Network Packet Capture Generation and Falsification

Elliot Thomas (w1372638)
Supervisor: Sean Tohill

University of Westminster

June 28, 2015

BSc Computer Science (Honours)
Computer Science Project (ECSC699)

Abstract

A document describing the design and development of a system to assist in the crafting of network packet captures for educational purposes.

This report is submitted in partial fulfillment of the requirements for the BSc (Hons) Computer Science Degree at the University of Westminster.

List of Figures

Listings

Contents

Acknowledgements

This document was typeset using LaTeX.

With thanks to Andy Thomas for proofreading, fellow students for keeping the author sane, tex.stackexchange.com for typesetting help and the Python Software Foundation for providing such a wonderful programming language.

Chapter 1

Introduction

Education is important. Education is one of the underpinnings of a progressive society, along with law and order and a few other things. Despite this, it seems human ingenuity is education's greatest enemy. Humans are lazy - most of us will strive to do as little work as possible for the greatest gain. We seek to make our lives better, easier, all to spend more time on the things we enjoy. For many, this will be leisure activities, for some, socialising, the humble few may take joy in altruism.

But motivations are irrelevant here, just the effects. In academia, plagiarism is a problem - people are unwilling to put the time in to do their own work. They realise that if someone else has solved a problem identical to theirs, they could just reuse that. What these people fail to realise, is that there is no substitute for experience. This problem is relevant in all fields, academic or applied, practical or theoretical; computer security and forensics is no exception.

1.1 Reader level

The reader is expected to be somewhat familiar with networking terminology and forensic analysis techniques, in addition to having some background in programming. Knowledge of specifically Python will be required to understand the design and implementation chapters.

1.2 The problem

Teaching network security and forensics is aided greatly by the presentation of example packet captures. These are used to learn and practice analytical skills, and to assess the level of understanding and knowledge that students hold. Creating such packet captures is not difficult, but there are few, if any tools available for automating the process. Changing this is the purpose of this project.

1.3 Existing Solutions

None, it would seem.

There are a number of tools for network packet capture, and a number of tools for analysing them. These tools do not solve the problem *per se*, but do allow packet capture manipulation to a certain extent, and therefore might be a useful aid in manipulating packet captures.

1.3.1 wireshark

Wireshark is packet sniffer and analysis tool[1], and contains a suite of utilities (some of which are mentioned below).

Wireshark is typically the tool that students would use to analyse packet captures in this scenario, and as such is an invaluable tool for verifying correctness.

As a tool for falsifying captures, it does provide one useful feature - the ability to change the apparent time a packet arrived.

1.3.2 editcap

Editcap is a program distributed as part of *Wireshark*.
From the manpage[2]:

> **Editcap** is a program that reads some or all of the captured packets from the infile, optionally converts them in various ways and writes the resulting packets to the capture outfile (or outfiles).

For the purposes of assisting generation, this program allows splitting a file into a series of files, according to the time they were sent; i.e., splitting every 2 seconds (where each set represents a packet capture and each number represents the time a packet arrived):
(1, 1, 2, 3, 5, 7, 8) → (1, 1, 2), (3), (5), (7, 8)

1.3.3 tshark

Tshark is a program distributed as part of *Wireshark*.
From the manpage[3]:

> **TShark** is a network protocol analyzer. It lets you capture packet data from a live network, or read packets from a previously saved capture file, either printing a decoded form of those packets to the standard output or writing the packets to a file. **TShark**'s native capture file format is **pcap** format, which is also the format used by **tcpdump** and various other tools.

As stated, it acts as a network traffic capture and analysis tool. It has functionality that will identify packets by protocol (and can reconstruct TCP streams) and allows the user to filter them. This is useful functionality, as it allows you to remove packets based on the protocols they contain, making captures simpler to understand.

1.3.4 Bit-Twist

Bit-Twist is a tool suite containing a packet capture 'generator' (a traffic replay program, not unlike tcpreplay[4]) and a pcap editor. From the website[5]:

> With Bit-Twist, you can now regenerate your captured traffic onto a live network! Packets are generated from tcpdump trace file (.pcap file). Bit-Twist also comes with a comprehensive trace file editor to allow you to change the contents of a trace file.

This tool is *very* close to the desired utility. It permits editing a packet capture in a handful of ways, specifically:

- Appending a 'payload' (arbitrary bytes) to the end of every packet.
- Removing a range of bytes from every packet.
- Recalculating checksums for (non-fragmented) IP, TCP, UDP and ICMP packets.
- Saving a specific range of packets either by their occurence (i.e. The fourth packet to the seventh packet) or by their timeframe (i.e. all packets between 2006/10/1 00:00:00 and 2006/10/31 10:30:00).
- Truncating packets to a specific 'layer' - where 2 is the link layer, 3 is the network layer and 4 is the application layer.
- Restricting edits to a specific 'header'- Ethernet, ARP, IP, ICMP, TCP and UDP are supported in this case.

1.4 Conclusions

There are tools that can help, but nothing particularly geared towards the problem. Most existing tools present a kind of 'read-only' interface toward existing captures, or allow modifying them while preserving packet integrity. Bit-Twist's pcap editor allows arbitrary modification, but is somewhat cumbersome to use, applying the same changes on all packets.

Chapter 2

Research

2.1 Approaches to the Problem

The problem '*How does one generate unique packet captures for teaching analysis to students?*' is open to multiple solutions. In this project, three possible approaches were considered. These can be described as *synthesis*, *generation* and *composition*.

2.1.1 Synthesis

This would work as a program that understands a *scenario* defined in a kind of declarative language, that creates a set of unique permutations of that scenario by varying some key data (names, hosts, times etc.) and the exact sequence of events. From this, it would synthesise a complete packet capture from each scenario - carefully constructing every packet - such that all generated captures represent the same abstract event (i.e. corporate espionage) but with wildly different specifics, hindering collusion.

This approach has the advantage of a flexible interface and fast implementation - at the cost of having to understand and reproduce the complexity of various networking protocols and producing a completely artificial output.

