-A	Flags specified for each platform	SP3	8	11
FR5-B	Dissectors support memory alignment	SP3	12	6.5
FR5-C	Dissectors support both little and big endian	SP3	6	4
FR5-D	Dissectors support different sizes from flags	SP3	14	2
FR6	Support parameters from command line		**51**	**24**
FR6-A	Support parameter for C header file	SP1	9	9
FR6-B	Support parameter for configuration file	SP1	28	8
FR6-C	Support batch processing of C header and configuration	SP2	7	4.5
FR6-E	Support C #defines and –Include from CLI	SP4	1	1
FR6-F	Only generate dissectors for structs with valid ID	SP4	4	1.5
	Total		327	236.5

Table 5.14: Optional Requirements Estimates

Req.	Description	Est. Hours
FR2-F	Display if struct member contains uninitialized memory	8
FR6-D	Do not regenerate dissectors across multiple runs	2
FR7-A	Find struct descriptions from Doxygen comments	20
FR7-B	Find configuration of #define enums from header files	20
	Total	50

CHAPTER 6 TEST PLAN

This chapter presents the test plan for our solution. The test plan is based on the standards set by the IEEE829-1998 standard for software testing [14], but with a few changes to better fit with our project. The purpose of this plan is to have a structured way of performing tests, as well as providing the developers with a list of specific component-behaviors. The tests will be based on functional as well as non-functional requirements, deterring architectural drift and enforcing our design plans for the system.

6.1 Methods for Testing

Regarding software testing, we have two different types of tests available, namely Black box and white box tests. This section is dedicated to the discussion of these two testing methodologies.

6.1.1 White Box Testing

White box testing is a method of software testing where you test internal structures or modules of an application, as opposed to its functions. White box testing requires the tester to have an internal perspective of the system, as well as sufficient programming skills. As the utility was required to be able to function with a variety of different input, as well as being used as a debugging tool itself, we chose to have every developer on the team write unit tests for their own code, and then have someone else on the team do the testing of their code in order to ensure correctness. Also, in order to get a proper overview over what and how many parts of the system that are covered by unit tests, the team decided to use a tool for measuring code coverage.

Attest

As a tool for creating white box unit tests, the team decided to use the Attest testing framework for python code. To create unit tests using Attest, you start off by importing Tests, assert_hook and optionally contexts from attest. You then create a variable and initialize it to an instance of Tests, which is the variable that will contain list functions that each constitutes one test that is to be run. To feed your test instance with functions for testing you then have to mark these functions with a decorator and feed it the .tests function of the Tests instance. After creating a unit test in this fashion you can run all of your unit tests through Attest from the command line by typing "python -m attest". This runs all of your unit tests through Attest and returns a message telling the user how many assertions failed, as well as what input made them fail. For more information read the user documentation of Attest.

Coverage

As a tool for calculating code coverage the team decided to use Coverage, which is a tool for measuring code coverage in python projects. In order to run Coverage from the command line with the tests for this utility, you would have to first navigate to the folder where you installed CSjark, before typing "Coverage run -m attest". This generates a file that will be used to Coverage for generating a html table displaying the coverage. In order to create this html table you would then have to type "coverage html", which generates a folder named htmlcov. This htmlcov folder again contains a file named index.html that contains a html table describing which parts of the system underwent testing and their code coverage.

6.1.2 Black Box Testing

Black box testing is a method of software testing where you test the functionality of a system, as opposed to its internal structures. Black box testing does in general not require the tester to have any intimate knowledge about the system or any of the programming logic that went into making it. Black box test cases are built around the specifications and requirements of a system, for example its functional, and in some cases, non-functional requirements. The team decided to use black box testing for both the functional and non-functional requirements of the utility, as the customer had already expressed thoughts on extending and understanding the non-functional parts of the utility themselves.

6.2 Non-Functional Requirements

As evaluating the non-functional requirements of a system through its source code is very difficult, it was decided that we should create test cases for them the same way we created black box test cases. Some of these tests would also require the team to have more manpower or resources than what can be expected by a group of students. It was therefore also decided that we would ask the customer for help regarding the testing of some of the non-functional requirements. These test cases would then be designed according to the wishes of the customer and what resources they were able to supply us with.

6.3 Templates for Testing

Table 6.1 , Table 6.2 and Table 6.3 are templates we will be using for testing purposes.

In order to standardize the testing process, the team decided on making templates for both the test cases themselves and for reporting their results. The ones responsible for testing were given the task of not only creating and running the test cases themselves, but also adhering to the standards set in this document.

Table 6.1 shows the template for each test case. All of the test cases written for the utility will be in this format, and executed according to this document.

Table 6.2 shows the template for reporting the result of each test case. Table 6.3 shows the template for reporting code coverage.

Table 6.1: Test Case Template

Header	Description
Description	Description of requirement
Tester	Team member responsible for the test
Prerequisites	Conditions that needs to be fulfilled before starting the test
Feature	Feature to test
Execution	Steps to be executed in the test
Expected result	The expected output of the test

6.4 Test Criteria

An item will be considered to have passed a test if the actual result from the test matches the expected result from the test. An item will be considered to have failed the test if the output varies from the expected result. If there are

Table 6.2: Test Report Template

Header	Description
Test ID	ID for the given test
Description	Description of requirement
Tester	Team member responsible for the test
Date	The date the testing took place
Result	The success or failure of the test, and a comment on the result if needed

Table 6.3: Code Coverage Report Template

Module	Statements	Missing	Excluded	Coverage
module 1	Statements ran	Statements not ran	Excluded statements	Percent code coverage
...	...	...	...	...
module n	Statements ran	Statements not ran	Excluded statements	Percent code coverage
Total	Statements ran	Statements not ran	Excluded statements	Percent code coverage

any specifics as to why the test passed/failed, which needs to be discussed, they will be listed as a comment to the result

6.5 Testing Responsibilities

Each team member is responsible for writing their own unit tests, while the test leader is responsible for the quality of the test plan and the tests. The tests will mainly not be executed by the same developers who wrote the code that is to undergo testing, but by others in the testing team with as little ownership of the code as possible.

6.5.1 Testing Routines

In each sprint we decided that the unit tests would be run continuously throughout the sprints in order to make sure that no new functionality broke any of the earlier code. As the test cases depended somewhat on having completed most of the functionalities for a sprint, as well as requiring a lot of manually generated testing data, it was decided that they would be run towards the end of the sprint. The test results would then be presented a few days before the sprint evaluation, giving the developers some time to fix any bugs discovered by the test cases.

6.6 Changelog

6.6.1 Sprint 1

During sprint 1, it became apparent that the customer would not be able to supply the team with any real traffic data to use for testing. The testing team therefore decided to use a hex-editor to generate their own data packets with the C-structs that would be used for testing the utility. This is done by manually writing the hex values for the individual bytes in a pcap packet, where the first byte indicates the version number, the second byte indicates the flag value of the packet and where the third and fourth byte indicates the message ID. The rest of the bytes in the packet then contain the member values of whatever struct was associated with the packet's message ID.

During the sprint, it also became apparent that Wireshark would be able to provide the developers and testers with feedback on syntax and user errors on both the dissectors created by the utility, as well as the traffic used for testing. This is done by Wireshark crashing and/or providing the team with error messages related to the code in which the dissector is faulty. There will also be displayed a warning or error message with the generated traffic-data if there are any faults with them. The team was therefore able to use Wireshark in assisting them in creating correct code early on before writing unit tests.

6.6.2 Sprint 2

As of sprint 2, it was decided that the team should use an automated tool for calculating code coverage. Code coverage is a measure describing the actual amount of, and which code that undergoes unit testing. As code coverage inspects code directly, it is considered a form of white box testing. In this project it will be used to ensure that an as big part as possible of the system actually undergoes testing, and that the unit tests associated with the different modules of the utility actually tests what they are supposed to. It was also added as a goal to have at least 80% code coverage at the end of the project, where the testers and developers would aim to increase the amount of code coverage from each previous sprint.

6.6.3 Sprint 3

During sprint 3, it was suggested that the team should make a C program that returns hex dumps. The hex dumps are used for producing traffic to test the utility with. It was therefore decided that the team would create such a program to generate data for the bigger and more complex C-header files. These header files could have a dozen of struct-members of various types. As it would be tedious to have to manually write down the hex values of each struct member, creating this small program would reduce the time for generating test traffic.

CHAPTER 7 ARCHITECTURAL DESCRIPTION

This chapter introduces the final architectural documents for the project. The initial architecture with its change log can be found in the appendix D.

The team followed the definition of software architecture defined by Len Bass, Paul Clements and Rick Kazman: “The software architecture of a program or computing system is the structure of structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them.” [9, p.3]

The purpose of this document is to describe our architecture in a structured way so that it can be used, not only by the team, but also as an aid for other stakeholders who are trying to understand the system.

7.1 Architectural Drivers

This section is dedicated to the discussion of the architectural drivers that were discovered during the project. The team chose Modifiability and Testability as quality attributes, but it later on became apparent that some care should also be taken considering Performance.

The reason for choosing Modifiability was that the development team would be unable to update or maintain the utility after completing this project. The code would also be distributed under a GPL that allows other developers to continue working on the utility and use it for their own applications in the future. It was therefore important that the code would be easy to understand, well documented and easy to modify. Not only will this promote the further development of the utility, but it would also make it easier for the customer to use and modify for their own purposes.

Testability would also be an important quality attribute as the utility was to be used by the customer for debugging purposes. It was therefore be very important that the utility it self contained as few bugs as possible so that the customer could be sure that the output given by the utility would be able to help them in analyzing and debugging. The developers of this project were also unable to test any given dissectors in a real environment, which made it even more necessary for the developers to do extensive testing of the utility's functionality. This was to ensure that the final product works properly even without the developers having had taken a good look on the data the utility will have to process after going public.

Performance became somewhat of an issue for the utility during the project as it became apparent that the customer would have to run the utility on several thousand header files at once. It would therefore be important for the utility to be able to run through all of the customer's header files in a reasonable amount of time, which in this case would be that the utility should be able to run through several thousand header files over one night of being run in batch mode.

7.1.1 Testability Tactics

The goal of using testability tactics was making it easier to test the system after finishing any given sprint, as well as generally raising the quality of the tests and the amount of coverage the tests give the system.

Specialize Access Routes/Interfaces

Using a specialized testing interface makes it possible to specify values for a component independently from its normal execution. This will in turn make it possible to test parts of an unfinished module as well as making it easier to get a clear overview over what data is flowing through individual parts of the system. This is important for this project as the utility must be able to run in a different environment than what the developers have access to. The testers must therefore be able to create input for each individual component of the system in order to ensure that it will work correctly with all kinds of input.

We incorporated the use of this tactic by using the Attest testing framework when creating unit tests. By using attest we were able to create instances of and test parts and modules of the system independently with the exact input we wanted without having to run the entire utility.

Code Coverage

By using a framework to see which parts and how much of the code is actually being run during the unit tests, it becomes easier to improve the quality of the unit tests. It could also be used as a checklist to see if the ones creating the unit tests have implemented some functionality that is currently not being tested.

We incorporated the use of this tactic by using the Coverage tool for python. By using Coverage we were able to measure the code coverage of the unit tests by having Coverage create a HTML table that showed which parts and percentage of the system actually underwent testing.

7.1.2 Modifiability Tactics

The goal of using modifiability tactics were to make it easer to extend and modify the software during development as well as after completing the product.

Anticipate Expected Changes

By trying to anticipate expected changes it is possible to make it easier for modules to be extended with new functionality later. It also makes it easier for the developers to anticipate the different ranges of input the modules are able to process. This was important for this project as it was being developed incrementally, with new functionality and code added every sprint.

This was handled in this project by first identifying all of the functionality that the utility would need in order to be considered a finished product. Then we had some discussions about what should be included in each following sprint, where it was also discussed in minor detail how the work items were to be implemented. This made it easier for the developers to figure out which changes might have to be done further down the line so that they could prepare their code beforehand.

Generalizing Modules

Generalizing the modules of a system makes it possible to reuse older modules when doing modifications to the system. The more general a module, the more likely it is that a needed change to the system can be implemented by just adjusting the input to the system, rather than having to modify existing or creating new modules.

This was implemented mostly by using inheritance where we reused a lot of the functionality in a class when making new classes that needed similar functionality.

Restrict Communication Paths

By restricting the number of modules that are able to collect data from a specific module, the less dependent the entire system becomes of that specific module. This makes it easier to swap out existing modules with new ones without having to make many widespread changes to the entire system. This is important for this project as the source code could change drastically after discovering new requirements in later sprints. By having a loose coupling we will minimize the amount of code that has to be rewritten after every sprint.

We followed this tactic by using code inspection. If we discovered during a sprint that the coupling between the modules were becoming too tight, it was decided to refactor some of the code early on in order to save time by not having to do any major refactoring later on.

Using Configuration Files

By using configuration files, it is possible to change the behaviour of the system without having to do any changes to its code. It is very important that this system uses configuration files as this was a requirement from the customer, as well as making it a lot more flexible for the end user.

In order to implement this in the project it was decided that we would use the YAML format for writing and parsing configuration files. These configuration files would then make it possible for the users of the utility to make several alterations to how data should be displayed in Wireshark as well as having their own custom LUA code that would get run inside of the generated dissectors.

7.1.3 Business Requirements

The following business requirements encompass the most important needs of the customer.

- The utility must be delivered on time as it is not possible for the developers to continue the development after the deadline
- The utility should be able to create dissectors for the C-structs in header files used by Thales
- The utility should be able to create dissectors that run on all of the platforms used by Thales and their customers
- Developers at Thales should be able to use Wireshark with the generated dissectors to display the values in C-structs passed through the system.

7.1.4 Design Goals

To help guide the design and the implementation we tried to follow these goals and guidelines:

- Smart data structures and dumb code works better than the other way around [15]!

- Clear and clean separation of the front-end and the back-end so in the future other parsers can be used to generate dissectors.
- Try to be pythonic, follow PEP8 [1] and PEP20[2].
- Now is better than never. Don't be afraid to write stupid or ugly code, we can always fix it later.
- The first version is never perfect, so don't wait until its perfect before you commit. Commit often!

7.2 Architectural Patterns

This section presents the different architectural patterns used in the utility

7.2.1 Pipe and Filter

The pipe and filter architectural pattern consists of a stream of data that in turn is processed sequentially by several filters. This is done in such a fashion that the output of one filter becomes the input of the other. It is a very flexible, yet robust way of processing data, with support for adding more filters if needed for future applications and processes. As the utility will only work on one piece of data that gradually changes, and is then converted into Lua-code at the end, this seemed like a good and structured way of processing data early on, while still being able to add new functionality further down the line.

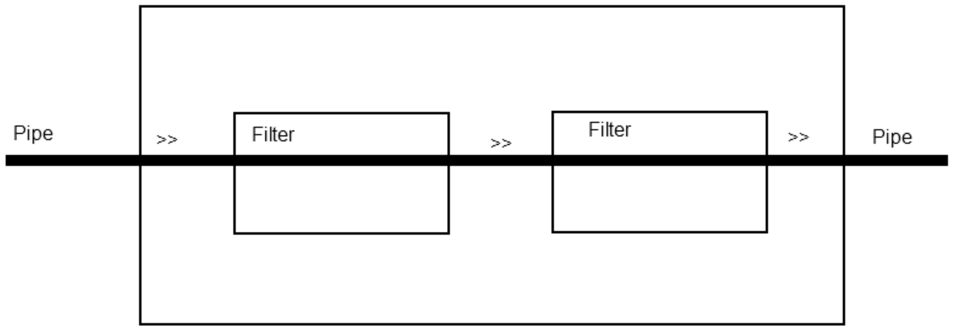


Figure 7.1: Pipe and Filter Pattern

[1]Style Guide for Python Code: `http://www.python.org/dev/peps/pep-0008/`
[2]The Zen of Python `http://www.python.org/dev/peps/pep-0020/`

7.2.2 Layered Architectural Pattern

The layered architectural pattern is a pattern that involves grouping several different classes that all share the same dependencies. This grouping of classes is called a layer, and the layers are structured so that the classes inside each layer only depend on the classes of their own layer level, or inside an underlaying one. Structuring the code in this way helps delegating responsibilities to different parts of the system in a logical way, making the code easier to understand and easier to navigate through.

Figure D.5 shows how the layered architectural pattern is used in the utility

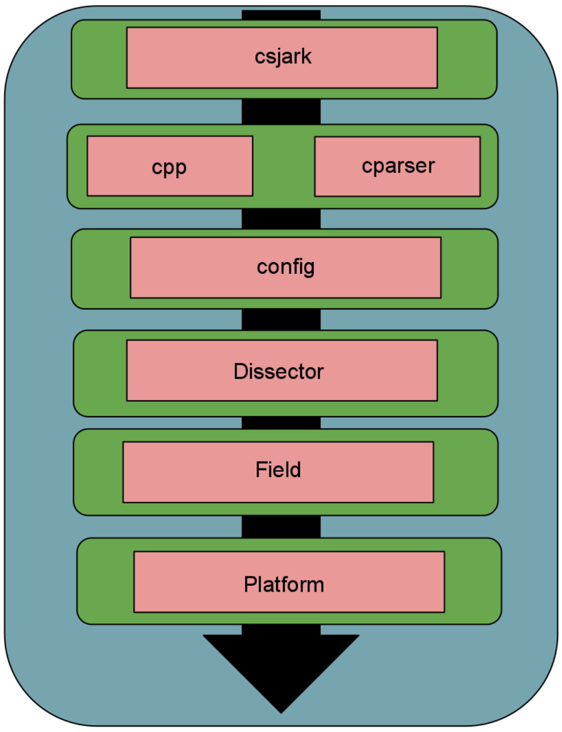


Figure 7.2: Layered Architectural Pattern in the Utility

7.3 Architectural Views

This section describes three different views the team used for this project: A logical view, process view and a deployment view.

7.3.1 Logical View

This view shown in Figure D.2. Command line takes the arguments for header file and configuration file as a string. The arguments are parsed in the command line parser. Header file is sent to ”C preprocessor & C parser”, the C header file is loaded and parsed by the C parser, which generates a parsing tree. Command line also call Configuration, which load the configuration file. The configuration will parse the configuration file and create configuration rules. The Lua script generator will generate a Lua script from the parsing tree and the config rules.

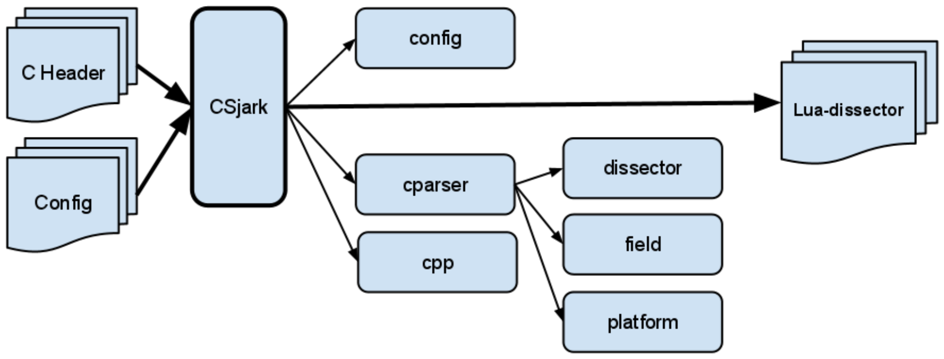


Figure 7.3: Overall Architecture

7.3.2 Process View

Figure D.3 shows the process view for our utility. Csjark takes header and config files as input and then uses the config and cparser to parse the files. CSjark then uses the cparser to find the structs in the header file and then creates dissectors for them. These dissectors are then written to a file and CSjark then reports to the user by sending a message to the command line.

7.3.3 Deployment View

Figure D.4 shows the deployment diagram for this project. CSjark takes header-files and config-files as input, and generates Lua dissectors. All these dissectors are added as plugins to Wireshark, extending the functionality. Wireshark will capture the data packet when Process A send data to Process B, the Lua dissectors is used to display these data packets correctly.

7.4 Architectural Rationale

The team decided to use the pipe and filter pattern as the architects felt that it was one of the only architectural patterns that would benefit the utility,

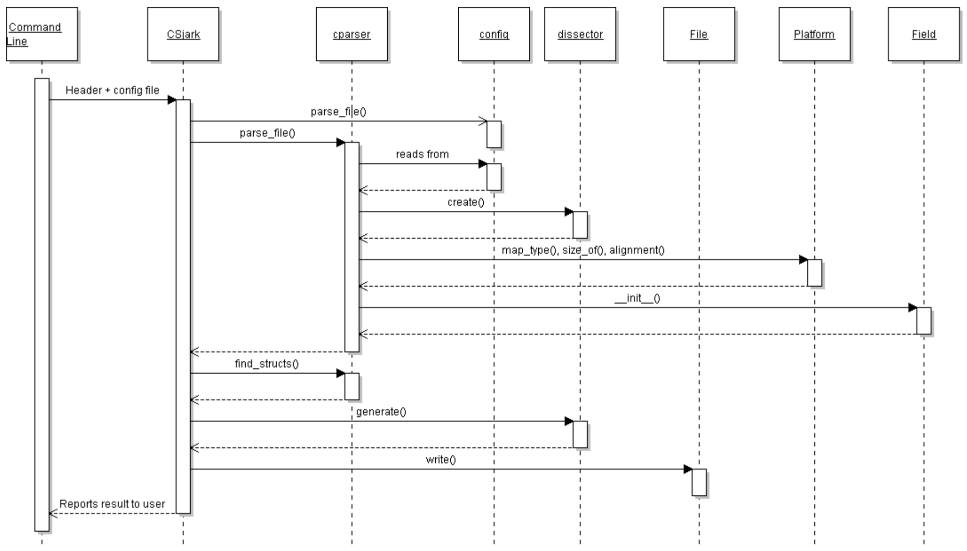


Figure 7.4: Data Flow During Regular Execution

without having to make it needlessly complex. The utility was supposed to take header files as input and then process the data from them several times, until the end result was a list of structs and members that could be used to make dissectors for Wireshark. This seemed like an excellent application to use the pipe and filter pattern with, as it would then be easy to add new filters to the header file for future increments of the development cycle without having to rewrite what had already been implemented in previous sprints.

The team also decided to use the layered architectural pattern as the code of the utility would have to stay logical and well structured through the entire project if we wanted the future development of the utility to go more

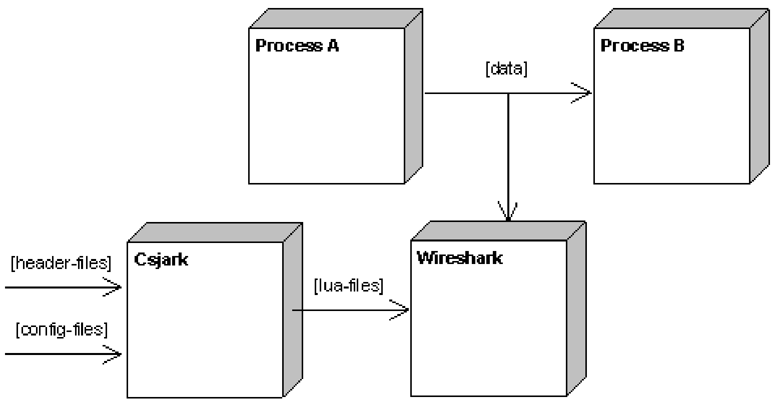


Figure 7.5: Deployment View

smoothly. By dividing the entire utility into several modules and designate layers between them, it became easier to decide which functionality would go where in the code, which could save the different developers some grief. It would also make the code easier to inspect by whomever that wishes to understand the code, and more specifically how the utility works.

For the views, the team decided to use a logical view, process view and deployment view. These views were chosen because the architects of the utility felt that these views alone could represent the system sufficiently, without creating too much overhead for the readers of the document. The logical view supplies the reader with a more in depth view of what the system is comprised of, which is useful for developers who need to figure out the workings of the system. The process view also seemed important for the developers and the testers of the utility, as it provides the reader with a more proper overview of the data flow in the system. This makes it easier and more clear to see which modules are run when, and to see which external calls dictate the modules' behaviour. Lastly, a deployment view was chosen to make it more clear for the reader of the document what the utility really produces as output and what other external applications it has to cooperate with.

Part II

Sprints

CHAPTER 8 SPRINT 1

In this chapter the first sprint is detailed. In section 8.1, the planning of the first sprint is explained, the design of sprint 1 is covered in section 8.2. In section 8.3, the features implemented in this sprint is described, and section 8.4 consists of the results from the tests. The feedback the team has got from the customer during the sprint is listed in section 8.5, and the evaluation of the first sprint is covered in section 8.6.

8.1 Sprint Planning

The first sprint was started with a sprint planning meeting September 14th. The team is planning to implement basic and fundamental parts of the utility. The most basic functionality was chosen from the product backlog, in order to end up with a utility that can generate simple Lua dissectors by the end of the sprint. This is important so we can see that we understood the project, and that the chosen preprocessor and parser is suitable for the utility.

8.1.1 Duration

Sprint 1 started September 14th, and lasted until September 27th. To avoid misunderstandings between our developers and the customer, we will have weekly meetings to show what we have accomplished and discuss further development.

8.1.2 Sprint Goal

The overall goal of the first sprint is to create a preliminary design and implement the core of the application. It will be possible to run it to generate

Lua dissectors, which can be used by Wireshark users. The first version will contain only the basic dissector generating features, for example, parsing C structs with basic data types as members. Also, some of the preprocessing capabilities will be implemented, such as handling #include directives. Basic support for configuration will be added.

8.1.3 Back Log

In the first sprint we will implement eight requirements. These are listed in Table 8.1. The time table for the sprint can be seen in Table 8.2.

8.2 System Design

This section introduces the the preliminary overall design, and the system design after the first sprint. In section 8.3, we describe how our implementation works and is to be used.

8.2.1 Preliminary Design

The following is a description of the preliminary design we had for our utility. We split our program into several parts, described in the following paragraphs. Their relationship is shown in Figure 8.1.

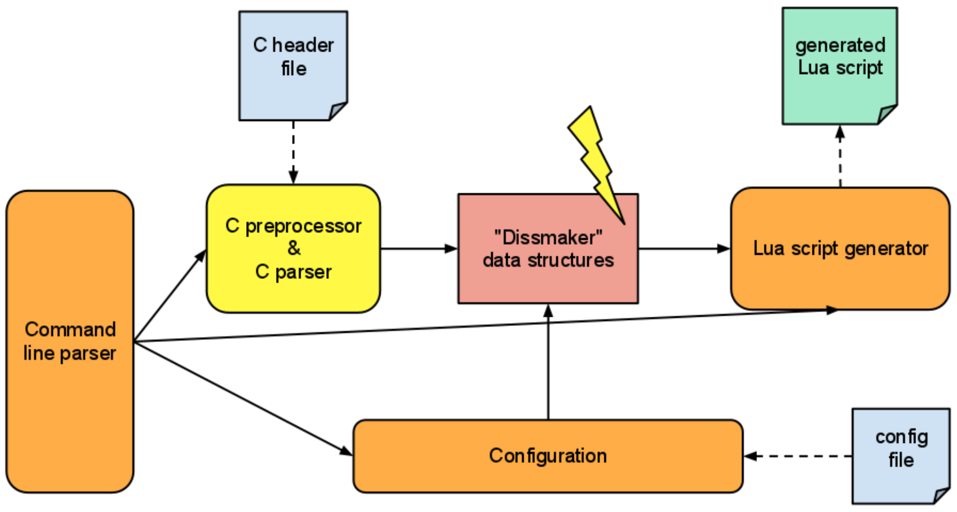


Figure 8.1: Overall Design

Command line interface The command line interface is where the user inputs which header files and configuration files he wants the utility to use. This module needs to ask the configuration to parse config files, and then ask

Table 8.1: Sprint 1 Requirements

		Hours	
User story	Req. and Description	Est.	Act.
US07	FR6-A: Command line shall support parameter for C header file	**9**	**9**
	Implementation	2	5
	Design	1	1
	Testing	2	2
	Documentation	4	1
US01	FR1-A: Support basic data types: int, float, char, boolean	**24**	**21**
	Implementation	8	13
	Design	4	4
	Testing	8	4
	Documentation	4	0
US02	FR2-A: The dissector shall be able to display simple structs	**28**	**25**
	Implementation	8	17
	Design	4	1
	Testing	8	6
	Documentation	8	1
US04	FR3-A: The utility shall support #include	**8**	**2**
	Implementation	2	0
	Design	2	1
	Testing	2	1
	Documentation	2	0
US05	FR3-B: Recognize invalid values for a struct member	**11**	**8**
	Implementation	11	8
US08	FR6-B: Command line shall support paramter for configuration file	**28**	**8**
	Implementation	8	4
	Design	8	2
	Testing	8	2
	Documentation	4	0
US06	FR4-A: Support valid ranges for struct members	**30**	**15**
	Implementation	8	8
	Design	6	3
	Testing	8	3
	Documentation	8	1
US03	FR2-D: Recognize invalid values for a struct member	**22**	**15**
	Implementation	6	9
	Design	4	4
	Testing	8	2
	Documentation	4	0
	Total:	108	95

the parser to parse C files. It should finally ask the Lua script generator to create Wireshark dissectors for the given header files. The Command Line

Table 8.2: Sprint 1 Timetable

	Hours	
Description	Est.	Act.
Design	30	10
Implementation	44	65
Testing	50	20
Documentation	36	3
Total:	160	98

Interface (CLI) should be able to accept some additional arguments like verbose, debug, nocpp and output options. The argument verbose should print out detailed information, debug should print out debugging information, and nocpp should disable the C preprocessor. If the CLI is provided with invalid arguments, it should print a message explaining the correct usage. If the program ran as expected it should output a message informing the user of the success.

Configuration The configuration should parse the configuration files given to the command line interface, and from this information modify the data structures generated by the parser.

Parser The most important part of the utility is the parser, it should be able to parse C files and look for struct definitions. The struct definitions will be put into a data structure that the Lua script generator will use when creating the dissectors. The parser should use pycparser and PLY libraries for parsing of C files. It should accept C header files and create an abstract syntax tree, which is traversed to find struct definitions and their members. The struct definitions and members are then filled into a suitable data structure.

Data structures The data structures should store the information the Lua script generator needs to generate Wireshark dissectors. It should have support for protocols and fields in Wireshark. The data structures should be easily modifiable by the configuration.

Lua script generator This is the part of the system that generates the actual Lua dissectors. It should use the information in the data structures, from the parser and configuration, to create Lua code that can dissect the header files specified by the user.

8.2.2 System Overview

Figure 8.2 illustrates the current class diagram for our utility after the end of the first sprint. The relationship between the modules can be seen in Figure 8.3. The csjark module contains the main method of the utility and is responsible for running the program. It is in this module we have implemented the functionality for the command line interface.

The utility will typically start off by using cparser to parse the C header file given to the utility as a command line argument. cparser will then use the config module to ensure that the parsing is done correctly after the configuration, and then generate protocols and fields to be used in the csjark module. The csjark module then generates a Wireshark dissector in Lua code by going through the protocols and fields generated earlier by the cparser module.

In this sprint we added configuration support for ranges, this was done by adding a RangeRule class to the config module. This class specifies how range rules should be written in the configuration. In the dissector module we created a RangeField class that interprets the rules and recognizes any invalid values when generating the dissectors. The RangeField class inherits from the Field class.

Csjark

The utility's *main()* function is a part of the csjark module. The function will start parsing of CLI arguments, start parsing of the configuration files, and start generating dissectors by calling *createDissectors()*. When all structs have been generated, it will print the results to the user on the command-line interface.

The Cli class will hold all the parameters, and take the parameters from the command-line interface, it will parse the arguments and return lists for header and config.

Cparser

The Cparser module is used to parse the header files, and has three methods and a class. The preprocessing is done in *parse_file()*, *parse()* is used to parse the file with pycparser library. To find a struct in the abstract syntax tree, the function *find_structs()* is used.

StructVisitor is a class that goes through the abstract syntax tree, which the pycparser library generates. The class will handle all the declaration, and return a struct with the members.

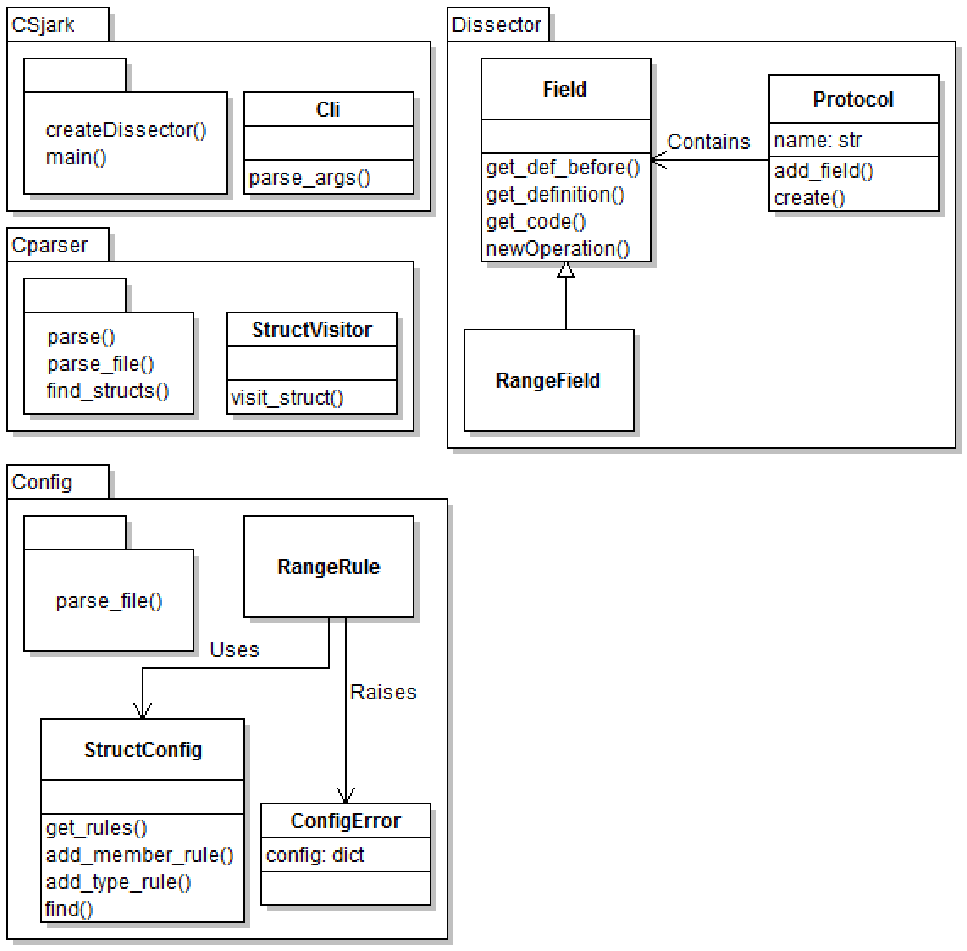


Figure 8.2: Sprint 1 Class Diagram

Dissector

This module has three classes that generates the Lua dissector.

- Protocol: This class generates the Lua dissector from a C struct. It will use Field class for each of the struct members.
- Field: Field is a class that generates code for each of the struct members. Field will be used to generate code for basic data types, and will be a superclass for other data types and for fields that are configured.
- RangeField: Is a subclass of Field. It is modified in order to generate a field that will give a warning for an invalid value in Wireshark.

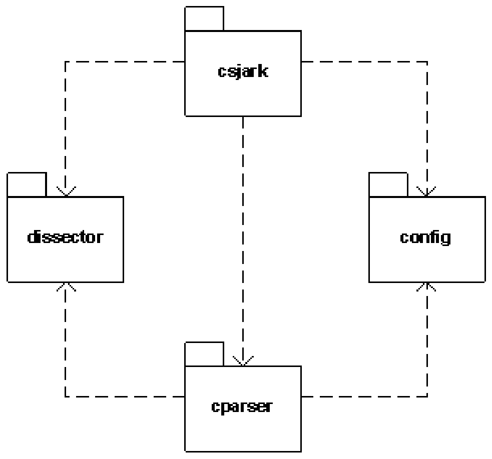


Figure 8.3: Sprint 1 Module Diagram

Config

This module consist of three classes, and a method to parse all the configuration files.

- StructConfig: This is a class used to store configuration for the C structs, and the configured members of the structs.
- RangeRule: This class is for range rules, stores information of minimum and maximum values, and the struct member.
- ConfigError: This class will raise an exception, if pyYAML is unable to parse the config file due to a error message.

8.2.3 User Stories

This section lists the user stories for the first sprint. They are displayed in Table 8.3 and Table 8.4. As we are developing a very technical utility, we have written user stories with an implementation level of abstraction. These user stories represent how we intend to add the functionality of each requirement to the utility. Each user story also contains information on how the modules of CSjark should interact with each other.

Table 8.3: User Stories - Sprint 1 Part 1

Header	Value
ID	US01
Requirements	FR1-A: The utility should support the following basic data types: int, float, char and boolean.
What	The administrator wants the utility to support structs with members of basic data types in input files. These are the basic data types in C that we support: int, float, char and boolean.
How	The C parser library, pycparser, provides this support for us. The input is fed to the cparser module, which extracts the definitions from an abstract syntax tree generated by the parser.
Result	The utility supports input with C structs with int, float, char and boolean members.
ID	US02
Requirements	FR2-A: The utility shall be able to display simple structs.
What	The developer wants Wireshark to display simple structs.
How	The dissector module shall generate Lua dissectors for Protocols created by the cparser module. The Lua dissectors shall use Wireshark's API to display structs with basic members.
Result	Simple C structs can be dissected in Wireshark by our auto generated Lua dissectors.
ID	US03
Requirements	FR2-D: Recognize invalid values for a struct member.
What	The developer wants Wireshark to give a warning if a struct contains invalid values.
How	The dissectors module will generate fields that will display warning in Wireshark when the value is outside the configured range. Wireshark will change the background color to yellow for fields with an invalid value, and give an error message.
Result	The developer can see when a value is out of range.
ID	US04
Requirements	FR3-A: The utility should support #include.
What	The administrator wants the utility to support the #include-statement inside a header file.
How	The cparser module feeds the input into an external tool, the C preprocessor, which supports #include-statements. It returns a C code file with all the included code from the external files.
Result	The utility supports #include.
ID	US05
Requirements	FR3-B: The utility should support #define and #if.
What	The administrator wants the utility to support #define- and #if-statement inside a header file.
How	The cparser module feeds the input into an external tool, the C preprocessor, which supports #define- and #if-statements. It returns a copy of the file with the statements removed, and their actions performed.
Result	The utility supports #define and #if functionality.

Table 8.4: User Stories - Sprint 1 Part 2

Header	Value
ID	US06
Requirements	FR4-A: Configuration must support valid ranges for struct members.
What	The administrator wants to be able to specify valid ranges for struct members in a configuration file.
How	The config module should read config files provided to the command line interface, and find any rules regarding valid ranges. The rules are used by the cparser when it translates struct definitions to Protocol and Field instances found in the dissector module.
Result	The administrator can specify valid ranges of struct members in the configuration.
ID	US07
Requirements	FR6-A: Command line shall support parameter for C-header file.
What	The administrator wants the utility to generate a dissector from a C-header file that he specifies.
How	The command line user interface will accept a number of arguments from the administrator. The argument that includes the path to an existing C header file is sent to the parser module. The parsing of arguments is done with the help of argparse library, which supports optional and positional arguments etc.
Result	The command line interface supports C-headers as input.
ID	US08
Requirements	FR6-B: Command line shall support parameter for configuration file.
What	The administrator wants to provide the utility with one ore more configuration files.
How	The command line interface should accept arguments that specifies the path of existing configuration files, and feed these to the config module. The config module should read the files and store them in configuration data structures. These data structures will be available to cparser to help when translating structs to Protocol and Fields from the dissector module.
Result	The administrator can input configuration files in the command line.

8.3 Implementation

In this sprint we have created a very naive implementation of the utility. It supports the most basic types of C structs. In addition, the utility supports some very basic configuration. The main reason for the naive implementation, was to find out that the libraries that the team had chosen was suitable for the utility, and that we had understood the task.

To get a better understanding of how the different requirements were implemented, look at the user stories for sprint 1 in subsection 8.2.3.

8.3.1 CLI Support for Header File

The tool uses a command line interface. The user inputs a C header file to the command line, and the program outputs a Wireshark dissector written in Lua. Below you can see a Figure 8.5 that illustrates how the program is run.

8.3.2 Support Basic C Data Types

In this first sprint the focus was to generate dissectors from structs with basic data types. These data types included integers, floats, char, boolean and arrays of chars. All different types of integers and floats were also implemented in the utility, most of the functionality was included in the pycparser library, and sizes for the different data types was specified in the utility.

8.3.3 Display Simple Structs

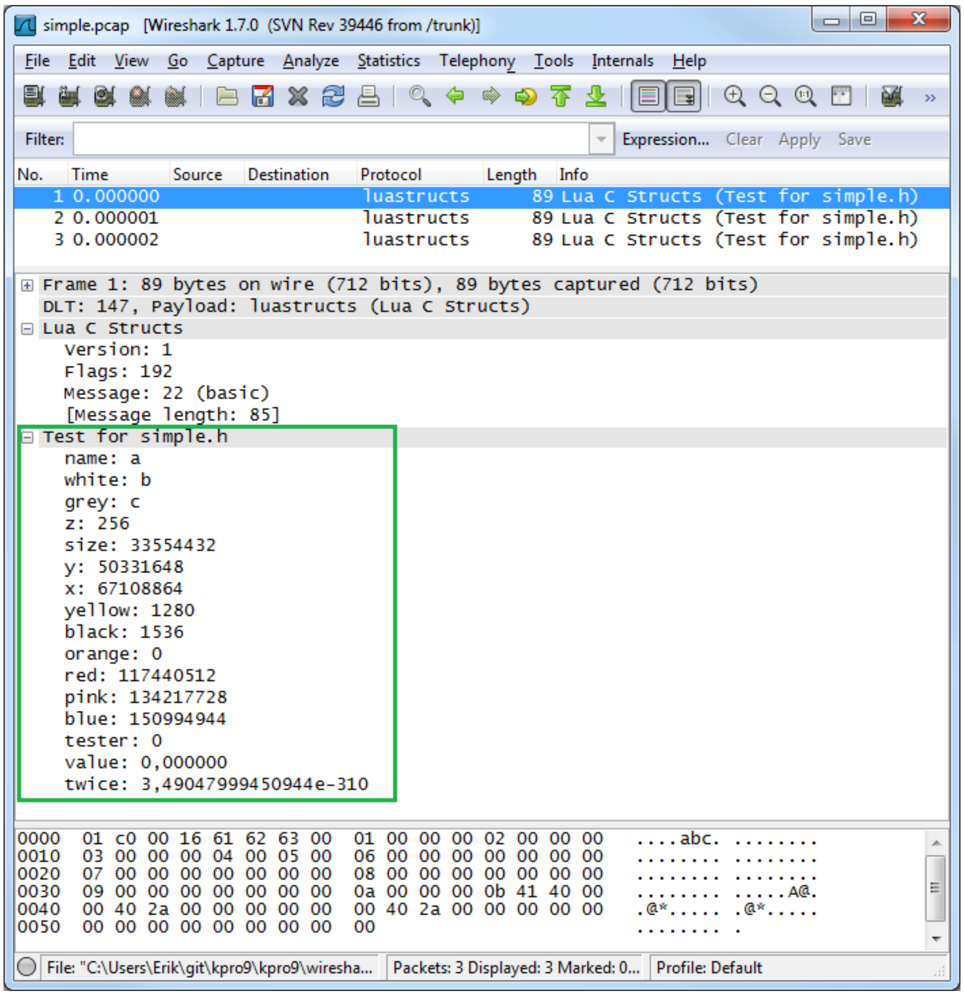
The dissectors that our utility generates is used to display the packets that are captured by Wireshark. Figure 8.4 shows an example of a packet capture. The example include a struct that is used to test different basic data types, for example, signed char, char, short, int, long int, float and double.

8.3.4 Support #include

The preprocessor in C is the first step of compilation. One of the most used preprocessor directives is #include, which include the content of a file in the compilation. Since the utility reads structs from header-files, it is possible that struct member may have been defined in other header-files, therefore #include is a important part of the utility. The preprocessor in this utility will replace the #include-line with the content of the included file.

8.3.5 Support #define and #if

These two preprocessor directives are important for our utility. #define is used to define a preprocessor macro that replaces a token with a sequence of


simple.pcap [Wireshark 1.7.0 (SVN Rev 39446 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

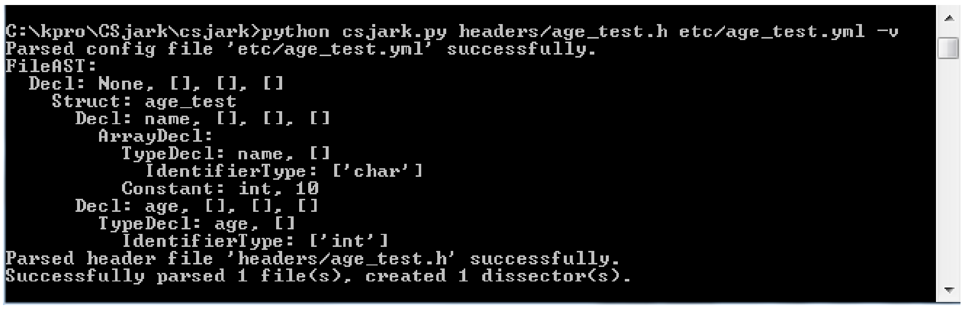
```
No.  Time        Source  Destination  Protocol     Length  Info
  1 0.000000                          luastructs       89 Lua C Structs (Test for simple.h)
  2 0.000001                          luastructs       89 Lua C Structs (Test for simple.h)
  3 0.000002                          luastructs       89 Lua C Structs (Test for simple.h)

⊞ Frame 1: 89 bytes on wire (712 bits), 89 bytes captured (712 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊟ Lua C Structs
    Version: 1
    Flags: 192
    Message: 22 (basic)
    [Message length: 85]
⊟ Test for simple.h
    name: a
    white: b
    grey: c
    z: 256
    size: 33554432
    y: 50331648
    x: 67108864
    yellow: 1280
    black: 1536
    orange: 0
    red: 117440512
    pink: 134217728
    blue: 150994944
    tester: 0
    value: 0,000000
    twice: 3,49047999450944e-310

0000  01 c0 00 16 61 62 63 00  01 00 00 00 02 00 00 00   ....abc. ........
0010  03 00 00 00 04 00 05 00  06 00 00 00 00 00 00 00   ........ ........
0020  07 00 00 00 00 00 00 00  08 00 00 00 00 00 00 00   ........ ........
0030  09 00 00 00 00 00 00 00  0a 00 00 00 0b 41 40 00   ........ .....A@.
0040  00 40 2a 00 00 00 00 00  00 40 2a 00 00 00 00 00   .@*..... .@*.....
0050  00 00 00 00 00 00 00 00  00                        ........ .

File: "C:\Users\Erik\git\kpro9\kpro9\wiresha...  Packets: 3 Displayed: 3 Marked: 0...  Profile: Default
```

Figure 8.4: Wireshark: Simple Lua dissector

characters. This can for example be used to define length of arrays, define values for enumeration or define a platform. #if and #ifdef are conditional directives that the utility will use to check if macros are defined. This will be used when the utility is generating dissectors for different platforms. The functionality for these preprocessor directives and the other preprocessor directives for C, was implemented by the library we use, which is cpp.exe for Windows and GCC for Mac and Linux.

