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>
*/
```

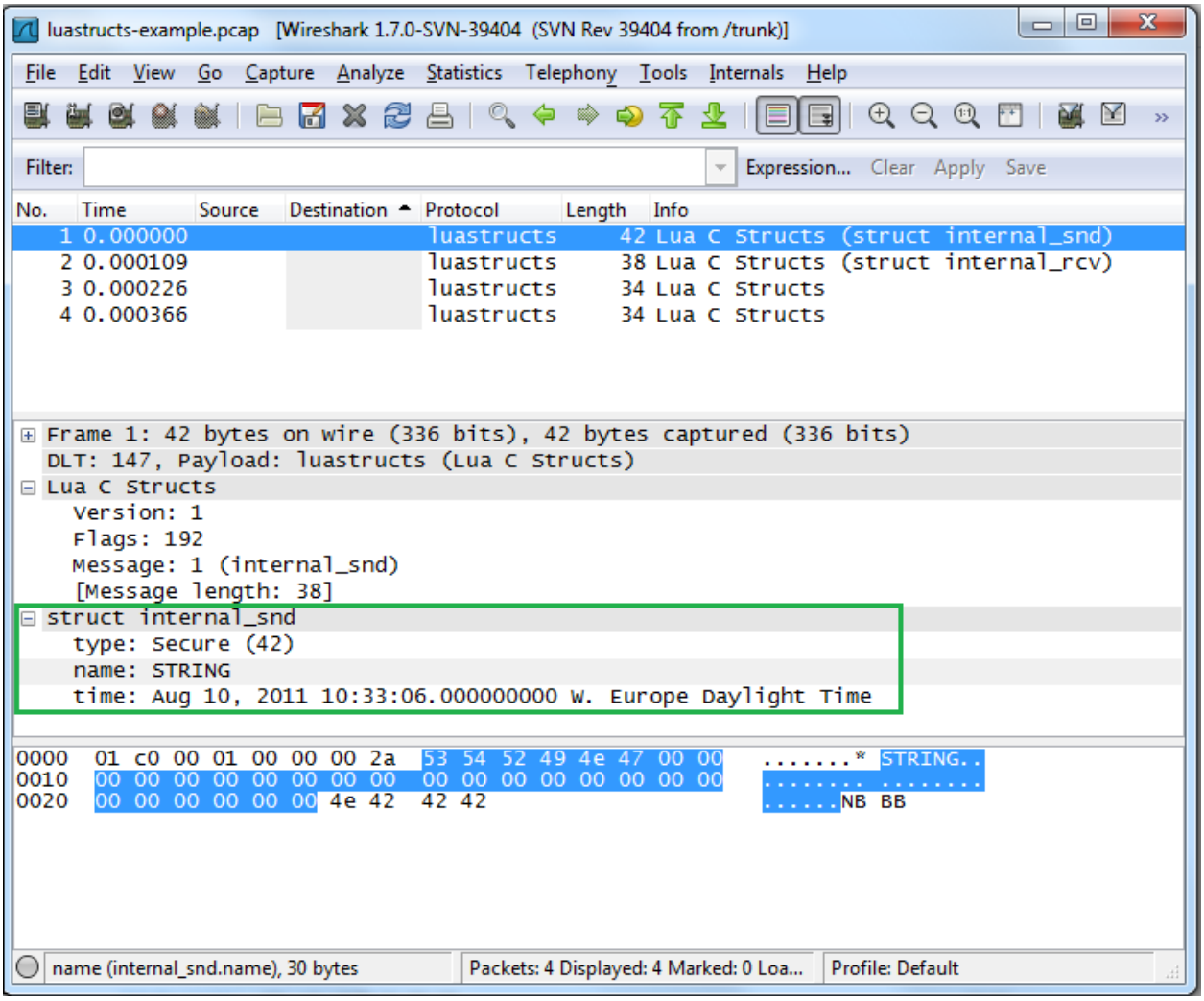


Figure 1.1: *Figure 1:* Wireshark

```
#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.