2.1.2 Generation

Much like *synthesis*, this would understand a declarative language for describing a scenario, but this approach would be to program a network of computers to actually do the actions in the scenario rather then synthesise their side effects. This would then run a packet sniffer on this network, generating perfectly authentic network captures of a real network, but where the human actors in a scenario are emulated by computers.

This approach has the advantage of flexibility and authenticity - at the cost of being very inefficient and fragile. It is more then likely such an approach would use a network of virtual machines running real-world operating systems and protocol implementations. This introduces a fair amount of unpredictability (which is both a good and bad thing), and would require changes if/when a used interface changes.

2.1.3 Composition

This was the first approach considered, and is perhaps the simplest. This would be a program that takes a set of existing packet captures (ideally, small purpose-built captures representing single transactions) and combines them while altering things like hosts, times and the like.

This approach has the advantage that real-world scenarios can be used - an existing library

of captures will still be useful to provide a data source for the program - and it retains a degree of authenticity. It has a lesser variant of the downside that *synthesis* has, in that it will need to identify and modify various networking protocols, but does not need to understand how they work - which removes a great deal of complexity. It also depends on existing captures - if these aren't available, the tool cannot function.

2.1.4 Chosen Approach

Synthesis and *generation* approaches are very complex - a good implementation would take a considerable amount of time to write and test, and allow a lesser degree of control to the user. For this reason, a *composition* based approach was chosen - it is comparatively simple and easy to understand; desirable qualities for any tool.

2.2 Programming language: Criteria

Choosing a programming language is, obviously, a decision that needs to be made, and there is no single correct choice. There are many factors influencing such a decision, each one needs to be considered. There were three major factors considered in this decision:

i) Requirements - what does the program require?

ii) Knowledge - how difficult will it be to program in and maintain?

iii) Availability - on what platforms can projects using this language be used?

2.2.1 Requirements influence

At its heart, this project is a data processing project. There are no requirements for real-time processing, no requirements for concurrency, no requirement to make use of or implement a specific API nor requirements for any kind of interactive behaviour. This project can very easily be designed and implemented as a batch program - give input, run program, get output.

Given the minimalistic requirements of the program, just about any Turing-complete language is serviceable. As such, the decision will have to be made predominantly on the other three factors. That's not to say entirely, some languages lend themselves quite well to generic data manipulation, while others are more specialised and geared towards specific purposes. For instance, Javascript is more suited towards web-based projects (given that its usual interpreter is a web browser), while C, Java and Python are more general purpose. Haskell, Java and Python have good support for abstract data structures while various assembly languages barely have the notion of a data type.

One important factor is for the program to be easily extended. While it is possible to write an extensible program in just about any language, it helps considerably if there is support for loading code from an external source at runtime, i.e. to allow building a plugin architecture. Most languages or platforms allow this in some manner (and practically all platforms depend on doing this in one way or another - this is how shared libraries work). While POSIX platforms provide functions like `void *dlopen(const char *filename, int flag);`, this interface is not consistently available on non-POSIX platforms, and requires code to be compiled first. Meanwhile, Python, Javascript and even Bash provide methods for loading new code programatically without the need to compile code first, allowing for a very flexible, powerful and user-friendly system to be developed.

2.2.2 Knowledge influence

Knowledge is a subjective factor that pertains to the programmer(s) developing the project. It is a limiting factor; a programmer proficient in C is not necessarily going to be able to understand a very different language such as Haskell. Also worth mentioning is the preference of the programmer; while this is not an overriding factor it can help shape a decision.

2.2.3 Availability influence

Implementation availability for most languages is somewhat of a non-issue. Given that portability is a desirable outcome, only languages which have a usable and consistent enough implementation across platforms will be considered. This eliminates some languages such as C#, Visual Basic (or any .NET language).

Regarding portability, are there provisions for file input/output? In the case of the various assembly languages, this is left to the programmer. Even if the assembly language itself is abstract and portable, it only dictates how the processor is used - assembly often leads to some *very* specific code.

```
mov rax, 1    ; system call constant for write on Linux64
mov rdi, 1    ; first argument: file descriptor to write to, eg: stdout.
mov rsi, msg  ; second argument: buffer holding data to be written.
mov rdx, len  ; third argument: number of bytes to write.
syscall       ; Do the system call. Similair to int 0x80.
```

Listing 2.1: *Written for x86-64 processors, this uses Linux's 64-bit system call convention to write data to 'stdout' (standard output)*

2.3 Programming Language: Assessment

Many languages were assessed for use in this project using the criteria above, with some being discarded on a single fatal flaw. These are:

- C#, Visual Basic, other .NET languages: These are fairly portable with the advent of Mono[12], but the author still has reservations about this.
- Haskell: An interesting language, but requires a substantially different approach to program design than the author is familiar with.
- PHP, Javascript: Languages that are perfectly serviceable for the task at hand, but have a heavy focus on web development rather then packet capture processing. A more neutral choice would be better.
- Assembly: These languages are very low level and require a great deal of effort to accomplish simple tasks. Additionally, they tend not to be portable.
- COBOL: An affront to language design, this is on the list purely as humour.

2.3.1 C

For a while, C was the *lingua franca* of the programming world. C is a fairly old language, dating back to 1972 and was used to write many parts of the UNIX operating system[6].

C is technically considered a high-level programming language, in that it gives you automatic variable allocation and permits arbitrarily complex arithmetic and logical expressions without explicitly storing intermediary results. The detail of a register (memory that is part of the CPU) is seldom expressed in C code; variables have an addressable place in main memory unless the register keyword is used in variable declaration.

C has the advantage of very little overhead - there is no garbage collector; all memory management is done manually. Given that networking protocols, particularly the ones relevant to this project, are often implemented in C; C seems like a fair choice despite its shortcomings.

Perhaps the biggest issue with using C is work to functionality ratio. Data structures available in other languages like linked lists and hash tables require implementing, and code generally has to be quite explicit. There are no abstractions such as 'iterate over every element in this array'

Although an optimising compiler *can* do this for you.

provided by default (although this is possible to some extent with preprocessor macros), and C is not an object oriented language - there is no real notion of classes, which are (if used correctly) a useful abstraction.

C is a compiled language, which means the project requires compilation to a platform specific executable. While the code in itself is no less portable for this, it does require testing on compliers for all supported platforms - most of which are not completely compatible with each other.