8.3.6 CLI Support for Configuration File

To be able to parse the configuration file, it is necessary to specify the configuration-file in the CLI. An example of this is shown in Figure 8.5, where a header-file and a config-file is input to the utility. The option ”-v” is specified, which will display the abstract syntax tree when the utility has generated the dissector.



```
C:\kpro\CSjark\csjark>python csjark.py headers/age_test.h etc/age_test.yml -v
Parsed config file 'etc/age_test.yml' successfully.
FileAST:
  Decl: None, [], [], []
    Struct: age_test
      Decl: name, [], [], []
        ArrayDecl:
          TypeDecl: name, []
            IdentifierType: ['char']
          Constant: int, 10
      Decl: age, [], [], []
        TypeDecl: age, []
          IdentifierType: ['int']
Parsed header file 'headers/age_test.h' successfully.
Successfully parsed 1 file(s), created 1 dissector(s).
```

Figure 8.5: Command Line Screenshot

8.3.7 Configuration of Valid Ranges

We decided to use YAML as our configuration format. In this sprint we have added support for range specification. A user can specify the ranges of the members of a struct, from a minimum to a maximum. Listing 8.1 shows an configuration in YAML for the header-file in Listing 8.2. The configuration specifes that the struct member age must have a value between 0 and 100.

Listing 8.1: Configuration of valid ranges

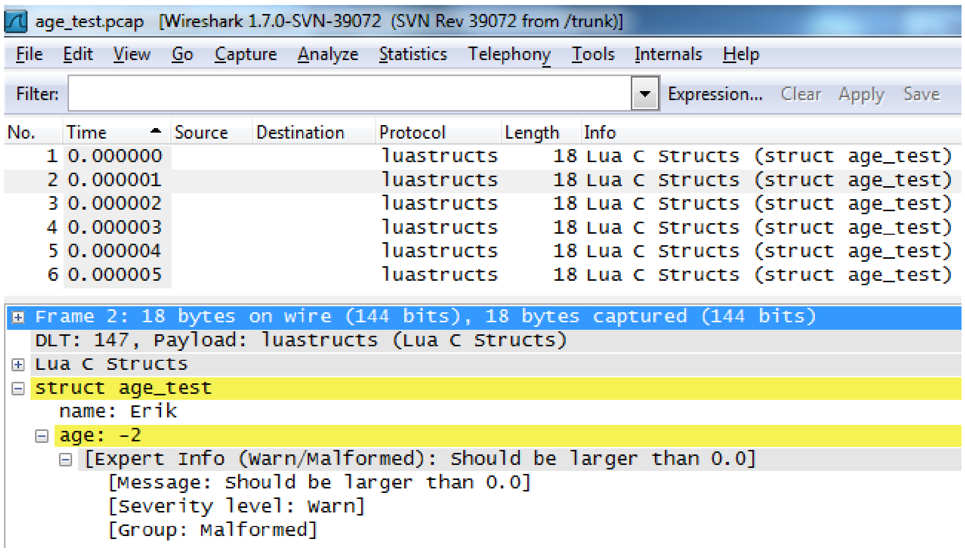
```
Structs:
  - name: age_test
    ranges:
      - member: age
        min: 0
        max: 100
```

Listing 8.2: Header-file for age_test

```
#define STRING_LEN 10

struct age_test {
    char name[STRING_LEN];
    int age;
};
```

8.3.8 Dissector Shall Recognize Invalid Values

In Figure 8.6, you can see how Wireshark displays a member that has an invalid range. When a invalid value exist, the dissector will warn the user with a message and description, this make it easier for debugging.


age_test.pcap [Wireshark 1.7.0-SVN-39072 (SVN Rev 39072 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time      Source  Destination  Protocol    Length  Info
   1 0.000000                       luastructs      18  Lua C Structs (struct age_test)
   2 0.000001                       luastructs      18  Lua C Structs (struct age_test)
   3 0.000002                       luastructs      18  Lua C Structs (struct age_test)
   4 0.000003                       luastructs      18  Lua C Structs (struct age_test)
   5 0.000004                       luastructs      18  Lua C Structs (struct age_test)
   6 0.000005                       luastructs      18  Lua C Structs (struct age_test)

⊞ Frame 2: 18 bytes on wire (144 bits), 18 bytes captured (144 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊞ Lua C Structs
⊟ struct age_test
    name: Erik
  ⊟ age: -2
    ⊟ [Expert Info (Warn/Malformed): Should be larger than 0.0]
        [Message: Should be larger than 0.0]
        [Severity level: Warn]
        [Group: Malformed]
```

Figure 8.6: Wireshark: Invalid Values

8.4 Sprint Testing

This section introduces the tests performed during the sprint and their results.

8.4.1 Test Results

During the sprint the team executed a total of seven test cases. An example of a test case run this sprint can be seen in Table 8.5 while the rest of the test cases can be found in Appendix C. These are the results for the tests the team ran for sprint 1 as according to Table 6.2 discussed in the test plan. The test results of the tests can be seen in Table 8.6.

Table 8.5: Test Case TID01

Header	Description
Description	Supporting parameters for C-header file
Tester	Lars Solvoll Tønder
Prerequisites	The utility must have been installed on the system and there needs to exist a header file associated with this test
Feature	Test that we are able to feed the solution with a C-header file and have it get dissected
Execution	1. Feed the utility with the name of the C-header file associated with this test through the command line 2. Read the output given by the program
Expected result	2. The user should be presented with some text expressing the success of creating a dissector

Table 8.6: Sprint 2 Test Results

Header	Description
Test ID	TID01
Description	Supporting parameters for C-header file
Tester	Lars Solvoll Tønder
Date	27.09.2011
Result	Failure. The Lua-file got created successfully but the user was not informed of the result
Test ID	TID02
Description	Supporting basic data types
Tester	Lars Solvoll Tønder
Date	27.09.2011
Result	Failure. The program supports the use of int, float, char and boolean, but did not inform the user of the result
Test ID	TID03
Description	Displaying simple structs
Tester	Lars Solvoll Tønder
Date	27.09.2011
Result	Success
Test ID	TID04
Description	Supporting C-header files with the #include directive
Tester	Lars Solvoll Tønder
Date	27.09.2011
Result	Failure. The program supports header files with the #include directive, but did not inform the user of the result
Test ID	TID05
Description	Supporting #define and #if
Tester	Lars Solvoll Tønder
Date	27.09.2011
Result	Failure. The program supports header files with the #define and #if directives, but did not inform the user of the result
Test ID	TID06
Description	Supporting configuration files
Tester	Lars Solvoll Tønder
Date	27.09.2011
Result	Failure. The program supports the use of configuration files but does not inform the user of any results
Test ID	TID07
Description	Recognizing invalid values
Tester	Lars Solvoll Tønder
Date	27.09.2011
Result	Success

8.4.2 Test Evaluation

Most of our tests failed because the developers had forgotten to implement usability features presenting the user with any textual information. With all core functionality in place, it only took three lines of code to fix this issue.

8.5 Customer Feedback

This section covers the feedback we got from the customer before and after the first sprint. Changes to the requirements are described in subsection 5.2.4.

8.5.1 Pre-sprint

The customer had no objections to the content of the first sprint, but was not completely satisfied with the feature descriptions. They thought that we should write more implementation details for each requirement, and that each requirement should be properly broken down. They felt that implementation and design should be two separate tasks if design is necessary. Corollary to this, our work items were too big. They also suggested a proper finish condition for each work item.

8.5.2 Post-sprint

We presented the result from the sprint 1 for the customer and they were very happy with the result. They had some ideas for how we could make our configuration files more compact, but said that it was not really important. Their other comments were mostly on what they wanted us to do for sprint 2 and how those features might be implemented.

8.6 Sprint Evaluation

This section contains the team evaluation of the first sprint.

8.6.1 Review

The first sprint is over and the team has implemented a working utility for simple structs. During the sprint planning we decided which requirements from the product backlog that we were to fulfill during the first sprint. The product backlog contains a prioritized list of requirements. We decided to include the requirements that had the highest prioritization. These requirements were basic, but essential and therefore highly prioritized.

The lack of prior knowledge to Scrum made planning and execution of the sprint complicated for the team. We did not agree within the team of how to do it, and wasted some time on discussions. In the end, we understood that we wrote a too high level description of the requirements, and had to redo previous work. This was time consuming, stealing person-hours from other parts of the sprint.

Each requirement is divided in four parts: design, implementation, testing and documentation. We currently have most of these parts covered, except documentation for all files and unit test for the dissector file. This work is postponed till the second sprint.

The first sprint resulted in a solid core for our project. The customer was happy with the first demo they got, the utility even run on Mac. We feel that we have a good start and are confident that we will be able to give the customer the product that they want in the end.

The burndown chart, Figure 8.7, shows the progress during the first sprint. The team made an effort in making a correct estimation. This is reflected in the chart, as the estimated and actual hours are following each other closely. In the end, actual hours stopped at 11 hours, which means we have some uncompleted tasks. These are put back in the product backlog, and will probably be part of the next sprint.

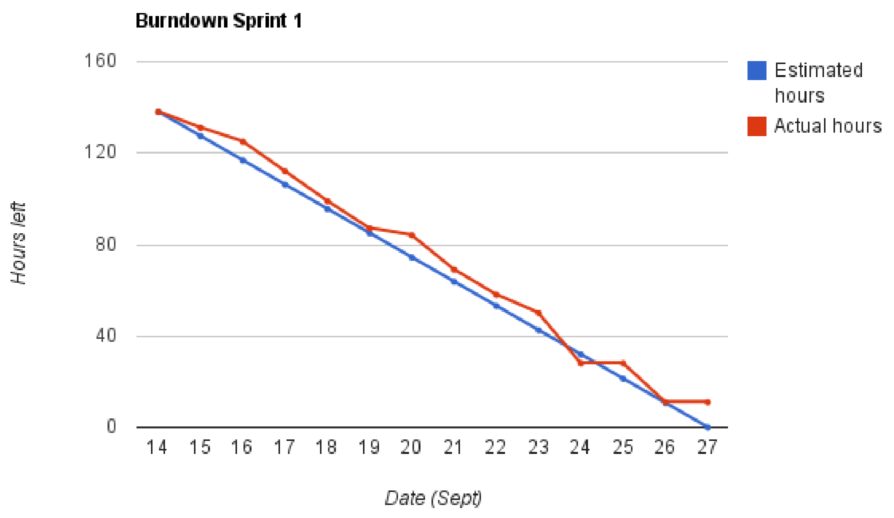


Figure 8.7: Sprint 1 Burndown Chart

8.6.2 Positive Experiences

- Good customer communication and relation from the beginning
- Implementation went smoothly
- Most of the requirements for this sprint were completed
  - all design, implementation and testing were completed
  - documentation had to be put back in the product backlog
- Errors and bugs were detected and corrected swiftly and with relative ease
- The team functioned well, with sufficient discussions and conflicts

8.6.3 Negative Experiences

- Hard for the team member to ”give” 25 person-hours each week
  - to understand that it is needed
  - to free up so many hours, and still have time to do other subjects
- Hard to find time for meeting/work where all team member was able to meet
- One member of the team was sick for a week
- We did not document the process and work during the first sprint well enough, which was a burden at the end of the sprint
- We did perhaps not understand Scrum properly, which resulted in extra work

8.6.4 Planned Actions

Below the planned actions for the following sprints are listed.

Better Sprint Planning

In order to avoid having to redo much of the work because of incorrect/poorly sprint planning, we have decided to do this properly next time. We have learned what we need to have in place and how to document it from this sprint.

Design Early in the Sprint

The design should be in place early in the sprint. This is related to better sprint planning, the planning should be so detailed/good that additional design is not necessary. This is important for understanding and being able to estimate hours and divide work.

Documenting in Parallel while Implementing

We suffered from the problem, code first then document. This is not a good practice in team divided work. We will try to do the documentation as we code. The documents for the project report also experienced a standstill while the team worked on implementation. Writing parts of sprint documentation while having the sprint is a much better way to work, then most of the documentation is already done before the sprint evaluation.

Split Coding and Report Writing Between Team Members

Not all the team members have to do coding. It is important to maintain a steady progress making the project report, while doing the implementation. Responsibilities for coding and report will be assigned next sprint.

8.6.5 Barriers

Some of the team member have experienced some technical difficulties with their Git client, and others had problems setting up PyCharm. Problems rose as we realized that it would be hard setting up the programs on different platforms.

We had problems with C parser library, pycparser, which did not support _Bool type, specified in the C99 standard. A patch for this was written by Even, and was later included in the pycparser library. There was also problems with the testing framework, attest, this framework did not have support for Windows command line prompt, a patch was written and was later added to their library.

Some team members had conflicting deadlines for deliveries in different courses.

CHAPTER 9 SPRINT 2

This chapter describes the results and process of sprint 2. In section 9.1, the planned changes from the first sprint is explained, the planning of this sprint is covered in section 9.2, and the design of the first sprint is explained in section 9.3. To give a better understanding of the requirements, section 9.4 explains the features implemented in this sprint. The results from the tests ran is described in section 9.5. The feedback from the customer is covered in section 9.6. In section 9.7, the results from the sprint retrospective is covered.

9.1 Pre-sprint

Sprint 1 gave the desired result, but the team was not satisfied with the way the Scrum process was conducted, especially the sprint planning. In sprint 1 retrospect we decided to conform more to, in our mind, proper Scrum. We will apply experience and advice from the first sprint, to get a better process.

In this planning meeting we will try to create more descriptive work items for the sprint backlog. This will ease the process of design, implementation, testing and documentation of the utility, and we do not have to redo any parts that will end up in the report in order to assist the reader. We focus on good user stories to ensure that the elements are of a low enough level.

9.2 Sprint Planning

The first sprint resulted in a solid core for the utility. During the next sprint iteration, the core will be extended with more advanced functionality. After this sprint, the utility will have most of the functionalities it need to work in a real environment, and will probably be able to aid Thales in some of their operations.

Since it is difficult to understand the complexity of all the requirements in the sprint backlog, the team ended in up with an uncertain person-hours estimate for some work objects. After the sprint is finished, we will see if we understood the complexity, and assigned enough hours to implement it. The more complex, but not so critical functionalities will be part of sprint 3 and 4.

9.2.1 Duration

According to the work breakdown structure, Table 3.1, the planning meeting of the second sprint should have been conducted the 5th of October. After a request from the customer to see our planning for the second sprint at the weekly customer meeting, which was scheduled to be before our planning meeting the same day, we decided to advance the planning to the 4th of October. This is to maintain the good relationship with the customer and submit to their preference.

The sprint started with the planning meeting the 4th of October and our work started the following day. The sprint duration is 14 days, and will end the 18th of October with a review meeting.

9.2.2 Sprint Goal

In the second sprint we will build on the core created in the first sprint. During the sprint, we will extend the functionality with more comprehensive and advanced features. Most of the requirements we intend to fulfill in this sprint had to be done subsequent to the first sprint, because the structure and design of the core had to be in place first. The requirements that we selected for this sprint are a natural advancement on the road to making the utility that the customer wants.

One of the most crucial functions to work in a real environment, is the support for nested header-files. The handling of the #include-statement gives the utility this feature. The goal of the sprint is to implement the #include and mainly to have support for enums, bit strings, endianness and batch mode.

9.2.3 Back Log

The second sprint we will implement thirteen requirements. These are listed in Table 9.1 and Table 9.2.

Table 9.1: Sprint 2 Requirements Part 1

			Hours	
#	Req.	Description	Est.	Act.
1	FR1-B	Support members of type enums	6	5
		Implementation	3	2
		Testing - unit	1	1
		Testing - end to end	2	2
2	FR1-C	Support members of type structs	7	3.5
		Implementation	6	3
		Testing - unit	1	0.5
3	FR1-F	Detect structs with same name	3	3.5
		Implementation	2	2.5
		Testing - unit	1	1
4	FR2-B	Support display of structs within structs	11	15
		Implementation	5	6
		Testing - unit	2	2
		Testing - end to end	4	7
5	FR4-F	Support enumerated named values	5	6.5
		Design	1	0.5
		Implementation	1	0.5
		Testing - unit	1	2.5
		Testing - end to end	1	1.5
		User documentation	1	1.5
6	FR4-G	Support for bit strings	10	11.5
		Design	2	2
		Implementation	3	6
		Testing - unit	2	1
		Testing - end to end	2	1
		User documentation	1	1.5
7	FR1-E	Support members of type array	7	12
		Implementation	3	7
		Testing - unit	1	1
		Testing - end to end	3	4

9.3 System Design

For sprint 2 the team decided to re-factor some of the code in order to make it easier to read and to split the functionalities of the utility in such a way that it reduces coupling within the system. Some new functionality was also added on the parser side in order to get the utility to recognize the datatypes mentioned in the sprint 2 backlog. This new functionality is represented by the addition of new classes in the dissector and config modules. These classes include the BitField, ArrayField, EnumField and

Table 9.2: Sprint 2 Requirements Part 2

#	Req.	Description	Hours	
			Est.	Act.
8	FR4-E	Structs with various trailers	18	15
		Design	3	2
		Implementation	6	6
		Testing - unit	2	1.5
		Testing - end to end	5	3.5
		User documentation	2	2
9	FR4-B	Support for custom Lua configuration	14	7
		Design	2	1
		Implementation	5	5
		Testing - unit	1	0
		Testing - end to end	4	1
		User documentation	2	0
10	FR4-D	Dissector ID	4	3
		Implementation	1	1
		Testing - unit	1	1
		User documentation	2	1
11	FR5-C	Endian handling	11	0.5
		Implementation	5	0.5
		Testing - unit	2	0
		Testing - end to end	6	0
12	FR6-C	Batch processing; folder support in the CLI	7	4.5
		Implementation	4	2
		Testing - unit	2	1.5
		User documentation	1	1
13	FR4-C	Support custom handling of specific data types	5	3
		Implementation	2	1.5
		Testing - unit	1	0.5
		Testing - end to end	1	1
		User documentation	1	0
14		Non-requirement programming tasks	26	25.5
		Refactoring	3	7
		Bug fixing		2
		Manual end-to-end tests	8	4
		Automatic end-to-end tests	4	3
		Misc unit tests	4	3
		Misc user documentation	7	6.5
		Total:	134	115.5

ProtocolField classes, which contain the functionality required to handle bitstrings, arrays, enums and structs within structs respectively. The classes

Table 9.3: Sprint 2 Timetable

	Hours	
Description	Est.	Act.
Sprint planning	**30**	**35.5**
Sprint 2 requirements	**134**	**115.5**
Design	8	5.5
Implementation	44	43
Testing	46	35
User Documentation	10	6.5
Other	26	25.5
Sprint review	**20**	**17.5**
Report documentation	**58**	**69**
Sprint 1 document	14	5
Sprint 2 document	44	54
Report document		10
Lectures	**25**	**27.5**
Meetings	**55**	**48**
Advisor meetings	28	26
Customer meetings	6	8
Stand-up meetings	21	14
Project Management	**20**	**40**
Total:	342	353

added to the config module include the Trailer, Bitstring, Custom and Enum classes, which handles the configuration needs for structs with trailers, bit strings, custom handling of data types and enums respectively. Other than that, there were no other changes or additions made to the design during sprint 2.

9.3.1 System Overview

Figure 9.1 shows the class diagram the team made for sprint 2. The main differences from sprint 1 are the additions of new classes, extending the functionality of the utility so it can handle more complex header files than it could before. The developers also generalized some of the modules and used inheritance to support the re-use of code. Figure 9.2 shows the dependencies between the modules, the only difference here, is that csjark do not use dissector, after some refactoring of the code.

The addition to the class diagram have been mainly to support more data types for C in the cparser and support for more configuration options in config module, and generation of the fields in dissector module.

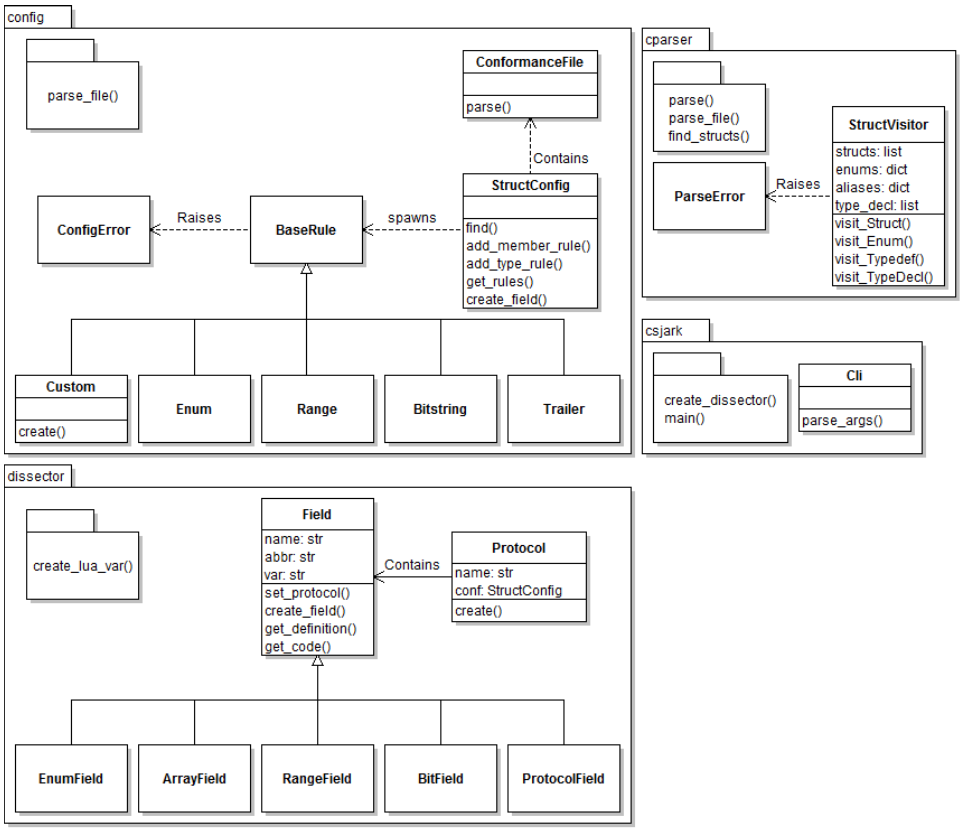


Figure 9.1: Sprint 2 Class Diagram

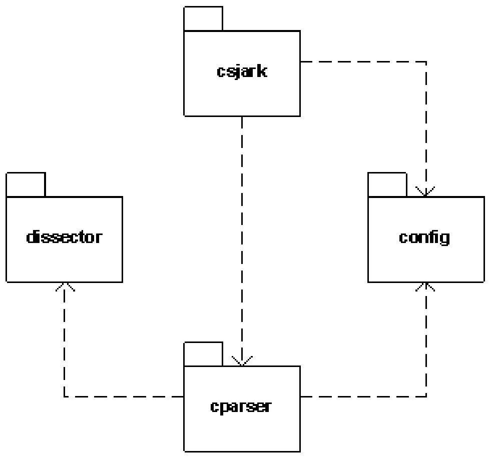


Figure 9.2: Sprint 2 Module Diagram

Csjark

The main differences in this module was changes in the Cli class, which now has functionality to find all files in a folder given as an argument when running CSjark, with the *files_in_folder()*. This method used for the support of batch mode generation.

Cparser

In the cparser module, a class named ParserError was added, its main purpose is to raise an exception when our utility does not support a C data type. StructVisitor has been extended to support more data types, for example structs, arrays and enums.

Config

For all the configuration added in this sprint, there has been added a class for each configuration. For this there has been added a superclass, named BaseRule. This class have all the functionality that the configuration rules share.

BaseRule has five subclasses, Custom, Enum, Range, Bitstring and Trailer. These subclasses have the configuration that are specific for these configuration types.

The last class added in this module is ConformanceFile, which hold the configuration written for custom Lua code.

Dissector

Four subclasses of Field was added in the dissector module. These classes are EnumField, ArrayField, BitField and ProtocolField, and they will generate different types of fields for the Lua dissector, which will display members differently in Wireshark.

Field and Protocol classes has been extended to support the new fields, and support for modifying configuration of the functionality in Wireshark.

9.3.2 User Stories

This section lists the user stories for the second sprint. They are displayed in Table 9.4, Table 9.5, Table 9.6 and Table 9.7. These user stories represent how we intend to add the functionality of the requirements for the second sprint, and explain how the modules in the utility should interact with each other in the different scenarios.

Table 9.4: User Stories - Sprint 2 Part 1

Header	Value
ID	US09
Requirement	FR1-B: The utility must support members of type enum
What	The administrator wants the utility to support structs with members of type enum.
How	When the cparser module detects an enum member in a struct, the cparser should search in an enum dictionary and the enum member will be mapped to the correct value found in the dictionary. A field representing the enum will be added to the prototype object corresponding to the enclosing struct.
Result	The utility supports members of type enum.
ID	US10
Requirement	FR1-C: The utility must support members of type struct
What	The administrator wants the utility to support structs with members of type struct.
How	When the cparser module detects a struct in the AST that has a struct member, the cparser searches for its definition in the dictionary of previously detected structs. When it finds it, it looks up the identification number and the size of the inner struct and creates a struct_field object inside the prototype object corresponding to the outer struct.
Result	The utility supports members of type structs.
ID	US11
Requirement	FR1-F: The utility should detect structs with the same name, and report it as an error
What	The administrator wants the utility to report an error if it discovers structs with the same name to avoid unforeseen name collisions.
How	When the cparser module traverses the AST to look for structs, it will detect if there are structs with the same name by searching in a database of all structs it has found so far. If a collision is detected the utility will crash with an error message.
Result	The utility will detect duplicated name of structs.
ID	US12
Requirement	FR2-B: The dissector shall be able to support structs within structs
What	The utility should be able to create a Lua dissector that correctly displays structs within structs in Wireshark.
How	For each struct definition encountered in cparser, a prototype object is created. This object will include an identifier number used to locate the Lua dissector for that struct. When a struct member is located inside an outer struct. The dissector module encodes the identification number and the size for the inner struct into the Lua dissector for the outer struct. The identification number is used to access the dissector for the inner struct when the outer struct dissector is used. The outer struct dissector uses the size of the inner struct to know how much of the network package to forward to the inner struct dissector. The size and identification number of the inner struct will be available in the struct field corresponding to the inner struct inside the protocol object corresponding to the outer struct.
Result	The dissector module supports nested structs
ID	US13
Requirement	FR4-F: Configuration must support integer members which represent an enumerated named value
What	The administrator wants to specify integer members, represented by an enumerated named value, in a configuration file.
How	The config module should read config files provided to the command line interface, and find any rules regarding enumerated integer values. The rules are used by the cparser when it translates struct definitions to Protocol, and makes the cparser create EnumFields instead of normal Fields for the specified members.
Result	Enum members can be specified in the configuration.

Table 9.5: User Stories - Sprint 2 Part 2

Header	Value
ID	US14
User doc	FR4-F: User documentation for writing configuration for integer members which represent an enumerated named value.
What	The administrator should be able to educate himself of how to give CSjark the necessary information to get integer values mapped to names in the generated dissector.
How	The administrator opens the user documentation and finds the section about configuration. From here he locates the sub section about enumerated names integer values. This section gives a good description of how to write such configuration, and the user is able to implement his desired configuration after reading through once and looking at provided examples.
Result	The administrator is now able to use the enumerated named value functionality.
ID	US15
Requirements	FR4-G: Configuration must support members which are bit strings
What	The administrator wants to specify members that represent bit strings in the configuration.
How	The config module should read config files provided to the command line interface, and find any rules regarding bit strings. The rules are used by the cparser when it translates struct definitions to Protocol and BitField instances found in the dissector module.
Result	The administrator can specify bit string members in the configuration.
ID	US16
User doc	FR4-G: User documentation for writing configuration for integer members which are bit strings.
What	The administrator should be able to educate himself of how to give CSjark the necessary information to get integer values mapped to bit string in the generated dissector.
How	The administrator opens the user documentation and finds the section about configuration. Then he locates the section about bit string values. This section gives a good description of how to write such configuration, and the administrator is able to implement his desired configuration after reading through once and looking at the examples.
Result	The administrator is now able to configure CSjark to generate dissector that recognises and formats bit strings correctly.
ID	US17
Requirements	FR1-E: The utility must support members of type array
What	The administrator wants the utility to support structs with members of type array.
How	When the cparser module finds an array declaration, it recursively traverses the tree until till it encounters the bottom of the declaration to discover the size of the array. The parser module creates an instance of an array field with the size and type of the array. From the array field the dissector module generates a dissector which has a sub tree for each level of the array.
Result	The utility support array members in structs.
ID	US18
Requirements	FR4-E: Configuration must support various trailers (other registered protocols)
What	The administrator wants to specify trailers to a C header file in the configuration.
How	The config module should read config files provided to the command line interface, and find any rules regarding trailers. A member in a struct will say how many packets of other protocols that follows the header. In the config-file it is specified which member contains this number, and what type of protocol the packet(s) belong to. When the dissector module generates a struct containing a trailer, the correct dissector for the trailer packet(s) will be called for the rest of the buffer.
Result	The utility can handle trailer packets following the header, specified in the configuration.

Table 9.6: User Stories - Sprint 2 Part 3

Header	Value
ID	US19
User doc	FR4-E: User documentation for how to specify trailers (other registered protocols)
What	The administrator should be able to find out how to specify trailers to a header struct by reading the user documentation.
How	The administrator opens the user documentation and finds the section about configuration. From here he locates the sub section about trailers. This section gives a good description of how to write such configuration for different types of trailers and provides sufficient examples, to make it clear to the administrator how to write the configuration he needs after reading the section.
Result	The administrator is now able to utilise the trailer feature of CSjark.
ID	US20
Requirements	FR4-B: Configuration must support custom Lua files for specific protocols
What	The administrator wants to specify custom Lua files in the configuration, which are to be used in complex cases where our utility is unable to generate a dissector for the C header.
How	The config module should read config files provided to the command line interface, and find any rules regarding the use of custom Lua files. When such a rule is found, the dissector module will use the Lua code found in the file(s) in addition to its own generated Lua code. This is done by reading the custom Lua file(s) and writing the content to the relevant parts of the dissector.
Result	The administrator can specify the use of custom Lua file in the configuration.
ID	US21
Requirements	FR4-C: Configuration must support custom handling of specific types.
What	The administrator will be able to specify that a certain type should be handled in a specific way specified in a configuration file. This configuration must be a Wireshark supported Lua field. The configuration could both be a global default value for that type, or specific for a struct member.
How	The config module should read config files provided to the command line interface, and find any rules regarding the use of custom handling of types. It will modify the field added to the prototype field representing the enclosing struct with the behaviour specified in the configuration.
Result	The administrator is able to configure custom behaviour for specific types.
ID	US22
User doc	FR4-C: User documentation for configuring custom handling of specific types.
What	The administrator should be able to find out how to specify custom handling for a specific type by reading the user documentation.
How	The administrator opens the user documentation and finds the section about configuration. From here he locates the sub section about custom type handling. This section gives a good description of how to write such configuration. There should also be some examples to clarify the description. After reading the section, the administrator has a good idea of how to do the desired custom handling.
Result	The administrator is able to use custom handling of specific types.
ID	US23
Requirements	FR4-D: Configuration must support specifying the ID of dissectors
What	The administrator wants to specify the ID of a dissector in a configuration file.
How	When the cparser finds a struct in the abstract syntax tree it looks for a configuration file for the struct. If a config-file is found, the ID of the dissector is mapped to the ID given in the config-file when generating the dissector.
Result	The administrator can specify the ID of dissectors in the configuration.

Table 9.7: User Stories - Sprint 2 Part 4

Header	Value
ID	US24
Requirements	FR5-C: Generate dissectors which support both little and big endian platforms
What	The administrator wants the utility to produce dissectors that can be used on both little and big endian platforms.
How	The administrator will specify the platform he is using in a configuration file by adding a message flag and endian pair. When the dissector for a struct is being generated, the dissector module in CSjark will encode a flag to endian dictionary inside the Lua dissector file. This dictionary will be used to look up the endian for a message given its flag. If no endian is found in the dictionary, it will use a default value.
Result	The dissectors are now able to support messages from platforms with different endian.
ID	US25
Requirements	FR6-C: Command line shall support batch processing of C header and configuration files
What	The administrator wants to set up the program to run automatically, in order for the program to create dissectors from the C header and configuration file(s) that are specified.
How	When the administrator feeds the command line with an input argument, header or configuration, the utility shall check if the input is a single file or a directory (folder). If it is a file, parse it. If it is a folder, retrieve all files in that folder and add them to a list, this list will be sent to the parser and all the files will be parsed one after another. A directory within a directory should be detected, and traversed recursively. With this approach we can start in a root folder and include all files, independent of the depth. The batch mode shall only include files with the extension .h (a header file) or .yml (config file), which are the files that are going to be parsed as input.
Result	The administrator can feed the utility with folders to make dissectors of all the headers found, also called batch mode.

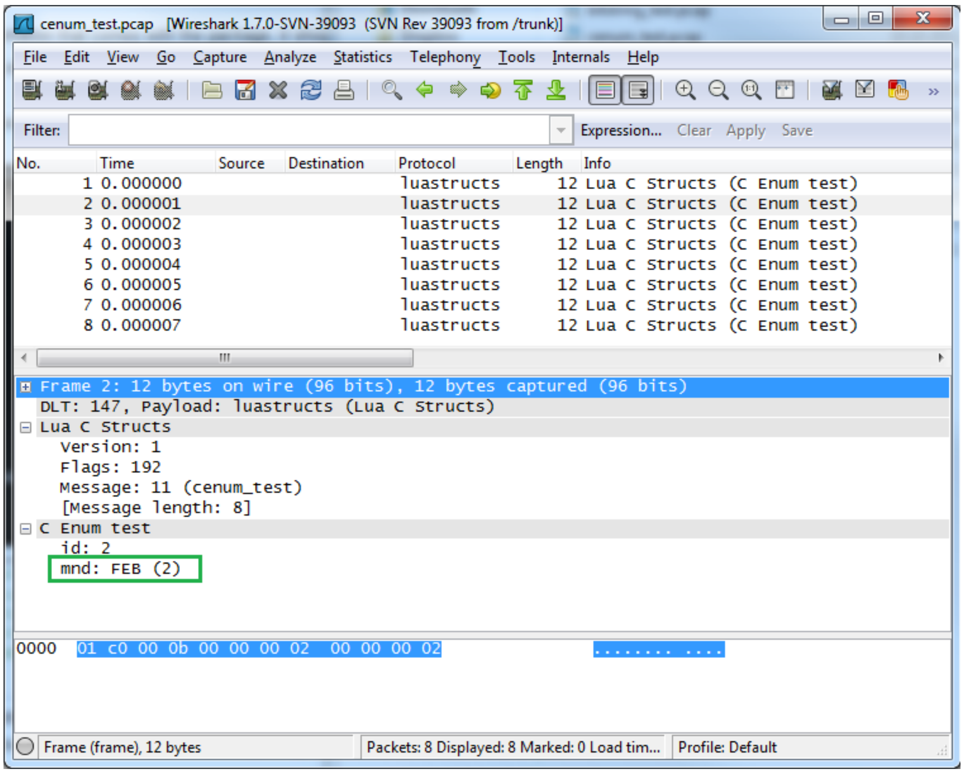
9.4 Implementation

In the previous sprint we focused on creating a naive implementation of the utility. In this sprint the focus was on implementing data types for the C programming language and making it possible to configure more options for how the dissector should function. This section will cover the requirements implemented, what the header and config files looks like, and what the ”output” looks like.

To get a better understanding of how the different requirements were implemented, look at the user stories for sprint 2 in subsection 9.3.2.

9.4.1 Support Members of Type Enum

Enum is a type declaration in C that specifies enumeration constants. Enum is supported because it is a basic data type in the C language. Listing 9.1 shows an example of an enum in a C-header file. The Wireshark dissector will display the named value, making it easier to read, an example is shown in Figure 9.3. The red rectangle shows the enumerated named value.


cenum_test.pcap [Wireshark 1.7.0-SVN-39093 (SVN Rev 39093 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save
Frame (frame), 12 bytes
Packets: 8 Displayed: 8 Marked: 0 Load tim...
Profile: Default

```
No.  Time      Source  Destination  Protocol     Length  Info
   1 0.000000                       luastructs       12 Lua C Structs (C Enum test)
   2 0.000001                       luastructs       12 Lua C Structs (C Enum test)
   3 0.000002                       luastructs       12 Lua C Structs (C Enum test)
   4 0.000003                       luastructs       12 Lua C Structs (C Enum test)
   5 0.000004                       luastructs       12 Lua C Structs (C Enum test)
   6 0.000005                       luastructs       12 Lua C Structs (C Enum test)
   7 0.000006                       luastructs       12 Lua C Structs (C Enum test)
   8 0.000007                       luastructs       12 Lua C Structs (C Enum test)
```

```
⊞ Frame 2: 12 bytes on wire (96 bits), 12 bytes captured (96 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊟ Lua C Structs
    Version: 1
    Flags: 192
    Message: 11 (cenum_test)
    [Message length: 8]
⊟ C Enum test
    id: 2
    mnd: FEB (2)
```

```
0000  01 c0 00 0b 00 00 00 02  00 00 00 02       ........ ....
```

Figure 9.3: Enumeration in Wireshark

Listing 9.1: Enum support

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
    SEP, OCT, NOV, DEC = 20 };

struct cenum_test {
    int id;
    enum Months mnd;
};
```

9.4.2 Support Members of Type Struct

Structs are an important part of the C language, a struct declaration consists of a group of different fields, these fields can have any type, also struct. This was therefore an important requirement to implement. An example is shown in Listing 9.2.

Listing 9.2: Struct support

```
#define NAME_LENGTH 10

struct postcode {
    int code;
    char town[NAME_LENGTH];
};

struct struct_member {
    int uid;
    struct postcode pcode;
};
```

9.4.3 Detect Structs with Same Name

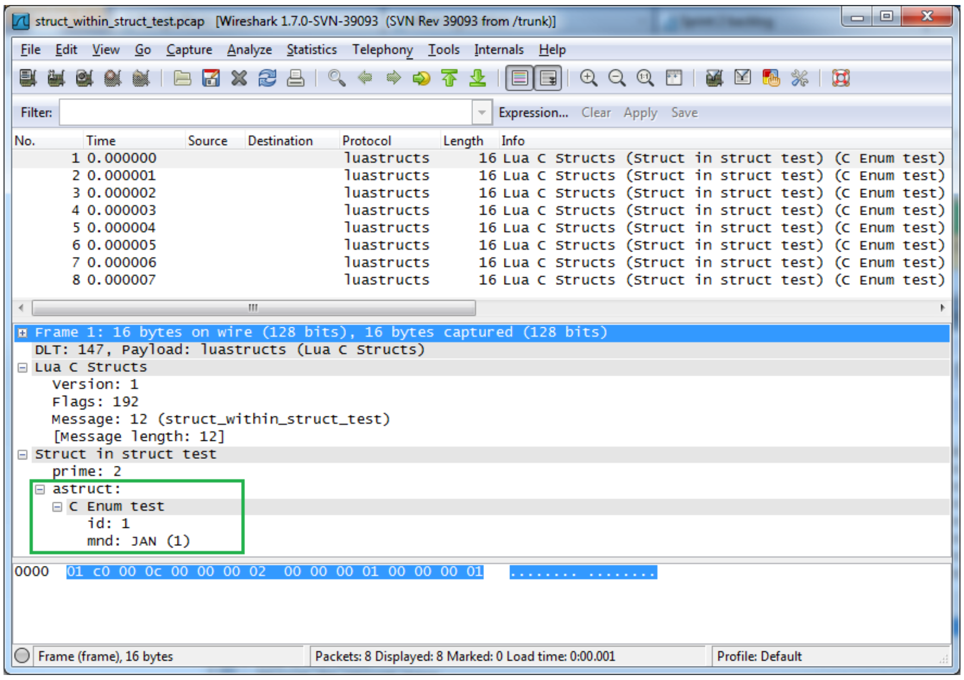
Two structs can have the same name, and therefore we needed a way of detecting it. If the parser finds two structs with the same name, an exception is raised, and the generation of the dissector is terminated.

9.4.4 Support Display of Structs within Structs

The utility is able to display structs within a struct in Wireshark, the member will be visible, and the struct will be in a subtree that can be expanded. Figure 9.4 is a screen shot of this dissector in Wireshark.

9.4.5 Support Enumerated Named Values

In C there are two ways to do enumerations, the first option was explained in subsection 9.4.1, the other way is to use #define which is shown in Listing 9.3. The advantage of using #define is that the values can be generated. Since


struct_within_struct_test.pcap [Wireshark 1.7.0-SVN-39093 (SVN Rev 39093 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.    Time      Source  Destination  Protocol    Length  Info
     1 0.000000                       luastructs      16 Lua C Structs (Struct in struct test) (C Enum test)
     2 0.000001                       luastructs      16 Lua C Structs (Struct in struct test) (C Enum test)
     3 0.000002                       luastructs      16 Lua C Structs (Struct in struct test) (C Enum test)
     4 0.000003                       luastructs      16 Lua C Structs (Struct in struct test) (C Enum test)
     5 0.000004                       luastructs      16 Lua C Structs (Struct in struct test) (C Enum test)
     6 0.000005                       luastructs      16 Lua C Structs (Struct in struct test) (C Enum test)
     7 0.000006                       luastructs      16 Lua C Structs (Struct in struct test) (C Enum test)
     8 0.000007                       luastructs      16 Lua C Structs (Struct in struct test) (C Enum test)
```

```
⊞ Frame 1: 16 bytes on wire (128 bits), 16 bytes captured (128 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊟ Lua C Structs
    Version: 1
    Flags: 192
    Message: 12 (struct_within_struct_test)
    [Message length: 12]
⊟ Struct in struct test
    prime: 2
  ⊟ astruct:
    ⊟ C Enum test
        id: 1
        mnd: JAN (1)
```

```
0000  01 c0 00 0c 00 00 00 02  00 00 00 01 00 00 00 01   ........ ........
```

```
Frame (frame), 16 bytes    Packets: 8 Displayed: 8 Marked: 0 Load time: 0:00.001    Profile: Default
```

Figure 9.4: Structs in Wireshark

this cannot be understood by the parser, it cannot be generated directly from the header file, so it have to be supported by configuration. Listing 9.4. The enum will be displayed in the same way as in subsection 9.4.1.

Listing 9.3: Enumerated named values

```
#define STRING_LEN 5

#define MONDAY 1
#define TUESDAY 2
#define WEDNESDAY 3
#define THURSDAY 4
#define FRIDAY 5
#define SATURDAY 6
#define SUNDAY 7

struct enum_test {
    int id;
    char name[STRING_LEN];
    int weekday;
};
```

Listing 9.4: Enumerated named values config

```
Structs:
  - name: enum_test
    id: 10
    enums:
      - member: weekday
        values: {1: MONDAY, 2: TUESDAY, 3: WEDNESDAY,
            4: THURSDAY, 5: FRIDAY, 6: SATURDAY, 7: SUNDAY}
```

9.4.6 Support for Bit Strings

All bits in a basic data type can represent different values. An integer is represented by four bytes(32 bits), each of these bits can for example represent 32 ”true/false’ values. Our utility support configuration of these bits. Bits can be in groups, so they can represent more than two values. Listing 9.5 shows how bit strings can be configured. Figure 9.5 shows an example of how bit strings are displayed in Wireshark. Each group of bits is masked, so it is easier to see the values. The values are also named, if they are configured.

Listing 9.5: Bitstring configuration

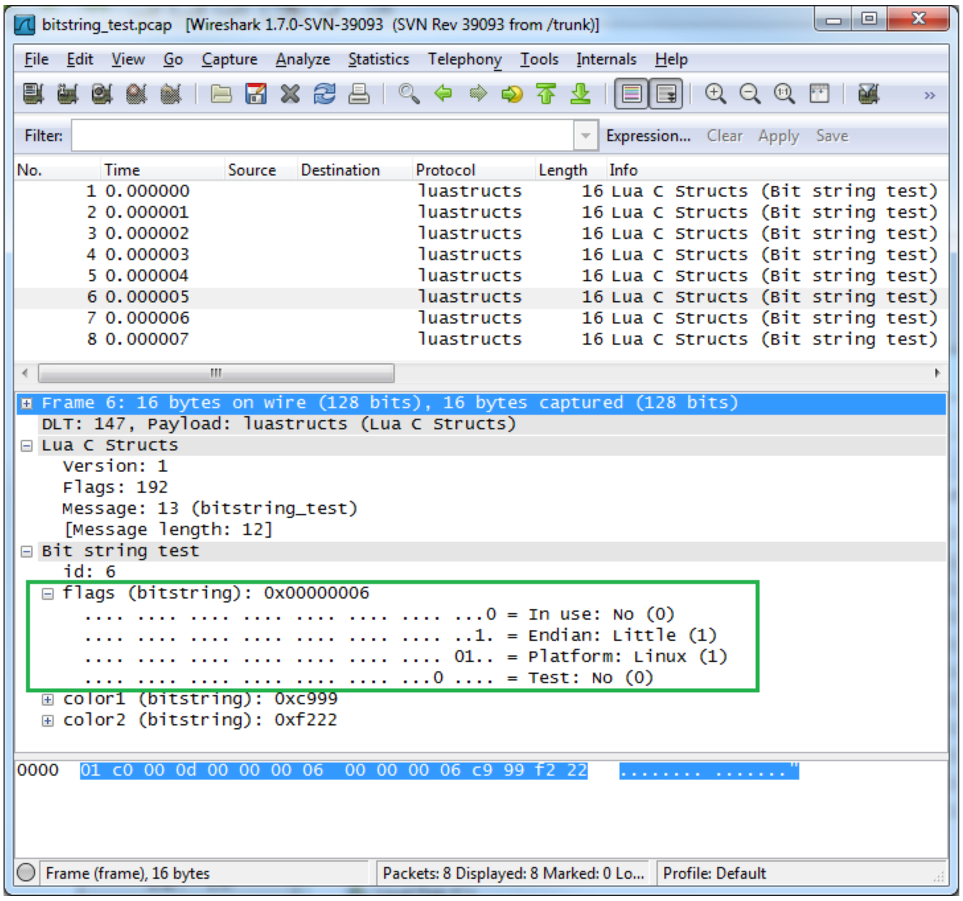
```
Structs:
  - name: bitstrings
    id: 13
    bitstrings:
      - member: flags
        32: In use
        31: [Endian, Big, Little]
        29-30: [Platform, Win, Linux, Mac, Solaris]
        28: [Test]
```

9.4.7 Support Members of Type Array

CSjark supports header-files with arrays, and is able to display them in Wireshark with the Lua-dissector. CSjark supports arrays of all data types implemented so far. The Wireshark dissector can display multidimensional arrays, and will create a new subtree for each dimension. A representation of arrays in Wireshark is displayed in Figure 9.6.

9.4.8 Struct with Various Trailers

The utility is able to support all kinds of trailers that Wireshark have built-in dissectors for. Trailers are data that follows a struct, this can be any kind of data, but only trailers that have built-in support in Wireshark can be displayed. To be able to use the Wireshark dissectors, they have to be configured. In the example below, the Wireshark dissector for ASN.1 BER


bitstring_test.pcap [Wireshark 1.7.0-SVN-39093 (SVN Rev 39093 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time      Source  Destination  Protocol    Length  Info
  1 0.000000                        luastructs      16 Lua C Structs (Bit string test)
  2 0.000001                        luastructs      16 Lua C Structs (Bit string test)
  3 0.000002                        luastructs      16 Lua C Structs (Bit string test)
  4 0.000003                        luastructs      16 Lua C Structs (Bit string test)
  5 0.000004                        luastructs      16 Lua C Structs (Bit string test)
  6 0.000005                        luastructs      16 Lua C Structs (Bit string test)
  7 0.000006                        luastructs      16 Lua C Structs (Bit string test)
  8 0.000007                        luastructs      16 Lua C Structs (Bit string test)
```

```
⊞ Frame 6: 16 bytes on wire (128 bits), 16 bytes captured (128 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊟ Lua C Structs
    Version: 1
    Flags: 192
    Message: 13 (bitstring_test)
    [Message length: 12]
⊟ Bit string test
    id: 6
  ⊟ flags (bitstring): 0x00000006
      .... .... .... .... .... .... .... ...0 = In use: No (0)
      .... .... .... .... .... .... .... ..1. = Endian: Little (1)
      .... .... .... .... .... .... .... 01.. = Platform: Linux (1)
      .... .... .... .... .... .... ...0 .... = Test: No (0)
  ⊞ color1 (bitstring): 0xc999
  ⊞ color2 (bitstring): 0xf222
```

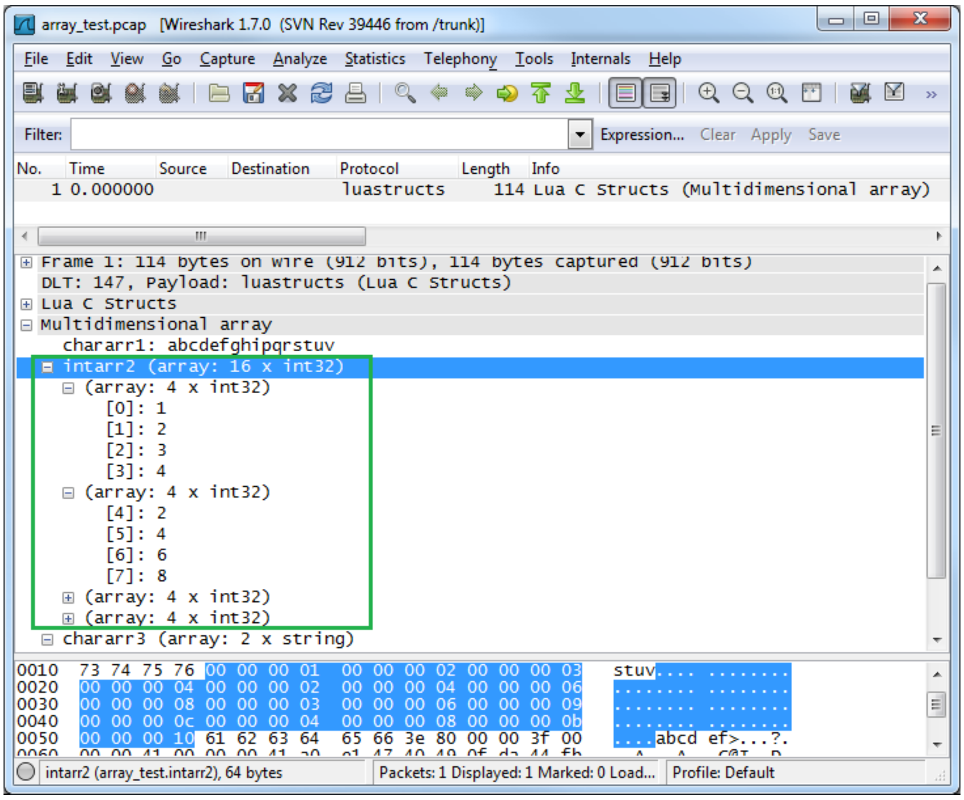
```
0000  01 c0 00 0d 00 00 00 06  00 00 00 06 c9 99 f2 22   ........ ......."
```

Figure 9.5: Bit string in Wireshark

is used. In Listing 9.6, we specify "asn1_count" as a member in the struct, this is used to tell the number of ASN.1 fields. The config in Listing 9.7 specifies fields with a size of six bytes, the number of fields are specified by the data sent with the struct. At the end there is a field with a size of five bytes. An example of ASN.1 in Wireshark, can be seen in Figure 9.7.