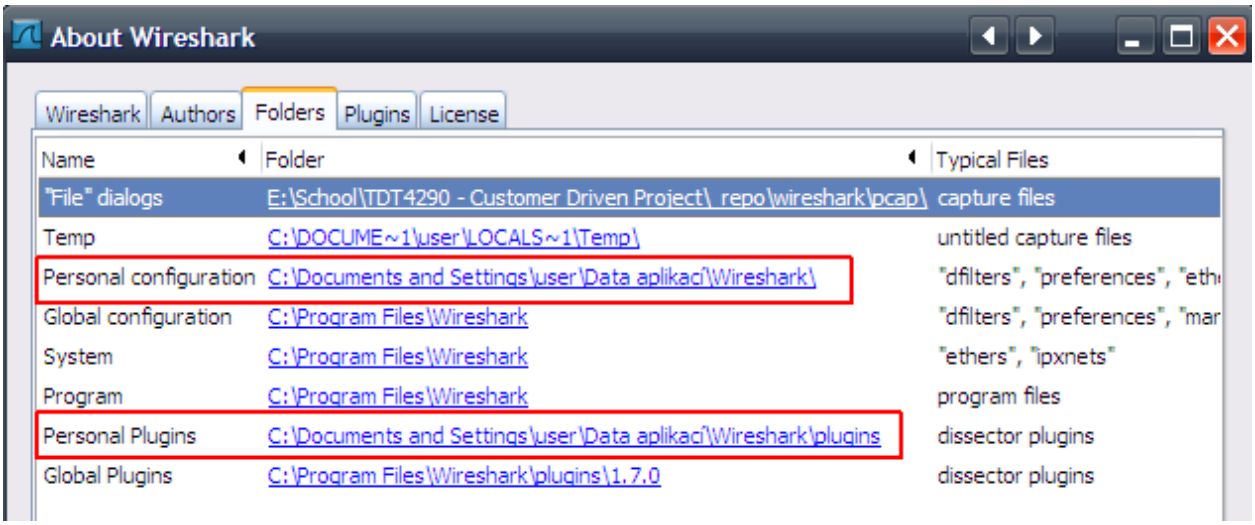
-A [argument [argument ...]], **-Additional** [argument [argument ...]]
Any additional C preprocessor arguments.

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).



   - 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.

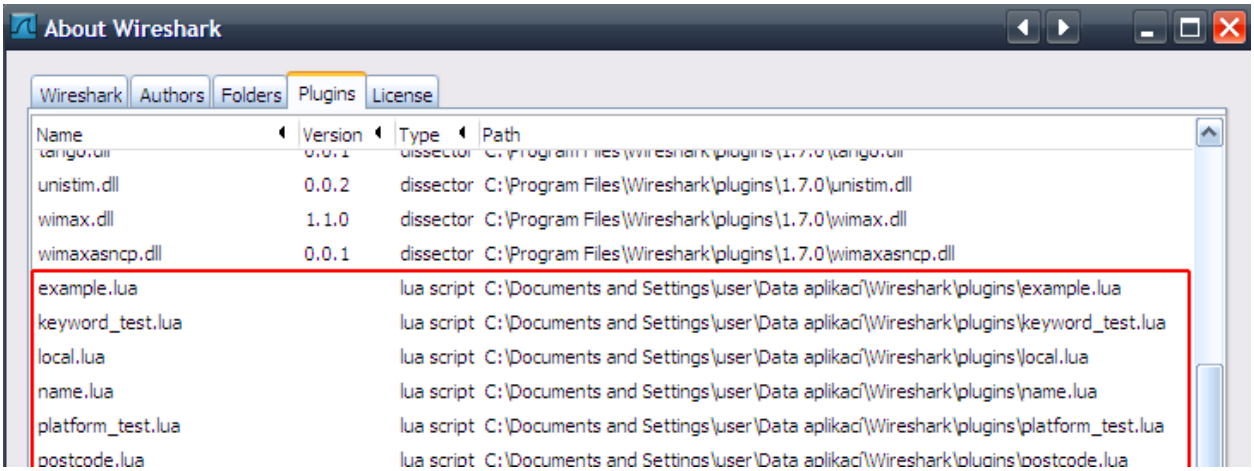
Note: Location of CSjark generated files is given by `-o` command line argument. More in section *Using CSjark*.

---

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".