C++

Often described as 'C with classes', C++ is a derivative (for the most part, a superset) of the C programming language[7]. C++ adds classes, namespaces, 'references' and templates to the C language. For the most part, it retains compatibility with the C language and can be freely mixed with C code.

C++ resolves some of the issues with using C for this project. However, the remaining issues of memory managment and a fairly 'low-level' interface make C/C++ difficult to choose.

2.3.2 Java

Java is a language usually compiled to *bytecode* and then interpreted by a virtual machine, the Java Virtual Machine (JVM). The virtual machine accepts programs in a standardised form, so Java bytecode can work on multiple implementations. This means that Java programs, once compiled, can be run anywhere. Java is a very popular language[9], used in enterprise environments and commonly taught to university students doing a course in computer science.

Java is a strongly typed, partially object oriented language - it supports all typical object oriented abstractions, but also supports *primitive types* and treats classes and methods differently to other objects (you cannot assign them like other objects).

Java has the advantage of executable portability that C and C++ do not, and is almost perfect for this task. It supports loading bytecode at runtime and has support for file I/O.

The only real issue is somewhat subjective - the author finds Java code needlessly verbose. This is discussed in the verbosity section (2.3.4).

2.3.3 Python

Python is an interpreted, multi-paradigm, fully object oriented language, developed and promoted by the Python Software Foundation[10]. Python also has its own philosophy - solutions are often deemed to be *pythonic* based on various factors such as elegance and being obvious. However, in Python, mechanism is separate from policy; the various Python conventions are exactly that - conventions.

Python code is usually distributed in source form, but can be compiled to a bytecode not unlike Java. Python as a language does not enforce any kind of information hiding or encapsulation, and does *not* require nor support typed variables. This is in contrast to Java, which provides these kind of access qualifiers. Instead of these, a convention is used - private and protected access qualifiers are signified by a single underscore before member names, and API documentation is generally used to specify argument and return types for functions and methods.

2.3.4 Code Verbosity

Code verbosity is a problem in certain languages, and can make even simple code hard to follow - the less you have to read, the quicker you can comprehend it. To illustrate this problem, consider

It should be noted that other compilers exist - GCJ[8] is such an example that compiles Java into machine code. Note that this does not mean python is weakly typed - Python has strong typing that requires explicit conversion.

a simple task: write a program that counts the number of lines in a text file. In the interest of fairness, the program will have specific behaviour; it should accept standard input or a file path as its first command line parameter, and work by looking at each character in the file and comparing it to a line feed. If reading from a file that does not exist, it should print an error message. Additionally, all variable identifiers should be the same or similar when translated to code.

This program was fully implemented for the three considered languages - C, Java and Python. The C implementation is as follows:

Listing 2.2: *Number of Lines program, in C*

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      FILE *f;
6      int lines = 0;
7      int chr;
8
9      if(argc > 1)
10     {
11         f = fopen(argv[1], "r");
12     }
13     else
14     {
15         f = stdin;
16     }
17
18     if(f == NULL)
19     {
20         printf("Error: %s - file not found.\n", argv[1]);
21         return 1;
22     }
23
24     do
25     {
26         chr = fgetc(f);
27         if(chr == '\n')
28         {
29             lines++;
30         }
31     } while(chr != -1);
32
33     fclose(f);
34
35     printf("Number of lines: %d\n", lines);
36     return 0;
37 }
```

The C version's program source code consists of 38 lines and occupies 514 bytes of storage. This version of the program produces the correct output of 38 when run on its own code, 50 and 22 when run on the other two versions' code.

The author recognises that code style has an impact on the metrics used (line count and source file size), and that all of these programs can be rewritten to occupy less space, however doing so would render them nigh unmaintainable.

The Java version, with full exception handling as requested by the Java compiler, looks like this:

Listing 2.3: *Number of Lines program, in Java*

```
1  import java.io.InputStream;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5
6  public class nol
7  {
8      public static void main(String[] argv)
9      {
10         InputStream f;
11         int chr;
12         int lines = 0;
13
14         try
15         {
16             if(argv.length > 1)
17             {
18                 f = new FileInputStream(argv[1]);
19             }
20             else
21             {
22                 f = System.in;
23             }
24
25             do
26             {
27                 chr = f.read();
28                 if(chr == '\n')
29                 {
30                     lines++;
31                 }
32             } while(chr != -1);
33
34             f.close();
35
36             System.out.printf("Number␣of␣lines:␣%d\n", lines);
37             System.exit(0);
38         }
39         catch (FileNotFoundException ex)
40         {
41             System.out.printf("Error:␣%s␣-␣file␣not␣found.\n", argv[1]);
42             System.exit(1);
43         }
44         catch (IOException ex)
45         {
46             System.exit(1);
47         }
48     }
49 }
```

The Java version is considerably larger, consisting of 50 lines and is 1113 bytes large. This version also produces correct behaviour.

The Python version, in Python 3, looks like this:

Listing 2.4: *Number of Lines program, in Python*

```
1  import sys
2
3  if len(sys.argv) > 1:
4      try:
5          f = open(sys.argv[1], "r")
6      except FileNotFoundError:
7          print("Error:␣%s␣-␣file␣not␣found." % sys.argv[1])
8          sys.exit(1)
9  else:
10     f = sys.stdin
11
12 char = f.read(1)
13 lines = 0
14
15 while char != '':
16     if char == '\n':
17         lines += 1
18     char = f.read(1)
19
20 print("Number␣of␣lines:␣%d" % lines);
21 sys.exit(0)
```

The python version is the smallest, at just 22 lines and 371 bytes in size. Again, correct behaviour is observed.

2.3.5 Chosen Language: Python

Time was the unspoken major factor in this decision, and given the liberty to choose any programming language means ones in which the programmer has experience will take precedence. This means that languages designed for unfamiliar paradigms (such as Haskell, being a purely functional programming language[11]) or languages that leave a considerable amount of work to their user (such as Assembly) were eliminated.

The choice was Python. This was driven by its status as a multi-paradigm language that offers different models of program abstraction. To illustrate this, shown below are two functionally equivalent and compatible prime number *iterators*;

Listing 2.5: *An object oriented approach to a prime generator.*