Listing 9.6: Enumerated named values

```c
struct trailer_test {
    float tmp[5];
    int asn1_count;
};
```


array_test.pcap [Wireshark 1.7.0 (SVN Rev 39446 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time      Source  Destination  Protocol    Length  Info
  1 0.000000                        luastructs     114 Lua C Structs (Multidimensional array)
```

```
⊞ Frame 1: 114 bytes on wire (912 bits), 114 bytes captured (912 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊞ Lua C Structs
⊟ Multidimensional array
    chararr1: abcdefghipqrstuv
  ⊟ intarr2 (array: 16 x int32)
    ⊟ (array: 4 x int32)
        [0]: 1
        [1]: 2
        [2]: 3
        [3]: 4
    ⊟ (array: 4 x int32)
        [4]: 2
        [5]: 4
        [6]: 6
        [7]: 8
    ⊞ (array: 4 x int32)
    ⊞ (array: 4 x int32)
  ⊟ chararr3 (array: 2 x string)
```

```
0010  73 74 75 76 00 00 00 01  00 00 00 02 00 00 00 03   stuv.... ........
0020  00 00 00 04 00 00 00 02  00 00 00 04 00 00 00 06   ........ ........
0030  00 00 00 08 00 00 00 03  00 00 00 06 00 00 00 09   ........ ........
0040  00 00 00 0c 00 00 00 04  00 00 00 08 00 00 00 0b   ........ ........
0050  00 00 00 10 61 62 63 64  65 66 3e 80 00 00 3f 00   ....abcd ef>...?.
```

intarr2 (array_test.intarr2), 64 bytes | Packets: 1 Displayed: 1 Marked: 0 Load... | Profile: Default

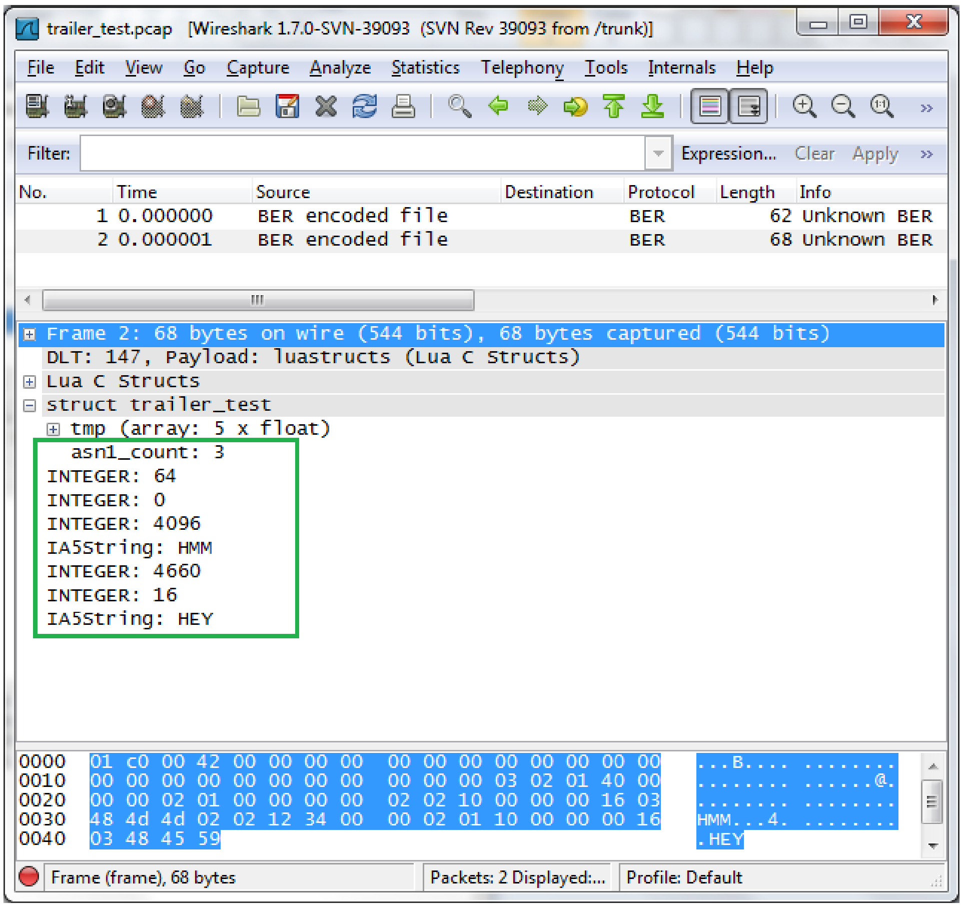
Figure 9.6: Arrays in Wireshark

Listing 9.7: Enumerated named values config

```
Structs:
  - name: trailer_test
    id: 66
    trailers:
      - name: ber
        count: asn1_count
        size: 6
      - name: ber
        count: 1
        size: 5
```

9.4.9 Custom Lua Configuration

CSjark can support custom Lua configuration, by including Lua-scripts from a file specified in the configuration file. The reason for supporting custom Lua configuration is to add support for protocols that are not built-in dissectors in Wireshark, and are not structs, or that display the members in a struct in a different way than our utility does. The only way to support this, is to write Lua code for it. To be able to insert the Lua code in correct places in


trailer_test.pcap [Wireshark 1.7.0-SVN-39093 (SVN Rev 39093 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply

```
No.   Time       Source            Destination   Protocol   Length   Info
    1 0.000000   BER encoded file                BER            62 Unknown BER
    2 0.000001   BER encoded file                BER            68 Unknown BER

Frame 2: 68 bytes on wire (544 bits), 68 bytes captured (544 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
Lua C Structs
struct trailer_test
  tmp (array: 5 x float)
    asn1_count: 3
  INTEGER: 64
  INTEGER: 0
  INTEGER: 4096
  IA5String: HMM
  INTEGER: 4660
  INTEGER: 16
  IA5String: HEY

0000  01 c0 00 42 00 00 00 00  00 00 00 00 00 00 00 00   ...B.... ........
0010  00 00 00 00 00 00 00 00  00 00 00 03 02 01 40 00   ........ ......@.
0020  00 00 02 01 00 00 00 00  02 02 10 00 00 00 16 03   ........ ........
0030  48 4d 4d 02 02 12 34 00  00 02 01 10 00 00 00 16   HMM...4. ........
0040  03 48 45 59                                        .HEY

Frame (frame), 68 bytes   Packets: 2 Displayed:...   Profile: Default
```

Figure 9.7: BER Trailer in Wireshark

the Lua files, the utility use a conformance file, which make it possible to specify where in the dissector the code should be inserted. This feature was not completely finished in this sprint, and will be finished in sprint 3.

9.4.10 Dissector ID

All luastructs-packets that Wireshark captures have a header. One of the fields in the header is the message id. This id is used to load the correct dissector when a packet is captured. Each dissector should have a unique id, to avoid possible conflicts. This functionality is implemented and the message id must be specified in the configuration file, Listing 9.8 is an example of how this is done.

Listing 9.8: Dissector ID config

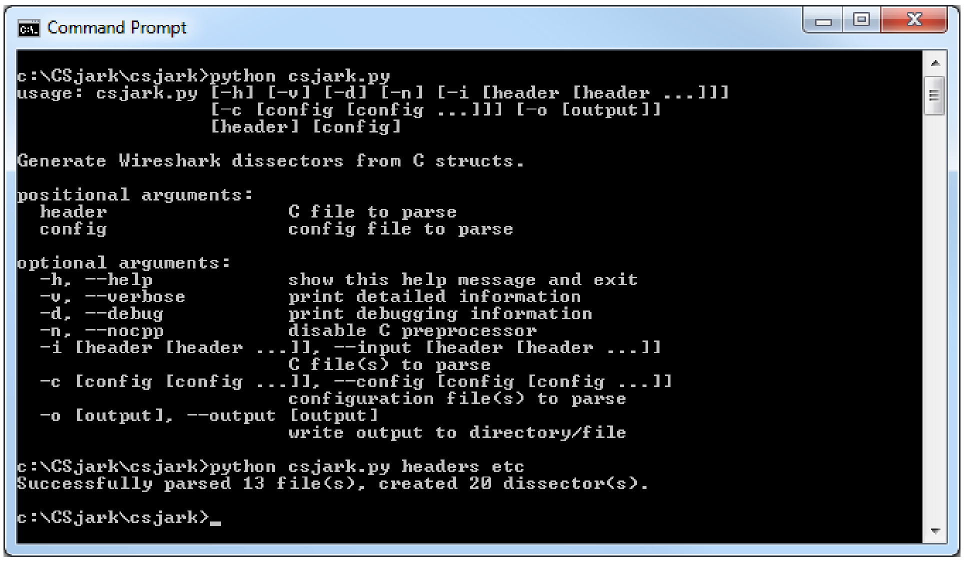
```
Structs:
  - name: structname
    id: 10
```

9.4.11 Endian Handling

Endian handling is postponed to the next sprint, because it is a platform specific problem, and should be implemented together with platform support.

9.4.12 Folder Support in the CLI

Folder support in the CLI has been implemented, so it is possible to generate Lua-scripts for all structs stored in a given folder. At the moment, all dissectors will be regenerated. Functionality to only generate modified or new header-files will be added in the next sprint. Figure 9.8 is an example usage of CSjark where the command first shows the usage of CSjark, and the second command generates dissectors from the folder "header/" and configurations from "etc/".

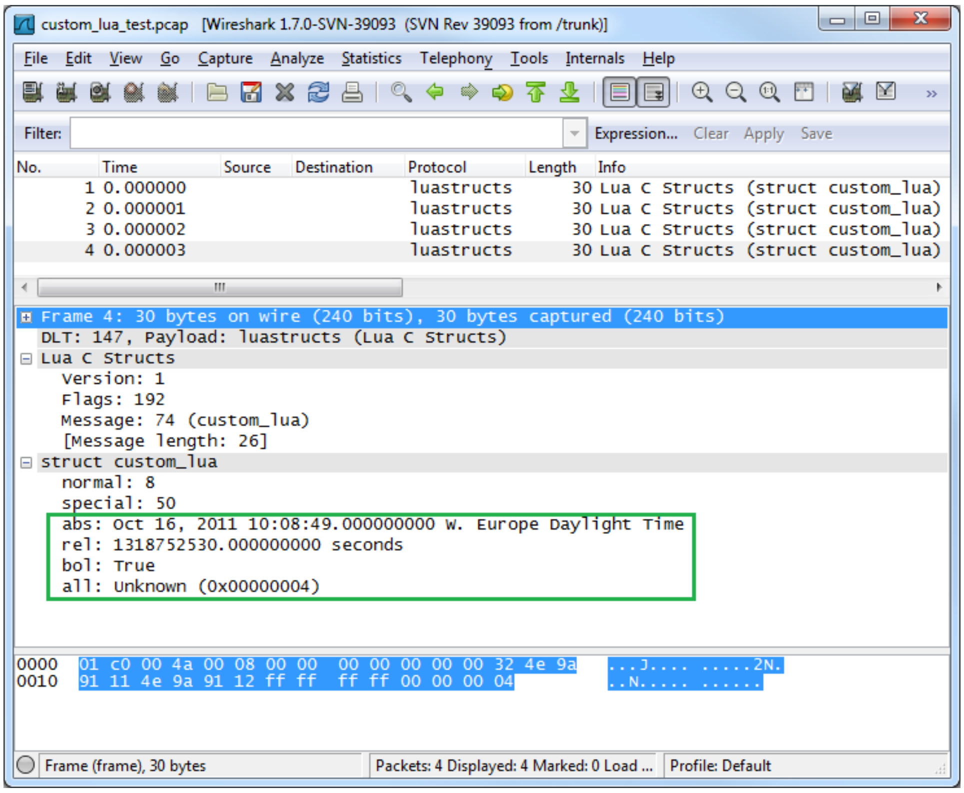


```
c:\CSjark\csjark>python csjark.py
usage: csjark.py [-h] [-v] [-d] [-n] [-i [header [header ...]]]
                 [-c [config [config ...]]] [-o [output]]
                 [header] [config]

Generate Wireshark dissectors from C structs.

positional arguments:
  header                C file to parse
  config                config file to parse

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         print detailed information
  -d, --debug           print debugging information
  -n, --nocpp           disable C preprocessor
  -i [header [header ...]], --input [header [header ...]]
                        C file(s) to parse
  -c [config [config ...]], --config [config [config ...]]
                        configuration file(s) to parse
  -o [output], --output [output]
                        write output to directory/file

c:\CSjark\csjark>python csjark.py headers etc
Successfully parsed 13 file(s), created 20 dissector(s).

c:\CSjark\csjark>_
```

Figure 9.8: Usage of Wireshark

9.4.13 Support Custom Handling of Specified Data Types

The utility supports custom handling of specific data types, this includes functionality to support time_t and nstime_t. All basic data types and struct members can be configured to be handled in a special way. Listing 9.9 shows an example of a struct with four members, two of them are time fields, and the last two is a BOOL and an integer to be handled in a custom way. This struct is configured in Listing 9.10, in the config the two time fields are configured to be respectively absolute time and relative time, and the BOOL type to have a size of four bytes. The struct member ”all’ is configured with an enumerated value, and will be visible as a hex-value. Figure 9.9 is a screen shot of the struct in Wireshark.


custom_lua_test.pcap [Wireshark 1.7.0-SVN-39093 (SVN Rev 39093 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time       Source  Destination  Protocol    Length  Info
   1 0.000000                        luastructs      30 Lua C Structs (struct custom_lua)
   2 0.000001                        luastructs      30 Lua C Structs (struct custom_lua)
   3 0.000002                        luastructs      30 Lua C Structs (struct custom_lua)
   4 0.000003                        luastructs      30 Lua C Structs (struct custom_lua)
```

```
⊞ Frame 4: 30 bytes on wire (240 bits), 30 bytes captured (240 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊟ Lua C Structs
    Version: 1
    Flags: 192
    Message: 74 (custom_lua)
    [Message length: 26]
⊟ struct custom_lua
    normal: 8
    special: 50
    abs: Oct 16, 2011 10:08:49.000000000 W. Europe Daylight Time
    rel: 1318752530.000000000 seconds
    bol: True
    all: Unknown (0x00000004)
```

```
0000  01 c0 00 4a 00 08 00 00  00 00 00 00 00 32 4e 9a   ...J.... .....2N.
0010  91 11 4e 9a 91 12 ff ff  ff ff 00 00 00 04         ..N..... ......
```

```
Frame (frame), 30 bytes   Packets: 4 Displayed: 4 Marked: 0 Load ...   Profile: Default
```

Figure 9.9: Custom Handling of Data Types

Listing 9.9: Struct for custom handling

```
#include "time.h"

typedef signed int BOOL;

struct custom_lua {
    time_t abs;
    time_t rel;
    BOOL bol;
    int all;
};
```

Listing 9.10: Config for custom handling

```
Structs:
  - name: custom_lua
    id: 74
    customs:
      - type: time_t
        field: relative_time
      - member: abs
        field: absolute_time
      - type: BOOL
        field: bool
        size: 4
      - member: all
        base: base.HEX
        values: {0: Monday, 1: Tuesday}
```

9.4.14 Typedef Support

CSjark is supporting the keyword typedef, which is a facility to create new data types names. Listing 9.11 shows examples of typedef's that CSjark supports.

Listing 9.11: Typedef example

```
typedef int BOOL;

typedef enum traffic_light_t { red, yellow, green }
    traffic_light_t;

typedef struct name_t {
     char first_name[20];
     char last_name[20];
} name_t;
```

9.5 Sprint Testing

This section introduces the tests performed during the sprint and their results. For sprint 2 it was also decided that the larger unit tests should also be documented and added to the test documents.

9.5.1 Test Results

During the sprint, the team executed a total of seven test cases. An example of a test case can be seen in Table 9.8 while the rest of the test cases can be found in Appendix C. The results are listed in Table 9.9.

Table 9.8: Test Case TID08

Header	Description
Description	Supporting members of type enum
Tester	Lars Solvoll Tønder
Prerequisites	The utility must have been installed on the system and there needs to exist a header file associated with this test
Feature	Test that the utility is able to support C-header files with enums
Execution	1. Feed the utility with the name of a C-header file which includes a struct using enums and its configuration file 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different packets and struct members that are displayed in Wireshark
Expected result	2. The program should provide the user with some text expressing the success of the Lua-file generation 5 The different packets should be displayed as having structs with enums, showing the value of the enum by its name and value

9.5.2 Test Coverage

This section introduces the amount of code covered by our unit tests and how it relates to the test coverage from the previous sprints.

Sprint 1

As the team had not started measuring code coverage in sprint 1, the testing team had to revert the project to an earlier version and retroactively check the code coverage from the first sprint. The following list contains the unit tests used in the sprint, and the coverage is shown in Table 9.10

- requirements.py
- test_cparser.py
- test_csjark.py
- test_config.py

Sprint 2

The following list contains the unit tests used in the second sprint, and the code coverage can be seen in Table 9.11. We also ran a comparison from the previous sprint and created a chart that we would use to monitor the progress of the code coverage. Tere was not a tremendous amount of progress

Table 9.9: Sprint 2 Test Results

Header	Description
Test ID	TID08
Explanation	Supporting members of type enum
Tester	Lars Solvoll Tønder
Date	15.10.2011
Result	Success
Test ID	TID09
Explanation	Supporting members of type array
Tester	Lars Solvoll Tønder
Date	15.10.2011
Result	Success
Test ID	TID10
Explanation	Supporting the display of structs within structs
Tester	Erik Bergersen
Date	17.10.2011
Result	Success
Test ID	TID11
Explanation	Supporting enumerated name values
Tester	Lars Solvoll Tønder
Date	17.10.2011
Result	Success
Test ID	TID12
Explanation	Supporting bit strings
Tester	Lars Solvoll Tønder
Date	17.10.2011
Result	Success
Test ID	TID13
Explanation	Supporting structs with various trailers
Tester	Erik Bergersen
Date	18.10.2011
Result	Success
Test ID	TID14
Explanation	Unit test covering all of the functionality implemented in sprint 2
Tester	Lars Solvoll Tønder
Date	18.10.2011
Result	Success

Table 9.10: Sprint 1 Coverage Report

Module	Statements	Missing	Excluded	Coverage
config	298	125	0	87%
cparser	215	50	0	74%
csjark	113	57	0	59%
dissector	460	79	0	38%
Total	1086	311	0	64%

from the previous sprint regarding code coverage. This is mostly due to the amount of new features that were added this sprint and the fact that using code coverage got implemented very late in the sprint, giving the developers and testers little time to use the metrics to improve the tests.

- black_box.py
- requirements.py
- test_config.py
- test_csjark.py
- test_dissector.py
- test_cparser.py

Table 9.11: Sprint 2 Coverage Report

Module	Statements	Missing	Excluded	Coverage
config	298	125	0	58%
cparser	215	50	0	77%
csjark	113	57	0	50%
dissector	460	79	0	83%
Total	1086	311	0	67%

9.5.3 Test Evaluation

For the second sprint the developers focused a lot more on testing during implementation. The team also decided that the testers should check how much of the code was covered in the unit tests. This made the team focus more on making proper unit tests that tests as much functionality as possible. This had a very positive effect on the tests that were run at the end of the sprint, where not a single test failed, except for one which ended up exposing a bug in Wireshark

9.6 Customer Feedback

In sprint 2 we were able to demonstrate many of the requirements for the customer, and as a result we got feedback on some changes and refinements. The customer also added some new features they would like to have. These changes and additions are described in subsection 5.2.2.

The customer was satisfied with the progress of the utility and was so far pleased with the functionality. They were also pleased that the team was flexible regarding feature refinement and additions.

9.7 Sprint Evaluation

This section contains the team evaluation of the second sprint.

9.7.1 Review

The first sprint evaluation resulted in some planned actions for the forthcoming sprints. Looking back at these at now, we quickly realized that we fell into the same pitfalls again: lack of documentation, bad work distribution and reverse engineering of vital parts. The documentation was improved to some degree, but we did not manage to continuously provide it for the advisors. This way we lost valuable feedback. These matters will be basis for planned actions for forthcoming sprints.

The sprint started out very good. We had a four-hour long planning meeting, and ended up with a good backlog and a common understanding of the technical aspects of the sprint. Even though the meeting was significantly better this time, we failed in doing the design early. User stories was postponed till the middle of the sprint and they were written by team members that had a limited understanding of the code. This is something we definitely have to change in the next sprints, if we want to do Scrum properly.

The rest of the sprint went as expected. Implementation went smoothly and the customer is satisfied with our progress.

In our sprint backlog, which we made at the sprint planning meeting, we listed a responsible for each task. In retrospect this did not work out very well. The problem we encountered was that the most experienced programmer was assigned almost all the programming tasks, leaving documentation tasks for the rest of the team. This could have worked out well, if the documentation and implementation were independent of each other. We know that the implementation and documentation are highly dependent, and the fact that it is not easy to write documentation for code you have not written yourself made it even worse. So all team members had to ask the implementer how he did it and how it works, which was not very efficient. This actually resulted in two bad experiences:

- Work distribution was uneven. Those with responsibilities were expected to do their task, but we did not think that a task might have been poorly estimated. Some tasks that seemed simple ended up being hard and consumed a lot of person hours.
- Not all items in the sprint backlog were completed. Because of the uneven work distribution, we realised too late that some of the backlog items would take considerably more time to complete. The person responsible for a task might have had a lot of other tasks that he needed to complete before starting on the current task, and thereby he could not get help from the other team members to complete it.

All in all we feel that we are still learning to do Scrum properly and if we take the new planned actions in true consideration, we most likely will perform considerable better at the next iteration.

The burndown chart, Figure 9.10, shows the progress during the second sprint. The estimation seem bad, but since we postponed one of the work items it does not reflect the true actual hours left. The postponed item was estimated 13 hours. Then the sprint end up being 27-13=14 actual hours left. We feel that the estimations, if we exclude the postponed work item, were good and accurate.

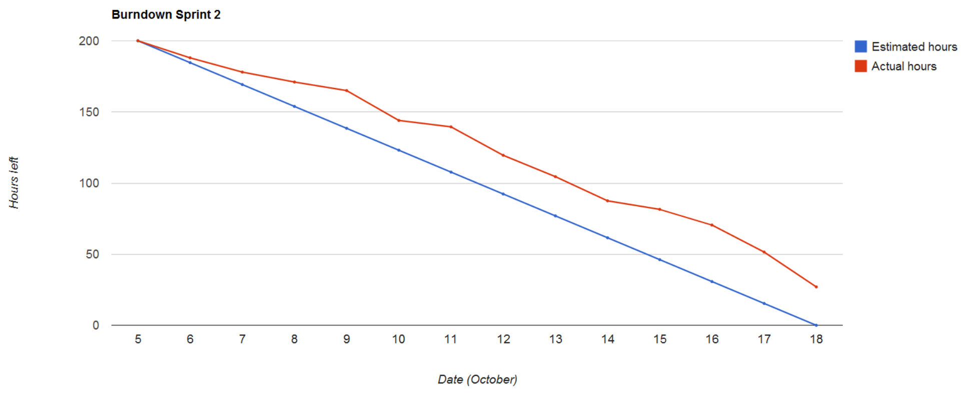


Figure 9.10: Sprint 2 Burndown Chart

9.7.2 Positive Experiences

- A significantly better sprint planning meeting
- All the team members have raised their effort, working more hours
- Accurate time estimates for most of the work items
- Successful presentation of the utility to the customer

  - Implementation of features are as intended
  - Good feedback from the customer
- Thriving team atmosphere

9.7.3 Negative Experiences

- The planning meeting was too short, which resulted in a shortcoming in the documentation.
- Documentation was postponed till the end again
- Lack of feedback to the advisor
- Poor work balance
- Could not complete all the tasks in the backlog

9.7.4 Planned Actions

We intend to complete these planned actions for the next sprint. To achieve better performance, it is crucial that the importance of these actions are not neglected.

Do the design in the planning

Last evaluation meeting we agreed to do the design early in the sprint. Since that did not work for us, we have decided to do the design in the planning meeting. Then it will not be possible to postpone it and the rest of the work will be more efficient. We will use as many hours as it takes to have a planning meeting where we end up with a good plan, proper documentation and detailed user stories.

Not assign responsibilities

We will not assign responsibilities for all the tasks at the sprint planning meeting, we rather pick one task each. This way no team members can assign themselves too many tasks, and in the end realize that they are not able to complete them. All team members will be able to see what is done and what is not, and can assign an unassigned task to themselves. We hope that this will make us capable of completing the whole sprint backlog within the given time.

List dependencies and prioritize the work items

As some tasks are dependent on others, we must plan in such a way that no team member must wait for others to do a task. It is crucial that we prioritize each task, in order to pick and complete the most important tasks first. We do not want to end up with uncompleted work items that have a high priority when the sprint is over. These changes are applied to the sprint backlog.

Provide more and better documentation to the advisors

We will strive to give our advisors more documentation to look at. We need their feedback and help in order to finish this project in a suitable manner.

9.7.5 Barriers

The work distribution As mentioned in the review, the team had an uneven work distribution. Some team members were eager to start implementing and took all the implementation tasks, leaving mostly documentation tasks for the rest of the team. The documentation tasks are closely connected to the implementation, so this resulted in problems. Documentation were postponed till the end. The planned actions section describes how we are planning to avoid that this happens again.

Wireshark We identified several bugs in Wireshark, which ended up crashing it when we tried to run our utility. Most of them were fixed by Stig Bjørlykke, as he works as a core developer for Wireshark. Unfixed bugs either will be fixed or dropped because they are not really relevant to us. Time can be lost while we wait for a patch.

Complexity consideration As always, design and estimation of future implementation is hard. At the planning meeting, we evaluated each work item and estimated the complexity of it. In hindsight, we saw that our complexity consideration for some of the work items was wrong, making it hard to complete all the items in the sprint backlog.

CHAPTER 10 SPRINT 3

This chapter explains what the team has done in the third sprint of the project. In section 10.1 the planning of this sprint is covered, the design is described in section 10.2, together with the user stories. The implemented requirements are explained in section 10.3, description of the test done can be found in section 10.4. Through the sprint the team has got feedback from the customer, this feedback can be found in section 10.5. The results from the evaluation of the sprint is explained in section 10.6.

10.1 Sprint Planning

For the third sprint we intend to implement the remaining requirements in the product backlog. We feel that the first and second sprint have resulted in a satisfying utility, but it is still missing important functionality.

After two sprint iterations, we are still trying to improve our approach to Scrum. Each sprint results in new ideas and better ways to do the process, and in this sprint we want everything to be correct and in the right order.

There will be two major changes this sprint:

- We will have a complete planning meeting. The meeting should result in a good planning document, user stories for all the requirements, complete set of work items in the sprint backlog and a early understanding of the design. This approach will be different from earlier sprints, where user stories were written in parallel with implementation. The user story should be in place before the implementation, and the implementation should be based on the user story. This will make documenting the process easier, and will in turn give the advisors more documentation of what we are doing. Then we can receive valuable feedback from them.

- In the sprint backlog we will have work items for every task that needs to be done throughout the sprint, including writing minutes, doing documentation, implementation and so on. Assignment of responsibilities for items in the backlog should not be done at the planning meeting, we rather only give responsibility for one item for each team member at a time. The rest of the items will be unassigned. At each stand-up meeting we pick a task, which we must complete before the next meeting. This will ensure efficiency and the work done by others are easier to check and revise. It will give us a better work balance, as no team member can gap over too many tasks and leave none for the others.

We think that these changes will improve our work efficiency, and make sprint 3 the best one so far.

10.1.1 Duration

The sprint started with the planning meeting the 19th of October and our work started the following day. The sprint duration is 14 days, and will end the 1st of November with a review meeting.

10.1.2 Sprint Goal

For the third sprint the team will update CSjark to version 0.3, which will extend the utility so that it contains the complete functionality requested by the customer at this phase of the project. In this sprint we will pick all of the current requirements from the product backlog, as all of the underlying functionality needed for them are already in place from the previous sprints. This means that we will also aim to create a draft of the final design of the system during the sprint.

The most important function that is going to be implemented in this sprint is being able to display packets from different originating platforms properly. This will be implemented by having every packet contain a flag specifying their originating platform, and by having our dissectors use this flag value to influence how it handles the data in the packet.

10.1.3 Back Log

The work items concerning features for this sprint are listed in Table 10.1. These are covered by user stories and are about a fourth of the work in this sprint. See the timetable for the other work items.
Timetable for this sprint: Table 10.2.

Table 10.1: Sprint 3 Requirement Work Items

		Hours	
User story	Req. and Description	Est.	Act.
Impl.		**56**	**48**
US29	FR5-A: Flags specified for each platform	5	11
US30	FR5-C: Dissector support both little and big endian	5	4
US31	FR5-D: Dissector support different sizes from flags	12	2
US32	FR3-C: Support WIN32, WIN64,SPARC etc	5	2.5
US20	FR4-B: Configuration supports custom Lua files	6	3.5
US28	FR2-C: Support Wireshark filter and search on attributes	3	1.5
US39	FR5-B: Dissectors support memory alignment	4	6.5
US26	FR1-D: Support members of type union	5	6
US27	FR2-A add: Display a wildcard type for valid C types that Wireshark has no support for	3	1.5
US34	FR4-D mod: Support specifying the ID of dissectors (name and function)	3	3
US37	FR6-D: Don't regenerate dissectors	1	1.5
US38	FR2: Handle Lua reserved definition names	2	3
Testing		**19**	**18.5**
	FR4-B: Custom Lua configuration	2	3
	FR5-C: Dissectors support both little and big endian	1	1
	FR5-D: Dissectors support different sizes from flags	2	2
	FR3-C: Support WIN32, WIN64, SPARC etc	4	2.5
	FR5-B: Dissectors support memory alignement	8	8
	FR1-D: Support members of type union	1	1
	FR2: Handle Lua reserved definition names	1	1
Doc.		**11**	**12**
	FR4-B: Custom Lua configuration	2	3.5
US33	FR4-C: Support custom handling of specific data types	2	1.5
US35	FR5: User documentation for what platform that the utility support	3	0
US36	FR5: Create developer manual from Python docstrings (autodoc plugin)	4	7
Fixes		**16**	**27**
	Total:	102	105.5

10.2 System Design

The system design defines the new modules and architecture that has to be in place to satisfy the specified requirements that we have included in the sprint 3 backlog. Most of the design are already done in earlier sprints; in this sprint we extend that and add some new elements.

Table 10.2: Sprint 3 Timetable

	Hours	
Description	Est.	Act.
Sprint planning	**30**	**47.5**
Sprint 3 requirements	**102**	**105.5**
Implementation	56	48
Testing	19	18.5
User Documentation	11	12
Fixes	16	27
Sprint review	**20**	**18**
Sprint documentation	**75**	**63**
Sprint 1 document	10	5.5
Sprint 2 document	14	13
Sprint 3 document	51	44.5
Report work	**42**	**52**
User stories to LaTeX	3	4
Architecture update	8	6.5
Glossaries and acronyms	16	20
Requirement review	15	-
Layout and correction	15	6.5
Lectures	**21**	**14**
Presentation outline	**2**	**2**
Meetings	**57**	**45**
Advisor meetings	28	21
Customer meetings	8	5
Stand-up meetings	21	19
Project management	**20**	**14**
Total:	367	361

10.2.1 System Overview

Now that the utility has both basic and advanced features, it is time to specialize and make support for environmental variables that can be found in Thales' source code. This basically include various platform specific support, endian handling and minor technicalities. The latter one is vital for the customer in order for the utility to be efficient and adequate.

The new design resulted in both major and minor changes to the class diagram, see Figure 10.1. Figure 10.2 shows the interaction between the modules, the main differences here is that platform is added, and that the config, cparser and dissector modules uses this module to generate correct dissectors for each platform.

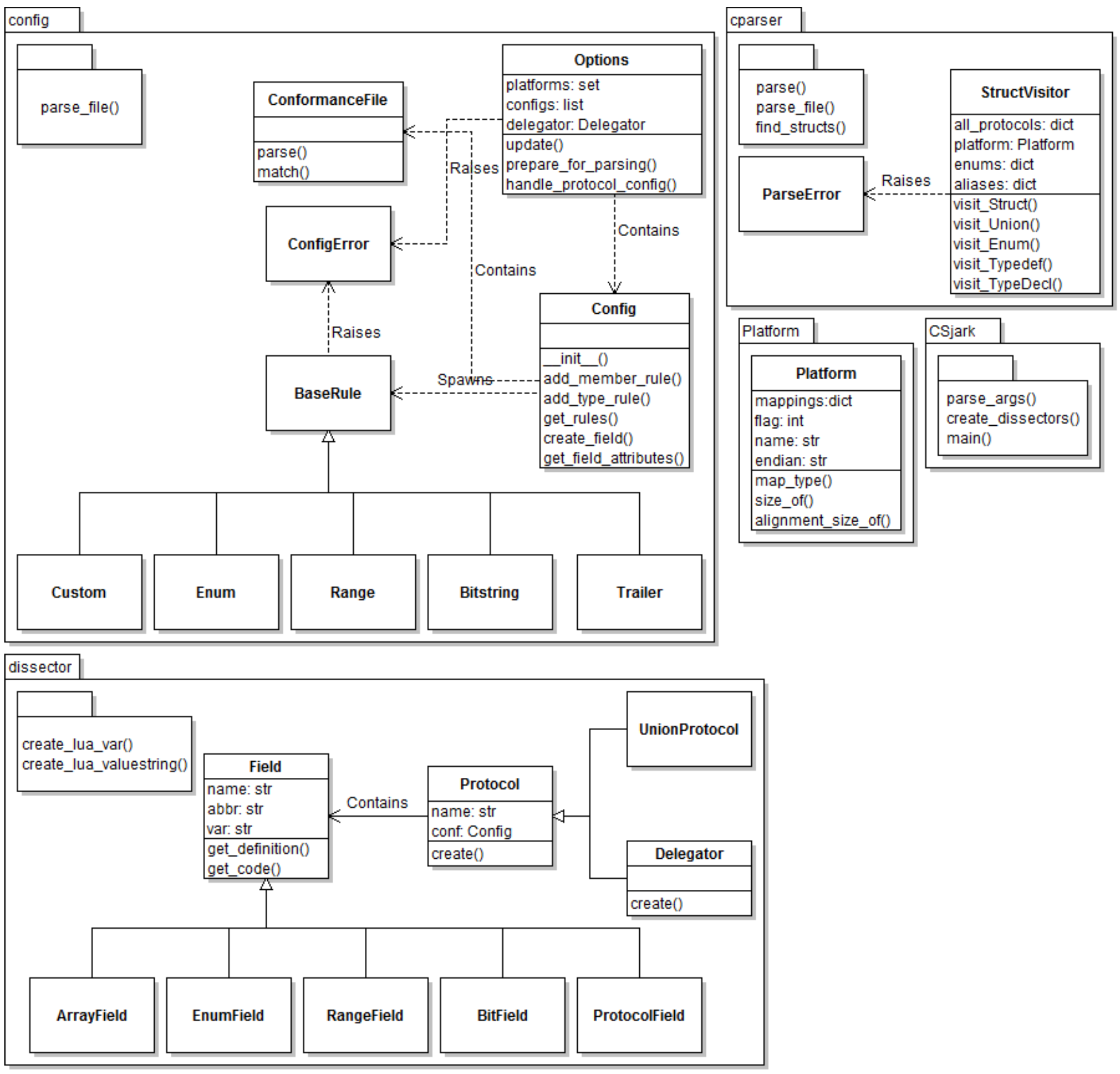


Figure 10.1: Sprint 3 Class Diagram

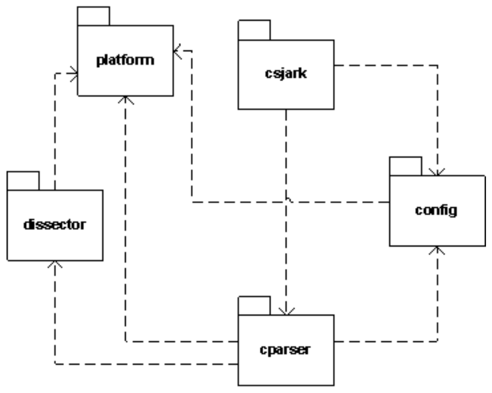


Figure 10.2: Sprint 3 Module Diagram

Major changes

- New module called Platform, to hold platform specific information. Earlier, some information about the platform was stored in the config module, but the platform specifications have become so extensive that it is best to separate it in its own module. See the section regarding platform specific support for more info.
- Moved the CLI class from csjark to config, and renamed it to Options. This class holds the attributes passed from the command line interface. This will make the code more loosely coupled, because the configuration attributes are stored in the correct module.
- A delegator class was added to dissector module. This class will be responsible for delegating dissecting to protocols. It will be a top-level dissector that delegates the task of dissecting specific messages to dissectors generated by protocol instances. It will be responsible for finding the correct dissector for a message based on the origin of it.
- UnionProtocol class was added to the dissector module. This class willl be responsible for holding the union and its members. The utility will be able to handle unions.
- In the config module the class StructConfig was removed and replaced by Options and Config. As mentioned, Options holds the attributes submitted from the command line or attributes loaded by one or more config-files. Config holds the configuration for a specific protocol.

Minor changes

- Config methods reduced to one.
- In the ConformanceFile, tokens and methods have been added.

- Trailers have a name variable.
- In the ProtocolField in the dissector module, the protocol is stored with a name instead of an id. This has been pointed out by Thales as the correct way to do it.
- Methods have been added to different classes to support the major changes. This is a change, more than an addition of new elements.

Platform Specific Support

We know that the various platforms behave differently, and this must be supported by our utility. The customer has mentioned at least three platforms that we have to support. As we emphasize extensibility and modifiability, it must also be expected that new platforms will be added or removed to the supported platforms in the future. Thereby we will try to make the set-up for support easily understandable.

We have decided that the platform definition will be a subpart of a flag-field in the structs header. To ensure modifiability we will assign a 16 bit field for this, giving the developers possibility to have 65536 unique platforms. We will make the platform flag point to the platform module, which has the following information about the actual platform:

- Endianness
- Length of fields
- Memory alignment
- Macros
- Types

All of these fields must be combined to enable the utility to generate a dissector that will dissect the struct correctly.

Dictionary Lookup

Deciding how to implement handling of different platforms ended in a design draft seen in Figure 10.3. The dissector module will make a dictionary lookup in the table of supported platforms. This table contains all platforms that the utility can make dissectors for, and their information of how the dissector shall be encoded to achieve correct handling of endianness, length of fields and memory alignment. We will start out by hard coding this into the source code, but in the end we want to have this feed to the utility through a configuration-file, which will be stored in the platform module. This is the way the customer has envisioned it.

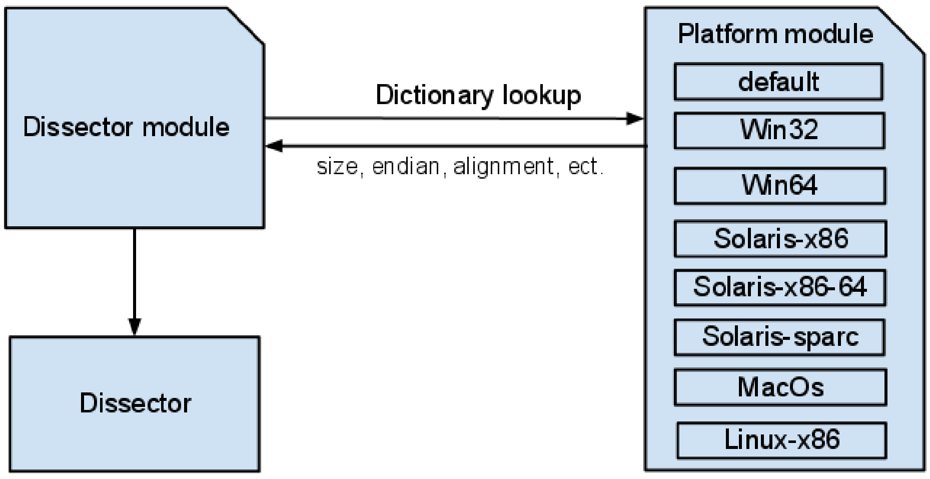


Figure 10.3: Platform Handling

Endianness

An important feature is the support for different endian handling. Endianness defines how the bytes are ordered. Big and little endian tells which bit that is the most significant, and thereby the value that the bits represent. Figure 10.4 [4] show the difference between big and little endian. Wireshark

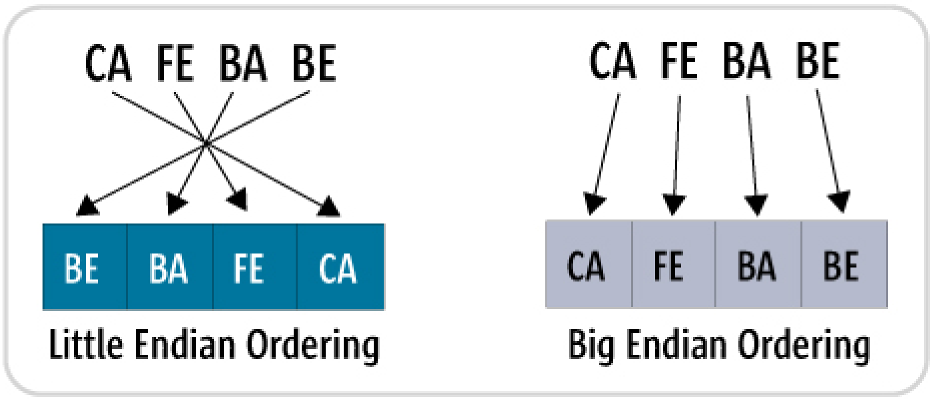


Figure 10.4: Endianness

has built-in support for reading big and little endian, and the dissector will tell which it should use. When the dissector module builds the abstract syntax tree, it must call the platform module to see what endian it shall use. In the class diagram the dissector module imports the platform module, from there it gets the information it needs regarding the endianness.

Batch Processing

The batch processing was implemented in the second sprint, but we have to modified it to ensure efficiency and reliability. When the utility encounter a header that it can not parse, it should skip that and move on to the next header in the queue. This will make the utility able to process almost all the headers in one batch, and ensure reliability. This is also according to the customer's specification.

The redesign of the config module will ensure better efficiency when the batch process is executed. It will also be easier to trace the connection between configuration and options.

10.2.2 User Stories

This section lists the user stories for the third sprint. They are displayed in Table 10.3, Table 10.4 and Table 10.5. These user stories represent how we intend to add the functionality of each requirement of the third sprint. How the different modules in CSjark should interact with each other is also detailed. Some of the user stories explains how the user documentation should be written, while one of them explains the modification of FR4-D, a requirement we implemented in sprint 2.

10.3 Implementation

The main focus in this sprint was to add support for different platforms. Several things are dependent on platform; endianness, the memory alignment and sizes of data types. It is also possible that structs can be defined differently for each platform. The utility will generate different dissectors for each platform. A dissector will detect the platform and use the correct platform dissector.

Support for the union data type, finishing implementation of custom Lua files and modification of functionality implemented in the previous sprint was also done in this sprint.

10.3.1 Specify Flags for Each Platform

It is necessary to specify flags for each platform to make it possible to correctly detect and display packages that Wireshark captures. In Wireshark the flags are used to tell which platform the packet is sent from, so that the right dissector is used to display the packets in Wireshark. In the utility the flag points to what kind of endianness is used, how memory is aligned and the different sizes that is used for data types on the platform. These data are used to generate a dissector for the specific platform. The luastructs protocol will read the platform flag, an find the dissector for the platform.

