CSjark Documentation

Release 0.3.1

Even Wiik Thomassen, Erik Bergersen, Sondre Johan Mannsverk,

November 13, 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. The goal with the dissectors is to make Wireshark able to nicely display the values of a struct sent over the network, along with member names and type. This can be a powerful tool for debugging C programs that communicates with strucs over the network.

For more information about Wireshark please visit http://www.wireshark.org.

Features (TBD)

- C header files
- Batch mode
- Searching and filtering in Wireshark
- ...

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)

CHAPTER

ONE

USER DOCUMENTATION

1.1 Installing CSjark

1.1.1 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 three 3rd party dependencies to get CSjark working:

1. **PLY** (Python Lex-Yacc)

   Required version 3.4

   Download location http://www.dabeaz.com/ply/

   Notes 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.

2. **pycpaser** <http://code.google.com/p/pycparser/>‘_

   Required version latest development version from pycparser repository

   Download location pycparser repository: http://code.google.com/p/pycparser/source/checkout

   Notes 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).

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**

   Required version 3.10

   Download location http://pyyaml.org/wiki/PyYAML

   Notes 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.

5. **Wireshark**

   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

Note 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.

1.1.2 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*.

1.2 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 ...]]] [-o [output]] [-p] [-n] [-C [cpp]]
                      [-i [header [header ...]]] [-I [directory [directory ...]]]
                      [-D [name=definition [name=definition ...]]]
                      [-U [name [name ...]]] [-A [argument [argument ...]]]
                      [header] [config]
```

The arguments here specify the following:

`header` a c header file to parse.

`config` a configuration file to parse.

Optional arguments:

`-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 [header [header ...]], --file [header [header ...]]`	Specifies that CSjark should look for struct definitions in the `header` files.
`-c [config [config ...]], --config [config [config ...]]`	Specifies that the program should use the `config` files as configuration.
`-o [output], --output [output]`	Writes the output to the specified file `output`.
`-p, --placeholders`	Generate placeholder config file for unknown structs
`-n, --nocpp`	Disables the C pre-processor.
`-C [cpp], --CPP [cpp]`	Specifies which preprocessor to use.
`-i [header [header ...]], --include [header [header ...]]`	Process file as Cpp `#include "file"` directive
`-I [directory [directory ...]], --Includes [directory [directory ...]]`	Directories to be searched for Cpp includes
`-D [name=definition [name=definition ...]], --Define [name=definition [name=definition ...]]`	Predefine name as a Cpp macro
`-U [name [name ...]], --Undefine [name [name ...]]`	Cancel any previous Cpp definition of name
`-A [argument [argument ...]], --Additional [argument [argument ...]]`	Any additional C preprocessor arguments

Example usage:

```
python csjark.py -v headerfile.h configfile.yml
```

1.3 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 1.7 dev or later as described in the installation section.
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 Personal configuration Personal Plugins and and note the folder paths. On a *nixbased system this may be ~/.wireshark/ and ~/.wireshark/plugins/ and on Windows it may be C:Users*YourUserName*AppDataRoamingWiresharkand C:Users*YourUserName*AppDataRoamingWiresharkpluginsIf the folders does not exist, create them.
3. Copy the `1-luastructs.lua` into the Personal configuration directory located in the previous step.
4. Copy your generated lua dissector to the Personal Plugins folder located in the first step.
5. Open the `init.lua` in the Personal configuration directory located in step 1 (create one if it does not exists). Insert the following code: dofile(`1-luastructs.lua`) . This ensures that the 1-luastructs.lua is loaded before the lua scripts in the Personal Plugins folder.
6. Restart Wireshark.

To check that the scrips are loaded, click on the `Help` button in the menubar and choose `About`. Navigate to the `Plugins` tab. The scripts should now appear in the list.

To add further dissectors, only step 2, 5 and 6 needs to be repeated.

For further information on the lua integration in wireshark, please visit: http://www.wireshark.org/docs/wsug_html_chunked/wsluarm.html

1.4 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

1.4.1 Configuration format

The configuration files are written in YAML which is a data serialization format designed to be easy to read and write. The format of the files are described below. The configuration should be put in a `filename.yml` file and specified when running CSjark with the `--config` command line argument.

Detailed specification can be found at YAML website.

The only part of configuration that is held directly in the code is the platform specific setup (file `platform.py`).

1.4.2 Configuration definitions

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 range or not. Value ranges are defined by the following syntax:

```
RangeRules:
  - struct: "Name of the struct"
    member: "Name of datameber"
    min: "Lowest allowed value"
    max: "Highest allowed value"
```

or, one could specify a type, and apply the value range to all the members of that type within the struct:

```
RangeRules:
  - struct: "Name of the struct"
    type: "Name of the type"
    min: "Lowest allowed value"
    max: "Highest allowed value"
```

Example:

```
RangeRules:
  - struct: example_struct
    member: percent
    min: 0
    max: 100

  - struct: example_struct
    type: int
    min: 0
    max: 100
```

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 2 mandatory and 1 optional value

```
- 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`)

Member Config Example of Struct definition with member named `weekday` and values defined as a list of key-value pairs.

```
Structs:
  - name: enum_example1
    id: 10
    description: Enum config example 1
    enums:
      - member: weekday
        values: {1: MONDAY, 2: TUESDAY, 3: WEDNESDAY, 4: THURSDAY, 5: FRIDAY, 6: SATURDAY, 7: SUNDAY}
```

Type config In this example we can see 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_example2
    id: 10
    description: Enum config example 2
    enums:
      - 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.

Member Config Below, there is an example of a configuration for the flags member of the struct example. 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.

```
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 Config This example specifies a bitstring for all data types of short.

```
Structs:
  - name: example
    id: 1000
    description: An example
    bitstrings:
      - type: short
        0: Red
        1: Green
        2: Blue
```

Dissector ID

In every struct-packet that Wireshark captures, there is a header. One of the fields in the header, the `id` field, specifies which dissector should be loaded to dissect the actual struct. This field can be specified in the configuration file. If no configuration file is given, the packet will be assigned a default dissector.

This is an example of the specification

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

One struct can be also dissected by multiple different dissectors. Therefore, it can contain a whole list of dissector 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.

More information how to write Lua code can be found in Lua reference manual.

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:

- The conformance file (as well as CSjark configuration files) follows YAML syntax specification.
- 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 defintion.
`DEF_BODY id`	Lua code to replace a Field defintion. 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 defintion
`DEF_EXTRA`	Lua code added after the last defintion
`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 (`DEF_*`) or field name (`FUNC_*`).

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


#.DEF_EXTRA
-- This was all the Field defintions
#.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 defintions 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 defintions

-- 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 add new proto fields to the dissector in Wireshark, 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.

Configuration of various trailers

CSjark only creates dissectors from c-struct, 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 trailer can be configured for a packet.

The following parameters is allowed in trailers:

- name: The 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, specifiy the total number of trailers or give a variable in the struct, which contains the amount of trailers. The two ways to configure trailers are listed below.

```
trailers:
  - name: "protocol name"
  - member: "variable in struct, which contain amount of trailers"
  - size: "size of the buffer"

trailers:
  - name: "protocol name"
  - count: "Number of trailers"
  - size: "size of the buffer"
```

Example: The example below shows an example with BER [1], which av 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, framenum

`Other types` float, double, string, stringz, bytes, bool, ipv4, ipv6, ether, oid, guid

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

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

[1] Basic Encoding Rules

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`).

1.4.3 Platform specific configuraion

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

CHAPTER

THREE

OTHER INFORMATION

3.1 What's New In CSjark 0.2.3

Author Even

Release 0.3.1

Date November 13, 2011

This article explains the new features in CSjark compared to previous version.

3.2 Limitations

CSjark currently have no way of specifying different type sizes or endian for different platform, and will therefore only produce correct dissector for a platform that conforms with our default values and communicates with a computer with the same endian and type sizes.

There are currently no support for typedef types except structs.

CSjark does not yet support struct members of type long double, as Wireshark does not support it.

3.3 License & Warranty

License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Warranty

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED

TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

3.4 Copyright

See *License & Warranty* for complete license and permissions information.

3.5 About these documents

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*