```
class primegen:
    def __init__(self, start, end):
        self.cur = start
        self.lim = end

    def __iter__(self):
        return self

    def __next__(self):
        prime = False
        while not prime:
            if self.cur >= self.lim:
                raise StopIteration()
            for i in range(2, self.cur/2):
                if self.cur % i == 0:
                    break
            else:
                retval = self.cur
                prime = True
```

```
            self.cur += 1
        return retval
```

Listing 2.6: *An imperative approach to a prime generator.*

```
def primegen(cur, lim):
    while cur < lim:
        for i in range(2, cur/2):
            if cur % i == 0:
                break
        else:
            yield cur
        cur += 1
```

2.3.6 Python 2 or Python 3?

Like most languages, Python has versions. The Python language currently exists as two distinct versions, Python 2 and Python 3. This is not to be confused with the multiple implementations of Python - which can support the same language and standard library - but the actual grammar of the language.

What is wrong with Python 2?

As a language, Python 2 made some odd decisions. Firstly, it had a 'print' statement - which sounds perfectly reasonable until you need to accommodate for edge cases and specific behaviour. Print statements have the form;

⟨*expressions*⟩ ::= ⟨*expression*⟩ | ⟨*expression*⟩ '`,`' ⟨*expressions*⟩

⟨*print statement*⟩ ::= '`print`' ⟨*expressions*⟩
| '`print`' ⟨*expressions*⟩ '`,`'

which means that

```
print "a", "b", "c"
```

is distinct from

```
print "a", "b", "c",
```

The former outputting "a b c" followed by a newline, and the latter outputting "a b c" without a newline! There is no convenient way of changing what the item separator is, requiring setting a field in the 'sys' module. The exec statement has similar issues.

Furthermore, Python 2 has some odd semantics for its 'input' function, which will read a line of standard input, and then *evaluate this input as a Python expression*. In order to read the unevaluated input, 'raw_input' is needed.

Python 2 evolved in such a way to have two different object orientation systems. Python 2 has *old-style* classes and *new-style* classes, observe:

```
>>> class Foo:
...  pass
...
>>> class Bar:
...  pass
...
>>> class Quux(object):
...  pass
...
```

```
>>> f = Foo()
>>> b = Bar()
>>> q = Quux()
```

Here, two (empty) *old-style* classes 'Foo' and 'Bar' are defined, along with a *new-style* class 'Quux'. Instantiation is identical for all of them, however...

```
>>> type(f)
<type 'instance'>
>>> type(b)
<type 'instance'>
>>> type(q)
<class '__main__.Quux'>
```

Their *type* differs. All *old-style* objects have a type of 'instance' while *new-style* objects have a type matching their class.

```
>>> type(f) == type(b)
True
>>> type(f) == type(q)
False
```

Python 2 also shares a limitation common to early, "lower level" programming languages like C in that it does not distinguish between a *string of characters* and a *string of bytes*. This makes handing either somewhat cumbersome. For instance,

Listing 2.7: *The pound sign, with a length of two characters.*

```
>>> len("£")
2
```

Which makes sense only when considered as a UTF-8 encoded characer, which exists as $(194, 163)$ representing codepoint 163 in unicode. Python 3 provides a better abstraction - strings are arrays of characters, 'bytes' are arrays of integers in the range $0 \rightarrow 255$, and strings can be encoded to bytes using a variety of different encodings.

Listing 2.8: *The pound sign, with a length of one character.*