Table 10.3: User Stories - Sprint 3 Part 1

Header	Value
ID	US26
Requirement	FR1-D: The utility must support members of type union
What	The administrator wants to generate dissectors that contain structs with unions as members.
How	When the administrator feeds the utility with a header containing a union, the cparser module should parse the union and its members to find the total size of the union, which equals the size of the largest member, and then create an instance of UnionField from the dissector module representing the union and its member
Result	The utility support union members in structs.
ID	US27
Requirement	FR2-A*Addition:* Display a wildcard type for valid C types that Wireshark has no support for.
What	The administrator should be able to give the tool a struct with a valid C-type even if Wireshark does not have a way to display that type in a natural way. The dissector should the just display the name of the type, the name of the member and the hex value from the packet.
How	The parser module will recognise if a type it encounters are supported in Wireshark or not. If it is not supported, it will add a wildcard field to the prototype object representing the enclosing struct.
Result	The administrator will be able to both run the utility and get some information from the dissector even if the type used is not supported by Wireshark.
ID	US28
Requirement	FR2-C: Filter and search on attributes (important to have descriptive abbreviations)
What	The developer wants to find specific attributes in Wireshark. The amount of data captured can be big. After the packets have been dissected, they are presented in Wireshark. The developer will have a hard time finding attributes by manual seeking. The built-in search and filter functionality needs to be supported to accommodate the developer.
How	The functionality is already in Wireshark. To utilize it, an abbreviation field must be provided to Wireshark. This abbreviation field will be in the dissector module and is included in the dissector generated by the utility. When Wireshark runs the dissector, all abbreviations are collected, which will make it possible to filter and search on attributes.
Result	The developer will be able to search and filter by attributes.
ID	US29
Requirement	FR5-A: Flags must be specified in configuration for each platform
What	The administrator wants to generate dissectors that are different depending on platform.
How	To be able to create dissectors that are different depending on the originating platform, the administrator needs to specify in the configuration which platforms he wants to support. The config module should accept such configuration and store it so other modules can use it.
Result	The administrator can specify what platform he is using by setting a flag in the configuration.

Table 10.4: User Stories - Sprint 3 Part 2

Header	Value
ID	US30
Requirement	FR5-C: Generate dissectors that support both little and big endian platforms
What	The administrator wants dissectors that handle both big and little endian encoding.
How	The dissector module will need to create different Lua code for big and little endian, when adding nodes to the Wireshark tree and when calling other dissectors. The dissector module shall have functionality that generates Lua code depending on endianness, and the different Field classes must use this function when adding nodes to the Wireshark tree.
Result	Dissectors can be created with platform specific endian.
ID	US31
Requirement	FR5-D: Generate dissectors that support different sizes depending on platforms
What	The administrator wants to generate dissectors for struct where members size depend on the originating platform.
How	When the administrator feeds the utility a header file and a config file with a set of platform he wants dissectors for, the config module will create new header files with C preprocessor directives for each platform. These files should define platform-specific macros that emulates parsing on the specific platform. The dissector module then create different dissectors for each message on each platform, and a mapping is added inside the master Lua file that maps message id and platform to the correct dissector.
Result	Dissectors can be created with platform specific sizes of members.
ID	US32
Requirement	FR3-C: Support for configuring a platform with a platform specific macro like WIN32, _WIN64, __sparc to be able to support different struct definitions for different platforms.
What	The administrator wants to be able to configure the utility to make dissectors that support structs that is defined differently on different platforms via platform specific macros like WIN32, _WIN64, __sparc.
How	The administrator specify the macro associated with a platform together with the platform definition configuration. The utility then make an auxiliary header file for each platform configuration with the specified macro definition. These headers are forwarded to the parser module, which uses them to generate one dissector for each platform for each struct. All dissectors dissecting the same structs are stored in the same file, but are added to a platform specific dissector table.
Result	The generated Lua files corresponding to structs includes one dissector for each platform defined in the configuration file.
ID	US33
User doc	FR4-C: User documentation for configuration custom handling of specific types.
What	The administrator should be able to find out how to specify custom handling for a specific type by reading the user documentation.
How	The administrator opens the user documentation and finds the section about configuration. From here he locates the sub section about custom type handling. This section gives a good description of how to write such configuration and what kind of configuration that could be done. There should also be some examples to clarify the description. After reading the section, the user has a good idea of how to do the desired custom handling.
Result	The administrator is able to use custom handling of specific types.

Table 10.5: User Stories - Sprint 3 Part 3

Header	Value
ID	US34
Requirement	FR4-D modified: Configuration must support specifying the ID of dissectors
What	The administrator wants to specify the ID of a dissector in a configuration file. The dissectors should not be given any ID if it has not been specifically configured.
How	When the cparser finds a struct in the abstract syntax tree it looks for a configuration file for the struct. If a config-file is found, the ID of the prototype field representing the dissector will be mapped to the ID given in the config-file. If it is not found, the ID will be sat to NONE.
Result	The administrator can specify the ID of dissectors in the configuration.
ID	US35
User doc	FR5 User documentation for how to add or remove support for a platform in the dissectors generated from the utilities.
What	The administrator should be able to find out how to add or remove support for a platform in the dissectors by reading the user documentation.
How	The administrator opens the user documentation and finds the section about configuration. From here he locates the sub section about platform support. This section gives a good description of how to add or remove support for a platform in the configuration, in order for the administrator to understand how to do this after reading the section.
Result	The administrator is able to configure the utility to add or remove platform support from the generated dissectors.
ID	US36
User doc	FR5 User documentation for what platform that the utility support.
What	The administrator should be able to find out what platforms he can add support for in the custom dissector files.
How	The administrator opens the user documentation and finds the section about configuration. From here he locates the sub section about platform support. This section gives a list of currently supported platforms by the utility. It should also have some information of where to find the documentation that describes how to add support for more platforms.
Result	The administrator is able to look up what platforms he can get the dissectors to support.
ID	US37
Requirement	FR6-D The utility should not regenerate dissectors within a single run.
What	The administrator should be able to specify folder that includes both a standalone header file with a struct definitions and another header file that includes the first header. The utility will only generate the dissector once for the struct inside the first header.
How	For each struct encountered, the utility will check the table of generated dissectors to see if there is an existing dissector generated for the struct name. It will only generate a new dissector if the table of dissectors is empty for that name.
Result	The utility will run faster as a result of not needing to regenerate dissectors.
ID	US38
Requirement	Handle Lua reserved definition names
What	The C structs could contain members with names that are reserved by Lua. The dissector module needs to avoid creating Lua variables with such names.
How	The dissector module has a method called create_lua_var. This method will ensure that variable names are valid, by comparing the variable names to a list of Lua reserved keywords, and if there is a match we need to add an underscore in front of the variable name.
Result	The utility can handle C header files that contain Lua reserved definition names.

Table 10.6: User Stories - Sprint 3 Part 4

Header	Value
ID	US39
Requirement	FR5-B: Generate dissectors with correct alignment depending on platform
What	The administrator wants to generate dissectors that are different depending on platform.
How	The dissector module will use the configured flags from the config module to modify the generated Lua dissectors, to have the right memory structure alignment and endianness. The dissector module shall have a function that calculates the offset for each field to align it, and functions that create fields for specific endianness.
Result	Dissectors can be created with platform specific memory structure alignment

10.3.2 Support Little and Big Endian

Different platforms can order bytes in either little(left-to-right) or big(right-to-left) endian. The Windows platform uses little endian, and SPARC uses big endian. Since the utility has to support both platforms, it was necessary to support handling of endianness. The Lua API in Wireshark has functionality to display data in both little and big endian. Therefore the utility has to read the specified flag for the platform, and generate a Lua dissector that displays the data correctly for the given platform.

10.3.3 Support Different Sizes from Flags

On different platforms, there can be different sizes on the data types. For example on Windows a *long double* is 8 bytes, and on SPARC it is 16 bytes. It is possible to specify sizes for data types for each platform. All these specifications is written in Python code, and are easy to modify for the user of the system.

10.3.4 Support Platform Specific Macros

All C compilers have predefined macros for different operating systems and processors. These macros are much used in code that need to be portabel for different platforms. When using these macros it is possible to create different struct members for each operating system, as shown in Listing 10.1, which will use different data types on each operating system. All platform specific macros are specified for each platform, so the dissector is generated correctly for each of the platforms. Currently all the platform specifications is done in Python code.

Listing 10.1: Macros in C

```
#if _WIN32
    float num;
#elif __sparc
    long double num;
#else
    double num;
```

10.3.5 Support Custom Lua Files

Implementation of this feature started in sprint 2, and support for using conformance file to add Lua code to correct places in the Lua dissector was finished in this sprint. With the conformance file it is possible to add code before and after an member in both the definition and function part of the dissector. It also possible to replace the code for a member in both of these sections. In the example below there is written custom Lua for a struct(temperature) with on integer member (Celsius). In the conformance file in Listing 10.2 there added three lines of comments as a custom Lua code, which are going to be added to the Lua dissector. In Listing 10.3 it is possible to see the custom Lua code that was added to the Lua dissector. The first comment is added after the member definition. The two other comments are added above and below the member in the function part of the Lua dissector.

Listing 10.2: Custom Lua conformance file

```
#.DEF_FOOTER celsius
-- This is below 'celsius'
#.END

#.FUNC_HEADER celsius
    -- This is above 'celsius' inside the dissector function.
#.END

#.FUNC_FOOTER celsius
    -- This is below 'celsius' inside dissector function
#.END
```

Listing 10.3: Custom Lua dissector code

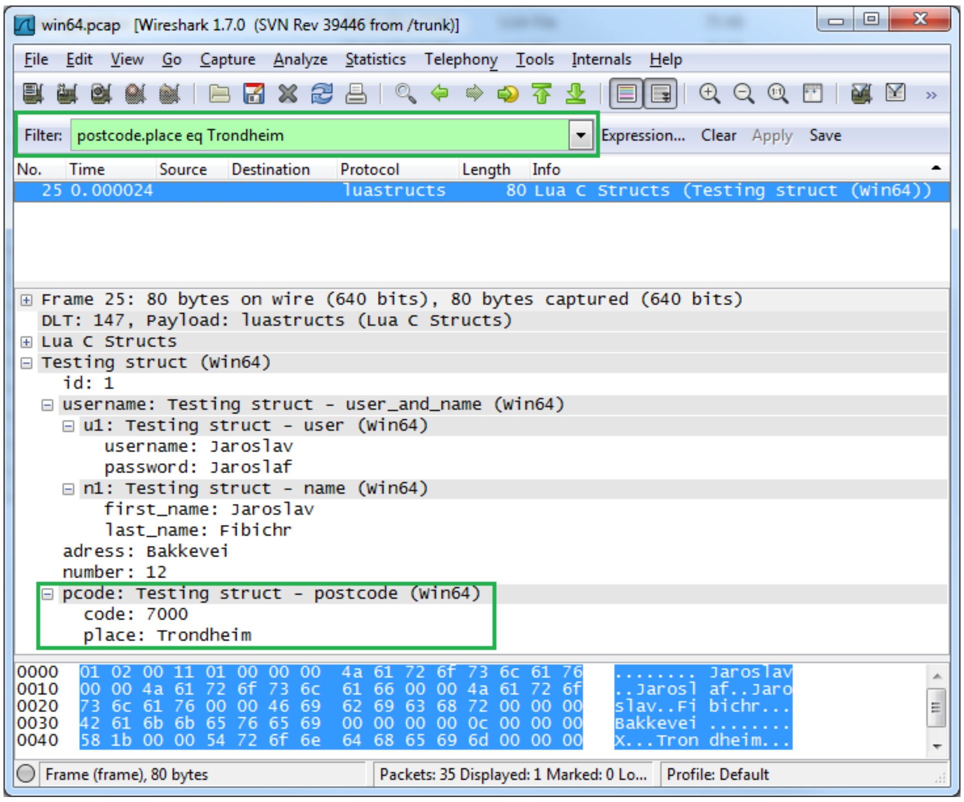
```
-- Dissector for win32.temperature: temperature (Win32)
local proto_temperature = Proto("win32.temperature", "↩
    temperature (Win32)")

-- ProtoField defintions for: temperature
local f = proto_temperature.fields
f.celsius = ProtoField.int32("temperature.celsius", "celsius")
-- This is below 'celsius'

-- Dissector function for: temperature
function proto_temperature.dissector(buffer, pinfo, tree)
    local subtree = tree:add_le(proto_temperature, buffer())
    if pinfo.private.caller_def_name then
        subtree:set_text(pinfo.private.caller_def_name .. ": " ↩
            .. proto_temperature.description)
        pinfo.private.caller_def_name = nil
    else
        pinfo.cols.info:append(" (" .. proto_temperature.↩
            description .. ")")
    end

    -- This is above 'celsius' inside the dissector function.
    subtree:add_le(f.celsius, buffer(0, 4))
    -- This is below 'celsius' inside dissector function
end

delegator_register_proto(proto_temperature, "Win32", "↩
    temperature", 7004)
```

10.3.6 Support Wireshark Filter and Search

Wireshark has a built-in display filter, where it is possible to use packet filtering. Each field in our generated dissectors has a abbreviation name that is connected to a struct. For each member of a struct, it is possible to filter on a value. An example is shown in Figure 10.5, this shows a filtering for packets where Trondheim is equal to the member *place* in the struct *postcode*.

10.3.7 Support Different Memory Alignment

Since the dissectors generated by the utility are going to be used for inter-process communication, it is important to handle memory structure alignment, because the packet that Wireshark capture is only a copy of the memory. Memory alignment is how the data are stored in the memory. Each member of a structure has a alignment. With the correct handling of memory alignment it is possible to display the structs correctly in Wireshark.


win64.pcap [Wireshark 1.7.0 (SVN Rev 39446 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: postcode.place eq Trondheim
Expression... Clear Apply Save

```
No.  Time      Source  Destination  Protocol    Length  Info
 25 0.000024                        luastructs      80 Lua C Structs (Testing struct (Win64))
```

```
⊞ Frame 25: 80 bytes on wire (640 bits), 80 bytes captured (640 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊞ Lua C Structs
⊟ Testing struct (Win64)
    id: 1
  ⊟ username: Testing struct - user_and_name (Win64)
    ⊟ u1: Testing struct - user (Win64)
        username: Jaroslav
        password: Jaroslaf
    ⊟ n1: Testing struct - name (Win64)
        first_name: Jaroslav
        last_name: Fibichr
    adress: Bakkevei
    number: 12
  ⊟ pcode: Testing struct - postcode (Win64)
      code: 7000
      place: Trondheim
```

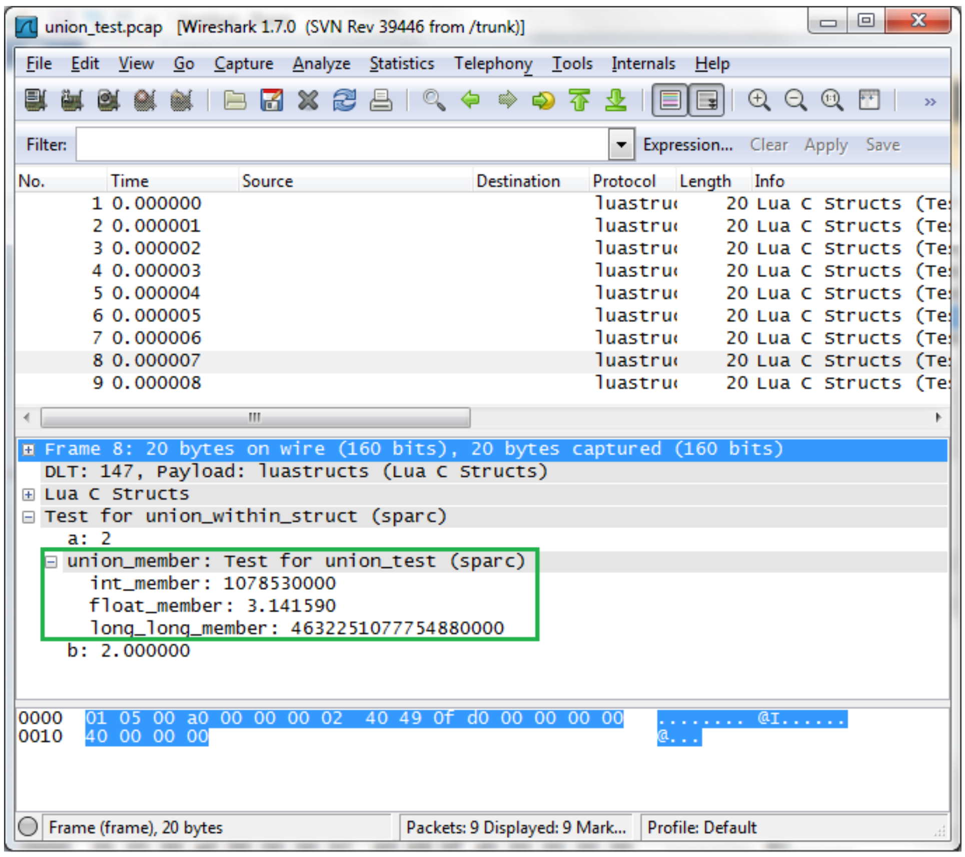
```
0000  01 02 00 11 01 00 00 00  4a 61 72 6f 73 6c 61 76   ........ Jaroslav
0010  00 00 4a 61 72 6f 73 6c  61 66 00 00 4a 61 72 6f   ..Jarosl af..Jaro
0020  73 6c 61 76 00 00 46 69  62 69 63 68 72 00 00 00   slav..Fi bichr...
0030  42 61 6b 6b 65 76 65 69  00 00 00 00 0c 00 00 00   Bakkevei ........
0040  58 1b 00 00 54 72 6f 6e  64 68 65 69 6d 00 00 00   X...Tron dheim...
```

Frame (frame), 80 bytes | Packets: 35 Displayed: 1 Marked: 0 Lo... | Profile: Default

Figure 10.5: Filter and Search in Wireshark

10.3.8 Support Union Type

The union type is a member that can contain different data types with different sizes. The union will allocate memory for the largest type defined in the union. Listing 10.4 shows an example of a header-file with a union type. The compiler is responsible for keeping track of size and alignment requirements [11, p.147] . Since it is not possible to find out which data type that is used in Wireshark, the utility has to generate a dissector that displays the values for each data type. Figure 10.6 displays the dissector generated from the struct in Listing 10.4. This shows the union with three members, all of them are listed with their values. In this case the float value is the correct one.


union_test.pcap [Wireshark 1.7.0 (SVN Rev 39446 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.     Time        Source                 Destination  Protocol  Length  Info
      1 0.000000                                        luastru       20 Lua C Structs (Te
      2 0.000001                                        luastru       20 Lua C Structs (Te
      3 0.000002                                        luastru       20 Lua C Structs (Te
      4 0.000003                                        luastru       20 Lua C Structs (Te
      5 0.000004                                        luastru       20 Lua C Structs (Te
      6 0.000005                                        luastru       20 Lua C Structs (Te
      7 0.000006                                        luastru       20 Lua C Structs (Te
      8 0.000007                                        luastru       20 Lua C Structs (Te
      9 0.000008                                        luastru       20 Lua C Structs (Te
```

```
⊞ Frame 8: 20 bytes on wire (160 bits), 20 bytes captured (160 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊞ Lua C Structs
⊟ Test for union_within_struct (sparc)
    a: 2
  ⊟ union_member: Test for union_test (sparc)
      int_member: 1078530000
      float_member: 3.141590
      long_long_member: 4632251077754880000
    b: 2.000000
```

```
0000  01 05 00 a0 00 00 00 02  40 49 0f d0 00 00 00 00   ........ @I......
0010  40 00 00 00                                        @...
```

Figure 10.6: Union Type Support

Listing 10.4: Union type

```
union union_test {
    int int_member;
    float float_member;
    unsigned long long long_long_member;
};

struct union_within_struct {
    int a;
    union union_test union_member;
    float b;
};
```

10.3.9 Display Types Wireshark Do Not Support

The utility is able to generate dissectors for C data types that Wireshark does not support. When such a data type occur, Wireshark will only display the packet data. An example of a data type that Wireshark does not support is *long double* on the SPARC platform, which is a 128-bit foat. For this Wireshark will display the 16 bytes in hexadecimal.

10.3.10 Support Specifying ID of Dissectors

Specifying dissector ID has been modified from sprint 2. The dissector will still use the ID given in the configuration for a struct. If the struct is not given an ID in the configuration, then the ID for the struct will be set to *NONE*. This is done to ensure that structs have an unique ID for their dissector. Struct-in-struct members do not need an ID since they are called from the struct by the dissector name.

10.3.11 Do Not Regenerate Dissectors

To increase the performance on generation of dissectors, dissectors that already are generated in batch mode, will only be generated once in a batch mode execution. This feature will be improved in the next sprint, so it only generates dissectors in batch mode that are modified or new since the last batch run.

10.3.12 Handle Lua Reserved Keywords

Lua has a list of reserved keywords, and some of these keywords are allowed in the C language. The utility is able to support this under generation of Lua code, when an identifier is a lua keyword, an underscore(_) is added, so the identifier starts with "_".

10.3.13 Support for Complex Arrays

Support for typedef was implemented in the previous sprint. In this sprint it has been improved to also support type definitions of complex arrays. Support for type definition of array is implemented, an example of such type definition in C is shown in Listing 10.5. Also support for arrays of enums, arrays, structs and pointers has been added.

Listing 10.5: Typedef of arrays

```c
typedef unsigned char BENQ;
typedef BENQ BENQTWO[2];
typedef BENQTWO BENQFOUR[2];

typedef int* INTPOINTER[3];
```

10.4 Sprint Testing

During sprint 3, the team executed a total of 11 test cases, but no additional testing features were added during the sprint.

10.4.1 Test Results

The test cases executed in this sprint can be found in Appendix C. An example of such a test case can be seen in Table 10.7. The results from the tests are listed in Table 10.8 and Table 10.9.

Table 10.7: Test Case TID15

Header	Description
Description	Support batch mode of C header and configuration files
Tester	Lars Solvoll Tønder
Prerequisites	The utility has have been installed on the system, there also needs to exist a header and configuration file for this test
Feature	Test that the utility is able to generate dissectors for all header-files in a folder, with configuration
Execution	1. Feed the utility the name of the two folders with header-files and configuration-files. 2. Read output from the utility
Expected result	2. The utility should provide the user with the amount of header files processed and the number of dissectors created. It should also provide the user with error messages for the header and configuration files it was unable to run

Table 10.8: Sprint 3 Test Results Part 1

Header	Description
Test ID	TID15
Description	Supporting batch mode of C header and configuration files
Tester	Lars Solvoll Tønder
Date	28.10.2011
Result	Success
Test ID	TID16
Description	Supporting custom Lua configuration
Tester	Lars Solvoll Tønder
Date	30.10.2011
Result	success
Test ID	TID17
Description	Supporting unions
Tester	Lars Solvoll Tønder
Date	30.10.2011
Result	success
Test ID	TID18
Description	Supporting filter and search in Wireshark
Tester	Lars Solvoll Tønder
Date	30.10.2011
Result	success
Test ID	TID19
Description	Supporting WIN32, _WIN64, _SPARC
Tester	Lars Solvoll tønder
Date	30.10.2011
Result	Success
Test ID	TID20
Description	Supporting the use of flags specifying platforms to display member values correctly
Tester	Lars Solvoll Tønder
Date	30.10.2011
Result	Failure. Most values were displayed correctly, but there were cases where the members and their values were different. Most notably in packet 2 and 3 in the win64 and win32 pcap-files
Test ID	TID21
Description	Supporting platforms with different endian
Tester	Erik Bergersen
Date	31.10.2011
Result	Success
Test ID	TID22
Description	Supporting alignments
Tester	Lars Solvoll Tønder
Date	30.10.2011
Result	Failure. All of the platforms that were used for testing were the same

Table 10.9: Sprint 3 Test Results Part 2

Header	Description
Test ID	TID23
Description	Handling Lua keywords
Tester	Lars Solvoll Tønder
Date	30.10.2011
Result	Success
Test ID	TID24
Description	Unit test encompassing all of the functionality implemented thus far in the utility
Tester	Lars Solvoll Tønder
Date	01.11.2011
Result	Success
Test ID	TID25
Description	Unit test covering all of the functionality imposed by the customer
Tester	Lars Solvoll Tønder
Date	01.11.2011
Result	Failure. Failed because one or more of the following tests failed: a.type should be equal to int32, b.type should be equal to string, c.type should be equal to float

Test Coverage

As can be seen in both Table 10.10, the team finally managed to hit the goal of having a code coverage of at least 80% in this sprint. This was achieved mostly by improving the black_box.py, which is a test of all the major parts of the utility. The following list shows the unit tests ran in sprint 3:

- black_box.py
- requirements.py
- test_config.py
- test_cparser.py
- test_csjark.py
- test_dissector.py

10.4.2 Test Evaluation

In this sprint we saw a massive improvement in code coverage, due to finally being able to use the coverage tool to monitor our unit tests. Many bugs were also discovered during this sprint, and while most of them were fixed, some of the bug fixing had to be pushed to next sprint due to time constraints. Some

Table 10.10: Sprint 3 Coverage Report

Module	Statements	Missing	Excluded	Coverage
config	309	25	0	98%
cparser	230	32	0	86%
csjark	113	51	0	55%
dissector	497	72	0	86%
Total	1149	180	0	81.25%

test cases also failed due to poor communication between the ones responsible for creating the test cases and the ones responsible for generating the test data. It was therefore decided that we should not split up these two types of tasks, but rather have the same person design a test case and create the required data for it as well.

10.5 Customer Feedback

This section covers the feedback we have got from the customer during the sprint. New requirements and descriptions of existing requirements are listed in subsection 5.2.3.

10.5.1 Pre-sprint

The customer was satisfied with the functionality we implemented in sprint 2. They feel that we are flexible and adapt well to their needs. Some of the features from sprint 2 were postponed to sprint 3 because we needed additional feedback from the customer. These include endianness and completing the handling of custom Lua files.

The customer wanted our utility to be as automatic as possible, and with as little configuration as possible. Additional requirements for sprint 4 should address optimization and testing of the utility.

10.5.2 Post-sprint

The customer stressed the importance of being able to run our utility on the header files they have provided. Once our utility is able to successfully parse all their files, they can begin testing the utility on their own data. We should expect more feedback on necessary fixes and improvements after the customer has tested the utility on their own source code tree.

In general the customer was satisfied with the progress we have made. Most of the requirements were implemented or close to complete.

10.6 Sprint Evaluation

After the sprint was finished, the team had an evaluation of the sprint. The results of this meeting is covered below.

10.6.1 Review

After some trying, failing and learning we feel that we understand the Scrum process and were able to do it more correctly. We did the planned actions from the prior sprints and achieved a considerable increase in work efficiency, work completion and had a well distributed workload.

After not assigning responsibilities at the planning meeting, the team members felt that it was easier to find a task that suited them, which increased the efficiency of task completion. We also listed all the work that needed to be done during the sprint, including meeting, minutes, sprint documentation and so on. This gave us a much better overview of the work.

Looking at the planned actions, we feel that we have managed to do all of them and we are very satisfied with that. The bad experiences mentioned in this sprint are only nitpicking compared to the ones in earlier sprints. We see this as a natural evolution, as we learn to do the Scrum steps correctly and efficiently, the team as a whole works better. But the raised effort from each team member is also indispensable and the improvement we have gained is a result of dedication towards the project.

The burndown chart for the sprint can be seen in Figure 10.7. The estimated and actual hours fit almost flawlessly. This is something we have been good at in all the sprints, but in this sprint it was perfected. Even though we estimated too many hours of work for this sprint, all the team members gave their best effort and we almost completed the whole backlog.

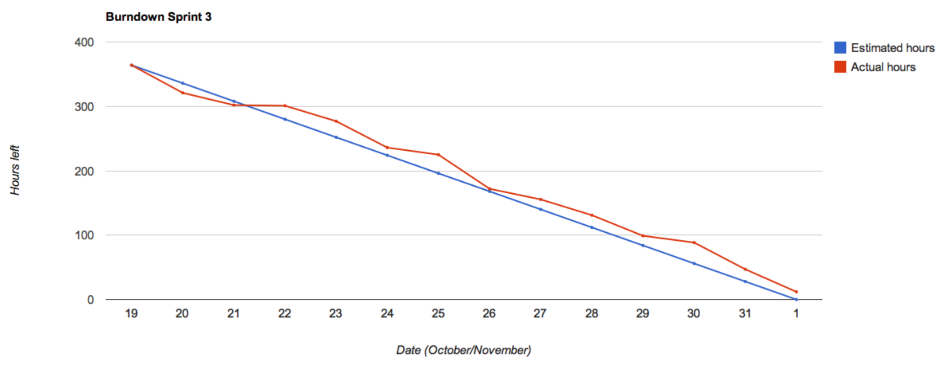


Figure 10.7: Sprint 3 Burndown Chart

10.6.2 Positive Experiences

- Even better planning.
- Positive and constructive customer feedback.
- The team functions well together.
- Planned actions completed.
- Easier to pick work items in the backlog after not assigning them at the planning meeting.
- Backlog contained all work that had to be done in the sprint. Implementation, documentation, testing and project management.
- Team members worked more hours.

10.6.3 Negative Experiences

- The documentation can still be better.
- We are not able to finish all tasks in the backlog without working more hours than anticipated.
- During the sprint, we had to add work items to the backlog. This is not correct according to Scrum procedure, but was necessary because of our lack of knowledge of the domain. This is explained in the barriers section.

10.6.4 Planned Actions

As mentioned in the review section; we accomplished all the planned actions. The new planned actions are more specific, because this is the last sprint in this project.

Project management We have underestimated the hours needed for project management in the prior sprints. We will remember this at the fourth sprint planning meeting and will try to give it a better estimation.

Completion of backlog Last sprint means no postponing of work items. We must do a good hour estimation and leave some buffer hours for unforeseen tasks.

Sprint focus The next sprint we would like to focus on testing and bug fixing. This is a natural shift, as the project goes into the completion phase.

10.6.5 Barriers

This sprint the barriers are all concerned with efficiency; bugs, roles and planning.

Technical problems Unforeseen and undiscovered bugs arise in Wireshark while we are developing the utility. We report these to the customer, and they patch Wireshark's source code. Our problem is that we have to wait until the customer has made the fix before we can continue.

Product owner In Scrum, a product owner should be assigned. In our case this is a team member. In a perfect world the product owner would be the customer. The product owner is responsible for prioritizing the product backlog and deciding which work items that should be included in the sprint backlog. We normally use six hours for the sprint planning meeting, and understand that this would be very costly if we were to occupy the customer for this time.

This results in wrong estimates of complexity and we have to add new work items to the backlog during the sprint, as we discover elements that we have overlooked.

Planning meeting It is hard to find a six hour slot where all of the team members can meet. If we do not do the design and proper planning at the planning meeting, we end up with a poor plan. As we get more familiar with Scrum, the planning will probably be more efficient and the hours needed will be less.

CHAPTER 11 SPRINT 4

This chapter describes the work done in sprint 4. The chapter is divided into sprint planning in section 11.1, design made in the sprint in section 11.2, the implementation in section 11.3, the result of the sprint testing in section 11.4, feedback from the customer in section 11.5 and evaluation of the sprint in section 11.6.

11.1 Sprint Planning

The fourth sprint will be the last iteration of this project. The sprint work hours will be split between implementing new features, improving existing features and making sure the utility work properly on Thales' source code.

Before the sprint the customer tried our utility on close to 200 C header files, but was only able to generate four dissectors. For the utility to be of value to Thales we must improve this.

Feedback from the customer before sprint 4 revealed that certain requirements were not fulfilled, and needed to be improved. These were added as work items to this sprint backlog in Table 11.1. In addition, we received new requirements that the customer would like us to implement.

As this is the last sprint, and with the new requirements from the customer it is clear we will not be able to complete them all. We made some of the new requirements optional as we only allowed 350 work hours of tasks in the sprint backlog. We are trying to improve the process by only adding tasks which we can complete during the sprint. We are prepared to work more for any new tasks and unforeseen bugs we must fix.

We have agreed to write suggestions on how to implement any optional tasks not completed during the sprint.

11.1.1 Duration

The sprint started with the planning meeting the 2nd of November and our work started the following day. The sprint duration is 14 days, and will end on the 15th of November with a review meeting.

11.1.2 Sprint Goal

The goal of this sprint will be to focus on fixing and implementing functions that the customer will need to use the utility on their source code. The most important thing to focus on in the beginning of the sprint will be to implement support for #pragma directives and support for including header-files that are not included by the preprocessor. This is important, as it will make it possible for the customer to fully test the utility.

Since the deadline of the project is 24th November, there will also be a focus on preparing a presentation and improving the report for the final delivery. The team are also going to hold a presentation for the customer's developers on the 17th of November, and it is important to also focus on this presentation.

11.1.3 Back Log

This section contains the sprint backlog in Table 11.1 and the timetable for the sprint in Table 11.2.

11.2 System Design

At the start of the fourth sprint our utility was only able to successfully parse a few of the header files in the customer's code base. We had to add functionality to support corner cases of features we already had, and to remove C code which pycparser does not accept, like #pragma directives.

This sprint the design will not be as comprehensive as in prior sprints, for two reasons:

- This was the last sprint, if we were to implement many new features we also had to use time for testing and writing documentation for them, which we did not have.
- The existing features had to be extensively tested and documented before the project was over, leaving less time for other work. Testing and bug fixing was essential this sprint.

After three sprints, the design was to a large degree set. To make the individual modules less complex, we re-factored some code and introduced some new modules, which is described below.

Table 11.1: Sprint 4 Requirement Work Items

User		Hours	
story	Req. and Description	Est.	Act.
Impl.		**47**	**24**
US56	FR2-E: Guess dissector from packet size	5	3
US40	FR3-A mod: Support #include of system headers	8	3
US41	FR3 mod: Ignore #pragma directives	2	1
US42	FR3-A mod: Find include dependencies which are not explicitly set	16	11
US47	FR4-B mod: Custom Lua files support inside a .cnf file	4	1
US49	FR4-D mod: Multiple message ID's for one dissector	2	1
US53	FR4-H:Automatic generation of placeholder configuration	1	0.5
US51	FR4-I: Support specifying the size of unknown struct members	4	1
US43	FR6-E: Support C #defines and –Include from CLI	1	1
US54	FR6-F: Only generate dissectors for structs with valid ID	4	1.5
Fixes		**35**	**43.5**
	TheFIX: Be able to process customer's files	12	17
	FR2-A: Improve generated Lua output	10	20
	FR1-E: Array bug in text	2	1.5
	FR1-E: Pointer support (array)	1	0.5
	FR1-E: Enum in arrays	1	1
	FR6-C: Batch mode (recursive search for subfolders)	8	2
	FR6-C: Support command line arguments for Cpp	1	0.5
	Exclude certain folders and files in batch mode	1	1
Testing		**16**	**7.5**
	Fixing existing tests	3	3.5
	Add more tests for csjark module	3	1
	Add more tests for cparser module	3	0.5
	Add more tests for the config module	2	0.5
	Add more tests for the dissector module	2	0.5
	Add more tests for platform module	1	0
	Add sprint 4 end-to-end tasks	2	1.5
Doc.		**15**	**17.5**
US44	Update command line interface document	1	1
US48	Update user documentation for custom Lua	1	1.5
US50	Update user documentation for message ID	1	1
US52	User documentation for struct size configuration	1	1
US55	User documentation for generating only struct with valid ID or dependencies	1	1
US36	Which platforms that the utility supports	2	1
US58	How to define new platforms to support	1	3.5
US37	Create developer manual from python docstrings	2	2
	Updates and polishing	5	5.5
	Total:	113	92.5

Table 11.2: Sprint 4 Timetable

	Hours	
Description	Est.	Act.
Sprint planning	**30**	**27**
Sprint 4 requirements	**113**	**92.5**
Implementation	47	24
Fixes	35	43.5
Testing	16	7.5
User Documentation	15	17.5
Sprint review	**20**	**20**
Sprint documentation	**46**	**52.5**
Sprint 3 document	4	4
Sprint 4 document	42	48.5
Report work	**38**	**49**
Update tables with actual hours	2	2
Abstract - improve	2	3
Importing user documentation to report	3	11
Introduction section	6	10
Report read-through from a technical perspective	6	6
Report read-through from a non-technical perspective	6	7
Glossary refinement	3	3
Acronym refinement	1	1
Write about optional requirements	4	2
References	3	2
Requirement agreement	2	2
Thales presentation	**28**	**28**
Meetings	**63**	**48.5**
Advisor meetings	28	20
Customer meetings	14	10
Stand-up meetings	21	18.5
Project management	**17**	**27**
Total:	351	344.5

11.2.1 System overview

The addition of new requirements from the customer resulted in changes in the utility's system overview. The need for more custom handling resulted in a re-factoring of the code. This is visible in the class diagram in Figure 11.1, which was extended from the third sprint. Two new modules was added, cpp and field. Major changes to the class diagram are described below. How the modules interact is described in subsection 11.2.2.

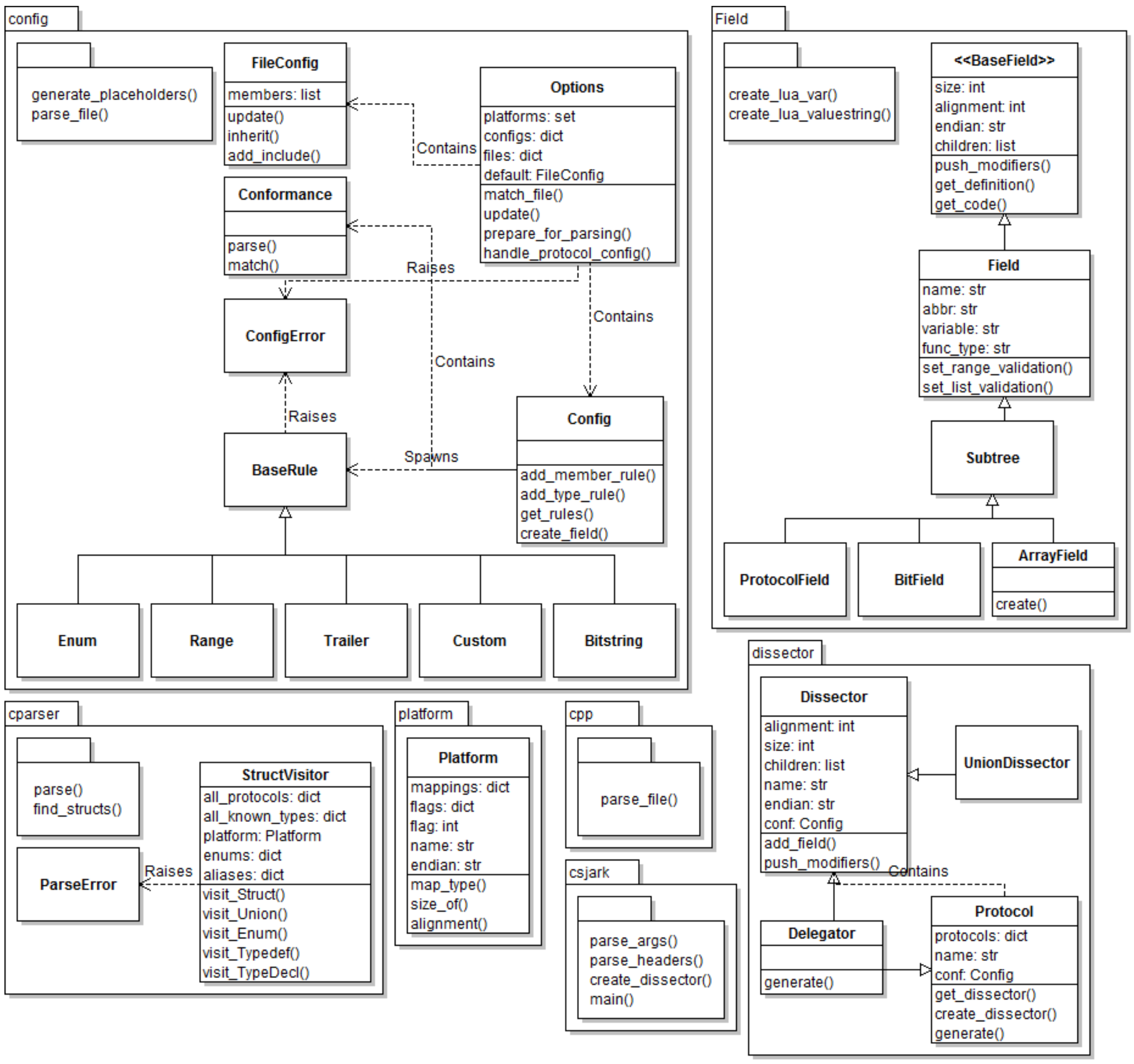


Figure 11.1: Sprint 4 Class Diagram

ccp module

The need for preprocessing C files before they are parsed have become so comprehensive, that we decided to create a dedicated module for it. All the existing cpp concerned code was relocated from the cparser module, and joined with the new implementation. The requirements listed inTable 11.1, made it necessary to have more functionality regarding the preprocessor step. Especially we need to remove parts of the output of the C preprocessor before forwarding it to the pycparser library.

field module

Field specific classes were moved from the dissector module to its own module, field. For the same reason as the ccp module; the amount of code concerning fields became so large, that an own module was appropriate. There was some refactoring in the field module. An interface, BaseField, was added and has Field as a subclass. The class Field is the most important class in this module, and generate code for most of fields used by dissector. Subtree is a class that generates fields for the dissector, that will need a subtree in Wireshark, this class has three subclasses, BitField, ArrayField and ProtocolField.

cparser module

Some changes was done in this module due to the refactoring, functionality for C preprocessing was moved, and some new methods was added.

config module

The class FileConfig was added, to hold the configuration for specific header-files. Some methods was also added for automatic generation of header-files.

dissector module

There was several changes, all the Field classes was added to it's own module. The Dissector-class was added, which holds fields representing one struct for a specific platform. Protocol is now a collection of Dissectors, one for each platform. UnionProtocol was renamed to UnionDissector.

11.2.2 Module Diagram

Figure 11.2 shows the dependencies between the modules. The main module for the utility is csjark, which have the main-method. This module uses config to parse and hold all the configuration, starts preprocessing in cpp module, and then uses cparser to parse the header files which returns all the generated protocols. At the end the dissector module generates Lua dissectors, which the csjark module writes to files.

The cpp module uses the config module to read options given for the c preprocessing.

The config module uses the two modules field and dissector to create configured dissector fields and the delegator class. The platform module is read, to get de predefined platform configurations.

When the cparser have parsed a header file it traverses the abstract syntax tree to create Protocol instances which represents Structs and Unions. To do this the cparser module depends on four modules: config, platform, dissector and field.

The dissector module uses the field module to generate fields for all the struct members. It also uses the platform module to a list of all platforms to support.

The field module depends on the platform module to test if field endian is big or little.

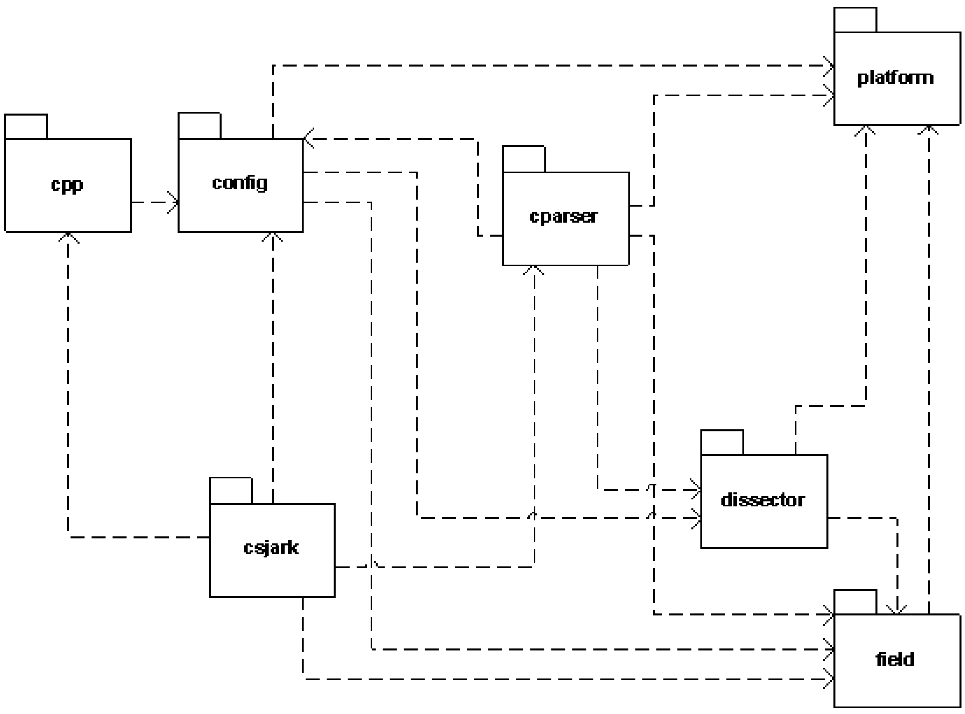


Figure 11.2: Sprint 4 Module Diagram

11.2.3 User Stories

This section lists the user stories for the fourth sprint, these are displayed in Table 11.3, Table 11.4, Table 11.5 and Table 11.6. These user stories represent how we intend to add the functionality of each requirement of the fourth sprint, and contains information on how the modules of CSjark should interact with each other. Some of the user stories explains how the user documentation should be written, while one of them explains the modification of the #include requirement we implemented in sprint 1.

11.3 Implementation

The implementation in this sprint has mainly been bug fixing, and adding features that the customer needs to be able to run the utility on their code. This section covers the added functionality and the most important fixes in this sprint.

11.3.1 Include Unsupported System-headers

CSjark must be able to support both Windows and Solaris operating system. To be able to generate dissectors from header-files, it was necessary to add system specific header-files for Windows and Solaris. This is done by adding the system-headers to the ”fake_libc_include”. ”fake_libc_include” is a fake library for standard header-files, that only contains default typedefs and default defines. Most of the header-files in the fake library are empty, so the preprocessor can find the files that are included.

11.3.2 Ignore #pragma Directives

Pragma directive is a preprocessor directive that is used to give options to the compiler. This can for example be used to ignore warnings or give the compiler version-information of the code. Since the pycparser library do not support pragma directives, these lines in the C-code is removed after the preprocessing. Removing these lines will not affect the utility, since the utility never compiles the code.

11.3.3 Include Dependencies

Include dependencies was the biggest issue in this sprint. The customer include all their header-files in the code file, and they only are going to generate Lua dissectors from header-files. The reason they only will generate from header-files is that some of the code is C++, and the size of the entire source code is about 1GB, which would take very long time to parse for an utility like CSjark.

This problem is very difficult to solve, since the include directive is needed to be able to parse header-files that depend on other header-files. To be able to solve this problem, the utility have to find the includes that the header-file depends on, and include them in the correct order, because the included

Table 11.3: User Stories - Sprint 4 Part 1

Header	Value
ID	US40
Requirement	FR3-A modification: #include system includes
What	The utility needs to support system dependent header-files even if these are not available on the platform that the utility is used on.
How	The administrator is able to specify a fake system header file with the defines they need to make their structs work correctly. This fake header is then used to represent the system header in the file so it is parsed correctly by pycparser.
Result	The administrator is now able to make the utility generate system dependent dissectors for headers with system dependent includes.
ID	US41
Requirement	FR3 modification: Ignoring #pragma directives
What	The utility needs to be able to support header files with the #pragma directive without necessarily having to support the functionality of the directive
How	Before feeding the header files to the parser, the utility needs to be able to run a pass through all of the headers that are to be parsed and remove all of the #pragma directives encountered in those header files.
Result	The user will be able to create dissectors for header files with the #pragma directive instead of having the utility be forced to skip them.
ID	US42
Requirement	FR3-A modification: Find include dependencies which are not explicitly set
What	It should be possible to generate dissector from header-files, that have definitions in header-files that are not included with a preprocessor directive in the header-file.
How	The cparser module has to be able to detect when an exception is raised in the pycparser library, if an exception is raised, cparser has to search through the header-files to find the declaration that the pycparser library failed on, and include this header file. The cparser module will have to do this procedure until the dissector is correctly parsed in the pycparser library.
Result	The utility shall be able to generate dissectors for these header-files
ID	US43
Requirement	FR6-E: Support C #defines and –Include from CLI
What	The administrator wants to pass C #define directives from the command-line to the preprocessor.
How	When CSjark is executed, it takes the arguments given in the command-line interface and store them in the config module. The #define directives must be added to the preprocessor arguments before the header-file is parsed in the pycparser library.
Result	The utility supports C #define directives passed from the CLI.
ID	US44
User doc	FR6-E: Support C #defines and –Include from CLI
What	The user wants to understand how the utility will handle C #defines and what #defines that are possible to pass to the CLI.
How	The user finds the correct section in the user documentation, describing the command-line interface and how C #defines are handled.
Result	The user understands how to use C #defines with the utility.

Table 11.4: User Stories - Sprint 4 Part 2

Header	Value
ID	US45
Requirement	FR7-A: Find struct descriptions from Doxygen comments
What	The utility will read Doxygen comments for a struct and use that to specify the description field for the proto object in the dissector.
How	The utility will search the header files for Doxygen comments before giving the file to the preprocessor. It will note what struct the comment corresponds to and add it to the config module. The dissector module will look up in the config module for each struct and use the description field there if it has been found.
Result	The dissectors now requires less manual configuration because it is able to use some of the text from the header files.
ID	US46
Requirement	FR7-B: Find configuration of #define enums from header files
What	The utility will read #define statements that define the allowed values and the names corresponding to those values for integers that are to be treated like enums, so that the user will not have to configure them manually.
How	The utility will search the header files for define statements that corresponds to a member that is configured to be handled as an enum. The statements needs to follow some configurable format. These statements are then used to auto generate a configuration file for the int member used to make an enum field for the int member when parsing the header file.
Result	The utility now requires less manual configurations to make dissectors interpret certain integers as enums.
ID	US47
Requirement	FR4-B modification: Fetch offset in custom Lua configuration
What	The administrator should be able to add configuration in the conformance file, so it is possible to add custom Lua code with correct offset values.
How	The conformance file must support a variable for offset, and a way to use this. The config module have to read this variable, so it can be used in the dissector module to generate a field in the dissector that uses the correct offset.
Result	The Lua dissector is generated with correct offset for the custom Lua code.
ID	US48
User doc	FR4-B modification: Fetch offset in custom Lua configuration
What	The administrator shall learn how to use offset values in custom Lua configuration.
How	The administrator reads the section in the user documentation, about how to use custom Lua in the utility.
Result	The user will understand how to add offset values in the conformance file.
ID	US49
Requirement	FR4-D modification: Support multiple message ID's for one struct
What	The administrator should be able to configure more than one message ID per struct. Therefore it is possible to use the dissector with several different messages (specified by different ID's).
How	The configuration file must support definition of multiple ID's. These ID's then have to be used for registering multiple protocols for one dissector.
Result	The Lua dissector can be used with multiple messages with different ID's.

Table 11.5: User Stories - Sprint 4 Part 3

Header	Value
ID	US50
User doc	FR4-D modification: Support multiple message ID's for one struct
What	The administrator should be able to find out how to specify multiple message ID's for a specific struct. This include the proper position and syntax of the message ID's specification. Also, he should be aware of the consequences of that definition.
How	The administrator reads the configuration section in the user documentation, about how to specify multiple message ID's for a specific struct.
Result	The administrator is able to specify and use multiple message ID's for a specific struct.
ID	US51
Requirement	FR4-I: Support specifying the size of unknown struct members
What	The administrator should be able specify in the configuration, how big (in bytes) the struct member is without having the member itself defined. Note: This is also a workaround for structs that are not parse-able.
How	The configuration should contain an optional attribute for each struct member which specifies the size of the member. If this member is a nested struct, and this struct is not defined, the size has to be specified. Otherwise the user should be informed about that.
Result	The Lua dissector can be used with C header that includes unspecified struct member. This member was only defined by its size, so it could be displayed as raw data in Wireshark.
ID	US52
User doc	FR4-I: Support specifying the size of unknown struct members
What	The administrator should be able to find out how to specify in the configuration, how big (in bytes) the struct member is without having the member itself defined.
How	The administrator reads the configuration section in the user documentation, about how to specify the size of the struct members.
Result	The user is able to specify the size of unknown struct member.
ID	US53
Requirement	FR4-H:Automatic generation of placeholder configuration
What	The utility will generate template configuration files if it encounters structs with no corresponding configuration file. This is to make it easier to make such a configuration file.
How	The cparser module checks if an encountered struct has a corresponding configuration in the config module. If not, the utility writes a template file for this struct.
Result	The user will now be able to use the auto generated template file to write the configuration for a struct instead of having to start from scratch.
ID	US54
Requirement	FR6-F: Only generate dissectors for structs with valid ID
What	The utility should only generate dissectors for structs that have a configuration file with a valid ID and their dependencies.
How	When the utility discovers a struct definition inside a header file it should check if there exists a configuration file for that struct and if it has a valid ID. If not then the utility should skip that struct and continue with the header file. If a struct with a valid configuration file and ID has a member that is not defined in the current header file, then the utility will check the includes in the current header for the missing structs and create a dissector for them as well.
Result	The utility will not generate dissectors for structs which have not been specified in the configuration. This gives the users of the system the ability to specify which structs they want to look at as well as shortening the time the utility needs to run.

Table 11.6: User Stories - Sprint 4 Part 4

Header	Value
ID	US55
User doc	FR6-F: Only generate dissectors for structs with valid ID
What	The administrator should be able to specify in the configuration which structs should have dissectors created for them.
How	The administrator reads the configuration section in the utility's user documentation that specifies how to specify which structs should have a dissector generated for them.
Result	The user will be able to specify which structs the utility will generate dissectors for.
ID	US56
Requirement	FR2-E: Guess dissector from packet size
What	The utility should be able to generate a Lua file that runs with Wireshark and guesses the dissector that is to be used for a packet, if it has a message ID that does not match any pre-existing dissector.
How	The luastructs.lua file that is generated by the utility should contain a dictionary of all dissectors, sorted by the size of the structs that they are associated with. When a packet with an unrecognized message ID is discovered by Wireshark, the code in the luastructs. Lua file should try to match the unidentified packet with a dissector that has been generated with the same size as the unidentified packet. The matching dissectors should then be run with the unidentified packet.
Result	Instead of only displaying the raw hex data from the unidientified packet, Wireshark should display the packet as containing all of the possible structs and member values the packet might really be containing, as dictated by the matching dissectors. This packet should also be displayed with a warning.
ID	US57
Requirement	FR2-F: Display if struct member contains uninitialized memory
What	The dissectors generated by the utility should be able to identify struct members that might possibly have uninitialized memory set as their values. These members and their values should be displayed with a warning in Wireshark to indicate that the values might have been uninitialized.
How	If the C-code that uses header files isn't using memset to set the initial values of different variables a parser might decide to fill uninitialized variables with some kind of patterned garbage data. This pattern might be possible to detect by the dissectors generated by the utility by adding a check to the dissector code which compares the member values with different known garbage-patterns generated by different parsers.
Result	The utility will now be able to generate dissectors which will make Wireshark display struct members and their values with a warning if they are suspected as being filled with uninitialized memory.
ID	US58
User doc	How to define new platforms
What	The Administrator should be able to define new platforms to support.
How	The user should look in the user documentation, and read the section about defining new platforms.
Result	After the administrator has defined the new platform, the utility should be able to generate dissectors for the new platform.

header-files can be dependent on each other. Another problem is that the parser library we use will raise an exception, this mean that the utility will have to solve the problem from the error message that is given from the parser.

The way CSjark solves this problem is to parse all the header-files. After one attempt of parsing all header-files, the utility will decode the error messages from pycparser library. From these error messages, the utility will try to find the missing includes, by going through the abstract syntax tree from the header-files that was parsed successfully in the first attempt. CSjark will try this procedure several times before it gives up.

This solution is not optimal, but was implemented due to limited time in this last sprint. A possible way to solve the dependencies if CSjark fails, is to add the missing includes to the header-files. We also added support for manually configure which includes a file need.

11.3.4 Improve Generated Lua Output

To improve the performance of the Lua dissectors in Wireshark, it was necessary to change how the different platforms were handled in the Lua dissector. Until this sprint the utility generated one dissector table for each platform. We changed the output to only create one dissector table for all the dissectors, and each dissector have different functions for each of the platforms. To change this, the dissector module in CSjark was modified to generate the dissectors correctly. By fixing this issue, it was also possible to use the auto complete feature in the expression field for filter and search.

11.3.5 Support Sub Folders in Batch Mode

A bug was discovered and fixed in batch processing on non-Windows platforms.

11.3.6 Fixed Proto Fields for Arrays

After adding support for complex arrays in sprint 3, some bugs occurred in the proto fields for arrays. The names for these proto fields were fixed, so it is possible to filter and search for values in arrays.

11.3.7 CLI Support for Include Folders

Support for specifying include folders was implemented in this sprint. This is given as an argument to the CLI with *-I* or *–Includes* followed by the folders to include. The folders included will be added as an argument to the preprocessor, so the preprocessor can search for files in these folders, when a file is given in an include directive(#include).

11.3.8 CLI Support for C Macro Definitions

The utility supports C Macro definitions as arguments from the CLI. A macro definition is also known as *#define*. This feature was added since it should be possible to add macro definitions to the preprocessor, instead of modifying several header-files. During a run of the utility with macro definitions specified, the definitions will be given for all the parsed header-files.

11.3.9 Support Multiple Dissector ID

The dissector ID for dissectors was modified in this sprint, so that a struct can have multiple dissectors. This was done since a struct can have multiple message ID's, in the system that the customer uses. After this fix it is possible to add a list of message ID's to the configuration file of the struct, and the Lua-dissector will add all message ID's to the dissector table.

11.3.10 Configuration of Size for Unknown Structs

Header-files that the utility parses, may have nested structs that are not defined in any other header file. To make it possible to generate a dissector for this case, the size of the struct needs to be specified in a configuration file. When the sizes are specified it will be possible to generate a struct that can display the defined members of the struct correctly in the utility, for the parts that are not defined only the hex value will be displayed.

This feature is added as a possible way to solve include dependencies that our utility is not able to solve. The user of the utility will get an error message when the utility is not able to find the include dependencies, and the user may add the size of struct to be able to generate a dissector for struct. An example of configuration of size for a struct, is shown in Listing 11.1

Listing 11.1: Configuration of struct size