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

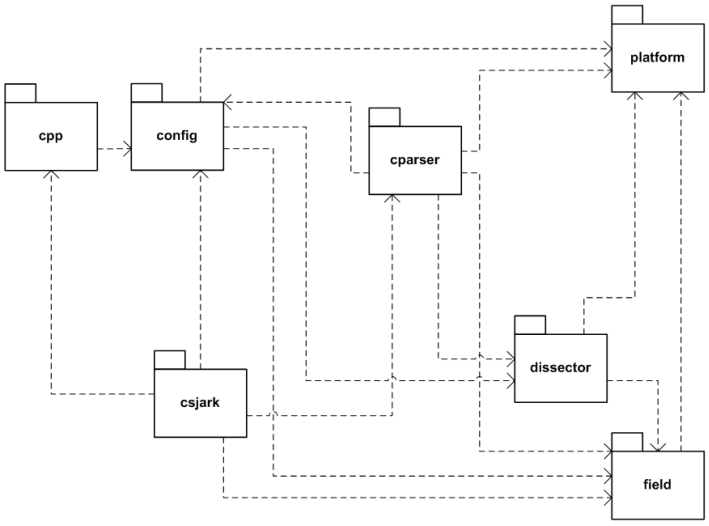


Figure 2.1: *CSjark: module diagram*

2.2.3 Class diagram

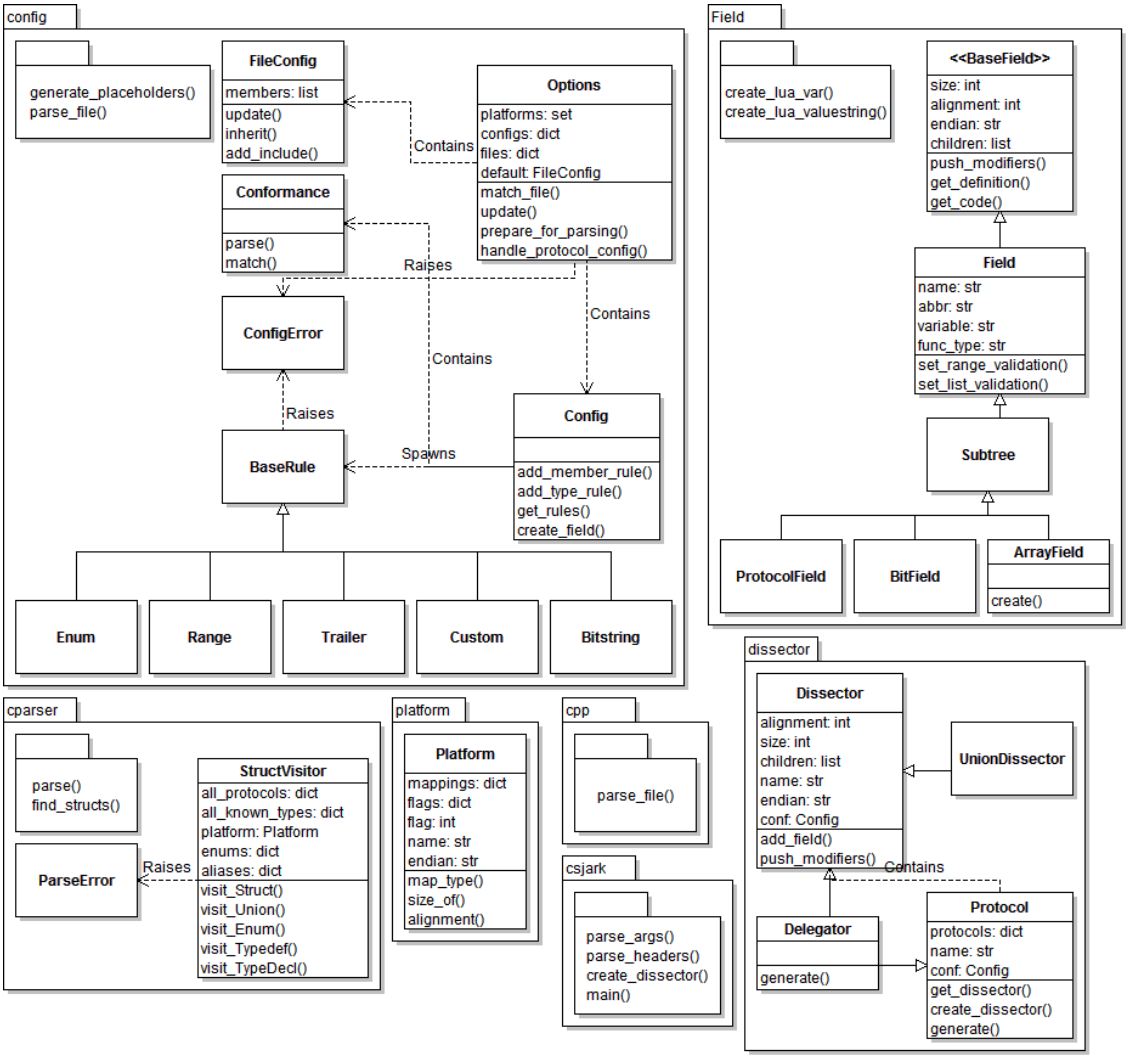


Figure 2.2: *CSjark: class diagram*

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/.