```
>>> len("£")
1
```

This is a rather important and useful distinction when dealing with predominantly binary data such as captured packets.

Perhaps the most compelling reason is that the reference implementation, CPython, currently exists as CPython 2.7 and CPython 3.4, with 2.7 being in maintenance mode[13] - that is, bug fixes only.

2.4 Architecture

There are many different ways of categorising programs, but most programs have a important, distinctive piece of behaviour that can place a program in one of two mutually exclusive categories. That is, are they *interactive* or not.

In the context of program design, an interactive program is one that accepts and alters its behaviour based on human input. A prime example of an interactive program is a text editor - it accepts input in the form of a series of keyboard presses and interprets each of these as a command to write a corresponding character.

Conversely, a non-interactive program is one that does *not* accept user input. For instance, a compiler is generally non-interactive - it takes input in the form of source code, and produces output in the form of compiled code.

The author, being a Linux aficionado, is particularly fond of following the *UNIX philosophy* where appropriate. This philosophy has no formal definition, but was once summarised, in the words of Peter H. Salus[14],

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

To this end, a system based around multiple small utilities that can be strung together to produce the desired output seems an appropriate architecture. Out of necessity, *text* will not be used as a model for data, but a ubiquitous packet capture format will be used. It is evident from the programs listed in section 1.3 that this format is *pcap*.

2.5 Development Methodology

The widely known 'waterfall model' of software development was, in its current form, presented as a flawed model that lacked development stage feedback[15]. The waterfall model is fairly simple, it suggests a requirements analysis stage, followed by a linear progression of design, implementation, testing and finally maintenance. It works in an idealised scenario where all requirements are specific, well understood and concrete, and that there are no unforeseen problems in implementation. Unfortunately, this is not how things work in practice.

As the project will consist of lots of small components, with a tree-like dependency graph, it is possible to design, build and test sets of components, without having designed a complete system. This is called *prototyping*, and is a practical approach to designing small systems. Doing this repeatedly to build a larger system is known as *incremental development*, and is essentially concurrent application of the waterfall model. Doing any design methodology repeatedly is called *iterative development*.

Ultimately, a prototype driven iterative and incremental development model seems like the most appropriate approach.

2.6 Libraries

The use of external libraries was considered for understanding file formats - namely libpcap, a C library, which supports the popular pcap file format and provides an interface for sniffing packets.

Python supports a FFI (Foreign Function Interface) module for C libraries under the name 'ctypes' and supports understanding C-like data structures using the 'struct' module, so this itself is not a limiting factor. However, existing bindings are outdated or are for Python 2, so a new set of bindings would have to be written. Libpcap does not have a very complex interface, but a lot of the functionality of libpcap is irrelevant - functionality related to querying network devices and capturing their output is not needed. The effort required to make a proper set of library bindings exceeds the effort to reimplement the desired functionality in pure python.

Instead, the documentation for libpcap's file format will be used to make an independent implementation.

2.7 Protocols

The project needs to deal with real network captures. Therefore, knowledge of protocols that occur on real networks is essential. Dissecting packets to extract and replace information is a task that requires specific knowledge. Fortunately, obtaining this is a fairly straightforward task.

The first problem is finding out *what* protocols are used. This is almost trivial, and can be determined in numerous ways; an empirical method would be simply to examining a live network,

this will tell you not only what protocols are in use, but how often they are used. Another method would be to simply gather some background knowledge - how networks operate and what they are used for. Internet access is almost ubiquitous, and the world wide web has become a prominent part of modern society - the world wide web is accessed using HTTP, which depends on TCP for its transport, which then depends on IP as its carrier. Additionally, for hosts implementing HTTP, DNS is used for resolving host names to IP addresses.

Local area networks typically have a homogeneous medium, the most common of which is Ethernet. From a data analysis point of view, this is one of the basic 'carrier protocols' that all other protocols are encapsulated within.

The next problem is understanding these protocols enough to identify them. This is also a straightforward task, most of these protocols are standardised by the IETF (Internet Engineering Task Force), and are publicly available as RFCs (Request For Comments).

As this project is about manipulating packet captures, it is essential to know what bits of data are important, and which are not. The three most important types of data are:

- Things that identify the payload - essential for further dissection.
- Things that identify hosts - these may need to be replaced for various reasons.
- Things that maintain data integrity - checksums and the like, which require updating if packet contents is modified.

There are other types of data with some significance. Protocols which split their payload over multiple segments (e.g. fragmented IP packets or TCP) will have some kind of identification in order to permit reassembly or to identify parts of a stream. This is important if reconstructing the whole transmission.

2.7.1 Ethernet II/IEEE 802.3

Ethernet is the common name for the IEEE 802.3 standards family, which is part of the broader family known as IEEE 802, covering technology used in Local/Metropolitan Area Networks.

Unlike the family of Internet Standards, Ethernet is a *physical* to *link-layer* specification maintained by IEEE-SA (Institute of Electrical and Electronics Engineers Standards Association), an organisational part of IEEE.

Figure 2.1: *Simplified diagram of the Ethernet frame format.*

The Ethernet *packet* format is specified in IEEE 802.3 Section 3.1.1[16], and consists of:

1. Preamble: 7 octets (8-bit bytes), with the pattern 10101010.
2. Start Frame Delimiter: 1 octet, with the pattern 10101011.
3. Destination Address: 6 octets. This is where the frame starts.
4. Source Address: 6 octets.
5. Ethertype/Length: 2 octets. This identifies what the payload contains, or how long it is (if less than 1500).
6. Payload: 46 to 1500 octets

It is worth noting that an IP router may also act as a gateway between two different networks, possibly with different media.

7. Frame Check Sequence: 4 octets, this is a CRC (Cyclic Redundancy Check) of the above frame data. This is where the frame ends.

These details are transmitted most significant bit first, most significant byte first (big-endian ordering).