```
Structs:
  - name: cpp_test
    id: [10, 12, 14]
    size: 24
```

11.3.11 Support Offset and Value in Custom Lua Files

After feedback from the customer, we added some more features to the handling of custom Lua files. This feature was that it should be possible to add new proto fields to the dissector Wireshark, with correct offset value and correct Lua variable. To be able to do this, it was necessary to add variables for offset and value in the conformance file. Use of the Lua variable and offset value is only possible in the functionality part of the Lua dissector. Listing 11.2 is an example of how value and offset can be used, and Listing 11.3 shows the result in the Lua dissector.

Listing 11.2: Custom Lua: offset and value

```
#.FUNC_FOOTER pointer
    -- Offset: {OFFSET}
    -- Field value stored in lua variable: {VALUE}
#.END
```

Listing 11.3: Custom Lua: Lua code for value and offset

```
local field_value_var = subtree:add(f.pointer, buffer(56, 4))
-- Offset: 56
-- Field value stored in lua variable: field_value_var
```

11.3.12 Support Array of Pointers

A pointer is a variable that contains the memory address for a variable. Support for arrays of pointers was added so the array can be correctly displayed in Wireshark, with correct sizes for different platforms.

11.3.13 Auto Generate Configuration Files

The auto generation of configuration file is a simple feature that could save the user of the utility some time, since the essential part of the configuration file is generated automatically. The utility will only create a new file, containing the names of the struct and lines to specify the ID for the dissector. To generate the configuration file, the utility must be run with *-p* or *–placeholders* as an option.

11.3.14 Only Generate Dissectors for Structs with Valid ID

Since sprint 2 the utility has been generating dissectors for all header-files found in a folder, when running in batch mode. We have added a strict mode where the utility only generates dissectors for structs that have a configuration file with an ID, and for structs that those structs depends on. This may speed up the generation of dissectors, since it only generates dissectors that Wireshark can use.

11.3.15 Guess Dissector From Packet Size

When Wireshark captures a packet with an unknown dissector ID, it should try to guess which packet that is used, from a list of all generated dissectors.

The luastructs protocol will guess the correct dissector from the size of the packet. All possible dissectors will be displayed in Wireshark with a warning. The reason this was implemented is that a packet can have several message IDs, and it should be possible to dissect a packet, even if an ID was not set.

11.3.16 CLI Support to Exclude Files or Folders

In this sprint it was added functionality to exclude files or folder from parsing. This is done by adding *-x* or *–exclude*, followed by the files or folders to the command-line interface when executing the utility. This makes it possible to generate dissectors, without doing changes to the folder containing all header files.

11.3.17 Configuration of Options

A feature to add some of the command-line arguments to the configuration files was implemented in this sprint. This was added because it is easier to write the options in a configuration file, instead of typing the commands each time a user execute the utility. Listing 11.4 shows an example of how Options can be added to the configuration file.

Listing 11.4: Configuration of Options

```
Options:
    use_cpp: True
    excludes: [examples, test]
    include_dirs: []
    includes: []
    defines: [CONFIG_DEFINED=3, REMOVE=1]
    undefines: [REMOVE]
    arguments: [-D ARR=2]
    files:
      - name: a.h
        includes: [b.h]
```

11.4 Sprint Testing

During sprint 4, the team executed 22 new test cases as well as re-running all of the test cases from the previous sprints. This was done in an effort to ensure that all of the functionality from the earlier sprints were still intact before ending the final sprint. Table 11.7 shows one test case run this sprint. Table 11.8, Table 11.9 and Table 11.10 shows new tests run this sprint. For more information about the test cases see Appendix C in the appendix.

Table 11.7: Test Case TID26

Header	Description
Description	Including system-headers
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system and there has to exist a pcap file which is associated with this test
Feature	Checking that the utility is able to support headers that use system headers
Execution	1. Feed the utility with the name of a C-header file that includes system-headers and its configuration file 2. Read the output 3. Copy the resulting dissectors into the plugins folder of the personal configuration in Wireshark 4. Run Wireshark with the pcap file associated with this test 5. Look at the resulting structs and members are displayed in Wireshark
Expected result	2. The user should be presented with some text expressing the success of generating dissectors 5. The structs and struct members defined in the system headers should be displayed as having a value and not just hex data

11.4.1 Test Evaluation

This sprint was very test intensive as it had been decided that the team would run regression tests with all of the test cases from the previous sprints. It was therefore heartening to see that all but two test cases failed, and that all of the functionality from the previous sprints were still intact. The bugs that caused the one use case to fail was also not deemed severe enough that it would warrant a fix. This was due to the fact that they both included processing invalid input and the complexity of having to fix the bugs. The customer had also expressed that not having CSjark function properly would be the least of their problems if they had header files with invalid C-code.

Table 11.8: Sprint 4 Test Results Part 1

Header	Description
Test ID	TID26
Description	Including system-headers
Tester	Lars Solvoll Tønder
Date	16.11.2011
Result	Success
Test ID	TID27
Description	Ignoring #pragma directives
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Success
Test ID	TID28
Description	Improve generated Lua output by removing platform prefix
Tester	Lars Solvoll Tønder
Date	16.11.2011
Result	Success
Test ID	TID29
Description	Recursive searching of sub-folders
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Success
Test ID	TID30
Description	Finding include dependencies which are not explicitly set
Tester	Lars Solvoll Tønder
Date	16.11.2011
Result	Success
Test ID	TID31
Description	Pointer support
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Success
Test ID	TID32
Description	Enums in arrays
Tester	Lars Solvoll Tønder
Date	16.11.2011
Result	Success
Test ID	TID33
Description	Supporting #define as a command line argument
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Success
Test ID	TID34
Description	Multiple message ID's for one dissector
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Success

Table 11.9: Sprint 4 Test Results Part 2

Header	Description
Test ID	TID35
Description	Allowing configuration for unknown structs
Tester	Lars Solvoll Tønder
Date	16.11.2011
Result	Success
Test ID	TID36
Description	Auto generating configuration files for structs that has no config file of their own
Tester	Even Wiik Thomassen
Date	09.11.2011
Result	Success
Test ID	TID37
Description	Only generating dissectors for structs with a valid ID
Tester	Lars Solvoll Tønder
Date	16.11.2011
Result	Success
Test ID	TID38
Description	Guessing dissectors from packet size
Tester	Lars Solvoll Tønder
Date	16.11.2011
Result	Success
Test ID	TID39
Description	Invalid header
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Failure. Syntax errors are caught by the utility, but declaring variables with the same name several times in the same struct is not caught
Test ID	TID40
Description	Invalid header during batch mode
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Success
Test ID	TID41
Description	Ambiguous struct IDs
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Failure. The user is not even presented with a warning if there are several structs with the same ID
Test ID	TID42
Description	Ambiguous platform IDs
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Success

Table 11.10: Sprint 4 Test Results Part 3

Header	Description
Test ID	TID43
Description	Running the utility on Windows
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Success
Test ID	TID44
Description	Running the utility on Solaris
Tester	Even Wiik Thomassen
Date	14.11.2011
Result	Success
Test ID	TID45
Description	Running the dissectors on Solaris
Tester	Even Wiik Thomassen
Date	14.11.2011
Result	Success
Test ID	TID46
Description	Running the dissectors on Windows
Tester	Lars Solvoll Tønder
Date	14.11.2011
Result	Success

Test Coverage

As can be seen in Table 11.11, the team managed to have a healthy increase in code coverage for sprint 4. This was done in spite of having to add a lot of new functionality to the utility during the sprint, as the team focused on improving the already existing unit tests as well as creating new ones. The following list shows the unit tests modules ran in sprint 4:

- black_box.py
- requirements.py
- test_config.py
- test_cparser.py
- test_csjark.py
- test_dissector.py
- test_platform.py

Table 11.11: Sprint 4 Coverage Report

Module	Statements	Missing	Excluded	Coverage
config	370	19	0	95%
cparser	230	32	0	92%
cpp	63	7	0	89%
csjark	239	85	0	64%
dissector	289	27	0	91%
Field	271	6	0	98%
Platform	73	5	0	98%
Total	1535	174	0	88.9%

11.4.2 Testing On-site at Customer

The customer tested our utility on their own source code before the start of the sprint, but encountered several bugs. They were only able to generate four dissectors from 200 C header files. As the utility worked on the example codes provided by the customer, we were unable to improve the situation without access to their code.

The customer requested that two team members should spend a day at their office, to fix bugs found when running the utility on their code. We had to fill out security clearance forms and non-disclosure agreements before we were allowed access to the code.

Some of the bugs we encountered were trivial to fix, but challenging to understand. For example header files with C++ includes, or "indent" directives without #pragma. We also spent some time fixing bugs that were caused by Python behaviour, which was different on Solaris. At the end of the day, we were able to parse about 40 of 200 files, but tries on the complete source tree failed with "too many open files" error. We agreed to provide one team member for one more day of on-site testing.

In the second day of testing, we fixed several bugs and set up some configuration for includes, defines and excludes. This resulted in our utility parsing 160 of 200 header files, generating over 500 dissectors. These files were the include folder in the customers source tree, and contained the majority of structs the customer wanted dissectors for. One of these files contained the infamous xcon struct, which the customer was very pleased we were able to generate a dissector for.

Attempts at parsing the whole source tree in one run, failed after parsing of 1400 out of 4000 header files, with a "out of memory" error. The virtual machine we tested on only allowed 512 MB memory for each process, which our utility reached.

Overall the customer was very pleased with our willingness to test the utility on-site, and with the results on their include folder. We agreed to investigate if it was possible to reduce the memory consumption, while the customer agreed to test our utility on a machine with more memory. Further improvements to the utility are described in subsection 12.2.2.

11.5 Customer Feedback

This sprint we got mostly feedback on implementations, and on new features the customer would like us to add. They also gave some thoughts on how to organize the sprint and how to test some of the non functional requirements. New requirements and descriptions of existing requirements are listed in subsection 5.2.4. Post-sprint feedback can be found in chapter 12.

The customer wanted the fourth and final sprint to be split into two one week sprints. This would provide the customer with a stable version of the utility after the first week for testing their files. Considering the overhead of an additional sprint, both in planning, evaluation and documentation, the team and the customer instead agreed to provide a stable release mid-sprint.

Some of the header files that the customer wants to generate dissectors for depends on header files not directly included (the dependency might be included above an include to this header file inside a source file). These dependencies must be found so that the file may be parsed correctly.

We also got feedback on how to complete the non-functional requirements. Testing on SPARC platform would not be necessary, the customer felt code inspection would be sufficient. For NR5 the customer would provide a suitable person to test the fulfillment of the requirement. For NR4 and NR6 we did not have resources to perform these tests, and so the customer agreed it would not be necessary.

11.6 Sprint Evaluation

The fourth sprint ended 15th of November with an evaluation meeting. This section gives a summary of our findings.

11.6.1 Review

As this was the last sprint, the pressure for completing the remaining work was high. Even though we agreed to focus on the documentation, we wanted to satisfy the customer as much as we could. Finishing the implementation and making the utility work on their source code would be important, in addition to completing the report work.

All the team members raised their effort this sprint, both in hours and efficiency. All the tasks in the backlog were completed by the end of the sprint. See the burndown chart in Figure 11.3.

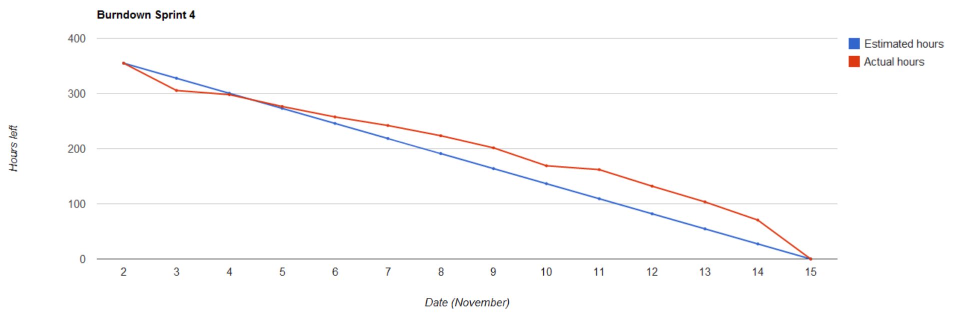


Figure 11.3: Sprint 4 Burndown Chart

The final testing and patching of the utility had to be done at the customer's site. Two of the team members was assigned to this task. To be allowed to see Thales source code, the team members had to sign a non-disclosure agreement and be under surveillance of the customer. As the customer had a demanding project on their own, they could not spare many hours for the testing.

It took long time to get the security clearance needed, so the testing was blocked until the last week of the sprint. The lack of testing time on the real code, could have ended in a unfinished utility. The lead programmer managed to fix all bugs and make the utility able to parse the source code in the end.

Some of the tasks had to be postponed, until the testing and bug fixing were completed, because of dependencies.

11.6.2 Positive Experiences

- We all completed a great amount of work.
- Finished all the items in the sprint backlog.
- Each team member assigned themselves work items from the backlog, and took responsibility for completing them.
- Customer was pleased with our work.

11.6.3 Negative Experiences

- Internal meetings were inefficient.
- Stumbled into blocked tasks because of dependencies.
- Too few attendants at the stand-up meetings.

11.6.4 Barriers

External factors The customer asked if we could do a presentation of the utility for the developers at Thales, when it was finished. This would be beneficial for the final presentation and the customer would be pleased, so we accepted. To make the presentation and rehearse for it, two team members had to be excluded from the sprint work for several hours. External factors like this are not always possible to foresee, and it affected the project.

Thales security It was necessary to have access to Thales source code for the last test- and bug fixing-phase. As a result of the strict rules at Thales, we had to wait with critical implementation and fixing until the middle of the sprint. We managed to have the utility work on their code in the end.

Part III

Conclusion & Evaluation

CHAPTER 12 CONCLUSION

This chapter describes the final state of the product. It also contains suggestions on how to improve the utility, as well as a short discussion on testing, and how it affected our utility.

12.1 System Overview

Figure 12.1 shows an overview of the final version of the product. Our utility consist of seven Python modules:

- csjark
- cpp
- cparser
- field
- dissector
- platform

The csjark module takes as input C header files and config files. Config files are forwarded to the config module, which reads them to find rules and options. The csjark module outputs Lua dissector files.

Csjark then forwards header files to the cpp module, which calls an external program, a C preprocessor. The output of the C preprocessor is given to the cparser module, which forwards it to the pycparser library. An abstract syntax tree is returned by pycparser, which is traversed by the cparser module when it searches for struct definitions.

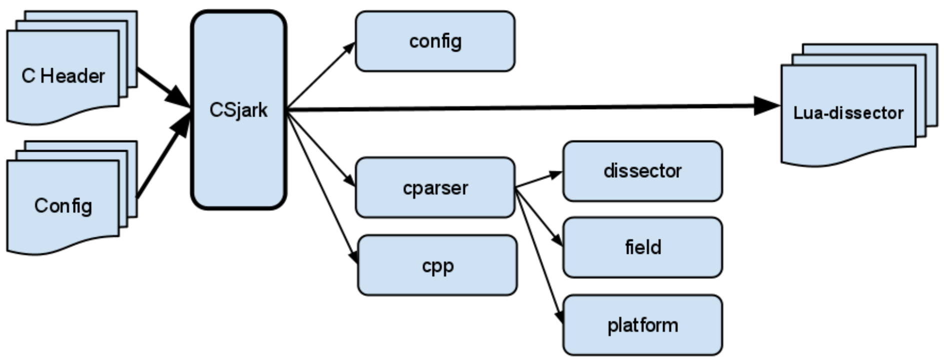


Figure 12.1: Overall Architecture

The cparser module creates a Protocol instance from the dissector module for each struct it finds. The Protocol instance is populated with Field instances or subclass instances for each struct member.

After all headers have been parsed, the csjark module takes the list of all protocols created by the cparser module, and writes to file the output of the generate() function. In the end, it writes the output of the generate function on a Delegator instance.

The platform module contains platform-specific details, which are used by the cparser when it creates new fields.

12.1.1 System summary

Originally, we had some design goals for the product. We wished to have some logical groupings of functionality into a front-end and back-end. We believe that we have achieved this goal. Everything specific for generating Lua dissectors are contained to the field and dissector modules, which contains no C specific code. These consist of smart data structures which are created by the front-end, the cpp and cparser module.

12.2 Further Development

This section describes possible improvements to our utility. In subsection 12.2.1 we describe how one might implement the remaining optional requirements, while subsection 12.2.2 list other improvements.

12.2.1 Optional Requirements

During the third sprint, as we had completed most of the requirements given to us, it became clear we would not have sufficient requirements for a fourth sprint. We requested more possible features from the customer, who provided us with a list of new functional requirements and their prioritization.

In the fourth sprint planning meeting, we estimated the work hours needed to complete each of the new requirements, including implementation, testing and documentation. Based on the customers prioritization and our estimates, we classified four of them as optional, as we did not deem it possible to fulfill them in the fourth sprint.

The customer asked us to provide a description of how the unfulfilled requirements could be implemented, which is listed below.

1. Don't regenerate dissectors across multiple runs
2. Use Doxygen comments for "Description"
3. Read int-enum config from header files
4. Display if struct member contains uninitialized memory

The following paragraphs describe how they can be implemented.

Don't regenerate dissectors across multiple runs To be able to decide if we have already generated dissectors in earlier runs, we need to store some state on disk.

We need to store the last modified timestamp, which the operating system reports, at the time we last read them. This needs to be done for each single input file, both header- and config files. Since command line arguments will affect the output, they must also be stored. The main challenge with this task is the fact that handling #include directives are performed by the external C preprocessor program, so we will not know which files need to be considered when evaluating, if files have been modified since last run.

One could look at #line directives outputted by the C preprocessor before we start parsing files, but the benefit of not regenerating dissectors would be diminished.

We estimate this task would require implementing our own C preprocessor or using a library instead of an external tool, to be able to extract the needed file dependencies. Our utility depends on PLY, which have a 95% finished implementation of a C preprocessor, which might prove valuable for this task.

When we are able to know which files depend on each other, and the last time they were modified, the task is simply to find a suitable data structure to store on disk between runs.

Use Doxygen comments for "Description" Comments are removed by the C preprocessor, which means we must parse them before it is run. As the preprocessor evaluates which files to open, we would be required to implement our own or use a library, or try to evaluate applicable files in all include folders.

The task, if such support was available, would simply be to search for doxygen comments, and when one is found parse it to extract the correct information.

Read int-enum config from header files Integers, which should act as enums in Wireshark, are defined by some specific C preprocessor macro define directives in the customer's current header files. This task is to automatically extract such information, to require less manual configuration of our utility.

Like the two previous task, this will require us to parse C preprocessor directives before they are removed by the C preprocessor, which means we must implement our own C preprocessor or use a library.

To avoid having to parse both C preprocessor directives and C code at the same time, we could design a syntax for describing which struct member(s) the macro define directives refer to. For example #define BEGIN_CSJARK_ENUM_FOR_NAME could be placed right before the current enum macro directives start, which would tell us that they refer to struct member NAME.

This task becomes trivial to implement if we had a custom C preprocessor we could modify.

Display if struct member contains uninitialized memory Uninitialized memory will look different depending on the compiler, so therefore we need to add support for specifying how it will look for each Platform instance in platform module. Since we can only evaluate the actual memory on the Wireshark end of things, most of the functionality must be written in Lua code.

These two conditions suggest that the dissector module should, inside the Delegator class, generate a suitable Lua function in luastructs.lua which accepts a buffer value for a field and the field node. This function should, if the buffer value match uninitialized memory, set an appropriate warning on the field node.

This new function must be called for every field defined in every dissector we generate, inside the appropriate dissector functions.

In addition to implementation, the task involves researching how uninitialized memory looks on different platforms we support, and creating pcap files for testing the functionality.

12.2.2 Additional Improvements

In subsection 11.4.2 we described testing of our utility on customer's code base. We discovered several corner cases which we did not support, and we also found problems with memory consumption when number of input files was over 1000. These problems could be solved by further development, which we describe in following paragraphs.

Reduce memory consumption We did some basic profiling, both of memory and CPU usage, to evaluate what could be improved. Almost all CPU usage was defined to the preprocessor and the pycparser library, which means we could not find any possible improvements in out utility.

Memory profiling on 1000 header files revealed that almost all memory was used by Python dictionaries and lists. Our utility builds up a list of Protocol instances which represent all structs we parse, before we write any to disk. This grows as more files are parsed. A simple, but effective, solution would be to write Protocols to disk as after we have parsed a single file. We would still need to store a few attributes for each Protocol, such as name, id, size. We believe this would reduce memory consumption sufficiently.

It is also possible that we leak some memory each time we fail to parse a file, which could be investigated and fixed afterwards.

Additional C parsing support We successfully parsed 160 of 200 header files in customer's include folder, the remaining files failed mostly from corner cases our utility did not support, for example typedef's we had not considered. We believe it would be relatively easy to add further support for them when they are discovered.

Less manual configuration Several of the optional requirements are focused on requiring less manual configuration. If we perform the C preprocessing ourself instead of delegating it to an external program, we would be able to read configuration from comments and #define's inside the header files.

12.3 Testing

At the beginning of the project all of the team members agreed that we were going to focus on doing extensive and proper testing of our utility. In this section we will discuss whether or not we were able to reach the goals we sat at the beginning and during the project.

12.3.1 Methods

At the beginning of the project we only used black box test cases and unit tests. We quickly found that even though we were able to uncover several bugs using these methods, it was hard to calculate what and how many parts of the system were actually undergoing testing. This again made it difficult to figure out the real quality of our tests and if our testing efforts were used to their true potential. When we then decided to use a tool for calculating code coverage, we noticed that there were large parts of the system that were still untested, even though the functionality of the system had been tested in our black box test cases. We therefore made a goal of trying to have a test coverage of at least 80% in order to catch as many bugs as possible. This proved to be a hard task at first, but as can be seen in Figure 12.2, once we got used to using a tool to calculate the coverage of our unit tests, we were able to improve the quality of our tests so that they covered more parts of the system.

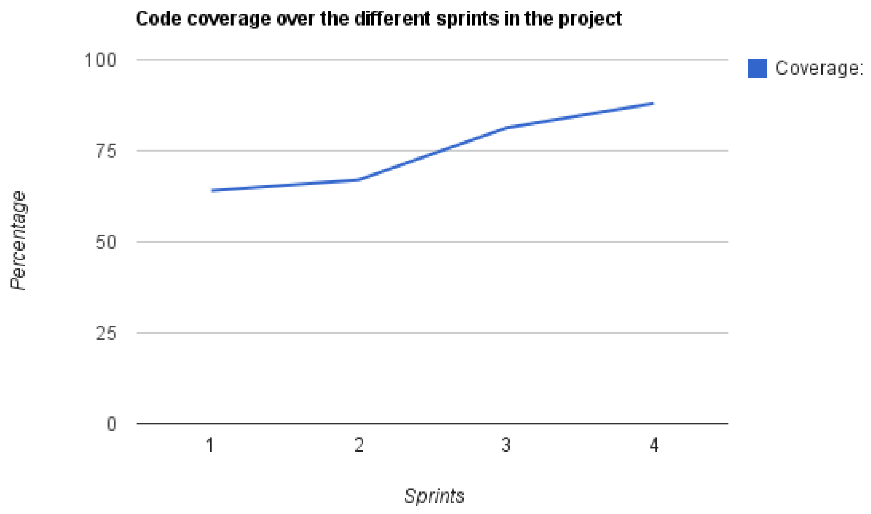


Figure 12.2: Code Coverage Progress from Previous Sprints

12.3.2 Testing Conclusion

After finishing the project we had uncovered several serious and many more non-critical bugs in our utility. We also discovered several bugs in the Lua-implementation of Wireshark that we needed to have our customer to fix. This proved to be very important as this severely reduced the amounts of bug fixes that had to be implemented, after being allowed to send our developers to work with the utility at the customer-site. We were therefore able to create a working product within the small time-frame we had left after testing the utility at the customer-site. We therefore conclude that it had indeed been necessary to focus on creating extensive tests for the utility and that the methods we had chosen for testing were sufficient to get a working end-product.

12.4 Summary

The team was given the task of creating an utility for automatic generation of dissectors for C structs in Wireshark. The dissectors were supposed to be created for structs contained in the customer's C header files so that they could be used to decode their inter-process communication.

After finishing the product, the customer reviewed the utility and they were satisfied with what we had delivered. We managed to complete all of the initial functional- and non-functional requirements. In addition we also completed all the non-optional functional requirements added by the customer before the last sprint.

The team firmly believes that the customer will be able to use the utility for its intended purpose, without having to do much additional work. This makes it feasible for them to use Wireshark for debugging inter-process communication. We therefore feel that we have delivered a solution to the task that the customer presented in the beginning of the project.

CHAPTER 13 PROJECT EVALUATION

This chapter gives an evaluation of the project. The main focus is on the team's work process, and on the issues that we encountered during this process.

13.1 Team Dynamics

The team dynamics and development of the team are described in this section.

13.1.1 Goals and Team Building

The project started out with randomly assigned student groups of six to seven people. This was done intentionally to learn the students to work in a realistic setting. Our group consisted of six Norwegian students and one Czech student.

As one of the team members was a foreign student, all the internal team communication had to be done in English. In addition, our advisors did not speak Norwegian, so the entire project was done in English. This was not a problem, because all the team members both spoke and wrote English fluently.

At the first meeting we decided to state our personal goals for the project. This resulted in the following list:

- Improve programming skills.
- Learn to develop software efficiently.
- Learn to work in a realistic environment.
- Fulfill the needs of the customer.

These goals match the course's goals, with some extensions. All these goals were common for the team members, which resulted in good cooperation from the start and an agreement of what we wanted to achieve in the project. The decision to characterize ourselves as a team came early in the project, which is defined to be a collection of people working interdependently and is committed to achieve one common goal.

None of the team members had any relations with each other when the project started. During the project we learned how to work together in a professional and efficient manner. Getting there was a demanding task and is described more thoroughly in the section regarding team evolution.

13.1.2 Team Evolution

During the first weeks of the project the team was in a good, but also uncertain stage. The team members did not know the boundaries of the others and tried to not end up in an argument. This resulted in a series of matters; responsibility for tasks were not taken and no one dared to ask why tasks was not done.

The problem was discovered and dealt with at an early stage of the project. We quickly realized that changes had to be made to ensure high work efficiency and effort. We started out by listing all tasks in an internal work sheet. Then everyone could see the tasks that had to be done before the end of the sprint, but there was an additional problem: we listed a person responsible for each task at the sprint planning meeting. This changed between sprint 2 and 3. We started to restrict the number of tasks that a person could be responsible for, from that moment, one person could only have one task a time. This increased the efficiency of the work flow for the rest of the project.

The work related problems ended in many discussions and arguments, which raised the conflict level within the team. Conflicts can to some degree raise the productivity, as illustrated in Figure 13.1 [12]. The team was aware of this, so we kept the conflicts to a moderate level.

Even though we had some difficulties in the start, this did not negatively influence the final product or report. The lack of hours used on task completion, early in the project, was improved in the later sprints, and the hours we used for discussions contributed to a better team dynamic and problem solving for the rest of the project.

13.2 Risk Handling

Some of the risks predicted in the planning phase occured to a certain degree during the project. This section will discuss those risks and how they were handled.

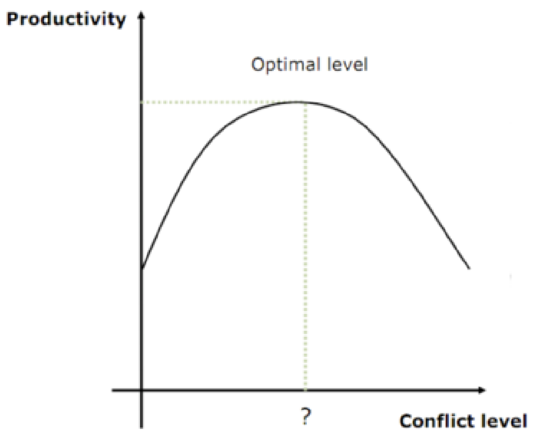


Figure 13.1: Optimal Conflict Level

R4. Illness/Absence In general, not many team members were absent for longer periods. One team member was away in northern Norway for a week on vacation, and another was sick for a week and a half. This caused some delay on a few tasks, as the absentees had to get up to date on the state of the project, but it did not majorly hinder the progress of the team. The consequence of this risk was also diminished by the fact that it occured early in the project, and that the team members who were absent did not work on critical tasks at the time.

R6. Conflicts within team In the start of the project, the team members were divided on which tools to use, and on which programming language to use. Because of this, the team had to spend some time discussing back and forth. After some constructive discussions the team was able to come to a mutual decision.

Also, some team members felt that it was not realistic that the course should demand 25 hours from each member. As the project progressed, it quickly became apparent that this number of hours were needed to complete the project in a satisfying manner. This led to an overall increase in work effort in the team.

R8. Miscommunication within team During sprint 3, the ones responsible for creating test data and the ones writing the test cases did not clearly communicate with each other. This led to having to make small changes in some of the test cases to be able to use the test data.

The test responsible wrote test cases for functionality that the programming team had not thought about. For example, checking that you are not allowed two platforms with the same platform ID. When the problem was detected, all essential functionality was added.

R10. Lack of experience with Scrum As the team had no previous experience with Scrum, the project got off to a slow start. After evaluating the first sprint, it quickly became apparent that the process was far from perfect. The second sprint was an improvement of the first, and the planning meeting was longer and more detailed, but in our opinion it was still not good enough. We felt that we did not adhere to proper Scrum, and that we did not properly explain the different tasks in the backlog. In the last two sprints, we felt that we had achieved a better understanding, and this really showed in the process. A more detailed discussion can be found in the Scrum section.

R11. Requirements added or modified late in the project At the customer meeting on the first day of sprint 4, the customer suggested several new requirements that they would like is to implement. Some requirements were also modified, as we had not implemented them exactly the way they wanted us to.

As we had to focus on tweaking some functionalities to work on their code, and also had to spend time on writing the report, we had to tell the customer that we would probably not be able to finish all the new requirements. This is because implementing a requirement would also require testing, user documentation and additional report work, which is something we could not afford to allot time for.

13.3 The Scrum Process

During the preliminary study, we decided to use an agile development method, to be able to adopt to changes in requirement during the project. Because of this, we chose to use Scrum. None of the team member had any previous experience with Scrum. In the beginning of the course, we had a lecture where we learned the basics of it.

When the first sprint planning was finished, we understood that we did not follow Scrum properly. The meeting was very short, and we did not do any design during the sprint planning. The work items from the sprint were wrongly estimated, and was not divided propely into tasks. For each sprint we improved, and in the last two sprints we felt that we understood Scrum, which led to a better process.

Some of team members feel that we followed Scrum too strictly, instead of doing what was best for the project. In the end of the project, we needed to use time on fixing the utility so it would run on the customer's code. And since we could not change our sprint plan, we ended up with a very high workload.

During the project we learned the advantages of using Scrum. The biggest advantage was that we could get weekly feedback on the work we had done from the customer, and then improve the features to what the customer wanted.

13.4 Time Estimation

In total, 2275 hours were estimated to use on this project. Our effort was even better than the estimate, and we used in total 2331 hours. The main reason was that we wanted the utility work for the customer and we also had to spend a significant amount of time on the report in the end of the project.

The work breakdown structures in Table 3.1, shows the estimated and actual hours for each task. The estimates are quite good on most of the tasks. For project management we have used 454 hours, and 275 hours were estimated. The reason that we used so many hours on project management is that we had a weekly meeting with both customer and advisor. In addition to this, we had internal meetings in the team, and all daily Scrum meetings were registered as Project Management. In the start of the project some of the effort was registered, so the amount of time spent on project management is actually lower.

In each of the sprints we estimated time for the work items. The estimation was based such that every team member should be able to finish the task on the estimated time. So in total the work items were overestimated, because some team members had more experience, and could finish the task much faster.

The time distribution by task is shown in Figure 13.2. Nearly 20% of the time was used on the project management. The planning, preliminary studies and requirement specificaiton was approximately 10% of the work load each. The actual amount is actually higher, because of some wrong registration of effort in the start of the project. The total time that was used for the sprints was over 40%., which is a good amount for the development of CSjark.

In the start of the project there was low activity, but effort per week increased during the project. Effort registered per week can be seen in Figure 13.3. To achieve the 2275 hours in total for the project, 175 hours per week was needed. The effort in week 12 was high because we needed to finish the last sprint and a significant work on the report was done.

13.5 Quality Assurance

At the start of the project we made several plans for ensuring quality of the project. This section discusses which of those plans did not work out and what we ended up doing instead.

Initially, we planned that we would have peer review of all of the code written for the utility. This plan was something that we were unable to follow

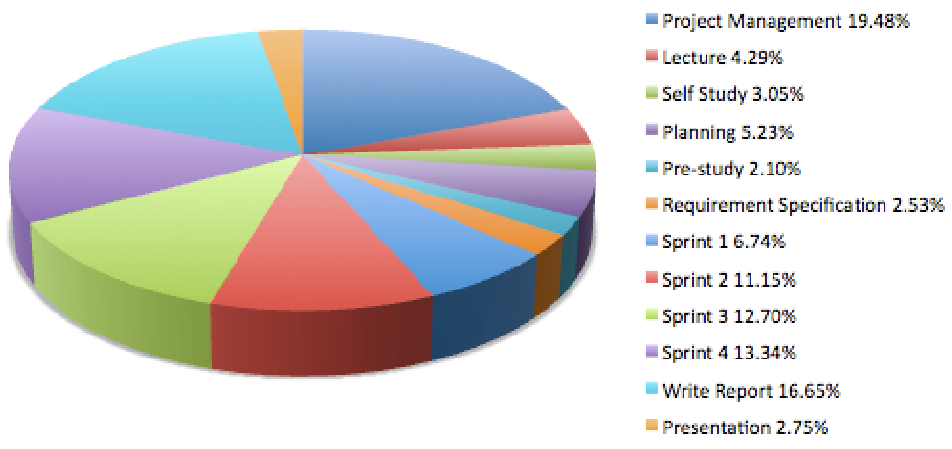


Figure 13.2: Time Distribution by Task

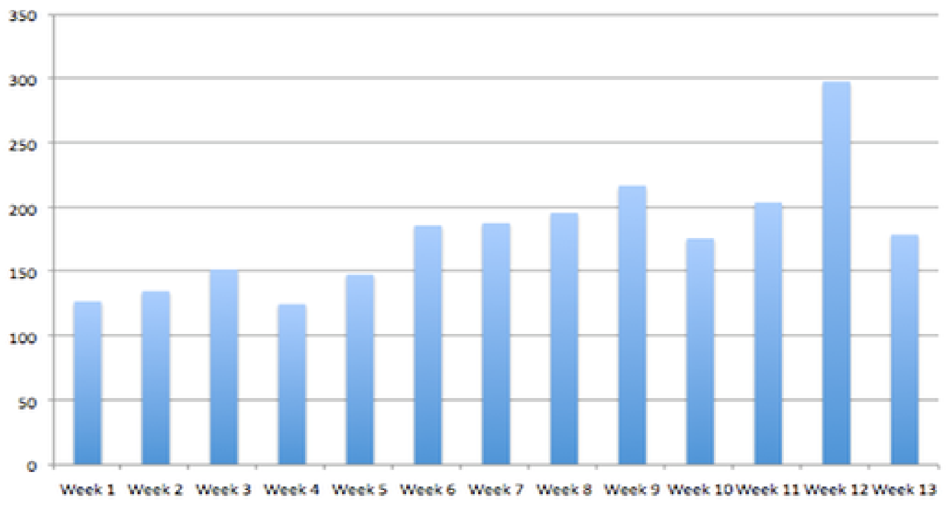


Figure 13.3: Time Distribution by Week

up on, mostly due to the poor planning meetings in sprint one and two where the time needed for pair review was not taken into consideration. This made it difficult to organize who was going to do the peer reviews and how we were supposed to find the time to do them. In the end, the lead programmer ended up reviewing most of the code written by the other developers, but the group consensus was that the quality of the code was good enough that it would have been irresponsible to spend more time on peer reviews.

During every sprint, the team was supposed to provide the advisor with up to date information regarding the report and the progress with the utility. This was not followed up on as the team members did not want to submit unfinished work to the advisor. This proved to be a problem towards the end of the project, forcing the team to start submitting more of the report work to the advisor. As this was not done earlier in the project, the team instead had to organize for several of the members to go through the early stages of the report and raise the quality of the documents internally.

It was planned that the team member responsible for the document was supposed to keep a bird's eye view of the report and go through the different entries before the end of each sprint. This proved to be too much work for just one person. We therefore also made sure that several other team members would support the one responsible with reviewing the report work.

13.6 Customer Relations

Our relationship with the customer was good throughout the entire project. We were assigned two customer representatives. As they were developers themselves, they both had a technical background. This made it easier to communicate with the customer. They knew exactly what they wanted, and were more than happy to give feedback and tips on the implementation of the different requirements. When we first received the initial requirements, we did not fully understand them. After some meetings and dicussions, both internally and with the customer, the team was able to provide the customer with a clear requirement specification that they accepted.

One of the customer contacts was a member of Wireshark's core development team. When we discovered bugs in Wireshark, he could fix them and apply a patch. He also had insight into how Lua dissectors were built, and how they interacted with Wireshark. This was a major asset to the team. When we encountered problems during implementation, the customer was able to give us invaluable feedback on how to proceed. This is something that a less technical customer would not have been able to give us.

Each week we had a meeting with the customer where we demonstrated the requirements we had implemented since last meeting. The customer then gave us feedback on the functionalities, and told us what we had to change or improve. This way, any misunderstandings were detected and dealt with in a swift manner.

The customer contacts also had access to our repository, thus they could test the utility themselves. This meant that they could get a first hand-experience of the utility, and could more easily detect bugs and other issues.

A problem we had was that, due to security issues, Thales could not allow us to test our utility on their source code. This meant that we had to do some guessing during the implementation, and getting the utility to work on their code was a difficult process. Luckily, Thales gave us the opportunity to send two of our members to them for a few days, as documented in the test section of sprint 4. That gave us an opportunity to fix most of the bugs that we encountered, and in the end the customer said that they were happy with our utility. They approved all the requirements we had listed, although a few of the optional requirements were not implemented due to time constraints. Overall the customer felt that we were flexible, and adhered to most of their needs.

The week before the end of the project, the team had a presentation at Thales for our advisor, the customer contacts, and some other employees of Thales. The employees that were there are probable users of the dissectors that our utility creates. Due to this, the presentation was focused on the demonstration of the utility, where we showed how the structs were dissected in Wireshark.

The customer also got one of their co-workers to read CSjark's user manual. The user manual was a little too unclear in some parts, so we used that feedback to improve it, and increase its quality.

The team felt that we were lucky with the customer that we were assigned. The fact that they knew exactly what they wanted, had a technical background and showed great enthusiasm for the project, made the project more manageable. This also increased the motivation for the team. We felt that they were interested in what we were developing, which ensured a good meeting atmosphere and a good working relationship. Throughout the entire project, they gave us essential feedback on the implementation and requirements, and they were not afraid to tell us if we did something that they disagreed with. The fact that we had two customer contacts, instead of one, increased the amount of feedback we received, and also meant that there was always one person available for us to contact. All in all, we could not be more satisfied with our customer.

13.7 Summary

We are overall very satisfied with what we have achieved in this project. Seven people with different personalities and skill sets were randomly assigned to a group. We received a task that we initially did not quite understand the scope off. But through hard work, and help and feedback from the advisors and the customer, we managed to create a utility that we are proud of. In the beginning, the process did not go as smoothly as we wanted it to, as we struggled to fully grasp and utilize Scrum. After a few sprints, we felt that we understood more, and this led to an increase in both work effort and cooperation. People took more responsibility, and we started to feel like a team with a common goal. During the road to the delivered solution we went through a challenging process that has given us invaluable work experience.

BIBLIOGRAPHY

[1] About Python. http://www.python.org/about/. [Online; accessed 09. November 2011].

[2] Asn2wrs. http://wiki.wireshark.org/Asn2wrsl. [Online; accessed 15. November 2011].

[3] Java. http://www.java.com. [Online; accessed 09. November 2011].

[4] Java New Input/Output. http://www2.sys-con.com/itsg/virtualcd/Java/archives/0902/krishnan/index.html. [Online; accessed 12. November 2011].

[5] Thales Norway AS. http://thales.no/pub/sites/index.php?siteID=4&m=1. [Online; accessed 09. November 2011].

[6] Waterfall 2006. http://www.waterfall2006.com/. [Online; accessed 21. November 2011].

[7] Wireshark - About. http://www.wireshark.org/about.html. [Online; accessed 09. November 2011].

[8] Lua: About. http://www.lua.org/about.html, 2011. [Online; accessed 09. November 2011].

[9] Lee Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition.* Addison-Wesley Professional, April 2003.

[10] NTNU IDI. Compendium: Introduction to course TDT4290 Customer Driven Project, Autumn 2011. Technical Report ISSN: 1503-416X, Department of Computer and Information Science, NTNU, August 2011.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, April 1988.

[12] Michael Sars Noru. Project Management. `http://www.idi.ntnu.no/emner/tdt4290/2011/slides/20110906projectmanagement/BearingPoint-ForelesningKPRO(NTNU)3.pdf`, 2011. [Online; accessed 22. November 2011].

[13] Tobias Oetiker, Hubert Partl, Irene Hyna, and Elisabeth Schlegl. *The Not So Short Introduction to LaTeX*, 4.13 edition, 10 September 2003.

[14] Software Engineering Technical Committee of the IEEE Computer Society. *IEEE Standard for Software Test Documentation*, ieee std 829-1998 edition, 16. September 1998.

[15] Eric Raymond. The Cathedral and the Bazaar. *Knowledge, Technology & Policy*, 12:23–49, 1999.

[16] Linda Rising and Norman S. Janoff. The Scrum Software Development Process for Small Teams. *Software, IEEE*, 17:25–32, 2000.

Part IV

Appendices

APPENDIX A ACRONYMS

ANTLR ANother Tool for Language Recognition. 31, 32, 38, 39
API Application Programming Interface. 31, 40, 135
ASN.1 Abstract Syntax Notation One. 27, 32, 109, 110

BER Basic Encoding Rules. 109, 112
BSD Berkeley Software Distribution. 34, 39, 41

CLI Command Line Interface. 79, 80, 86, 87, 98, 113, 128, 156, 160, 161, 163
CORBA Common Object Request Broker Architecture. 27
CVS Concurrent Versions System. 11

GCC GNU Compiler Collection. 31–33, 39, 87
GPL General Public License. 40, 41

H High. 22–25, 42, 44, 45
HTML HyperText Markup Language. 35, 36

IDE Integrated Development Environment. 36, 37
IDI Institute for Computer Science and Information Technology. 12
IDL Interface Description Language. 26
IP Internet Protocol. 7
IPC inter-process communication. 1
IT Information Technology. 8

JavaCC Java Compiler Compiler. 31

L Low. 22, 24, 42, 44, 45

M Medium. 22–25, 42, 44, 45

MIT Massachusetts Institute of Technology. 33, 39, 41

NATO North Atlantic Treaty Organization. 7
NTNU Norwegian University of Technology and Science. i, 7, 8, 12, 15, 16

PLY Python Lex-Yacc. 31, 32, 39, 41, 80

SPARC Scalable Processor Architecture. 45, 125, 135, 140, 142
SVN Subversion. 11

UML Unified Modeling Language. 13

VIM Vi IMproved. 37

XML Extensible Markup Language. 33, 35

YAML YAML Ain't Markup Language. 33, 39, 40, 88

APPENDIX B GLOSSARY

#define A C directive that can be used to define a constant or create a macro. 44, 60, 84, 86, 90, 107, 156, 157, 161, 194, 208
#if A C directive that executes a statement if a given expression holds true. 44, 60, 84, 86, 87, 90, 194
#ifdef A C directive that checks if a given token has been defined. 87, 201
#include A C directive that includes other header files to the current file. 44, 60, 78, 79, 84, 86, 90, 96, 150, 155, 156, 160, 194

Abstract Syntax Notation One A data representation format. 27
abstract syntax tree A tree represention of a compiled program. 33, 38, 40, 80, 84, 104, 134
argparse A Python module for writing a command line interfaces. 85
array A data type that can hold a collection of elements. 44, 46, 60, 86, 87, 97, 103, 109, 111, 117, 141, 150, 160, 196, 198, 208

batch mode Automatic execution of a series of programs. 45, 96, 105, 140–142, 150, 162, 166, 199, 211
batch processing See batch mode. 131, 211
binary Two base arithmetic using the digits 0 and 1. 3, 7, 9, 42, 44
bit string An array that stores individual bits. 44, 96–98, 103, 109, 117, 197, 198
boolean A data type that represents logical truth, it can have the values True or False. 44, 60, 79, 84, 86, 90, 193
branch A feature of a version control system that enables modifications of files in parallel, by duplicating the originating code. 11, 33, 35, 39

C A programming language. i, 1, 3, 4, 6, 7, 9, 10, 26, 27, 30–33, 38–42, 44, 50, 60, 65, 66, 73, 78–87, 89, 90, 94, 103–107, 116, 125, 132–135, 140–142, 155, 156, 158, 159, 161, 193–201, 203–210, 212, 213, 217, 219

C++ A programming language. 7, 31–33, 38, 155
C99 A modern extension of C. 32, 39, 94
char A data type in C that contains a character or a small integer. 44, 60, 79, 84, 86, 90, 193
clang A compiler front-end for different C programming languages. 31, 33, 39
Common Object Request Broker Architecture A standard for enabling pieces of software written in different languages to work together as a single application. 27
cron A program in Unix that enables the user to schedule the execution of command-line programs.. 47

data serialization The process of converting a data structure to a storable format. 33
dissector Code that decodes packet data and makes it readable by humans. i, 1–4, 6, 7, 9, 10, 26, 27, 30, 31, 40, 42, 44–48, 50, 51, 60, 66, 68, 71, 73, 74, 77–81, 84–88, 92, 97, 98, 102–107, 109, 111–113, 116, 118, 124, 125, 128–137, 139–141, 148, 150, 155–162, 166, 167, 193–213, 215, 217–221
distributed repository model A distributed approach to a version control system. 11
double A data type in C that contains a double precision floating point number. 86

Eclipse An application aiding computer programmers in software development. 37
endian See endianness. 44, 47, 60, 98, 105, 113, 125, 126, 130, 133, 135, 142, 203
endianness Refers to the ordering of bytes in a word. A big-endian machine stores the most significant byte first, and a little-endian the least significant.. 96, 129–131, 133, 135, 144, 203
enum See enumerated named value. 44, 60, 96–98, 102, 106–108, 116, 117, 141, 150, 157, 195, 197, 208
enumerated named value A value of a enumerated type in C. 60, 97, 106
Extensible Markup Language A markup language. 33

float A data type in C that contains a floating point number. 44, 60, 79, 84, 86, 90, 139, 143, 193

GCC-XML Acronym. 33
GNU Compiler Collection A compiler front-end that supports many different programming languages. 31

header A file that contains C declarations and macros that can be shared by several source files. 1, 6, 9, 21, 26, 30–32, 38–40, 42, 44, 46–48, 60, 66,

72–74, 78–81, 84–90, 96, 99, 103–106, 108, 109, 112, 113, 116, 129, 131–134, 139, 141, 142, 144, 149, 150, 155–162, 166, 193–213, 217, 219–221
hex dump A hexadecimal view of computer data. 13, 66
hexadecimal A number system where sixteen is the base. 140

int See integer. 44, 60, 79, 84, 86, 90, 157, 193
integer A data type in C that contains an integer. 47, 86, 109, 114, 136, 157
inter-process communication The exchange of data that happens between processes. 1, 9, 30, 42, 179

Java A programming language. 30–33, 38, 39
Javascript A scripting language. 33

lexer A lexer is a program that converts a sequence of characters into a sequence of tokens. 31, 32, 38
library A collection of pre-written code for aiding programmers in the development process. 1, 31–35, 37–41, 80, 84–87, 94, 155, 156
Linux An operating system. 87
Lua A programming language, often used for making scripts. i, 2–4, 6, 7, 9–11, 13, 21, 30, 31, 40–42, 44, 45, 47, 60, 71, 73, 77, 80, 81, 84, 86, 90, 98, 102, 104, 105, 109, 111–113, 116, 125, 131, 133–136, 140, 142–144, 150, 155, 157–162, 195, 199, 204, 206, 218–220

Mac A brand of personal computers. 87
makefile A file that helps the make utility in the creation of executables from source code. 27
markup language A language for specifying the processing, definition and presentation of text. 12
member A representation in C that contains data or behaviour of a struct. 3, 4, 9, 44, 47, 60, 66, 73, 78–80, 84–86, 88, 97, 102–104, 107, 110, 113, 114, 116, 117, 125, 128, 132–137, 139, 140, 142, 157–159, 161, 195–205, 207–209, 211, 213, 221

nested struct A struct within another struct. 10

Objective-C A programming language. 33
Objective-C++ A programming language. 33

packet Small block of data transmitted over a network. 2–4, 7, 21, 47, 65, 66, 86, 103, 112, 116, 124, 131, 132, 137, 140, 142, 150, 159, 166, 195–198, 201–203, 207–209, 211, 213
parser A program that receives input, checks it for correct syntax and builds a data structure representing the input. 31–33, 36, 38, 39, 73, 77, 80, 84, 85, 94, 97, 103, 105, 107, 108, 132, 133, 156, 159, 219
pcap-file See capture file. 13, 116, 142, 195–205, 207–209, 211, 213

Perl A programming language. 33
PHP A scripting language. 33
post-dissector A dissector that is run after every other dissector has been run. 30
preprocessor A program that prepares code files for compilation. 10, 32, 38, 39, 41, 44, 60, 73, 77, 80, 84, 86, 87, 133, 149, 155–157, 160, 161, 219
process A program running on a computer. 2, 4
protocol A system of rules for exchanging messages between machines. 2, 4, 6, 7, 27, 30, 44, 47, 102–104, 128, 129, 157
pycparser A C parser written in Python. 1, 31, 32, 39–41, 80, 86, 94, 149, 153, 156
Python A programming language. 1, 21, 26, 30–36, 38–41, 125, 135
Python Lex-Yacc A Python library for creating lexers and parsers. 31

repository A central storage area where data is kept and maintained. 11, 20, 21
Ruby A programming language. 33

script A list of commands that are executed by a certain program, usually as an extension of the original functionality. i, 9–11, 13, 21, 73, 111, 113, 219
Scrum A software development methodology. 17, 18, 23, 25, 28, 29, 38, 91, 93, 95, 123, 145–147
Solaris An operating system by Sun Microsystems. 45, 167, 213
string A string in C is a character string stored as an array containing the characters. 4, 143
struct Short for structure, it is a type that groups several members into a single object. i, 2–4, 6, 7, 9, 10, 27, 30, 32, 40, 42, 44, 46–48, 60, 65, 66, 73, 78–80, 84–86, 88, 90, 97, 98, 102–105, 107–111, 113, 114, 116, 117, 129, 131–137, 139–141, 150, 156–159, 161, 162, 166, 193, 195–202, 204–213, 217, 219, 221
Sun RPC The Unix equivalent of Remote Procedure Call. 27

trailers define this. 46, 60, 98, 109, 117, 198

union A struct where all the members share the same memory, ensuring that only one member is valid at the same time. 44, 60, 125, 128, 131, 132, 139, 142, 200
utility A small program that supports larger applications by doing certain tasks. i, 1–3, 6, 7, 26, 27, 30, 32–45, 47, 48, 50, 51, 62–68, 70–73, 75, 77–79, 81, 83–88, 91, 92, 95–97, 99, 102–107, 109, 113, 116, 119, 120, 122–126, 128, 129, 131–135, 137, 139–141, 143, 144, 147–150, 155–162, 167, 193–213, 215–219, 221, 222

version control system A system that ensures consistency of files when several people are collaborating on them. 11, 12

wildcard A character that can be used as a substitute for any other character. 125, 132
Windows An operating system by Microsoft. 87, 135, 167, 212, 213
Wireshark Program used to analyze packet data sent between network nodes. i, 1–4, 6, 7, 9, 13, 21, 26, 27, 30, 31, 41, 42, 44, 45, 47, 51, 60, 66, 70, 73, 74, 77, 79–81, 84, 86, 88, 89, 102, 104, 106–114, 116, 118, 122, 125, 130–133, 135, 137, 139, 140, 142, 147, 158–162, 193, 195–209, 211, 213, 217, 220, 221

x86 The instruction set architecture used by Intel processors. 45
x86-64 An extension of the x86 instruction set that is compatible with 64-bit processors. 45

APPENDIX C
TEST CASES

This section introduces the test cases we ran during the different sprints of the project. All of the test cases follow the template mentioned in the test plan on Table 6.1.

C.1 Sprint 1 Tests

In this sprint we ran the test cases shown in Table C.1-C.7. The test cases made for this sprint covers most of the extremely simple functionality needed in order to have a working utility.

C.2 Sprint 2 Tests

In this sprint we ran the test cases shown in Table C.8-C.14. These are tests to see that the basic functionalities of the utility are in place.

C.3 Sprint 3 Tests

In this sprint we ran the test cases shown in Table C.15-C.23. These are tests to see that the utility is working properly even with the most advanced features.

C.4 Sprint 4 Tests

In this sprint we ran the test cases shown in Table C.26-C.48. These are tests to see that the utility has been fixed in accordance to the extra requirements made by the customer in sprint 3 as well as the final touches that were needed in order to get the utility to work in a real environment. This section also includes the test cases made for the non-functional requirements.

Table C.1: Test case TID01

Header	Description
Description	Supporting parameters for C-header file
Tester	Lars Solvoll Tønder
Prerequisites	The utility must have been installed on the system and there needs to exist a header file associated with this test
Feature	Test that we are able to feed the solution with a C-header file and have it get dissected
Execution	1. Feed the utility with the name of the C-header file associated with this test through the command line 2. Read the output given by the program
Expected result	2. The user should be presented with some text expressing the success of creating a dissector

Table C.2: Test case TID02

Header	Description
Description	Supporting basic data types
Tester	Lars Solvoll Tønder
Prerequisites	The utility must have been installed on the system and there needs to exist a header file associated with this test
Feature	Test that our utility will be able to make a dissectors for C-header files including the following basic data types: int, float, char and boolean
Execution	1. Feed the utility with the name of a C-header file associated with this test which includes the aforementioned basic data types 2. Read the output given by the program
Expected result	2. The program should provide the user with some text expressing the success of creating a dissector

Table C.3: Test case TID03

Header	Description
Description	Displaying simple structs
Tester	Lars Solvoll Tønder
Prerequisites	The utility has already made a dissector for the struct inside the simple.h header file and wireshark must have been configured to use this dissector
Feature	Test that our utility is able to generate dissectors that displays simple structs
Execution	1. Open Wireshark 2. Load the pcap file which contains the captured traffic with a simple structs 3. Read the output
Expected result	1. Wireshark should start without presenting the user with any warnings or errors about the dissector used in this test 3. Wireshark should display the data inside the structs sent in the capture data as proper values instead of just binary or hex-data

Table C.4: Test case TID04

Header	Description
Description	Supporting #include
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to have been installed on the system and there needs to exist a header file associated with this test
Feature	Test that our utility supports C-header files with the #include directive
Execution	1. Write the name of the C-header file associated with this test which contains an #include directive 2. Read the output
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector

Table C.5: Test case TID05

Header	Description
Description	Supporting #define and #if
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to have been installed on the system and there needs to be a header file associated with this test
Feature	Test that our utility supports C-header files with #define and #if directives
Execution	1. Write the name of the C-header file associated with this test which contains a #define and #if directive 2. Read the output
Expected result	2. The program should provide the user with some text expressing the success of creating a dissector

Table C.6: Test case TID06

Header	Description
Description	Supporting configuration files
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to have been installed on the system and there needs to exist a header file with a struct that has a configuration file tied to it
Feature	Test that our utility supports reading data from a configuration file
Execution	1. Feed the utility with the name of the header file associated with this test and its config file 2. Read the output
Expected result	2. The program should provide the user with some text expressing the success of creating a dissector

Table C.7: Test case TID07

Header	Description
Description	Recognizing invalid values
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to have been installed on the system and there needs to exist a header file with a struct that has a configuration file tied to it. Wireshark must also have been installed on the system and be configured to being able to run dissectors written in Lua. and there needs to exist a pcap file associated with this test
Feature	Test that our utility recognizes invalid values for struct members specified in the config file.
Execution	1. Feed the utility with the name of a header and a config-file, where the config file sets restrictions on the members of the header file. 2. Copy the resulting dissector to the personal plugins folder of Wireshark. 3. Run Wireshark with the pcap file associated with this test 4. Inspect the different packets displayed in Wireshark
Expected result	3. Wireshark should display the struct members with invalid values marked as invalid

Table C.8: Test case TID08

Header	Description
Description	Supporting members of type enum
Tester	Lars Solvoll Tønder
Prerequisites	The utility must have been installed on the system and there needs to exist a header, config and pcap file associated with this test
Feature	Test that the utility is able to support C-header files with enums
Execution	1. Feed the utility with the name the C-header file associated with this test, which includes a struct using enums and its configuration file 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different packets and struct members that are displayed in Wireshark
Expected result	2. The program should provide the user with some text expressing the success of the Lua-file generation 5 The different packets should be displayed as having structs with enums, showing the value of the enum by its name and value

Table C.9: Test case TID09

Header	Description
Description	Supporting members of type array
Tester	Lars Solvoll Tønder
Prerequisites	The utility must have been installed on the system and there needs to exist a header, config and pcap file associated with this test
Feature	Test that the utility iis able to support C-header files with arrays
Execution	1. Feed the utility with the name of the C-header file associated with this test, which includes a struct using arrays 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different packets and struct members that are displayed in Wireshark
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector for the struct in the header file 5 The different packets should be displayed as having structs with arrays, showing the values of the cells in the array. Multidimensional arrays should also be displayed as being arrays with subtrees of other arrays. This should be indicated by the multidimensional arrays having a clickable "+" box to the left which when pressed shows the values of the cells of the inner arrays

Table C.10: Test case TID10

Header	Description
Description	Supporting the display of structs within structs
Tester	Erik Bergersen
Prerequisites	The utility must have been installed on the system and there needs to exist a header, config and pcap file associated with this test
Feature	Test that the utility is able to support C-header files with structs that have other struct members and display it properly in Wireshark
Execution	1. Feed the utility with the name of the C-header file associated with this test, which includes a struct with another struct member 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different struct members that are displayed in Wireshark
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector for the struct in the header file 5. The struct members within structs should be displayed as subtrees, indicated by having a clickable ”+” box to the left which when clicked, displays the contents of the given struct

Table C.11: Test case TID11

Header	Description
Description	Supporting enumerated named values
Tester	Erik Bergersen
Prerequisites	The utility must have been installed on the system and there needs to exist a header, config and pcap file associated with this test
Feature	Test that the utility iis able to support C-header files which uses integers as enums without declaring them as enums
Execution	1. Feed the utility with the name of the C-header file associated with this test, which includes enumerated named values 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different struct members that are displayed in Wireshark
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector for each of the structs inside the header file 5. The different packets should be displayed as containing enumerated named values expressed by their name and not value

Table C.12: Test case TID12

Header	Description
Description	Supporting bit strings
Tester	Erik Bergersen
Prerequisites	The utility must have been installed on the system and there needs to exist a header, config and pcap file associated with this test
Feature	Test that the utility iis able to support C-header files with bit strings
Execution	1. Feed the utility with the name of the C-header file associated with this test, with a struct using bit strings 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different struct members that are displayed in Wireshark
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector for each of the structs inside the header file 5. The packets should be displayed as containing bit strings. Bit strings should be displayed as subtrees indicated by a clickable "+" box to the left which shows the values of the different bits in the bit string when pressed

Table C.13: Test case TID13

Header	Description
Description	Supporting structs with various trailers
Tester	Erik Bergersen
Prerequisites	The utility must have been installed on the system and there needs to exist a header, config and pcap file associated with this test
Feature	Test that the utility is able to support C-header files with trailers
Execution	1. Feed the utility with the name of a C-header file with a struct that has a trailer 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different struct members that are displayed in Wireshark
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector for each of the structs inside the header file 5. The packets should be display the data inside the struct, and the data from the trailer should be displayed below

Table C.14: Test case TID14

Header	Description
Description	Sprint 2 functionality test
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to have been installed on the system as well as the attest testing framework
Feature	Checking that the utility is able to create a valid dissector from header files with all of the data types that were to be supported for sprint 2, including: trailers, bit strings, enumerated named values, structs within structs and arrays
Execution	1. Navigate to the folder where CSjark is installed through the terminal or command line 2. type ”python -m attest” into the terminal or command line and then press enter 3. Read the output
Expected result	3. The user should be presented with some text expressing the failure of 0 assertions

Table C.15: Test case TID15

Header	Description
Description	Support batch mode of C header and configuration files
Tester	Lars Solvoll Tønder
Prerequisites	The utility has have been installed on the system, there also needs to exist a header and configuration file for this test
Feature	Test that the utility is able to generate dissectors for all header-files in a folder, with configuration
Execution	1. Feed the utility the name of the two folders with header-files and configuration-files. 2. Read output from the utility
Expected result	2. The utility should provide the user with the amount of header files processed and the number of dissectors created. It should also provide the user with error messages for the header and configuration files it was unable to run

Table C.16: Test case TID16

Header	Description
Description	Supporting custom Lua configuration
Tester	Sondre Mannsverk
Prerequisites	The utility must have been installed on the system and there needs to exist a header, config and pcap file associated with this test
Feature	Test that the utility is able to support custom Lua files by configuration
Execution	1. Look at the custom Lua file(s) and understand what they do 2. Feed the utility with a C header-file, that needs to have specific parts of it dissected by custom Lua file(s), and a configuration file that specifies what Lua file(s) should be used. 3. Read the output 4. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 5. Open the pcap-file for this test with Wireshark 6. Look at the different struct members that are displayed in Wireshark
Expected result	3. The program should provide the user with some text expressing the success of the Lua file generation 6. Assert that the custom Lua file(s) have affected the display of the struct members in the expected way.

Table C.17: Test case TID17

Header	Description
Description	Supporting unions
Tester	Lars Solvoll Tønder
Prerequisites	The utility must have been installed on the system and there needs to exist a header, config and pcap file associated with this test
Feature	Test that the utility is able to support C-header files with unions
Execution	1. Feed the utility with the name of the header file associated with this test which contains a unions and its configuration file 2. Read output from the utility 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different struct members that are displayed in Wireshark
Expected result	2. The utility should provide the user some text expressing the success of generating a dissector for each of the structs inside the header file 5. The unions should be displayed as subtrees with the names of all the union members and their values, even the ones that haven't been explicitly set

Table C.18: Test case TID18

Header	Description
Description	Support filter and search in Wireshark
Tester	Erik Bergersen
Prerequisites	Wireshark has to be up and running with the trailer_test.lua
Feature	The dissector must support Wireshark's built-in filter and search on attributes
Execution	1. Open the pcap-file, which contains trailer_test packets. 2. Type the following in the filter-textbox: "luatructs.message == 66" and click apply 3. Look on the packet view.
Expected result	3. Only trailer_test packets will be visible in the packet view.

Table C.19: Test case TID19

Header	Description
Description	Support WIN32, _WIN64, _sparc etc
Tester	Lars Solvoll Tønder
Prerequisites	The utility must have been installed on the system and there needs to exist a header, config and pcap file associated with this test
Feature	Test that the utility is able to support C-header files with platform definition preprocessor macros (e.g. _WIN32, _WIN64, _sparc, etc.)
Execution	1. Feed the utility with C-header files with platform definition preprocessor macros (e.g. _WIN32, _WIN64, _sparc, etc.). 2. Read output from the utility 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different struct members that are displayed in Wireshark
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector for each of the structs inside the header file. 5. The different structs should be displayed as having the right struct members according to the C-header definitions. For example, the member defined surrounded by `#ifdef _WIN32 ... #endif` directives should be displayed correctly in WIN32 struct.

Table C.20: Test case TID20

Header	Description
Description	Supporting the use of flags specifying platforms to display member values correctly
Tester	Lars Solvoll Tønder
Prerequisites	A dissector for dissecting a header according to it's configuration, as well as a pcap-file for that header needs to be in place
Feature	Test that the utility is able to create dissectors that support the use of different flags. These flags should specify the originating platform of the packet and be used to display the member values properly.
Execution	1. Open a pcap-file which includes packets with different flags but the same structs and member values just with different sizes according to the originating platform specified in the flags 2. Look at how the different packets are displayed in Wireshark
Expected result	2. The different packages should have the exact same member values just with different flag-values

Table C.21: Test case TID21

Header	Description
Description	Supporting platforms with different endian
Tester	Erik Bergersen
Prerequisites	There have to exist a header-file, config-file and pcap-file to do the test.
Feature	Generate dissectors which support both little and big endian platforms
Execution	1. Feed the utility with the name of the C-header file associated with this test 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different members in Wireshark that have data types that can be affected by endianness.
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector for each of the structs inside the header file 5. The packets sent from platforms with different endian, should display the same values.

Table C.22: Test case TID22

Header	Description
Description	Supporting alignments
Tester	Erik Bergersen
Prerequisites	There have to exist a header-file, config-file and pcap-file to do the test.
Feature	Generate dissectors on platforms with different alignment
Execution	1. Feed the utility with the name of the C-header file associated with this test 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Look at the different packets from platforms that uses different alignments.
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector for each of the structs inside the header file 5. The packets sent from platforms that uses different alignments, should display equal values.

Table C.23: Test case TID23

Header	Description
Description	Handling Lua keywords
Tester	Erik Bergersen
Prerequisites	There have to exist a header-file, config-file and pcap-file to do the test.
Feature	The utility should support struct members, that have equal names as keywords in Lua
Execution	1. Feed the utility with the name of the C-header file associated with this test which contains a struct 2. Read the output 3. Move the resulting dissector into the plugins folder in the personal configuration folder of Wireshark 4. Open the pcap-file for this test with Wireshark 5. Check that the dissector work as it should.
Expected result	2. The program should provide the user with some text expressing the success of generating a dissector for each of the structs inside the header file 5. Wireshark should not display an error message for invalid Lua code.

Table C.24: Test case TID24

Header	Description
Description	Sprint 3 functionality test
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system as well as the attest testing framework.The file black_box.py must also be present in the test folder of CSjark
Feature	Checking that the utility is able to create a valid dissector from header files with all of the data types that were to be supported for sprint 3
Execution	1. Navigate to the test folder inside the folder where CSjark is installed through the terminal or command line 2. type ”python -m attest” into the terminal or command line and then press enter 3. Read the output
Expected result	3. The user should be presented with some text expressing the failure of 0 assertions

Table C.25: Test case TID25

Header	Description
Description	Sprint 2 functionality test
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to have been installed on the system as well as the attest testing framework. The file requirements.py must also be present in the test folder of CSjark
Feature	Checking that the utility is able to support all of the features required by the customer
Execution	1. Navigate to the folder where CSjark is installed through the terminal or command line 2. type ”python -m attest” into the terminal or command line and then press enter 3. Read the output
Expected result	3. The user should be presented with some text expressing the failure of 0 assertions

Table C.26: Test case TID26

Header	Description
Description	Including system-headers
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system and there has to exist a pcap-file which is associated with this test
Feature	Checking that the utility is able to support headers which use system headers
Execution	1. Feed the utility with the name of a C-header file that includes a system-headers and its configuration file 2. Read the output 3. Copy the resulting dissectors into the plugins folder of the personal configuration in Wireshark 4. Run Wireshark with the pcap-file associated with this test 5. Look at the resulting structs and members are displayed in Wireshark
Expected result	2. The user should be presented with some text expressing the success of generating dissectors 5. The structs and struct members defined in the system headers should be displayed as having a value and not just hex data

Table C.27: Test case TID27

Header	Description
Description	Ignoring #pragma directives
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system and there needs to exist a pcap-file which is associated with this test
Feature	Making sure that the utility is able to parse header files with the #pragma directive by just ignoring that directive
Execution	1. Feed the utility with the name of a C-header file that contains a #pragma directive and it's configuration file 2. Read the output
Expected result	2. The user should be presented with some text expressing the success of generating dissectors

Table C.28: Test case TID28

Header	Description
Description	Improve generated Lua output by removing platform prefix
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Making sure the utility only generates one dissector for the struct instead of several, but still keeping all of the functionality
Execution	1. Feed the utility with any C-header file and it's configuration 2. Read the output 3. Copy the resulting dissectors into the plugins folder of the personal configuration in Wireshark 4. Run Wireshark 5. Open the dissector tables menu entry from the Internals menu 6. Click the luastructs tree entry 7. Inspect its contents
Expected result	2. The user should be presented with some text expressing the success of generating dissectors 7. There should only be one tree entry for each dissector, not one for each platform as well

Table C.29: Test case TID29

Header	Description
Description	Recursive searching of subfolders
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system and there needs to exist a folder with folders that all have header files in them
Feature	Checking that it is possible for the utility to be fed a folder with header files that has subfolders which are in turn inspected
Execution	1. Feed the utility with the name of a folder of header files which again has subfolders with other header files 2. Read the output 3. Inspect the generated Lua files
Expected result	2. The user should be presented with some text expressing the success of generating dissectors for every header file in the subfolders 3. There should be 1 file for each struct contained in the different header files located in the header folder and its subfolders

Table C.30: Test case TID30

Header	Description
Description	Finding include dependencies which are not explicitly set
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Check that the utility is able to identify include dependencies which are not explicitly set and use that information to parse files over again correctly
Execution	1. Feed the utility with a C-header file that has include dependencies which are not explicitly set, and it's configration file 2. Read the output 3. Copy the resulting dissectors into the plugins folder of the personal configuration in Wireshark 4. Run Wireshark with the pcap-file associated with this test 5. Inspect the different packets, their structs and member values
Expected result	2. The user should be presented with some text expressing the success of generating dissectors 5. All structs and member values should be displayed as having proper values and not just hex data

Table C.31: Test case TID31

Header	Description
Description	Pointer support
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Checking that the utility is able to support the use of pointers in header files
Execution	1. Feed the utility with a C-header file that has pointers, and it's configuration file 2. Read the output 3. Copy the resulting dissectors into the plugins folder of the personal configuration in Wireshark 4. Run Wireshark with the pcap-file associated with this test 5. Inspect the different packets, their structs and member values
Expected result	2. The user should be presented with some text expressing the success of generating dissectors 5. All structs and member values should be displayed as having proper values and not just hex data

Table C.32: Test case TID32

Header	Description
Description	Enums in arrays
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Checking that the utility is able to support the use of enums inside of arrays
Execution	1. Feed the utility with a C-header file that has an array of enums, and it's configuration file 2. Read the output
Expected result	2. The user should be presented with some text expressing the success of generating dissectors

Table C.33: Test case TID33

Header	Description
Description	Supporting #define as a command line argument
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Checking that it is possible to feed the utility with a #define argument and have it force pre-processor to add a corresponding argument to the header files it is processing
Execution	1. Feed the utility with a C-header file, it's configuration file and a #define directive 2. Read the output 3. Copy the resulting dissectors into the plugins folder of the personal configuration in Wireshark 4. Run Wireshark with the pcap-file associated with this test 5. Inspect the different packets, their structs and member values
Expected result	2. The user should be presented with some text expressing the success of generating dissectors 5. All structs and their member values should be displayed as having proper values and not just hex data

Table C.34: Test case TID34

Header	Description
Description	Multiple message ID's for one dissector
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Checking that the utility supports having more than one message ID per dissector
Execution	1. Feed the utility with a C-header file and it's configuration file which includes more than one message ID 2. Read the output
Expected result	2. The user should be presented with some text expressing the success of generating dissectors

Table C.35: Test case TID35

Header	Description
Description	Allowing configuration for unknown structs
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Checking that the utility supports being able to configure the size of unknown structs
Execution	1. Feed the utility with a C-header file that has unparseable members and it's configuration file which includes the size of the struct itself 2. Read the output 3. Copy the resulting dissectors into the plugins folder of the personal configuration in Wireshark 4. Open Wireshark with the pcap-file associated with this test 5. Inspect the different packets, their structs and member values
Expected result	2. The user should be presented with some text expressing the success of generating dissectors 5. All of the members of each packets should have proper values except for the unparseable members which should only be displayed as containing hex data

Table C.36: Test case TID36

Header	Description
Description	Auto generating configuration files for structs that has no config file of their own
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Checking that the utility is able to create template configuration files for all structs that does not currently have a configuration
Execution	1. Feed the utility with a C-header file which has several structs without any configuration files 2. Open the configuration folder where CSjark is installed 3. Inspect the configuration files
Expected result	3. There should now be one configuration file present for each struct in the C-header file that has an empty template for filling in values for configuration

Table C.37: Test case TID37

Header	Description
Description	Only generating dissectors for structs with a valid ID
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Making sure that the utility only creates dissectors for files that have a valid ID specified in its configuration
Execution	1. Feed the utility with a C-header file and it's configuration file that has no valid ID 2. Read the output
Expected result	2. The user should be presented with a message saying that no dissectors were created and why

Table C.38: Test case TID38

Header	Description
Description	Guessing dissectors from packet size
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system and Wireshark has to have been loaded with a dissector for a struct of the same size as the one associated with this test
Feature	Making sure the utility is able to guess which dissector to use based on packet size if there are no dissectors specified for the packet
Execution	1.Start Wireshark with the pcap-file associated with this test 2. Inspect the different packets, their structs and member values
Expected result	2. All of the packets should contain a struct with different members and values instead of just raw hex data

Table C.39: Test case TID39

Header	Description
Description	Invalid header
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Making sure the utility crashes if it receives an invalid header
Execution	1. Feed the utility with an invalid header file 2. Read the output
Expected result	2. The utility should crash and give an error message explaining why it crashed

Table C.40: Test case TID40

Header	Description
Description	Invalid header during batch mode
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on the system
Feature	Making sure the utility skips headers it is unable to parse during batch processing
Execution	1. Feed the utility with a folder containing several invalid header files 2. Read the output
Expected result	2. The utility should skip the files it is unable to parse and present the user with a message saying why it skipped the files it was unable to parse

Table C.41: Test case TID41

Header	Description
Description Tester Prerequisites Feature	Ambiguous struct IDs Lars Solvoll Tønder The utility has to be installed on the system Making sure the utility crashes if there are several structs that all have been configured to using the same ID
Execution	1. Feed the utility with a pair of headers and configuration files that both have the same struct ID 2. Read the output
Expected result	2. The utility should present the user with an error message saying that there exists 2 structs which have been configured to using the same ID

Table C.42: Test case TID42

Header	Description
Description Tester Prerequisites Feature	Ambiguous platform IDs Lars Solvoll Tønder The utility has to be installed on the system Making sure the utility crashes if there are several platforms that all share the same ID
Execution	1.Add another platform platform.py which has the same platform ID as one of the previous ones 2. Feed the utility with any header file and its config 3. Read the output
Expected result	3. The utility should present the user with an error message saying that there exists two platforms with the same name

Table C.43: Test case TID43

Header	Description
Description Tester Prerequisites Feature	Running the utility on Windows Lars Solvoll Tønder The utility has to be installed on a Windows system Making sure the utility runs on the latest version of Windows as per 24.11.2011
Execution	1. Feed the utility with any C-header file and its config 2. Read the output
Expected result	2. The utility should present the user with some text expressing the success of creating a dissector for each of the structs inside the header file

Table C.44: Test case TID44

Header	Description
Description	Running the utility on Solaris
Tester	Lars Solvoll Tønder
Prerequisites	The utility has to be installed on a Solaris system
Feature	Making sure the utility runs on the latest version of Solaris as per 24.11.2011
Execution	1. Feed the utility with any C-header file and its config 2. Read the output
Expected result	2. The utility should present the user with some text expressing the success of creating a dissector for each of the structs inside the header file

Table C.45: Test case TID45

Header	Description
Description	Running the dissectors on Solaris
Tester	Lars Solvoll Tønder
Prerequisites	Wireshark has to have been installed with the dissectors associated with this test
Feature	Making sure the dissectors runs on the latest version of Solaris as per 24.11.2011
Execution	1. Run Wireshark with the pcap-file associated with this test 2. Inspect the different packets, their structs and member values
Expected result	2. The packets should be displayed as containing structs with members that all have proper values and not just hex data

Table C.46: Test case TID46

Header	Description
Description	Running the dissectors on Windows
Tester	Lars Solvoll Tønder
Prerequisites	Wireshark has to have been installed with the dissectors associated with this test
Feature	Making sure the dissectors runs on the latest version of Solaris as per 24.11.2011
Execution	1. Run Wireshark with the pcap-file associated with this test 2. Inspect the different packets, their structs and member values
Expected result	2. The packets should be displayed as containing structs with members that all have proper values and not just hex data

Table C.47: Test case TID47

Header	Description
Description	Quality of user documentation
Tester	Lars Solvoll Tønder
Prerequisites	The user documentation has to have been written
Execution	1. Read the user documentation for one hour 2. Try using the utility to generate a dissector for a header file
Expected result	2. The user should be able to generate a dissector configured to display the values from a given c-struct properly

Table C.48: Test case TID48

Header	Description
Description	Docstrings in code
Tester	Lars Solvoll Tønder
Prerequisites	None
Execution	1. Read through all of the class and method definitions in the code
Expected result	1. All classes and methods should have a docstring explaining what they do and how they are used

APPENDIX D ARCHITECTURAL DESCRIPTION

This chapter introduces the architectural documents pertaining to our solution. The team followed the definition of software architecture defined by Len Bass, Paul Clements and Rick Kazman: “The software architecture of a program or computing system is the structure of structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them.” [9, p.3]

The purpose of this document is to describe our architecture in a structured way so it can be used, not only by the team, but also as an aid for other stakeholders who are trying to understand the system.

D.1 Architectural Drivers

This section is dedicated to the discussion of the architectural drivers. The team has chosen Modifiability and Testability as the quality attributes for this utility.

The reason for using Modifiability is that the development team is not going to be the ones updating or maintaining the utility after completing the project. It is therefore important that the code is easy to understand, well documented and easy to modify. This makes it easier for our customer to later modify or extend the utility.

Testability is important since the utility will be used for debugging by the customer. Since it is not possible for the team to test the dissectors in a real environment, it is very important that we do extensive testing of the utility’s functionality. This is to ensure that the final product works correctly with both expected and unexpected input.

D.1.1 Testability Tactics

The goal of using testability tactics is making it easier to test the system after finishing an increment of the software development.

Specialize Access Routes/Interfaces

Using a specialized testing interface makes it possible to specify values for a component independently from its normal execution. This will in turn make it possible to test parts of an unfinished module as well as making it easier to get a clear overview over what data is flowing through individual parts of the system. This is important for this project as the utility must be able to run in a different environment than what the developers have access to. The testers must therefore be able to create input for each individual component of the system in order to ensure that it will work correctly with all kinds of input.

D.1.2 Modifiability Tactics

The goal of using modifiability tactics is to make it easier to extend and modify the software during the development and after the completion of a working product.

Anticipate Expected Changes

By trying to anticipate expected changes, it is possible to make it easier for modules to be extended with new functionality later. It also makes it easier for the developers to anticipate the different ranges of input the modules are able to process. This is important for this project as it is being developed incrementally, with new functionality and code added every sprint.

Limit Possible Options

By limiting the range of possible modifications to the system it becomes easier to generalize the structure of different modules. This will in turn make it easier to constrict the wide ranging effect of new modifications to the system, giving the developers a clearer view over what a given change will actually do to the system. This is important for this project as the developers have a limited time window to implement the utility, making it important to be able to limit the scope of the utility while still being able to add the functionality required by the customer.

Generalizing Modules

Generalizing the modules of a system makes it possible to reuse older modules when doing modifications to the system. The more general a module, the more likely it is that a needed change to the system can be implemented by just adjusting the input to the system, rather than having to modify existing or creating new modules.

Restrict Communication Paths

By restricting the number of modules that are able to collect data from a specific module, the less dependent the entire system becomes of that specific module. This makes it easier to swap out existing modules with new ones without having to make many widespread changes to the entire system. This is important for this project as the source code could change drastically after discovering new requirements in later sprints. By having a loose coupling we will minimize the amount of code that has to be rewritten after every sprint.

Using Configuration Files

By using configuration files, it is possible to change the behaviour of the system without having to do any changes to its code. It is very important that this system uses configuration files as this was a requirement from the customer, as well as making it more flexible for the end user.

D.1.3 Business Requirements

The following business requirements encompass the most important needs of the customer.

- The utility must be delivered on time as it is not possible for the developers to continue the development after the deadline
- The utility should be able to create dissectors for the C-structs in header files used by Thales
- The utility should be able to create dissectors that run on all of the platforms used by Thales and their customers
- Developers at Thales should be able to use Wireshark with the generated dissectors to display the values in C-structs passed through the system.

D.1.4 Design Goals

To help guide the design and the implementation we tried to follow these goals and guidelines:

- Smart data structures and dumb code works better than the other way around [15]!

- Clear and clean separation of the front-end and the back-end so in the future other parsers can be used to generate dissectors.
- Try to be pythonic, follow PEP8 [1] and PEP20[2].
- Now is better than never. Don't be afraid to write stupid or ugly code, we can always fix it later.
- The first version is never perfect, so don't wait until its perfect before you commit. Commit often!

D.2 Architectural Patterns

This section presents the different architectural patterns used in the utility

D.2.1 Pipe and Filter

The pipe and filter architectural pattern consists of a stream of data that in turn is processed sequentially by several filters. This is done in such a fashion that the output of one filter becomes the input of the other. It is a very flexible, yet robust way of processing data, with support for adding more filters if needed for future applications and processes. As the utility will only work on one piece of data that gradually changes, and is then converted into Lua-code at the end, this seemed like a good and structured way of processing data early on, while still being able to add new functionality further down the line.

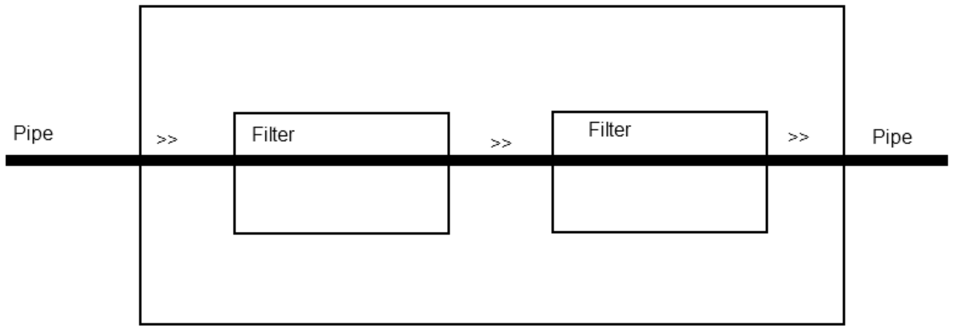


Figure D.1: Pipe and Filter Pattern

[1]Style Guide for Python Code: `http://www.python.org/dev/peps/pep-0008/`
[2]The Zen of Python `http://www.python.org/dev/peps/pep-0020/`

D.3 Architectural Views

This section describes three different views: Logical view, process view and deployment view.

D.3.1 Logical View

Figure D.2 shows the logical view of the system. The command line takes the arguments for header file and configuration file as a string. The arguments are parsed in the command line parser. Header file is sent to ”C preprocessor & C parser”, the C header file is loaded and parsed by the C parser, which generates a parsing tree. The command line also calls Configuration, which load the configuration file. The configuration will parse the configuration file and create configuration rules. The Lua script generator will generate a Lua script from the parsing tree and the config rules.

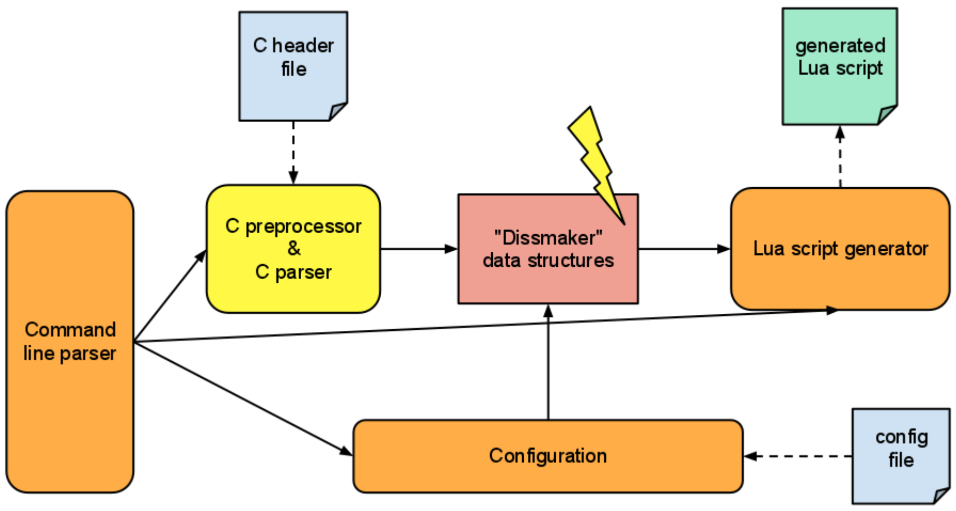


Figure D.2: Overall Architecture

D.3.2 Process View

Figure D.3 shows the process view for our utility. CSjark takes header and config files as input and then uses the config and cparser to parse the files. CSjark then uses the cparser to find the structs in the header file and then creates dissectors for them. These dissectors are then written to a file and CSjark then reports to the user by sending a message to the command line.

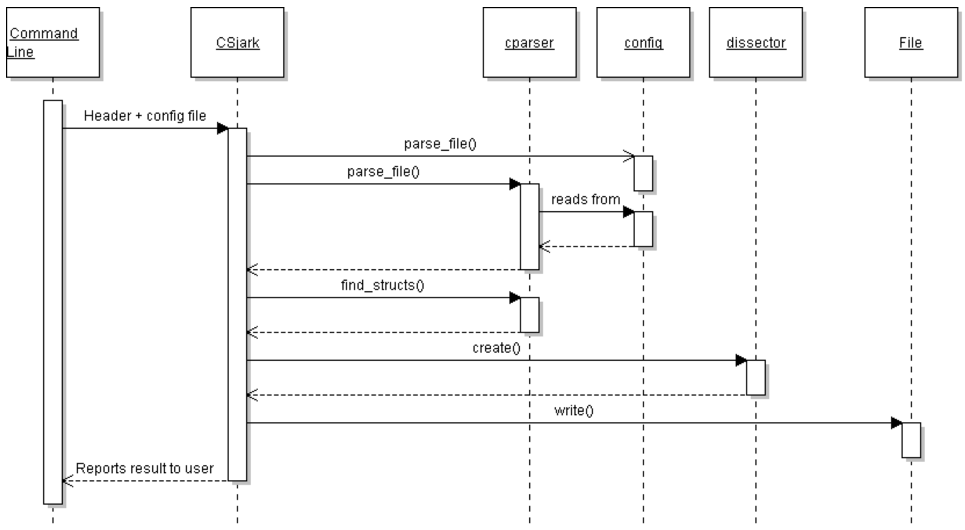


Figure D.3: Data Flow During Regular Execution

D.3.3 Deployment View

Figure D.4 shows the deployment diagram for this project. CSjark takes header-files and config-files as input, and generates Lua dissectors. All these dissectors are added as plugins to Wireshark, extending the functionality. Wireshark will capture the data packet when Process A send data to Process B, the Lua dissectors is used to display these data packets correctly.

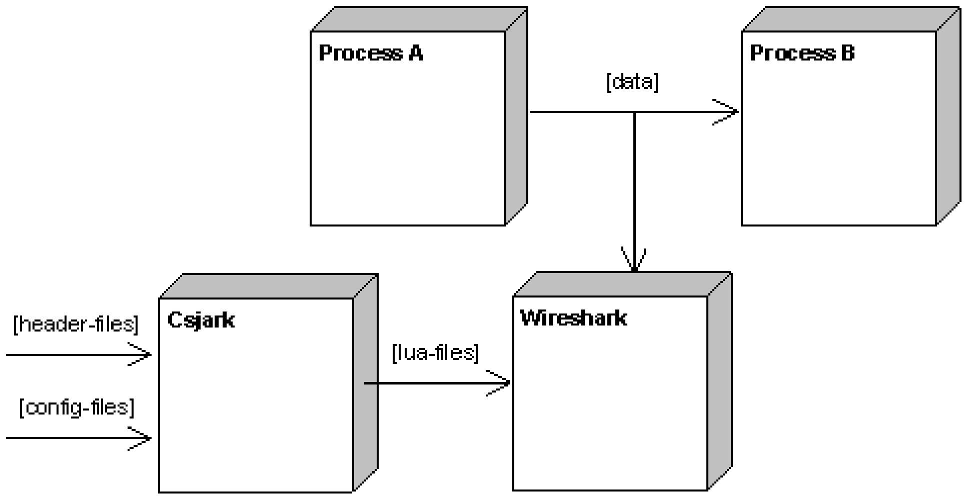


Figure D.4: Deployment View

D.4 Architectural Rationale

The team decided to use the pipe and filter pattern as the architects felt that it was the only architectural pattern that would benefit the utility without having to make it needlessly complex. The utility was supposed to take header files as input and then process the data from them several times, until the end result was a list of structs and members that could be used to make dissectors for Wireshark. This seemed like an excellent application of the pipe and filter pattern, as it would then be easy to add new filters to the header file for future increments of the development cycle, without having to rewrite what had already been implemented in previous sprints.

For the views the team decided to use a logical view, process view and deployment view. These views were chosen because the architects of the utility felt that these views alone could represent the system sufficiently without creating too much overhead for the readers of the document. The logical view supplies the reader with a more in depth view of what the system is comprised of, which is useful for developers who need to figure out the workings of the system. The process view also seemed important for the developers and the testers of the utility, as it provides the reader with a more proper overview of the data flow in the system. This makes it much easier to see which modules are run when, and to see which external calls dictate the modules' behaviour. Lastly, a deployment view was chosen to make it more clear for the reader of the document what the utility really produces as output and what other external applications it has to cooperate with.

D.5 Changelog

An architectural document should be thought of as a living document and treated as such. We will therefore introduce the changes done to the architecture during each sprint in the following subsections.

D.5.1 Sprint 1

As we in sprint one was mostly concerned with getting the basic functionality in place, there were no changes done to the architecture.

D.5.2 Sprint 2

During the second sprint it became apparent that we needed to structure our code in a better and more organized way. If not, it would become very hard to add new functionality to the utility in a straight forward and logical way. We therefore decided to introduce the layered architectural pattern (D.5.3), as it is often used to resolve issues just as the ones we faced in sprint 2. We also decided to add metrics for code coverage as a tactic for testability, in order to improve our unit tests.

Code Coverage

By using a framework to see which parts and how much of the code is actually being run during the unit tests, it becomes easier to improve the quality of the unit tests. It could also be used as a checklist to see if the ones creating the unit tests have implemented some functionality that is currently not being tested.

D.5.3 Layered Architectural Pattern

The layered architectural pattern involves grouping several different classes that all share the same dependencies. This grouping of classes is called a layer, and the layers are structured so that the classes inside each layer only depend on the classes of their own layer level, or inside an underlaying one. Structuring the code in this way helps delegating responsibilities to different parts of the system in a logical way, making the code easier to understand and easier to navigate through.

Figure D.5 shows how the layered architectural pattern is used in the utility.

D.5.4 Sprint 3

There were no additions or subtractions done to the architecture of the utility during sprint 3.

D.5.5 Sprint 4

There were no additions or subtractions done to the architecture of the utility during sprint 4.

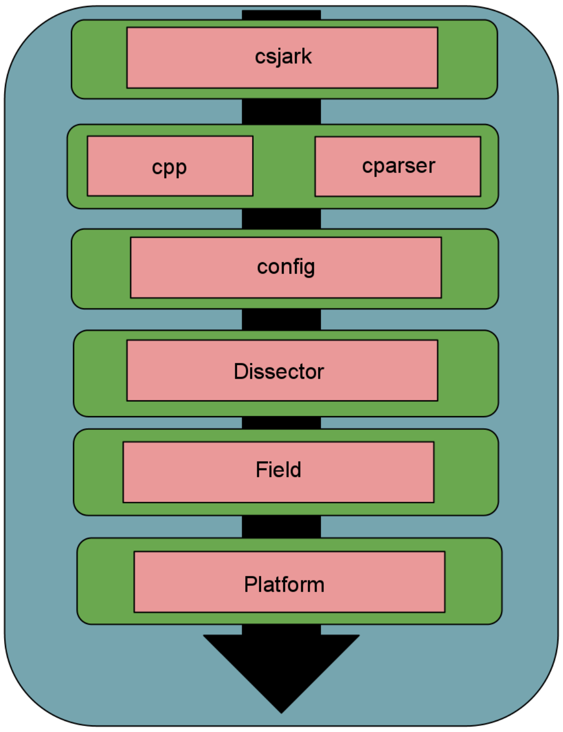


Figure D.5: Layered Architectural Pattern in the Utility

APPENDIX E INITIAL LIST OF REQUIREMENTS

The customer provided an initial requirements specification for the utility at the start of the project, which can be seen in section E.1.

We made some initial changes to the format, created some non-functional requirements and added priority and complexity to each requirement. This resulted in the initial function requirements listed in Table E.1 and initial non-functional requirements listed in Table E.2. These changes were approved by the customer before the start of the first sprint.

E.1 Requirements from Customer

The customer provided the following list of requirements, for the utility we should create, at the start of the project.

F01 The utility shall be able to read basic C language struct definitions, and generate a Wireshark dissector for the binary representation of the structs.

F02 The utility shall support structs with any of the basic data types (e.g. int, boolean, float, char) and structs.

F03 The utility shall be able to follow #include <...> statements. This allows parsing structs that depend on structs or defines from other header files.

F04 Each struct may be connected to one or more references (integer value). For instance, a member parameter 'type' can have names for a set of values.

F05 The dissector shall be able to recognize invalid values for a struct member. Allowed ranges should be specified by configuration. An example is an integer that indictates a percentage between 0 and 100.

F06 A struct may have a header and/or trailer (other registered protocol). This must be configurable.

F07 The dissector shall be able to display each struct member. Structs within structs shall also be dissected and displayed.

F08 It shall be possible to configure special handling of specific data types. E.g. a 'time_t' may be interpreted to contain a unixtime value, and be displayed as a date.

F09 An integer member may indicate that a variable number of other structs (array of structs) are following the current struct.

F10 Integers may be an enumerated named value or a bit string.

F11 The dissectors produced shall be able to handle binary input from at least Windows 32bit and 64bit, Solaris 64bit and Sparc. Example: BOOL is 1 byte on Solaris and 4 bytes on Win32. Endian and alignment also differs between the architectures.

E.2 Initial Requirements

Initial function requirements are listed in Table E.1 and initial non-functional requirements are listed in Table E.2.

Table E.1: Initial Functional Requirements

ID	Description	Pri.	Cmp.
FR1	The utility must be able to read basic C language struct definitions from C header files.	H	
FR1-A	The utility must support the following basic data types: int, float, char and boolean.	H	L
FR1-B	The utility must support members of type enums.	H	L
FR1-C	The utility must support members of type structs.	H	M
FR1-D	The utility must support members of type unions.	M	M
FR1-E	The utility must support member of type array.	H	M
FR2	The utility must be able to generate lua-script for Wireshark dissectors for the binary representation of C struct.	H	
FR2-A	The dissector shall be able to display simple structs.	H	L
FR2-B	The dissector shall be able to support structs within structs.	M	M
FR2-C	The dissector must support Wireshark's built-in filter and search on attributes.	H	L
FR3	The utility must support C preprocessor directives and macros.	H	
FR3-A	The utility shall support #include.	H	L
FR3-B	The utility shall support #define and #if.	H	L
FR3-C	The utility shall support , `_WIN32`, `_WIN64`, `__sparc__`, `__sparc` and `sun`.	M	H
FR4	The utility must support user configuration.	M	
FR4-A	The dissector shall be able to recognize invalid values for a struct member. Allowed ranges should be specified by configuration.	L	L
FR4-B	Configuration must support integer members which represent enumerated named value or a bit string.	M	L
FR4-C	Configuration must support custom handling of specific data types. E.g. a 'time_t' may be interpreted to contain a unixtime value, and be displayed as a date.	L	M
FR5	A struct may have a header and/or trailer (other registered protocol). The configuration must support the use of integer members to indicate the number of other structs that will follow in the trailer	L	H
FR6	The dissectors must be able to handle binary input which size and endian depends on originating platform.	M	
FR6-A	Flags must be specified for each platform.	M	M
FR6-B	Flags within message headers should signal the platform.	M	H
FR7	The utility shall support parameters from command line.	H	
FR7-A	Command line shall support parameters for C header file.	H	L
FR7-B	Command line shall support for configuration file.	H	L
FR7-C	Command line shall support batch mode of C header and configuration file.	L	M
FR7-D	When running batch mode, dissectors that already are generated, shall not be regenerated, if the source are not modified since last run.	L	M

Table E.2: Initial Non-Functional Requirements

ID	Description	Pri.	Cmp.
NR1	The utility shall be able to run on latest Windows and Solaris operating system.	M	L
NR2	The dissector shall be able to run on Windows x86, Windows x86-64, Solaris x86, Solaris x86-64 and Solaris SPARC.	M	M
NR3	The utilities user interface shall be command line.	H	L
NR4	The configuration shall have sufficient documentation to allow a person with no previous knowledge of the system to be able to use it to generate LUA-scripts after five hours of reading.	M	M
NR5	The configuration should have sufficient documentation to allow a person, already proficient with the system, to understand the code well enough to be able to extend it's functionality after four hours of reading.	M	M
NR6	The utility code should follow standard python coding convention as specified by PEP8, and try to follow python style guidelines defined by PEP20.	H	L
NR7	The utilities code should be documented by python docstrings which should explain the use of the code. Python modules, classes, functions and methods should have docstrings.	M	L

APPENDIX F
USER AND DEVELOPER MANUAL

CSjark Documentation

Release 0.4.2

Erik Bergersen **Jaroslav Fibichr**
Sondre Johan Mannsverk **Terje Snarby**
Even Wiik Thomassen **Lars Solvoll Tønder**
Sigurd Wien

November 21, 2011

CONTENTS

CSjark is a tool for generating Lua dissectors from C struct definitions to use with Wireshark. Wireshark is a leading tool for capturing and analysing network traffic. CSjark provides a way to display the contents of the C struct data coming from an IPC packet in human-readable form in Wireshark.

You can find more about CSjark functionality in the *Introduction* section.

CHAPTER

ONE

USER DOCUMENTATION

1.1 Introduction

This part is a technical introduction to CSjark. It gives a concise explanation of the most important terms used in the documentation. The first section briefly explains Wireshark, dissectors and how dissectors are used in Wireshark. The connection between Wireshark and the Lua structs protocol is also explained. The second section describes how the Lua code works and how it is generated by our utility.

1.1.1 Wireshark and dissectors

This section gives a brief introduction to Wireshark and dissectors. The rst part describes what Wireshark is and what it can be used for. The second part explains exactly what a dissector is, and how a dissector can be used to extend Wireshark.

Wireshark

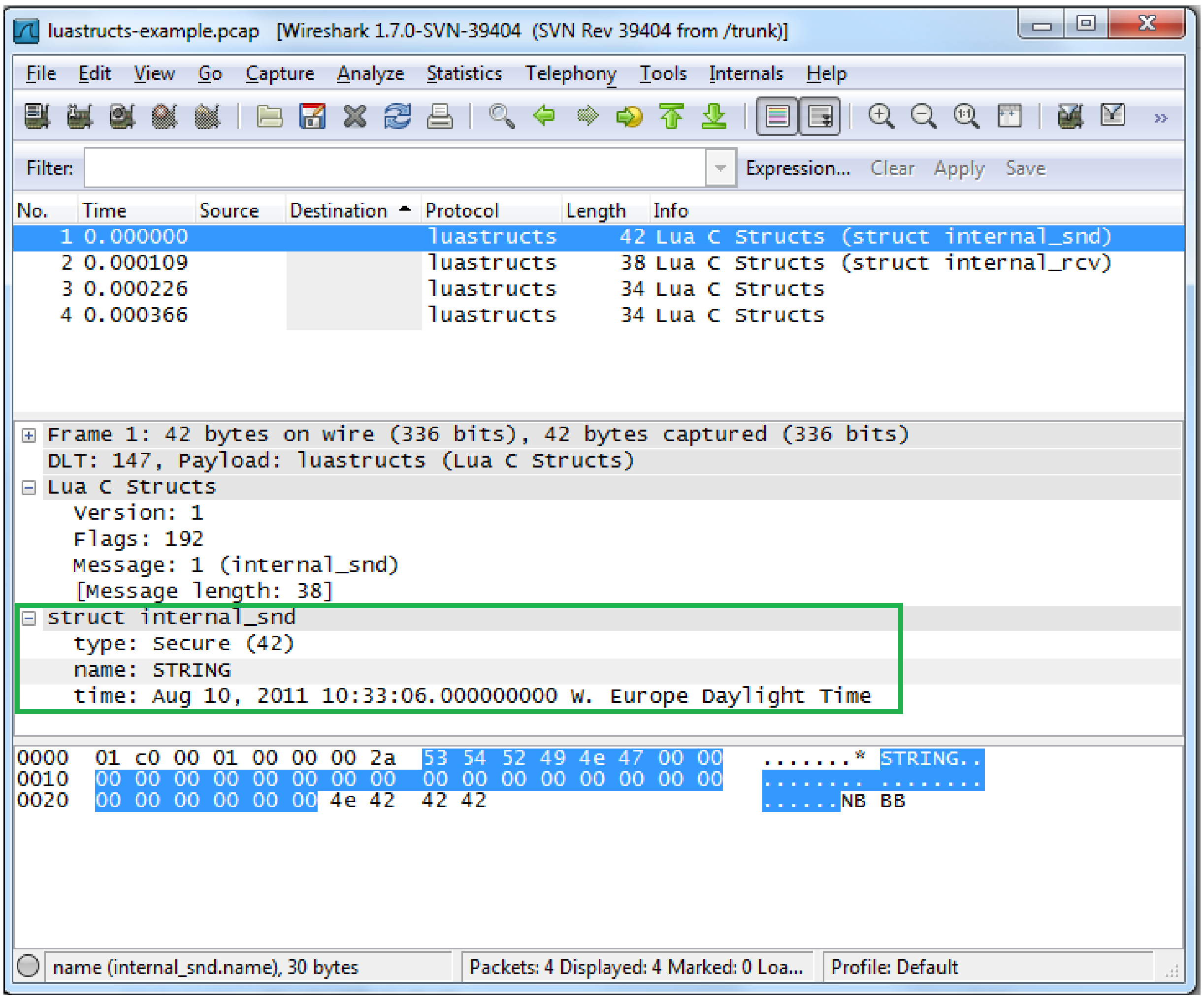
Wireshark is a program used to analyze network traffic. A common usage scenario is when a person wants to troubleshoot network problems or look at the internal workings of a network protocol. An important feature of Wireshark is the ability to capture and display a live stream of packets sent through the network. A user could, for example, see exactly what happens when he opens up a website. Wireshark will then display all the messages sent between his computer and the web server. It is also possible to filter and search on given packet attributes, which facilitates the debugging process.

In Figure 1, you can see a view of Wireshark. This specific example shows a capture file with four messages, or packets, sent between internal 2 processes, in other words it is a view of messages sent by inter-process communication. Each of the packets contain one C struct. To be able to display the contents of the C struct, Wireshark has to be extended. This can be accomplished by writing a dissector for the C struct.

Dissector

In short, a dissector is a piece of code, run on a blob of data, which can dissect the data and display it in a readable format in Wireshark, instead of the binary representation. The Figure 1 displays four packets, with packet number 1 highlighted. The content of the packet is a C struct with two members, name and time, and it is displayed inside the green box. The C code for the struct is shown below.