The frame format is extended by IEEE 802.1Q and further extended by IEEE 802.1AD. 802.1Q adds support for VLANs (Virtual Local Area Networks) and 802.1AD adds support for an additional layer of VLANs. Logically, the information these extensions add is placed before the Ethertype, and consist of a 2 octet differentiating identifier ($0x8100$ for 1Q, $0x88A8$ for 1AD) in place of the Ethertype, and a 2 octet 'VLAN tag'. This identifier is used to determine if the frame has any such extensions. 802.1Q adds a single VLAN tag, 802.1AD adds an additional identifier and VLAN tag preceding the 802.1Q field.

This all means that an Ethernet *frame* has a variable size, with a range of 64 to 1564 octets (+4 or +8 octets, for 802.1Q and 802.1AD respectively).

The source and destination address fields are fairly self-explanatory, the Ethertype describes either the contents of the payload, or its length. The former seems to be the typical case, and is invaluable to interpreting Ethernet frames on networks carrying multiple protocols.

2.7.2 Address Resolution Protocol

The Address Resolution Protocol, commonly known as ARP, was first proposed and standardised as RFC 826[17].

ARP's primary purpose is to map *hardware* host addresses to *protocol* addresses, originally proposed for mapping MAC addresses to IPv4 addresses.

ARP packets consist of two 16 bit type identifiers, two 6 bit length fields and a 16 bit opcode field, followed by four variable length address fields.

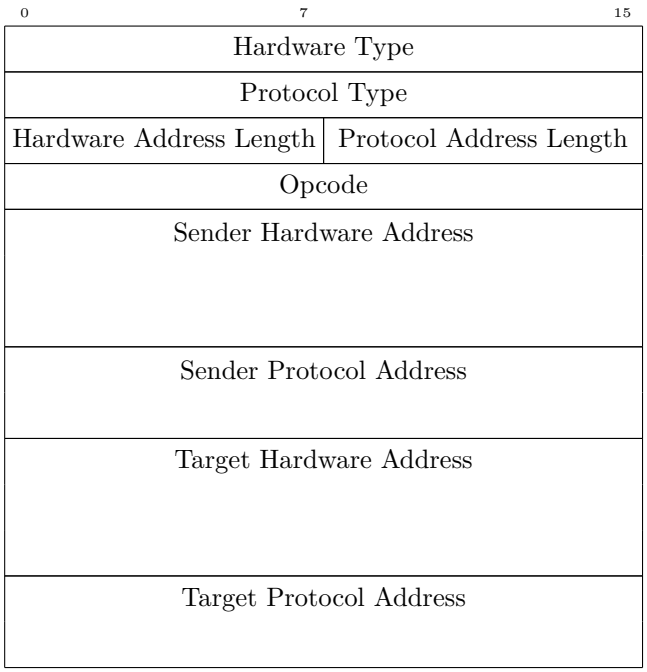

Figure 2.2: *Typical ARP packet, with 6 octet hardware addresses and 4 octet protocol addresses.*

As far as dissection goes, ARP is a terminal protocol and is also quite boring - it does not encapsulate anything, and is rarely interesting in terms of forensics, but it does contains four *host identities*. These are significant if modification is required.

It is interesting to note that in principle, ARP can work with any hardware or protocol address up to 255 bytes in length - its design allowing for arbitrary sizes for either, however in practice it is only ever used for MAC/IPv4 resolution. For IPv6, the same problem was solved with a new protocol, NDP (see 2.7.8)

2.7.3 Internet Protocol (version 4)

The Internet Protocol, commonly known as IPv4 or just IP, is specified and explained in RFC 791[18].

Suffice to say, the Internet Protocol is very important, and unsurprisingly, has a rather complex form.

Discussions about routing and IP addresses are outside the scope of this document, but unlike Ethernet and ARP, knowledge of protocol behaviour is needed to properly identify data transferred over IP packets.

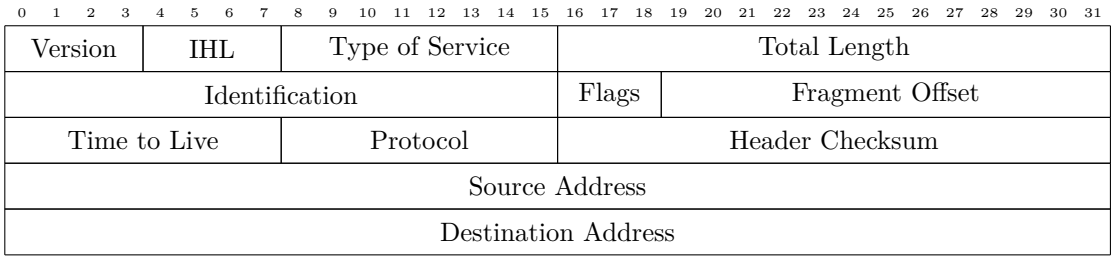

Figure 2.3: *Typical IPv4 packet format, without any optional sections*

In real terms, IPv4 usually operates as a kind of datagram transfer protocol, with flags and fields telling networked hosts how important the data it contains is and how many times the packet should be rerouted (this is represented in the Time to Live (TTL) field, and is known as a *hop count*). IPv4 also contains a specification for various optional parts, which occur after the destination address. For checksumming purposes, and the Internet Header Length field, these are counted as part of the header. These optional fields are always padded to the next 32 bit boundary.

In addition to this mode of operation, IP also specifies behaviour when packets need to traverse media with different MTU's (Maximum Transfer Unit). It does this by splitting the packet up into smaller pieces. The fragmented packet can then be reassembled by using the second 32-bit word (see 2.3) of the IP header, which contains an 'identification' field and two fields used for managing these fragments (additionally, the protocol field is added as part of this identity - meaning that the identification field need only be unique on a per-protocol basis). The flags field contains a bitfield holding two options, `MF` and `DF`. Bit 2 is the `DF` bit, or 'Don't Fragment', which instructs the receiver to discard the packet if it needs fragmentation. Bit 1 is the `MF` field, or 'More Fragments', and identifies the packet as one which has been fragmented and requires reassembly. The last fragment in a fragmented packet has `MF` bit set to 0.

IPv4 headers also contain a *checksum*. The checksum algorithm is simple, and is defined as:

> The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

It is worth noting that the same checksum algorithm is used in other protocols, notably TCP and UDP. For a Python implementation of this algorithm, see 5.1

For dissection purposes, supporting the reassembly of this fragmentation is useful. The packet contains two IP addresses, which can be altered but will require recalculation of the checksum. The protocol field is necessary to determine how to interpret the payload.

2.7.4 Internet Protocol (version 6)

The Internet Protocol version 6, or just IPv6, is specified in RFC 2460[19].

Despite being defined in 1997, IPv6 has been adopted very slowly. IPv6 was designed to be a solution to the IPv4 address exhaustion problems anticipated at the time. (Top level IPv4 addresses were depleted on the 3rd of February, 2011[20])

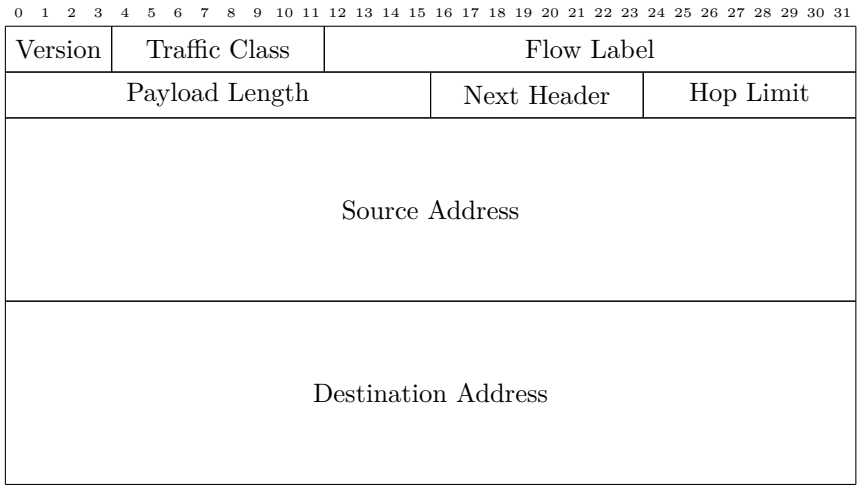

Figure 2.4: *IPv6 header format.*

The IPv6 header format is considerably simpler then the one of IPv4. Much like IPv4, it contains a field indicating the protocol it carries (the 'next header'). Unlike IPv4, the source and destination addresses are 16 bytes long and no fragmentation is supported in the core protocol.

IPv6 has a few option extensions defined, but these options are communicated in a different way to IPv4 options. IPv6 options are just additional protocols, which contain their own 'next header'.

2.7.5 User Datagram Protocol

The User Datagram Protocol (UDP) is specified in RFC 768[21].

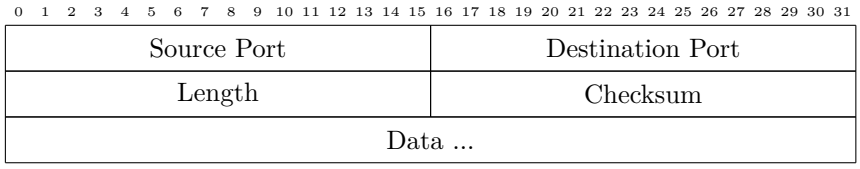

Figure 2.5: *UDP header format.*

The user datagram protocol exists as a relatively thin layer over IP. It offers data integrity and not much else.

UDP has a checksum field with an algorithm that depends on the carrier protocol. The UDP checksum algorithm makes use of a 'psuedoheader' consisting of the source and destination addresses, the length of the entire UDP segment and the IP protocol number for UDP (as 16 bit integers for IPv4, or 32 bit integers for IPv6). Because UDP datagrams can have an odd length, a byte with a value of 0 should be appended to the segment for the purposes of the calculation.

The UDP introduces the concept of a 'port'. This is little more then a flexible protocol identification field, with well-known protocols operating over well-known ports. For further analysis, the *destination port* should be used for payload identification.

2.7.6 Transmission Control Protocol

The Transmission Control Protocol (TCP) is specified in RFC 793[22].

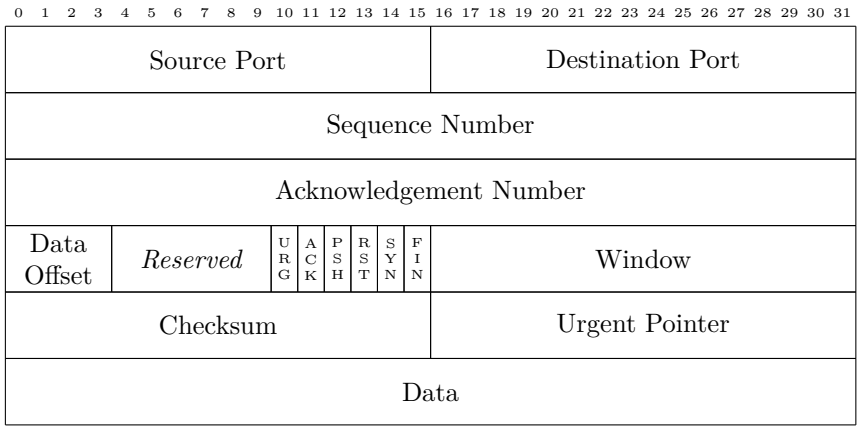

Figure 2.6: *Typical TCP segment header format, sans options.*

The Transmission Control Protocol is a reliable stream transport protocol, used over an unreliable carrier such as IP. TCP headers have many fields as seen in figure 2.6, but there are seven important ones: source port, destination port, sequence number, acknowledgement number, data offset, flags and checksum. These are used to establish a kind of 'circuit-switched' connection over a packet-switched network. TCP has a three-way handshake between client and server, used to negotiate this connection.

Client machines initiate a connection by sending a TCP segment with the SYN (synchronise) flag set, the destination port set to the port number used for the desired service, and the source port set to a random, ephemeral value.

Upon receiving a SYN, the TCP implementation on the server will, if said service is supported, will send a SYN-ACK (Synchronise/Acknowledgement) segment back; with the destination port set to the source port of the initial segment and the source port set to a random, ephemeral port. Finally, the client then responds with an ACK and the handshake is complete.