```
/*
* Sample header file for testing Lua C structs script
* Copyright 2011 , Stig Bjorlykke <stig@bjorlykke.org>
*/
```


luastructs-example.pcap [Wireshark 1.7.0-SVN-39404 (SVN Rev 39404 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time      Source  Destination  Protocol    Length Info
  1 0.000000                        luastructs      42 Lua C Structs (struct internal_snd)
  2 0.000109                        luastructs      38 Lua C Structs (struct internal_rcv)
  3 0.000226                        luastructs      34 Lua C Structs
  4 0.000366                        luastructs      34 Lua C Structs

⊞ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits)
  DLT: 147, Payload: luastructs (Lua C Structs)
⊟ Lua C Structs
    Version: 1
    Flags: 192
    Message: 1 (internal_snd)
    [Message length: 38]
⊟ struct internal_snd
    type: Secure (42)
    name: STRING
    time: Aug 10, 2011 10:33:06.000000000 W. Europe Daylight Time

0000  01 c0 00 01 00 00 00 2a  53 54 52 49 4e 47 00 00   .......* STRING..
0010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0020  00 00 00 00 00 00 4e 42  42 42                     ......NB BB

name (internal_snd.name), 30 bytes | Packets: 4 Displayed: 4 Marked: 0 Loa... | Profile: Default
```

Figure 1.1: *Figure 1:* Wireshark

```
#include <time.h>

#define STRING_LEN 30

struct internal_snd {
    int type;
    char name [STRING_LEN];
    time_t time;
};
```

The dissector takes the C struct, decodes its binary representation and makes it readable by humans. Without a dissector, Wireshark would just display the struct and struct members as a binary blob.

All the packets containing C structs belong to the protocol called `luastructs`. When opening a capture file in Wireshark, this protocol maps the id of the messages to the correct dissector, and calls them.

1.1.2 From struct definition to Lua dissector

This section explains what happens under the hood of a Lua dissector.

Lua dissectors

The code below shows what the code for the Lua dissector, displayed in packet 1 in Figure 1, looks like. The `Proto` variable defines a new protocol. In this example, a dissector for the internal_snd struct, called internal_snd, is created. The different fields of the struct are created as instances of `ProtoField`, and put in `Protocol.fields`. For example, the `name` variable is a string in C, and as such it is created as a ProtoField.string with the name `name`.

The protocol dissector method is the method that does the actual dissecting. A subtree for the dissector is created, and the description of the dissector is appended to the information column. All the ProtoFields are added to the subtree. Here you can see that the `type`, `name` and `time` fields are added to the subtree for the `internal_snd` dissector. The buffer size allocated to the fields is the size of the members in C.

```
--
--  A sample dissector for testing Lua C structs scripts
--  Copyright 2011, Stig Bjorlykke <stig@bjorlykke.org>
--

local PROTOCOL = Proto ("internal_snd", "struct internal_snd")
local luastructs_dt = DissectorTable.get ("luastructs.message")

local types = { [0] = "None", [1] = "Regular", [42] = "Secure" }

local f = PROTOCOL.fields
f.type = ProtoField.uint32 ("internal_snd.type", "type", nil, types)
f.time = ProtoField.absolute_time ("internal_snd.time", "time")
f.name = ProtoField.string ("internal_snd.name", "name")

function PROTOCOL.dissector (buffer, pinfo, tree)
   local subtree = tree:add (PROTOCOL, buffer())
   pinfo.cols.info:append (" (" .. PROTOCOL.description .. ")")

   subtree:add (f.type, buffer(0,4))
   subtree:add (f.name, buffer(4,30))
   subtree:add (f.time, buffer(34,4))
end
```

```
luastructs_dt:add (1, PROTOCOL)
```

Note: Lua dissectors are usually files with extension `.lua`.

For further information on the Lua integration in Wireshark, please visit: Lua Support in Wireshark.

More information programming in Lua in general can be found in Lua reference manual.

1.2 Features

This is a list of CSjark features:

- C header files
- Batch mode
- Searching and filtering in Wireshark
- ...

1.2.1 Currently supported platforms

- Windows 32-bit
- Windows 64-bit
- Solaris 32-bit
- Solaris 64-bit
- Solaris SPARC 64-bit
- MacOS
- Linux 32 bit

(additional platforms can be added by configuration)

1.3 Installing CSjark

1.3.1 Dependencies

CSjark is written in Python 3.2, and therefore needs Python 3.2 (or later) to run. Latest implementation of Python can be downloaded from Python website. For installing please follow the instruction found there.

There are 4 third party dependencies to get CSjark working:

1. **PLY (Python Lex-Yacc)** PLY is an implementation of lex and yacc parsing tools for Python. It is required by pycparser. Instructions and further information can be found on the page linked above.

Required version	3.4
Download location	http://www.dabeaz.com/ply/

2. **pycpaser** Pycparser is a C parser (and AST generator) implemented in Python. Due to the continuous development, CSjark requires the latest development version (not the release version).

Required version	latest development version from pycparser repository
Download location	pycparser repository: http://code.google.com/p/pycparser/source/checkout

3. **C preprocessor** CSjark requires a C-preprocessor. The way how to get one depends on operating system used by the user:

Windows	Bundled with CSjark.
OS X, Linux, Solaris	Needs to be installed separately. For example, as a part of GCC

4. **pyYAML**

   pyYAML is a YAML parser and emitter for the Python programming language. YAML is a standard used to specify configurations to CSjark. The website includes both a way to download the software and also instructions of how to install it.

Required version	3.10
Download location	http://pyyaml.org/wiki/PyYAML

1.3.2 Wireshark

Wireshark is an open source protocol analyzer which can use the Lua dissectors generated by CSjark. To get the proper integration of Lua dissectors, the latest development version of Wireshark is required.

Required version	1.7 dev (build 39446 or newer)
Download location	http://www.wireshark.org/download/automated/, on the page, browse for the required platform version

1.3.3 CSjark

CSjark can be obtained at git CSjark repository: https://github.com/eventh/kpro9/. CSjark itself requires no installation. After the steps described in the dependencies section is completed. It can be ran by opening a terminal, navigating to the directory containing `cshark.py` and invoking as described in section *Using CSjark*.

1.4 Using CSjark

CSjark can be invoked by running the `csjark.py` script. The arguments must be specified according to:

```
csjark.py [-h] [-v] [-d] [-s] [-f [header [header ...]]]
          [-c [config [config ...]]] [-x [path [path ...]]]
          [-o [output]] [-p] [-n] [-C [path]] [-i [header [header ...]]]
          [-I [directory [directory ...]]]
          [-D [name=definition [name=definition ...]]]
          [-U [name [name ...]]] [-A [argument [argument ...]]]
          [header] [config]
```

Example usage:

```
python csjark.py -v -o dissectors headerfile.h configfile.yml
```

Batch mode

One of the most important features of CSjark is processing multiple C header files in one run. That can be easily achieved by specifying a directory instead of a single file as command line argument (see above):

```
python csjark.py headers configs
```

In batch mode, CSjark only generates dissectors for structs that have a configuration file with an ID (see section *Dissector message ID* for information how to specify dissector message ID), and for structs that depend on other structs. This speeds up the generation of dissectors, since it only generates dissectors that Wireshark can use.

Required arguments

`header` a C header file to parse or directory which includes header files

`config` a configuration file to parse or directory which includes configuration files

Both `header` and `config` can be:

- file - CSjark processes only the specified file
- directory - CSjark recursively searches the directory and processes all the appropriate files found

Optional argument list

`-h`, `--help`	Show a help message and exit.
`-v`, `--verbose`	Print detailed information.
`-d`, `--debug`	Print debugging information.
`-s`, `--strict`	Only generate dissectors for known structs.
`-f`, `--file`	Additional locations of header files.
`-c`, `--config`	Additional locations of configuration files.
`-x`, `--exclude`	File or folders to exclude from parsing.
`-o`, `--output`	Location for storing generated dissectors.
`-p`, `--placeholders`	Automatically generates config files with placeholders.
`-n`, `--nocpp`	Disables the C pre-processor.
`-C`, `--Cpppath`	Specifies the path to C preprocessor.
`-i`, `--include`	Process file as Cpp *#include "file"* directive
`-I`, `--Includes`	Additional directories to be searched for Cpp includes.
`-D`, `--Define`	Predefine name as a Cpp macro
`-U`, `--Undefine`	Cancel any previous Cpp definition of name
`-A`, `--Additional`	Any additional C preprocessor arguments

Optional argument details

`-h, -help`
Show a help message and exit.

`-v, -verbose`
Print detailed information.

`-d, -debug`
Print debugging information.

`-s, -strict`
Only generate dissectors for known structs. As known structs we consider only structs for which exists valid configuration file with ID defined. Also, CSjark generates dissectors for structs that depend on known structs.

`-f` `[path [path ...]]`, **`-file`** `[path [path ...]]`
Specifies that CSjark looks for struct definitions in the *path*. There can be more than one path specified, separated by whitespace. As *path* there can be file and directory. In case of a directory, CSjark searches for header files recursively to maximum possible depth.

All header files found are added to the files specified by the required `header` argument.

Example:

```
csjark.py -f hdr/file1.h dir1 file2.h
```

-c `[path [path ...]]`, **-config** `[path [path ...]]`
Specifies that CSjark looks for configuration definition files in the *path*. There can be more than one path specified, separated by whitespace. As *path* there can be file and directory. In case of a directory, CSjark searches for configuration files recursively to maximum possible depth.

All configuration files found are added to the files specified by the required `config` argument.

Example:

```
csjark.py -c etc/conf1.yml dir1 conf2.yml
```

-x `[path [path ...]]`, **-exclude** `[path [path ...]]`
File or folders to exclude from parsing.

When using the option, CSjark will not search for header files in the *path*. There can be more than one path specified, separated by whitespace. As *path* there can be file and directory. In case of a directory, CSjark will skip header files also in its subdirectories.

-o `[path]`, **-output** `[path]`
Sets location for storing generated dissectors.

If *path* is a directory, CSjark saves the output dissectors into this directory, otherwise CSjark saves the output dissectors into one specified file named *path*. If file with this name already exists, it is rewritten without warning.

Default: CSjark root directory (when the *csjark.py* file is located)

-p, -placeholders
Automatically generates configuration files with placeholders for structs without configuration.

More in section *Configuration file format and structure*.

-n, -nocpp
Disables the C pre-processor.

-C `[path]`, **-Cpppath** `[path]`
Specifies the path to the external C preprocessor.

Default:

- Windows, the path is *../utils/cpp.exe* (uses cpp bundled with CSjark).

-i `[header [header ...]]`, **-include** `[header [header ...]]`
Process *header* as if *#include "header"* appeared as the first line of the input header files

-I `[directory [directory ...]]`, **-Includes** `[directory [directory ...]]`
Additional directories to be searched for Cpp includes.

Add the directory *directory* to the list of directories to be searched for header files. These directories are added as an argument to the preprocessor. The preprocessor can search there for those files, which are given in an *#include* directive of the C header input.

-D `[name=definition [name=definition ...]]`, **-Define** `[name=definition [ name=definition ...]]`
Predefine *name* as a Cpp macro, with definition *definition*.

-U `[name [name ...]]`, **-Undefine** `[name [name ...]]`
Cancel any previous Cpp definition of name, either built in or provided with a `-D` option.

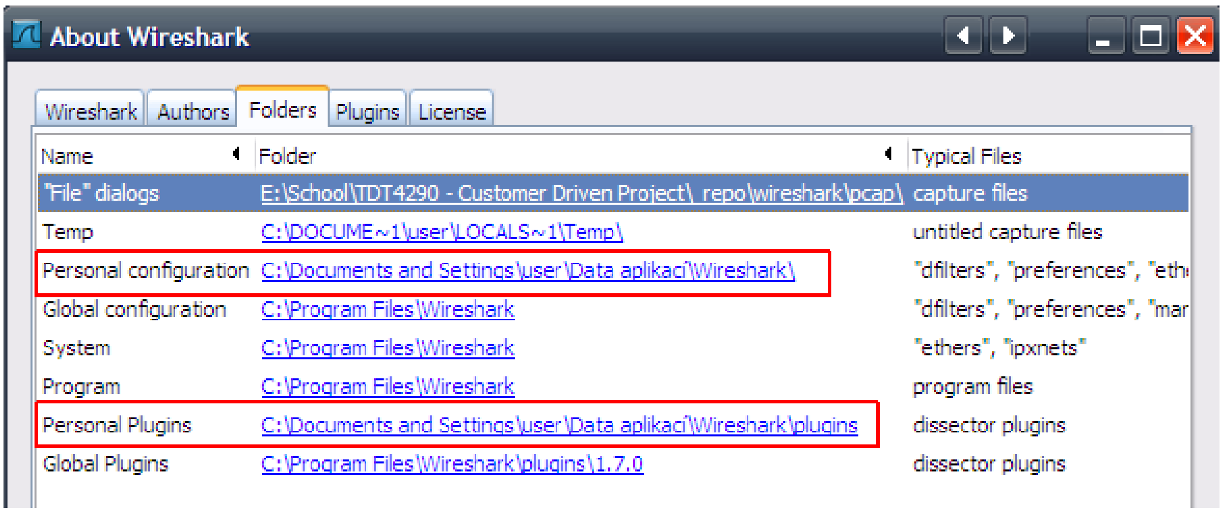
-A `[argument [argument ...]]`, **-Additional** `[argument [argument ...]]`
Any additional C preprocessor arguments.

Adds any other arguments (additional to *-D*, *-U* and *-I*) to the preprocessor.

1.5 Using the generated Lua files in Wireshark

These are the steps needed to use a Lua dissector generated by CSjark with Wireshark.

1. Get the latest version of Wireshark as described in the installation section *Wireshark*.
2. Locate the Personal configuration and the Personal Plugins directories. To do this, start Wireshark and click on `Help` in the menubar and then on `About Wireshark`. This should bring up the About Wireshark dialog. From there, navigate to the `Folders` tab. Locate folders `Personal configuration` and `Personal Plugins` and note their paths (see below).



Name	Folder	Typical Files
"File" dialogs	E:\School\TDT4290 - Customer Driven Project_repo\wireshark\pcap\	capture files
Temp	C:\DOCUME~1\user\LOCALS~1\Temp\	untitled capture files
Personal configuration	C:\Documents and Settings\user\Data aplikací\Wireshark\	"dfilters", "preferences", "eth
Global configuration	C:\Program Files\Wireshark	"dfilters", "preferences", "mar
System	C:\Program Files\Wireshark	"ethers", "ipxnets"
Program	C:\Program Files\Wireshark	program files
Personal Plugins	C:\Documents and Settings\user\Data aplikací\Wireshark\plugins	dissector plugins
Global Plugins	C:\Program Files\Wireshark\plugins\1.7.0	dissector plugins

   - on Linux/Unix system it may be `~/.wireshark/` and `~/.wireshark/plugins/`
   - on Windows it may be `C:\Users\*YourUserName*\AppData\Roaming\Wireshark\` and `C:\Users\*YourUserName*\AppData\Roaming\Wireshark\plugins\`

   If the folders does not exist, create them.

3. Copy CSjark generated file `luastructs.lua` into the `Personal configuration` folder located in step 2.

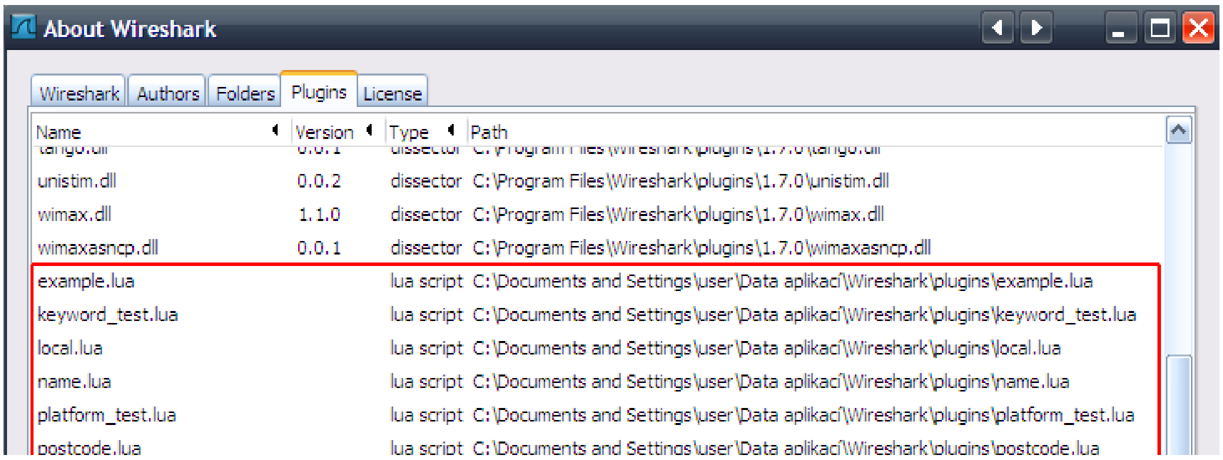
Note: Location of CSjark generated files is given by `-o` command line argument. More in section *Using CSjark*.

4. Copy CSjark generated Lua dissector files into the `Personal Plugins` folder located in step 2.
5. Open the file `init.lua` located in the `Personal configuration` folder which you found in step 2. Insert the following code:

   ```
   dofile("luastructs.lua")
   ```

   This ensures that the `luastructs.lua` is loaded before all other Lua scripts. `luastructs.lua` is a protocol that maps the id of the messages to the correct dissector, and calls them.

6. Restart Wireshark. To check that the scripts are loaded, navigate to `Help` -> `About` -> `Plugins`. The scripts should now appear in the list as "lua script".



Name	Version	Type	Path
tango.dll	0.0.1	dissector	C:\Program Files\Wireshark\plugins\1.7.0\tango.dll
unistim.dll	0.0.2	dissector	C:\Program Files\Wireshark\plugins\1.7.0\unistim.dll
wimax.dll	1.1.0	dissector	C:\Program Files\Wireshark\plugins\1.7.0\wimax.dll
wimaxasncp.dll	0.0.1	dissector	C:\Program Files\Wireshark\plugins\1.7.0\wimaxasncp.dll
example.lua		lua script	C:\Documents and Settings\user\Data aplikací\Wireshark\plugins\example.lua
keyword_test.lua		lua script	C:\Documents and Settings\user\Data aplikací\Wireshark\plugins\keyword_test.lua
local.lua		lua script	C:\Documents and Settings\user\Data aplikací\Wireshark\plugins\local.lua
name.lua		lua script	C:\Documents and Settings\user\Data aplikací\Wireshark\plugins\name.lua
platform_test.lua		lua script	C:\Documents and Settings\user\Data aplikací\Wireshark\plugins\platform_test.lua
postcode.lua		lua script	C:\Documents and Settings\user\Data aplikací\Wireshark\plugins\postcode.lua

To add further dissectors, only step 4, 5 and 6 needs to be repeated.

For further information on the Lua integration in Wireshark, please visit: Lua Support in Wireshark.

1.6 Configuration

Because there exists distinct requirements for flexibility of generating dissectors, CSjark supports configuration for various parts of the program. First, general parameters for utility running can be set up. This can be for example settings of variable sizes for different platforms or other parameters that could determine generating dissectors regardless actual C header file. Second, each individual C struct can be treated in different way. For example, value of specific struct member can be checked for being within specified limits.

Contents

1.6.1 Configuration file format and structure

Note: Besides the configuration described below, one part of the configuration is held directly in the code. It represents the platform specific setup (file `platform.py`) - see Platform specific configuration.

Format

The configuration files are written in YAML which is a data serialization format designed to be easy to read and write. The configuration must be put in a `.yml` file and specified when running CSjark as a command line argument (more about CLI in section *Using CSjark*).

Basic YAML syntax is shown on following example:

```
# comments can start anywhere with number sign (#) and continues until the end of the
# line

key1: value1                    # associative array of two key-value pairs
key2: value2                    # pairs are separated by colon (:) and space ( ) on a
                                # separate line

{key3: value3, key4: value4}    # inline format of specifying associative arrays
                                # pairs are separated by comma (,) and enclosed into
                                # curly braces ({})

- item1                         # list of two items
- item2                         # each item starts with hyphen (-) and space ( ) on a
                                # separate line

[item3, item4]                  # inline format of the list
                                # items are separated by comma (,) and enclosed into
                                # square brackets ([])
```

Data structure hierarchy in YAML is maintained by outline indentation (whitespace is used, tab not allowed). All the basic elements can be combined to create a hierarchy:

```
Options:
    use_cpp:                True
    generate_placeholders:  True

Structs:
  - name:   struct1
    id:     [10, 12, 14]
  - name:   struct2
    id:     [11, 13, 15]
```

Strings are ordinarily unquoted, but may be enclosed in double-quotes ("), or single-quotes ('). The specific number of spaces in the indentation is unimportant as long as parallel elements have the same left justification and the hierarchically nested elements are indented further. This sample defines an associative array with 2 top level keys: one of the keys, "Structs", contains a 2 element array (or "list"), each element of which is itself an associative array with differing keys.

Detailed specification of YAML syntax can be found at YAML website.

Structure

CSjark configuration files might consist of two main parts:

- The first part is used for specifying all the configuration corresponding CSjark processing in general. More about CSjark options in Options Configuration.
- The second part contains configuration for individual C struct definitions. That is described in section Struct Configuration.

The configuration file may have following structure:

```
Options:
  # there will be all your CSjark processing configuration
  use_cpp: True
  ...

Structs:
  # there will be a sequence of Struct definition configurations
  - name: struct1
    id: [10, 12, 14]
    # another struct1 config
  - name: struct2
    id: [11, 13, 15]
    # another struct2 config
```

Automatic generation of configuration files

Automatic generation of configuration file is a simple feature, that could save the user of the utility some time, since the essential part of the configuration file is generated automatically. The utility will only create a new file containing the name of the struct and line to specify the ID for the dissector. To generate the configuration file, the utility must be run with `-p` or `--placeholders` as an option (see *Using CSjark* for more about CSjark CLI).

1.6.2 Struct Configuration

Each individual C struct processed by CSjark can be treated in different way. All the configuration settings must be done in the `Structs` section of the configuration file. Every Struct definition is one item of the sequence and may contain these attributes:

Attribute name	Description
name	C struct name (required field)
id	Dissector message id - more in Dissector message ID
description	Struct name displayed in Wireshark
size	Size of the struct in memory - more in Unknown structs handling
cnf	Conformance file name - more in External Lua dissectors
ranges	Value ranges limitations - more in Value ranges
enums	Enumeration definitions - more in Enums
bitstrings	Bitstrings definitions - more in Bitstrings
trailers	Trailers definitions - more in Trailers
customs	Definitions for custom struct member handling - more in Custom handling of data types

General notes

- Definition of `Structs` part of configuration is not mandatory. However, the user must be aware that if a struct configuration is not defined (namely the `id` attribute), it can be dissected only as a part of other struct (as its struct member). Otherwise there will be no dissectors registered for the struct.
- If there exists a configuration for a struct member and also configuration for the type of this member, the behaviour is not defined. It is up to the user to ensure the definitions are exclusive for each struct member. For example, in the *Value ranges* section example, if the `percent` is defined as *float*, the configuration would be ambiguous and there would be no guarantee that `percent` value is between 0 to 100 or -10 to 10.
- If a struct contains another struct as its member, none of the configuration valid for the outer struct is applied on the nested struct. The same goes for unions. In order to configure the nested struct, the user must define separate struct configuration for it. In this example, the configuration valid for the members of *person* struct is not valid for members of *address* struct

  ```
  struct address {
      int housenum;
  ```

```
        string street;
    };

    struct person {
        string name;
        address adr;
        int age;
    };
```

Value ranges

Some variables may have a domain that is smaller than its given type. You could for example use an integer to describe percentage, which is a number between 0 and 100. It is possible to specify this to CSjark, so that the resulting dissector will tell Wireshark if the values are in the specified range or not. Value ranges are defined by the following syntax:

```
Structs:
  - name: "Name of the struct"
    id: 989
    ranges:
        - member | type: "Name of struct member / type"
          min: "Lowest allowed value"
          max: "Highest allowed value"
```

When the definition specified as a type, the value range is applied to all the members of that type within the struct.

The value ranges configuration is valid only for data types that are meaningful for this purpose (e.g. integers, float, enums). Definitions for other data types are not taken into account.

Example:

```
Structs:
  - name: example_struct
    id: 90
    ranges:
        - member: percent
          min: 0
          max: 100
        - type: float
          min: -10.0
          max: 10.0
```

Value explanations

Some variables may actually represent other values than its type. For example, for an enum it could be preferable to get the textual name of the value displayed, instead of the integer value that represent it. Such example can be an enum type or a bitstring.

Enums

Values of integer variables can be assigned to string values similarly to enumerated values in most programming languages. Thus, instead of integer value, a corresponding value defined in configuration file as a enumeration can be displayed.

The enumeration definition can be of two types. The first one, mapping specified integer by its struct member name, so it gains string value dependent on the actual integer value. And the second, where assigned string values correspond to every struct member of the type defined in the configuration.

The enum definition, as an attribute of the `Structs` item of the configuration file, always starts by `enums` keyword. It is followed by list of members/types for which we want to define enumerated integer values for. Each list item consists of 2 mandatory and 1 optional values

```
- member | type: member name | type name
  values: [value1, value2, ...] | { key1: value1, key2: value2, ...}
  strict: True | False
```

where

- `member name`/`type name` contains string value of integer variable name for which we want to define enumerated values
- `[value1, value2, ...]` is comma-separated list of enumerated values (implicitly numbered, starting from 0)
- `{ key1: value1, key2: value2, ...}` is comma-separated list of key-value pairs, where `key` is integer value and `value` is it's assigned string value
- `strict` is boolean value, which disables warning, if integer does not contain a value specified in the enum list (default `True`)

Example of enums in struct definition contains: - member named `weekday` and values defined as a list of key-value pairs. - definition of enumerated values for `int` type. Values are given by simple list, therefore numbering is implicit (starting from 0, i.e. `Blue` = 2). Warning in case of invalid integer value *will* be displayed.

```
Structs:
  - name: enum_example1
    id: 10
    description: Enum config example
    enums:
      - member: weekday
        values: {1: MONDAY, 2: TUESDAY, 3: WEDNESDAY, 4: THURSDAY, 5: FRIDAY,
                6: SATURDAY, 7: SUNDAY}
      - type: int
        values: [Black, Red, Blue, Green, Yellow, White]
        strict: True # Disable warning if not a valid value
```

Bitstrings

It is possible to configure bitstrings in the utility. This makes it possible to view common data types like integer, short, float, etc. used as a bitstring in the wireshark dissector.

There is two ways to configure bitstrings, the first one is to specify a struct member and define the bit representation. The second option is to specify bits for all struct members of a given type.

These rules specifies the config:

- The bits are specified as 0...n, where 0 is the most significant bit
- A bit group can be one or more bits.
- Bit groups have a name
- It is possible to name all possible values in a bit group.

Below, there is an example of a configuration for the member named `flags` and all the members of `short` type belonging to the struct `example`.