From then on, either side can send segment headers with the PSH (push) flag set. These contain the data being sent over the logical stream. Communication is full duplex, either side can send data segments. Each PSH should be reciprocated with an ACK from the other side, returning the acknowledgement number of the received push segment. Each time any TCP PSH segment transfer is carried out, the sequence number is incremented.

The FIN (Finish) flag can be sent to indicate that no further PSH flags will occur in communication, effectively closing one end of the bi-directional stream.

TCP has a checksum that depends on its carrier protocol. The TCP checksum is defined in a similar way to the UDP checksum, with a 'psuedoheader' consisting of the source and destination addresses, the length of the entire TCP segment and the IP protocol number for TCP (again, as 16 bit integers for IPv4, or 32 bit integers for IPv6). Like a UDP datagram, a TCP segment can have an odd length, to account for this, a single byte with a value of 0 should be appended for the purposes of the calculation.

TCP also has the notion of a port. The initial destination port (sent in a segment with the SYN flag) indicates the type of service. Further segments use ephemeral ports, so in order to reliably identify higher level protocols, all parts of a TCP stream need to be tracked and presented as a unit.

2.7.7 Domain Name System

The Domain Name System (DNS) is specified in RFC 1035[23].

This protocol normally operates over UDP port 53, but can use TCP as an alternate transport.

The domain name system is most commonly used to map a set of named *hosts*, each belonging to a *domain*, to a set of IP addresses.

`blackhole.time-domain.co.uk` → `81.142.251.220`

Example of a domain name resolution.

The domain name system exists as a distributed hierarchical database, with DNS servers local to a network (the *name server*) possibly querying servers higher up the tree. There is more to this, but a detailed discussion on how DNS works is beyond the scope of this document.

DNS messages have a common header format, indicating what kind of request or reply is being sent. DNS messages can contain a wide variety of record types, such as 'A' records for IPv4 addresses, 'AAAA' records for IPv6 addresses, and many others. Supporting DNS message falsification is considered a luxury goal for this project.

2.7.8 Neighbour Discovery Protocol

The Neighbour Discover Protocol (NDP) is specified in RFC 4861[24].

NDP fills a similar role to ARP (2.7.2) in the IPv6 role (among others), and has a message structure comparable to that of DNS (2.7.7). NDP also provides other features for networks, such a router advertisement and address autoconfiguration. Supporting NDP falsification is considered a luxury goal for this project.

Chapter 3

Requirements

Ultimately, specifics of the project were left to the author. Sean Tohill proposed the following workflow:

> You run small captures of individual sessions e.g. email.
>
> You sanitise by removing some packets e.g. arp, proxy.
>
> You change ip addresses of the host to something else.
>
> You put them together to make a larger capture, you need to alter times and stuff here.

Based on this proposed workflow, it was determined that at least three programs would need to exist:

i) A program that can filter certain protocols from packet capture files.

ii) A program that can alter *host identities* from packet capture files.

iii) A program that can merge packet capture files.

These are more formally specified in the functional and non-functional requirements below.

3.1 Functional Requirements

FR1: The programs shall be able to read a packet capture file in at least pcap format as program input.

FR2: The programs shall be able to write a packet capture file in at least pcap format as program output.

FR3: The programs shall be able to identify protocols used in individual packets, and based on this information should...

FR4: ... be able to selectively remove packets belonging to a predefined set of protocols.

FR5: ... be able to alter the contents of packets based on a set of predefined substitutions.

FR6: There shall be a program that accepts multiple inputs, and merge them into a single output.

FR7: There shall be a program that alters the timestamp metadata for individual packets.

FR8: There shall be a program that selectively removes packets pertaining to specific protocols.

FR9: There shall be a program that finds and replaces host identifiers within packets contained in a capture, while recalculating any checksums that need updating.

FR10: Programs shall support loading external support code (extensions) for handling new protocols at runtime.

FR11: The programs shall be able to operate on streaming data in addition to static files.

3.2 Non-functional Requirements

NFR1: The programs shall be usable on any python implementation supporting at least Python 3.4.

NFR2: The programs shall be usable on any platform supported by such an implementation.

NFR3: The programs should scale well and have, as a worst case, a small coefficient linear memory usage growth rate in respect to input data size.

NFR4: The programs should scale well and have a linear time complexity in respect to input data size.

NFR5: The source code shall be designed in a maintainable manner, such that...

NFR6: ... components not related to file handling make no assumptions about the input file format.

NFR7: ... components not related to file handling make no assumptions about the output file format.

NFR8: ... support for identifying new protocols requires few, if any alterations to existing code.

NFR9: ... support for new file formats requires few alterations to existing code.

NFR10: The programs (and their source code) should be structured and reusable with-well defined interfaces.

NFR11: The source code should be well documented and adequately commented, to assist future developers in understanding the code.

NFR12: The programs should be resilient - they should handle invalid data gracefully.

NFR13: The programs should be secure - they should handle malicious data in a safe way.

3.3 Functional Requirement No. 3

FR3: *packet identification* is quite an open ended requirement and leaves an important detail to consider: *which protocols?*

Section 2.7 talks about common protocols, but understanding just one of these protocols takes a considerable amount of time. Because of this, the author decided to *initially* focus on a small number of protocols, based on their frequency of occurrence and importance; these are Ethernet, IPv4, TCP, UDP and ARP, possibly considering more should the time be available.

Additionally, partial *identify only* support should be included for at least IPv6 and ICMP. This selection provides reasonable filtering and adjustment capabilities for common capture files.

Chapter 4

Design

It is important to note that the design evolved with the implementation - challenges discovered in implementation will be reflected in the design. One of the most important factors in this project was building it so that future developers could work on it, so a fair amount of work has gone into structuring the project well.

4.1 Structure

Ultimately, code maintainability relies on a logical layout. Firstly, a clear distinction between *application code* and *library code* was decided. All *library code* is to be contained in its own Python package.

Much like the programs, this code has several well differentiated tasks - *reading and understanding packet captures*, *dissecting and altering packets* and *processing streams of packets*. Code that handles these abstract tasks are contained as their own subpackage, *capfile*, *identity* and *pipeline* respectively.