- member `flags`: This example has four bits specified, the first bit group is named "In use" and represent bit 0. The second group represent bit 1 and is named "Endian", and the values are named: 0 = "Big", 1 = "Little". The last group is "Platform" and represent bit 2-3 and have 4 named values.
- type `short`: Each of the 3 bits represents one colour channel and it can be either "True" or "False".

```
Structs:
  - name: example
    id: 1000
    description: An example
    bitstrings:
      - member: flags
        0: In use
        1: [Endian, Big, Little]
        2-3: [Platform, Win, Linux, Mac, Solaris]
      - type: short
        0: Red
        1: Green
        2: Blue
```

Dissector message ID

Every packet with C struct captured by Wireshark contains a header. One of the fields in the header, the `id` field, specifies which dissector should be loaded to dissect the actual struct. The value of this field can be specified in the configuration file.

This is an example of the specification

```
Structs:
    - name: structname
      id: 10
```

More different messages can be dissected by one specific dissector. Therefore, the struct configuration can contain a whole list of dissector message ID's, that can process the struct.

```
Structs:
    - name: structname
      id: [12, 43, 3498]
```

Note: The `id` must be an integer between 0 and 65535.

External Lua dissectors

In some cases, CSjark will not be able to deliver the desired result from its own analysis, and the configuration options above may be too constraining. In this case, it is possible to write the lua dissector by hand, either for a given member or for an entire struct.

Note: To be able to understand and write external Lua dissectors, the user should be familiar with basics of Lua programming and Lua integration into Wireshark. More information how to write Lua code can be found in Lua reference manual. For further information on the Lua integration in Wireshark, please visit Lua Support in Wireshark.

A custom Lua code for desired struct must be defined in an external conformance file with extension `.cnf`. The conformance file name and relative path then must be defined in the configuration file for the struct for which is the custom code applied for. The attribute name for the custom Lua definition file and path is `cnf`, as shown below:

```
# CSjark configuration file

Structs:
    - name: custom_lua
      cnf: etc/custom_lua.cnf
      id: 1
      description: example of external custom Lua file definition
```

Writing the conformance file implies respecting following rules:

- Each section starts with `#.<SECTION>` for example `#.COMMENT`.
- Unknown sections are ignored.

The conformance file implementation allows user to place the custom Lua code on various places within the Lua dissector code already generated by CSjark. There is a list of possible places:

`DEF_HEADER id`	Lua code added before a Field definition.
`DEF_BODY id`	Lua code to replace a Field definition. Within the definition, the original body can be referenced as `%(DEFAULT_BODY)s` or `{DEFAULT_BODY}`
`DEF_FOOTER id`	Lua code added after a Field definition
`DEF_EXTRA`	Lua code added after the last definition
`FUNC_HEADER id`	Lua code added before a Field function code
`FUNC_BODY id`	Lua code to replace a Field function code
`FUNC_FOOTER id`	Lua code added after a Field function code
`FUNC_EXTRA`	Lua code added at end of dissector function
`COMMENT`	A multiline comment section
`END`	End of a section
`END_OF_CNF`	End of the conformance file

Where `id` denotes C struct member name . The `END` token is only optional, it does not have to be placed at the end of each section. However, all code after `END` token which is not part of another section defined above is discarded.

Example of such conformance file follows:

```
#.COMMENT
    This is a .cnf file comment section
#.END

#.DEF_HEADER super
-- This code will be added above the 'super' field definition
#.END

#.COMMENT
    DEF_BODY replaces code inside the dissector function.
    Use %(DEFAULT_BODY)s or {DEFAULT_BODY} to use generated code.
#.DEF_BODY hyper
-- This is above 'hyper' definition
%(DEFAULT_BODY)s
-- This is below 'hyper'
```

```
#.END

#.DEF_FOOTER name
-- This is below 'name' definition
#.END

-- This text would be discarded.

#.DEF_EXTRA
-- This was all the Field definitions
#.END


#.FUNC_HEADER precise
    -- This is above 'precise' inside the dissector function.
#.END


#.COMMENT
    FUNC_BODY replaces code inside the dissector function.
    Use %(DEFAULT_BODY)s or {DEFAULT_BODY} to use generated code.
#.FUNC_BODY name
    --[[ This comments out the 'name' code
    {DEFAULT_BODY}
    ]]--
#.END

#.FUNC_FOOTER super
    -- This is below 'super' inside dissector function
#.END

#.FUNC_EXTRA
    -- This is the last line of the dissector function
#.END_OF_CNF
```

This conformance file when run with this C header code:

```
struct custom_lua {
    short normal;
    int super;
    long long hyper;

    char name;
    double precise;

};
```

...will produce this Lua dissector:

```
-- Dissector for win32.custom_lua: custom_lua (Win32)
local proto_custom_lua = Proto("win32.custom_lua", "custom_lua (Win32)")

-- ProtoField definitions for: custom_lua
local f = proto_custom_lua.fields
f.normal = ProtoField.int16("custom_lua.normal", "normal")
-- This code will be added above the 'super' field definition
f.super = ProtoField.int32("custom_lua.super", "super")
-- This is above 'hyper' definition
f.hyper = ProtoField.int64("custom_lua.hyper", "hyper")
```

```
-- This is below 'hyper'
f.name = ProtoField.string("custom_lua.name", "name")
-- This is below 'name' definition
f.precise = ProtoField.double("custom_lua.precise", "precise")
-- This was all the field definitions

-- Dissector function for: custom_lua
function proto_custom_lua.dissector(buffer, pinfo, tree)
    local subtree = tree:add_le(proto_custom_lua, buffer())
    if pinfo.private.caller_def_name then
        subtree:set_text(pinfo.private.caller_def_name .. ": " .. proto_custom_lua.description)
        pinfo.private.caller_def_name = nil
    else
        pinfo.cols.info:append(" (" .. proto_custom_lua.description .. ")")
    end

    subtree:add_le(f.normal, buffer(0, 2))
    subtree:add_le(f.super, buffer(4, 4))
    -- This is below 'super' inside dissector function
    subtree:add_le(f.hyper, buffer(8, 8))
    --[[ This comments out the 'name' code
        subtree:add_le(f.name, buffer(16, 1))
    ]]--
    -- This is above 'precise' inside the dissector function.
    subtree:add_le(f.precise, buffer(24, 8))
    -- This is the last line of the dissector function
end

delegator_register_proto(proto_custom_lua, "Win32", "custom_lua", 1)
```

Support for Offset and Value in Lua Files

Via External Lua dissectors CSjark also provides a way to reference the proto fields of the dissector, with correct offset value and correct Lua variable.

To access the fields value and offset, `{OFFSET}` and `{VALUE}` strings may be put into the conformance file as shown below:

```
#.FUNC_FOOTER pointer
    -- Offset: {OFFSET}
    -- Field value stored in lua variable: {VALUE}
#.END
```

Adding the offset and variable value is only possible in the parts that change the code of Lua functions, i.e. `FUNC_HEADER`, `FUNC_BODY` and `FUNC_FOOTER`.

Above listed example leads to following Lua code:

```
local field_value_var = subtree:add(f.pointer, buffer(56,4))
    -- Offset: 56
    -- Field value stored in lua variable: field_value_var
```

Note: The value of the referenced variable can be used after it is defined.

Trailers

CSjark only creates dissectors from C structs defined as its input. To be able to use built-in dissectors in Wireshark, it is necessary to configure it. Wireshark has more than 1000 built-in dissectors. Several trailers can be configured for a packet.

The following parameters are allowed in trailers:

name	Protocol name for the built-in dissector
count	The number of trailers
member	Struct member, that contain the amount of trailers
size	Size of the buffer to feed to the protocol

There are two ways to configure the trailers - specify the total number of trailers or give a variable in the struct, which contains the amount of trailers. Both ways to configure trailers are shown below. In case the variable `trailer_count` equals 2, the definitions has the same effect.

```
trailers:
  - name: proto1
    member: trailer_count
    size: 32

trailers:
  - name: proto1
    count: 2
    size: 32
```

Example: The example below shows an example with BER [1], which has 4 trailers with a size of 6 bytes.

```
trailers:
  - name: ber
  - count: 4
  - size: 6
```

Custom handling of data types

The utility supports custom handling of specified data types. Some variables in input C header may actually represent other values than its own type. This CSjark feature allows user to map types defined in C header to Wireshark field types. Also, it provides a method to change how the input field is displayed in Wireshark. The custom handling must be done through a configuration file.

For example, this functionality can cause Wireshark to display `time_t` data type as `absolute_time`. The displayed type is given by generated Lua dissector and functions of `ProtoField` class.

List of available output types follows:

`Integer types` uint8, uint16, uint24, uint32, uint64, int8, int16, int24, int32, int64, framenum

`Other types` float, double, string, stringz, bytes, bool, ipv4, ipv6, ether, oid, guid, absolute_time, relative_time

For `Integer` types, there are some specific attributes that can be defined (see below). More about each individual type can be found in Wireshark reference.

The section name in configuration file for custom data type handling is called `customs`. This section can contain following attributes:

- Required attributes

[1] Basic Encoding Rules

Attribute name	Value
`member` | `type`	Name of member or type for which is the configuration applied
`field`	Displayed type (see above)

- Optional attributes - all types

Attribute name	Value
`abbr`	Filter name of the field (the string that is used in filters)
`name`	Actual name of the field
`desc`	The description of the field (displayed on Wireshark statusbar)

- Optional attributes - Integer types only:

Attribute name	Value
`base`	Displayed representation - can be one of `base.DEC`, `base.HEX` or `base.OCT`
`values`	List of `key:value` pairs representing the Integer value - e.g. `{0: Monday, 1: Tuesday}`
`mask`	Integer mask of this field

Example of such a configuration file follows:

```
Structs:
  - name: custom_type_handling
    id: 1
    customs:
      - type: time_t
        field: absolute_time
      - member: day
        field: uint32
        abbr: day.name
        name: Weekday name
        base: base.DEC
        values: { 0: Monday, 1: Tuesday, 2: Wednesday, 3: Thursday, 4: Friday}
        mask: nil
        desc: This day you will work a lot!!
```

and applies for example for this C header file:

```
#include <time.h>

struct custom_type_handling {
    time_t abs;
    int day;
};
```

Both struct members are redefined. First will be displayed as `absolute_type` according to its type (`time_t`), second one is changed because of the struct member name (`day`).

Unknown structs handling

The header files that the utility parses, may have nested struct that is not defined in any other header file. To make it possible to generate a dissector for this case, the user must be able to specify the size of the struct in a configuration file. When the sizes are specified it will be possible to generate a struct that can display the defined members of the struct correctly in the utility, for the parts that are not defined only the hex value will be displayed. This feature is added as a possible way to solve include dependencies that our utility is not able to solve. The user of the utility will get an error message when the utility is not able to find include dependencies, and the user may add the size of struct to be able to generate a dissector for the struct.

The size of unknown struct may be defined directly in the struct configuration as `size` attribute, similar to the example below:

```
Structs:
    - name: unknown struct
      id: 111
      size: 78
```

Note: Size must be defined as a positive integer (or 0).

1.6.3 Options Configuration

CSjark processing behaviour can be set up in various ways. Besides letting the user to specify how the CSjark should work by the command line arguments (see section *Using CSjark*), it is also possible to define the options as a part of the configuration file(s).

Configuration file field	CLI equivalent	Value	Description
`verbose`	`-v`	`True/False`	Print detailed information
`debug`	`-d`	`True/False`	Print debugging information
`strict`	`-s`	`True/False`	Only generate dissectors for known structs
`output_dir`	`-o`	`None` or path	Definition of output destination
`output_file`	`-o`	`None` or file name	Writes the output to the specified file
`generate_placeholders`	`-p`	`True/False`	Generate placeholder config file for unknown structs
`use_cpp`	`-n`	`True/False`	Enables/disables the C pre-processor
`cpp_path`	`-C`	`None` or file name	Specifies which preprocessor to use
`excludes`	`-x`	List of excluded paths	File or folders to exclude from parsing
`platforms`		List of platform names	Set of platforms to support in dissectors
`include_dirs`	`-I`	List of directories	Directories to be searched for Cpp includes
`includes`	`-i`	List of includes	Process file as Cpp #include “file” directive
`defines`	`-D`	List of defines	Predefine name as a Cpp macro
`undefines`	`-U`	List of undefines	Cancel any previous Cpp definition of name
`arguments`	`-A`	List of additional arguments	Any additional C preprocessor arguments

The last 5 options can be also specified separately for each individual input C header file. This can be achieved by adding sequence `files` with mandatory attribute `name`.

Below you can see an example of such `Options` section:

```
Options:
    verbose: True
    debug: False
    strict: False
    output_dir: ../out
    output_file: output.log
    generate_placeholders: False
    use_cpp: True
    cpp_path: ../utils/cpp.exe
    excludes: [examples, test]
    platforms: [default, Win32, Win64, Solaris-sparc, Linux-x86]
```

```
include_dirs: [../more_includes]
includes: [foo.h, bar.h]
defines: [CONFIG_DEFINED=3, REMOVE=1]
undefines: [REMOVE]
arguments: [-D ARR=2]
files:
  - name: a.h
    includes: [b.h, c.h]
    define: [MY_DEFINE]
```

Note: If you give CSjark multiple configuration files with the same values defined, it takes:

- for attributes with single value: a value from *last processed config file* is valid
- for attributes with list values: lists are *merged*

1.6.4 Platform specific configuration

To ensure that CSjark is usable as much as possible, platform specific

Entire platform setup is done via Python code, specifically `platform.py`. This file contains following sections:

1. Platform class definition including it's methods
2. Default mapping of C type and their Wireshark field type
3. Default C type size in bytes
4. Default alignment size in bytes
5. Custom C type sizes for every platform which differ from default
6. Custom alignment sizes for every platform which differ from default
7. Platform-specific C preprocessor macros
8. Platform registration method and calling for each platform

When defining new platform, following steps should be done. Referenced sections apply to `platform.py` sections listed above. All the new dictionary variables should have proper syntax of Python dictionary:

Field sizes Define custom C type sizes in section 5. Create new dictionary with name in capital letters. Only those different from default (section 3) must be defined.

```
NEW_PLATFORM_C_SIZE_MAP = {
    'unsigned long': 8,
    'unsigned long int': 8,
    'long double': 16
}
```

Memory alignment Define custom memory alignment sizes in section 6. Create new dictionary with name in capital letters. Only those different from default (section 4) must be defined.

```
NEW_PLATFORM_C_ALIGNMENT_MAP = {
    'unsigned long': 8,
    'unsigned long int': 8,
    'long double': 16
}
```

Macros Define dictionary of platform specific macros in section 7. These macros then can be used within C header files to define platform specific struct members etc. E.g.:

```
#if _WIN32
    float num;
#elif __sparc
    long double num;
#else
    double num;
```

Example of such macros:

```
NEW_PLATFORM_MACROS = {
    '__new_platform__': 1, '__new_platform': 1
}
```

Register platform In last section (8), the new platform must be registered. Basically, it means calling the constructor of Platform class. That has following parameters:

```
Platform(name, flag, endian, macros=None, sizes=None, alignment=None)
```

where

`name`	name of the platform
`flag`	unique integer value representing this platform
`endian`	either `Platform.big` or `Platform.little`
`macros`	C preprocessor platform-specific macros like _WIN32
`sizes`	dictionary which maps C types to their size in bytes

Registering of the platform then might look as follows:

```
# New platform
Platform('New-platform', 8, Platform.little,
         macros=NEW_PLATFORM_MACROS,
         sizes=NEW_PLATFORM_C_SIZE_MAP,
         alignment=NEW_PLATFORM_C_ALIGNMENT_MAP)
```

CHAPTER

TWO

DEVELOPER DOCUMENTATION

2.1 Development rules

This document describes coding requirements and conventions for working with the CSjark code base. Please read it carefully and ask back any questions you might have.

2.1.1 Coding Style Standard

The programming language used to implement the utility is Python. All the code should as much as possible follow the coding style described in the Style Guide for Python Code (PEP8). In addition we decided that the design should attempt to be pythonic, as detailed by PEP20.

2.1.2 Code base

The entire CSjark project is hosted on GitHub. In order to get the source code, you can:

- clone the Git repository: https://github.com/eventh/kpro9
- download most recent tar archive
- download most recent zip archive

Note: When commiting to the repository, always write good log messages. It will help people that will read the diffs in the future.

2.1.3 Issue tracking

For issue tracking, we use bug tracking capabilities of GitHub. You can register bugs, discuss issues or see what's going on for the next milestone, both from an email and from a web interface.

https://github.com/eventh/kpro9/issues

2.2 Design Overview

This part describes the design of the CSjark to help all the future developers to understand what is under the hood.

2.2.1 Textual description

CSjark starts with the *csjark* module. It takes as input command line arguments, including file and folder names for C header files and configuration files. It replaces folder names with the file names of the files inside the folders, and checks that all the files exists, then it gives the names of the configuration files to the *config* module.

The *config* module parses configuration files and stores them in suitable config data structures in memory. It takes as input file names, and it opens those files to read their content.

Csjark module then gives file names of C header files to the *cpp* module. The *cpp* module gives the file names and some other arguments to an external program, the C preprocessor (*cpp.exe* on Windows). This external program opens the header files, and when it encounters *#include* directives it searches for the right file and opens it as well. The output of this external program is just a long string of C code, which *cpp* module returns to the *csjark* module.

The *csjark* module then gives the C code string to *cparser* module, which forwards the string to pycparser. Pycparser parse the C code to generate an abstract syntax tree, which it returns to *cparser* which returns it to *csjark*.

Csjark module then tells *cparser* to find all struct definitions in the abstract syntax tree, which it does by traversing the tree. Each time it finds a struct, it asks *dissector* module to create a protocol for it. Afterwards *cparser* holds a list of all the created protocols.

Csjark then gets the list of protocols from *cparser*, and for each one asks *dissector* module to generate lua code for Wireshark dissectors. It writes these dissectors to files, and finishes with a status message informing the user how it all went.

The new *field* module is simply to move class *Fields* and its sub-classes into their own module to make *dissector* module smaller and less complex.

Summary

csjark module writes Lua files, *config* module opens and reads YAML files, *cpp* module starts an external program which reads C header files. The structure as well as the associations among the classes are shown on following module and class diagrams.

2.2.2 Module diagram

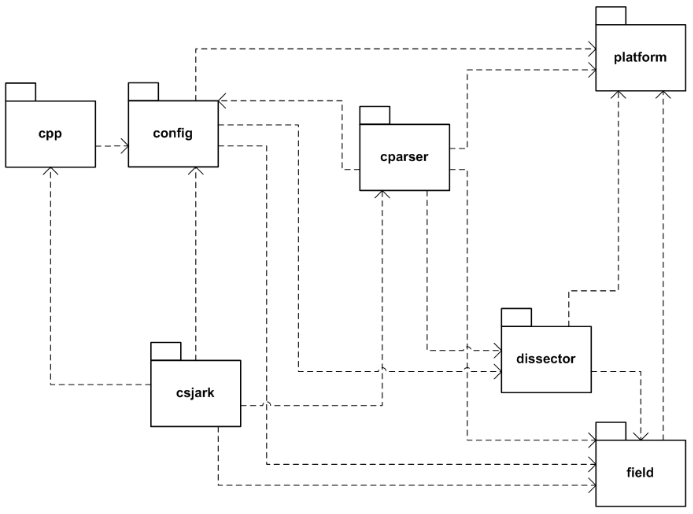


Figure 2.1: *CSjark: module diagram*

2.2.3 Class diagram

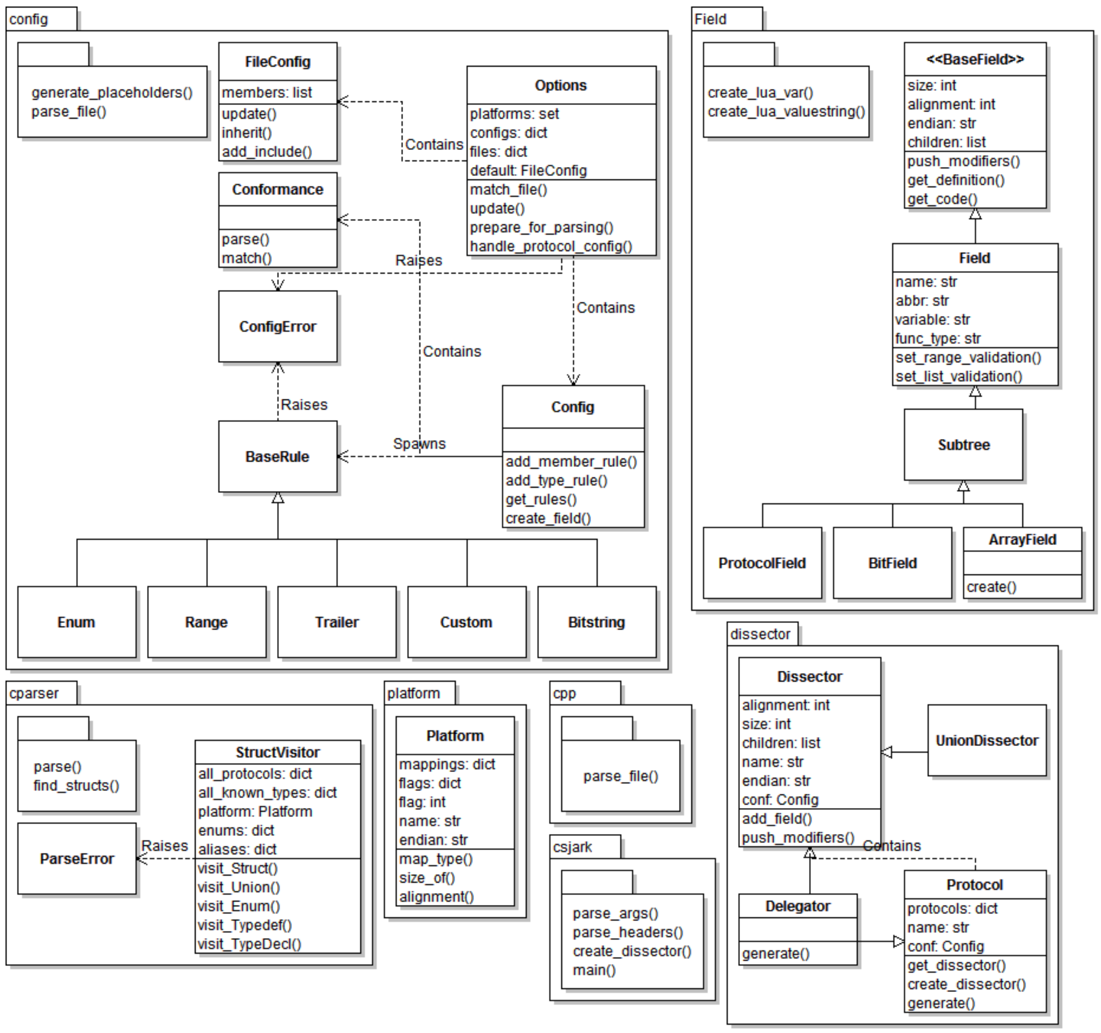


Figure 2.2: *CSjark: class diagram*

2.3 Testing

There are several types of tests done for verification of CSjark functionality:

2.3.1 White box testing

This type of white box testing basically means verification of functionality of specific section of the code, usually at the function level. As a tool for creating white box unit tests, the team decided to use the Attest testing framework for python code.

2.3.2 Black box testing

In general black box testing does not require the tester to have any intimate knowledge about the system or any of the programming logic that went into making it. Black box test cases are built around the specifications and requirements of a system, for example its functional, and in some cases, non-functional requirements.

For CSjark, black box testing means feeding the utility with input C header file, corresponding configuration file and generating the output (Lua dissector). Then, the generated output is compared to expected output.

Another way how to do the black box testing to use the generated dissectors in Wireshark. For this purpose, we also need a `pcap` file containing the IPC packets that corresponds to the input C header files. You can read more about whole process in the User Manual section *Introduction*. A short guide how to create `pcap` file for the C struct can be found in the project wiki.

With the generated Lua dissectors, it should possible to display the contents of the C struct within the IPC packets. Also, according to the utility requirements, it should be possible to filter and search by any variable name or its value. This way of testing is based on manual checking of the individual variable values. The process involves several manual steps and therefore cannot be automated.

2.3.3 Regression testing

The test must be written in a way that it should be possible to run them repeatedly after the first run. That can ensure that all the implemented functionality is still working well after changes in the code.

2.3.4 Creating tests

To create tests using Attest, you start by importing `Tests`, `assert_hook` and optionally `contexts` from `attest` library. You then create a variable and initialize it to an instance of Tests, which is the variable that will contain list functions that each constitutes one test that is to be run. To feed your test instance with functions for testing you then have to mark these functions with a decorator and feed it the `.tests` function of the Tests instance. After creating a unit test in this fashion you can run all of your unit tests through Attest from the command line by typing:

```
python -m attest
```

This runs all of your unit tests through Attest and returns a message telling the user how many assertions failed, as well as what input made them fail. For more information read the user documentation of Attest.

2.3.5 Testing code

All the test code, the testing configuration files and the testing input files are located in folder

> CSjark/csjark/test

It contains modules for unit/module testing of each of the CSjark modules as well as modules for bundled white/black box testing.

2.4 Source code overview

Program modules

`csjark`	CSjark is a tool for generating Lua dissectors from C struct

Continued on next page

Table 2.1 – continued from previous page

`config`	A module for configuration of our utility.
`cpp`	Module for performing the C preprocessor step on C header files.
`cparser`	A module for parsing C files to find struct definitions.
`field`	A module for classes which represents values in a packet.
`dissector`	A module for generating Lua dissectors for Wireshark.
`platform`	A module which holds platform specific configuration.

Testing modules

Modules for testing are located in

CSjark/csjark/test

2.4.1 csjark

CSjark is a tool for generating Lua dissectors from C struct definitions to use with Wireshark.

`csjark.`**`parse_args`**`(`*args=None*`)`
: Parse arguments given in sys.argv.

  ‘args’ is a list of strings to parse instead of sys.argv.

`csjark.`**`parse_headers`**`(`*headers*`)`
: Parse ‘headers’ to create a Wireshark protocol dissector.

`csjark.`**`create_dissector`**`(`*filename*, *platform*, *folders=None*, *includes=None*`)`
: Parse ‘filename’ to create a Wireshark protocol dissector.

  ‘filename’ is the C header/code file to parse. ‘platform’ is the platform we should simulate. ‘folders’ is a set of all folders to -Include. ‘includes’ is a set of filenames to #include. Returns the error if parsing failed, None if succeeded.

`csjark.`**`_write_dissector`**`(`*name*, *proto*`)`
: Write a single dissector to file.

`csjark.`**`write_dissectors_to_file`**`(`*all_protocols*`)`
: Write lua dissectors to file(s).

`csjark.`**`write_delegator_to_file`**`()`
: Write the lua file which delegates dissecting to dissectors.

`csjark.`**`write_placeholders_to_file`**`(`*protocols*`)`
: Write a placeholder file for ‘protocols’ with no configuration.

`csjark.`**`main`**`()`
: Run the CSjark program.

2.4.2 config

A module for configuration of our utility.

Should parse config files and create data structures which the parser can use when translating C struct definitions to Wireshark protocols and fields.

Config class holds configuration for specific struct by name. FileConfig holds C preprocessor options for specific files by path. Options holds global utility configuration, include dictinaries for the Config and Fileconfig instances.

Additionally there is the BaseRule class and its subclasses which holds specific rules specified by configuration for members in structs.

exception `config.`**`ConfigError`**
: Exception raised by invalid configuration.

class `config.`**`Config`**(*name*)
: Holds configuration for a specific protocol.

    `add_member_rule`(*member*, *rule*)
    : Add a new rule for a specific member.

        'member' is the member of a struct to match 'rule' is the new rule to add

    `add_type_rule`(*type*, *rule*)
    : Add a new rule for all members of a specific type.

        'type' is the C type to match members against 'rule' is the new rule to add

    `get_rules`(*member*, *type*)
    : Return all rules which match 'member' or 'type'.

    `create_field`(*proto*, *name*, *ctype*, *size*, *alignment*, *endian*)
    : Create a field depending on rules.

class `config.`**`BaseRule`**(*conf*, *obj*)
: A base class for rules referring to protocol fields.

class `config.`**`Range`**(*conf*, *obj*)
: Rule for specifying a valid range for a member or type.

class `config.`**`Enum`**(*conf*, *obj*)
: Rule for emulating enum with int-like types.

class `config.`**`Bitstring`**(*conf*, *obj*)
: Rule for representing ints which are bit strings.

class `config.`**`Trailer`**(*conf*, *obj*)
: Rule for specifying one or more trailer protocol(s).

class `config.`**`Custom`**(*conf*, *obj*)
: Rule for specifying a custom field handling.

    `create`(*proto*, *name*, *ctype*, *size*, *alignment*, *endian*)
    : Create a new Field based on this rule.

class `config.`**`ConformanceFile`**(*conf*, *file*, *config_file=''*)
: A class for parsing a conformance file.

    A conformance file specifies custom lua code for fields. It can give custom code for the definition, and inside the dissector function. For these two cases, it supports header, body, footer and extra sections which places code above, instead of, below, or at the end of the section.

    Each section starts with #.<SECTION> for example #.COMMENT. Unknown sections are ignore, to be compatible with Asn2wrs .cnf files.

    `t_def_hdr` = **'DEF_HEADER'**

    `t_def_body` = **'DEF_BODY'**

    `t_def_ftr` = **'DEF_FOOTER'**

    `t_def_extra` = **'DEF_EXTRA'**

    `t_func_hdr` = **'FUNC_HEADER'**

`t_func_body` = **'FUNC_BODY'**

`t_func_ftr` = **'FUNC_FOOTER'**

`t_func_extra` = **'FUNC_EXTRA'**

`t_comment` = **'COMMENT'**

`t_end` = **'END'**

`t_end_cnf` = **'END_OF_CNF'**

`def_tokens` = **['DEF_HEADER', 'DEF_BODY', 'DEF_FOOTER']**

`func_tokens` = **['FUNC_HEADER', 'FUNC_BODY', 'FUNC_FOOTER']**

`store_tokens` = **`def_tokens`** + **`func_tokens`** + **['DEF_EXTRA', 'FUNC_EXTRA']**

`valid_tokens` = **`store_tokens`** + **['COMMENT', 'END', 'END_OF_CNF']**

`_get_token`(*line*)
Find the token and the field it refers to.

`parse`()
Parse the conformance file's sections and content.

`match`(*name*, *code*, *definition=False*, *field=None*)
Modify fields code if a cnf file demands it.

class `config.`**`FileConfig`**(*name*)
Holds options for specific files.

`members` = **('include_dirs', 'includes', 'defines', 'undefines', 'arguments')**

`update`(*obj*)
Update variables with config from a yml file.

`inherit`(*parent*)
Update variables with config from another FileConfig instance.

classmethod `add_include`(*filename*, *include*)
Add a new 'include' to 'filename' config.

If the 'filename' has no FileConfig, creates one.

class `config.`**`Options`**
Holds options for the whole utility.

These options are set by either command line interface or one or more configuration yaml files.

`verbose` = **False**

`debug` = **False**

`strict` = **False**

`output_dir` = **None**

`output_file` = **None**

`generate_placeholders` = **False**

`use_cpp` = **True**

`cpp_path` = **None**

`excludes` = **[]**

`platforms` = **set()**

delegator = None

configs = {}

files = {}

default = <config.FileConfig object at 0x02225210>

classmethod **match_file** (*filename*)
Find file config object for 'filename'.

classmethod **update** (*obj*)
Update the options from a config yaml file.

classmethod **prepare_for_parsing** ()
Prepare options before parsing starts..

classmethod **handle_protocol_config** (*obj*, *filename='*')
Handle rules and configuration for a protocol.

config.**generate_placeholders** (*protocols*)
Generate placeholder config for unknown structs.

config.**parse_file** (*filename*, *only_text=None*)
Parse a configuration file.

2.4.3 cpp

Module for performing the C preprocessor step on C header files.

The parse_file() function calls the external C preprocessor program, while post_cpp() function removes output from the preprocessor which pycparser does not support.

cpp.**_get_cpp** ()
Find the path and args to the C preprocessor.

cpp.**parse_file** (*filename*, *platform=None*, *folders=None*, *includes=None*)
Run a C header or code file through C preprocessor program.

'filename' is the file to feed CPP. 'platform' is the platform to simulate. 'folders' is directories to -Include. 'includes' is a set of filename to #include.

cpp.**post_cpp** (*lines*)
Perform a post preprocessing step, removing unsupported C code.

2.4.4 cparser

A module for parsing C files to find struct definitions.

The parse() function asks pycparser to parse a piece of C code, and returns an Abstract Syntax Tree (AST). The find_structs() function walks the AST to find any struct defininition.

The StructVisitor class is used to traverse an AST generated by pycparser, and looks for structs, enums, unions and type definitions. When it finds a struct or a union it creates a Dissector instance from the dissector module, which can generate Lua dissectors for respective C code sections.

This module requires PLY 3.4 and pycparser 2.07.

exception cparser.**ParseError**
Exception raised by invalid input to the parser.

cparser.**parse** (*text*, *filename='‘*, *parser=<pycparser.c_parser.CParser object at 0x02352710>*)
Parse C code and return an AST.

cparser.**find_structs** (*ast*, *platform=None*)
Walks the AST nodes to find structs.

class cparser.**StructVisitor** (*platform*)
A class which visit struct nodes in the AST.

The Visitor traverse the Tree, and when it finds Struct, Enum, Union, Typedef or TypeDecl nodes it calls the respective methods in this class.

It will populate all_protocols class member with Dissector-instances representing all the relevant C data structures it found.

The alll_know_types class member is used to discover which C file should be included if we fail parsing because of unknown types.

all_protocols = {}

all_known_types = {}

_last_visitor = None

_last_diss = None

_last_proto = None

visit_Struct (*node*)
Visit a Struct node in the AST.

visit_Union (*node*)
Visit a Union node in the AST.

_visit_nodes (*node*, *union=False*)
Visit a node in the tree.

visit_Enum (*node*)
Visit a Enum node in the AST.

visit_Typedef (*node*)
Visit Typedef declarations nodes in the AST.

visit_TypeDecl (*node*)
Keep track of Type Declaration nodes.

handle_type_decl (*node*, *proto*)
Find member details in a type declaration.

handle_array_decl (*node*, *depth=None*)
Find the depth, size and type of the array.

‘node’ is a pycparser.c_ast.ArrayDecl instance ‘depth’ is a list of elements already traversed It returns a list with count of elements in in each level, and a Field instance.

handle_protocol (*proto*, *name*, *proto_name*)
Add an protocol field or union field to the protocol.

handle_array (*proto*, *depth*, *field*, *name=None*)
Add an ArrayField to the protocol.

handle_pointer (*node*, *proto*)
Find member details in a pointer declaration.

handle_enum(*proto*, *name*, *enum*)
Add an EnumField to the protocol.

handle_field(*proto*, *name*, *ctype*, *size=None*, *alignment=None*)
Add a field representing the struct member to the protocol.

_find_protocol(*node*)
Check if the protocol already exists.

_create_protocol(*node*, *union=False*)
Create a new protocol for 'node'.

_create_field(*name*, *ctype*, *size=None*, *alignment=None*)
Create a new field representing the given 'ctype'.

_create_enum(*name*, *enum*)
Create a new enum field.

_create_protocol_field(*name*, *proto_name*)
Create a new protocol field.

_get_type(*node*)
Get the C type from a node.

_get_array_size(*node*)
Calculate the size of the array.

_register_type(*node*, *name=None*)
Register the type 'name' in the known types mapping.

2.4.5 field

A module for classes which represents values in a packet.

Field class and its subclasses represent a value or a list of values in packets to be dissected by Wireshark. They represent Wireshark's ProtoField instances.

field.**create_lua_var**(*var*, *length=None*)
Return a valid lua variable name.

field.**create_lua_valuestring**(*dict_*, *wrap=True*)
Convert a python dictionary to lua table.

class field.**BaseField**(*size*, *alignment*, *endian*)
Interface for Fields and list of Fields.

add_var
Get the endian specific function for adding a item to a tree.

push_modifiers()
Push prefixes and postfixes down to child fields.

get_definition()
Get the ProtoField definition for this field.

get_code(*offset*, *store=None*, *tree='subtree'*)
Get the code for dissecting this field.

class field.**Field**(*name*, *type*, *size*, *alignment*, *endian*)
Represents Wireshark's ProtoFields which stores a specific value.

prefixes = ['var_prefix', 'abbr_prefix', 'name_prefix']

`postfixes` **= ['name_postfix', 'var_postfix', 'abbr_postfix']**

`infixes` **= ['_name', '_var', '_abbr']**

`members` **= ['type', 'size', 'alignment', 'endian', 'base', 'values', 'mask', 'desc', 'offset',**

'range_validation', 'list_validation'] + `prefixes` + `postfixes` + `infixes`

`name`
Get the name of the field.

`abbr`
Get the fields abbr.

`variable`
Get the variable to store the field in.

`func_type`
Get the lua function to read values from buffers.

`get_definition`()
Get the ProtoField definition for this field.

`get_code`(*offset*, *store=None*, *tree='subtree'*)
Get the code for dissecting this field.

'offset' is the buffer offset the value is stored at 'store' is the lua variable to store the tree node in 'tree' is the tree we are adding the node to

`_store_value`(*var=None*, *offset=None*)
Create code which stores the field value in 'var'.

If 'offset' is not provided, must be run after get_code().

`set_range_validation`(*min_value=None*, *max_value=None*)
Set validation that field value is between a given range.

`_create_range_validation`()
Create code which validates the field value inside the range.

`set_list_validation`(*values*, *strict=True*)
Set validating that field value is a member of 'values'.

`_create_list_validation`()
Create code which validates fields value in valuestring.

class `field.`**`Subtree`**(*tree*, **args*, ***vargs*)
A Subtree is a Field with a list of fields as children.

`push_modifiers`(*push_children=True*)
Push prefixes and postfixes down to child fields.

`get_definition`()
Get the ProtoField definition for this field.

`get_code`(*offset*, *store=None*, *tree=None*)
Get the code for dissecting this field.

'offset' is the buffer offset the value is stored at 'store' is the lua variable to store the tree node in 'tree' is the tree we are adding the node to

class `field.`**`ArrayField`**(*children*, *tree='arrtree'*, *parent='subtree'*)
ArrayField is a Subtree with visible indices.

push_modifiers()
Push prefixes and postfixes down to child fields.

get_code(*offset*, *store=None*, *tree=None*)
Get the code for dissecting this field.

'offset' is the buffer offset the value is stored at 'store' is the lua variable to store the tree node in 'tree' is the tree we are adding the node to

classmethod create(*depth*, *field*, *name='array'*)
Recursively create a tree of arrays of 'depth'.

class field.**BitField**(*bits*, *name*, *type*, *size*, *alignment*, *endian*)
BitField is a Subtree with field for each relevant bit.

class field.**ProtocolField**(*name*, *proto*)
A ProtocolField is a field for a protocol.

This class allows part of a packet to be dissected by another protocol, used for structs and unions which is a member of another.

Fake
alias of `FakeProto`

get_definition()
Get the ProtoField definition for this field.

get_code(*offset*, *store=None*, *tree='subtree'*)
Get the code for dissecting this field.

'offset' is the buffer offset the value is stored at 'store' is the lua variable to store the tree node in 'tree' is the tree we are adding the node to

2.4.6 dissector

A module for generating Lua dissectors for Wireshark.

The Disssector class is a container of platform-specific Wirehsark fields instances and subclasses. The Protocol class is a collection of dissector-instances for each platform it should support. The Delegator class is a subclass of both these classes, and generates 'luastructs.lua' which decides which Wireshark dissector to call from each message id.

class dissector.**Dissector**(*name*, *platform*, *conf=None*)
A Dissector is a collection of fields and code.

It's used to generate Wireshark dissectors written in Lua, for dissecting a packet into a set of fields with values.

alignment
Find the alignment size of the fields in the protocol.

size
Find the size of the fields in the protocol.

add_field(*field*)
Add a field to the dissectors list of field.

push_modifiers()
Push prefixes and postfixes down to child fields.

get_definition()
Get the ProtoField definition for this field.

get_code(*offset*, *store=None*, *tree='subtree'*)
Get the code for dissecting this field.

get_padding(*field*, *offset*)
Get padding for correct alignment.

_trailers(*rules*, *offset*)
Add code for handling of trailers to the protocol.

class dissector.**UnionDissector**(**args*, ***vargs*)
A Dissector where each field does not increase the offset.

size
Find the size of the fields in the protocol.

class dissector.**Protocol**(*name*, *id=None*, *description=None*)
A Protocol is a collection of platform specific dissectors.

It's used to generate Wireshark dissectors written in Lua, for dissecting a packet into a set of fields with values.

REGISTER_FUNC = 'delegator_register_proto'

protocols = {}

get_dissector(*platform*)
Get a dissector for a given 'platform'.

classmethod create_dissector(*name*, *platform=None*, *conf=None*, *union=False*)
Create a new dissector and protocol if needed.

generate()
Returns all the code for dissecting this protocol.

_legal_header()
Add the legal header with license info.

_header_defintion()
Add the code for the header of the protocol.

_fields_definition()
Add code for defining the ProtoField's in the protocol.

_dissector_func()
Add the code for the dissector function for the protocol.

_register_dissector()
Add code for registering the dissector in the dissector table.

class dissector.**Delegator**(*platforms*)
A class for delegating dissecting to protocols.

Creates the top-level lua dissector which delegates the task of dissecting specific messages to dissectors generated by Protocol instances.

This top-level dissector contains code for finding the platform the message originates from, and finds which specific dissector handles that platform and message.

generate()
Returns all the code for dissecting this protocol.

_header_defintion()
Add the code for the header of the protocol.

_register_function()
Add code for register protocol function.

_dissector_func()
Add the code for the dissector function for the protocol.

2.4.7 platform

A module which holds platform specific configuration.

It holds the Platform class which holds specific configuration for one platform, and a list of all supported platforms.

It is used when creating dissectors for messages which can originate from various platforms.

class `platform.`**`Platform`**(*name*, *flag*, *endian*, *macros=None*, *sizes=None*, *alignment=None*, *types=None*)

Represents specific attributes of a platform.

Platform here refers to a combination of Operating System, Hardware platform and Compiler. An instance of this class is an abstraction of all of these. It inceases the number of platforms if one wish to support many, but the utility only need to handle one at a time.

`big` = **'big'**

`little` = **'little'**

`mappings` = **{}**

`flags` = **{}**

`map_type`(*ctype*)

Find the Wireshark type for a ctype.

`size_of`(*ctype*)

Find the size of a C type in bytes.

`alignment`(*ctype*)

Find the alignment size of a C type in bytes.

`platform.`**`merge`**(*a*, **dicts*)

Merge several dictinaries into a new one.

2.5 Changing documentation

2.5.1 Documentation files in your local checkout

Most of the CSjark's documentation is kept in *CSjark/docs*. You can simply edit or add '.rst' files which contain ReST-markuped files. Here is a ReST quickstart but you can also just look at the existing documentation and see how things work.

2.5.2 Automatically test documentation changes

We automatically check referential integrity and ReST-conformance. In order to run the tests you need sphinx installed. Then go to the local checkout of the documentation directory and run the Makefile:

```
cd CSjark/docs
make html
```

If you see no failures chances are high that your modifications at least don't produce ReST-errors or wrong local references. Now you will have *.html* files in the *_build* documentation directory which you can point your browser to!

Additionally, if you also want to check for remote references inside the documentation issue:

```
make linkcheck
```

which will check that remote URLs are reachable.

2.5.3 Automatic builds on Read The Docs

The CSjark project documentation is hosted at ReadTheDocs. Each commit to the repository triggers a new build of the documentation, therefore it always remains up to date.

Note: Due to lack of support of the latest version of Python language specification (v3) by ReadTheDocs, the online developer manual does not contain source code overview section.

CHAPTER THREE

OTHER INFORMATION

3.1 Copyright

3.2 License

3.3 About these documents

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

These documents are hosted at *ReadTheDocs <http://www.readthedocs.org/>*. You can find it at http://csjark.readthedocs.org/.

APPENDIX G

TEMPLATES

Meeting agenda

TDT4290 – Customer Driven Project
Group 9

Place: Thales
Time: 9:00 - 11:00
Date: 9.11.2011

Meeting responsible: Sigurd Wien,   47254625

Agenda:

1. Presentation and demo 9:00-10:00
2. Status on "the fix".
3. Requirement completion agreement.
4. Customers agenda.

Attending (from group 9):

Sigurd
Jaroslav
Sondre
Erik
Lars
Even
Terje

Meeting minutes

TDT4290 – Customer Driven Project
Group 9

Room: 236, IT-Syd
Time: 10.30-12.00
Date: 14.10.2011

Attendants group:

● Erik Bergersen,	91748305
● Jaroslav Fibichr,	45126314
● Sondre Johan Mannsverk,	94815506
● Terje Snarby,	91527390
● Even Wiik Thomassen,	99161929
● Lars Solvoll Tønder,	97600317
● Sigurd Wien,	47254625

Not present: None.

Attendants staff:

● Daniela Soares Cruzes,	94249891
● Maria Carolina Mello Passos	

Agenda:

1. Approval of agenda
2. Approval of minutes of meeting from last advisor meeting
3. Comments to the minutes from last customer meeting or other meetings
4. Approval of status report
5. Review/approval of attached phase documents
    - a. User stories
    - b. Daniela's thoughts about our predelivered report
    - c. Sprint 2
6. Other issues

Minutes:

- Carol says they want to focus on problems we may have.
    - User stories

        - They are user stories with an implementation view
        - Our program is a utility, not a common, commercial product
        - The user is a system analyst
        - Even asks if what we have made is actual user stories, right now they are more like how we implemented it. He also says they are more for us than the user.
        - Carol thinks we are on the right track with the user stories
        - Daniella says they are called user stories because they are stories that tell how the system will be used.
        - We can write that we have used an implementation level of abstraction
        - She has talked with Andreas, and he also thinks our way of writing the user stories is okay.
        - We can make them even more detailed though
        - We don't need to have just one user story per requirement, we can have multiple
    - Sprint planning document
        - We need to explain more how we are doing the planning
        - Carol talks about how they usually wrote planning documents
            - They had 2 plans: One is the selection of tasks, what tasks should be implemented in the upcoming sprint. This could be influenced by customer priority, what the team think is important to implement in the sprint, the size of the task, and the number of hours the group can spend total.
            - The second plan is how the team is actually planning to do it.

….

Next meeting:

Date: 21.10.2011
Time: 10.30
Room: 236, IT-Syd

Referent: Sondre Johan Mannsverk

Status report

TDT4290 – Customer Driven Project

Group 9

Week: 12

Summary:

Since last advisor meeting the group has been focusing on:

1. Sprint 4
    a. Completing the backlog; finishing last implementation in the project, patching old implementation to make it work as intended, testing that everything works and write good user documentation.
2. Presentation
    a. We used quite some time to prepare for the Thales presentation. Sigurd and Terje had the task of creating the presentation and performing it.
3. Report
    a. We have made our own backlog for the things that we have to change/improve in the backlog. Some of the team members have been working on this in parallel with the sprint work period. The sprint backlog show that we have completed 335 hours of work, but the actual work hours are much greater.

Work done in this period:

Documents:

- Sprint 4
- Introduction section
- Report (many changes)

Meetings:

15.11.2011: Sprint 4 evaluation meeting

Other activities:

Internal work meetings

Problems:

None.

Planning of work for next period:

Meetings:

- Rehearsal meeting with advisors, date: TBA
- Work meetings most of the days next week to complete all the work that remains.

Activities:

Presentation 24.11.2011

Other:

The end is near!