```
src
└── packet
    ├── capfile
    │   ├── core.py
    │   └── pcap.py
    ├── identity
    │   ├── arp.py
    │   ├── core.py
    │   ├── eth.py
    │   ├── ip4.py
    │   ├── ip6.py
    │   ├── tcp.py
    │   └── udp.py
    ├── pipeline
    │   ├── filter.py
    │   ├── identify.py
    │   └── merge.py
    ├── common.py
    └── memorymap.py
```

Figure 4.1: *A speculative code layout.*

4.2 Abstractions

When dealing with a problem, it is helpful to simplify details to a more abstract form. This way, it is possible to handle *packets* instead of *pcap-formatted ethernet packet data*.

For this project, there are three major abstract entities: packets, packet sources and protocols.

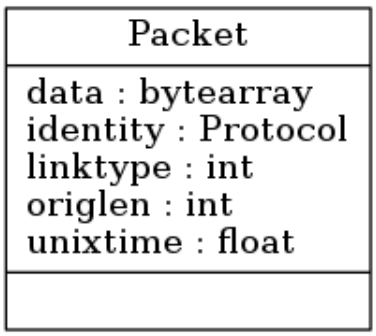

Figure 4.2: *The packet class*

4.2.1 Packets

Packets are represented by instances of a *Packet class*. The Packet class will consist of a collection of fields; the *linktype*, the time the packet arrived, the recorded packet data, and the packet's original length along with a special field, *identity*.

Here, *linktype* is an integer constant that represents one of many network media types. An important design decision was to make as few assumptions as possible about program input, so this field exists to identify what kind of packet this is. The most common value for this will be 1, which corresponds to Ethernet (`LinkType.ETHERNET`) but other values are possible and should be supported.

The *identity* is a field holds the root (i.e. the bottom most) *protocol instance* for this packet. This should be set to None by default, signifying the packet has not been identified. More on this in the *Protocols* section (4.2.3).

4.2.2 Packet Sources and Sinks

In order to help ensure scalability, the flow of the system should work around processing individual packets, rather then entire collections of packets. This means a packet is only resident in memory for the time it is needed - permitting processing very large files without too much memory usage.

Packet sources are the realisation of this goal. Packet sources are objects that support Python's *iterator protocol*, that yield the packet objects mentioned in the previous section. Packet sources are presented by the library as building blocks for higher level applications. Packet sources represent a kind of lazily-evaluated interface towards streams of packets - this cuts down on memory usage, improving scalability. Sources that depend on other sources can be chained without restriction.

A certain type of packet source, a packet reader, is to represent sources based off the various packet capture file formats. These classes implement methods in an abstract class, PacketReader. This abstract class will implement the interface of a packet source using `read_packet`, and can be used just like one. Along with a packet reader, a packet writer class is available as a *packet sink* - there are no semantics beyond implementing the `close` and `write_packet` methods in the PacketWriter class.

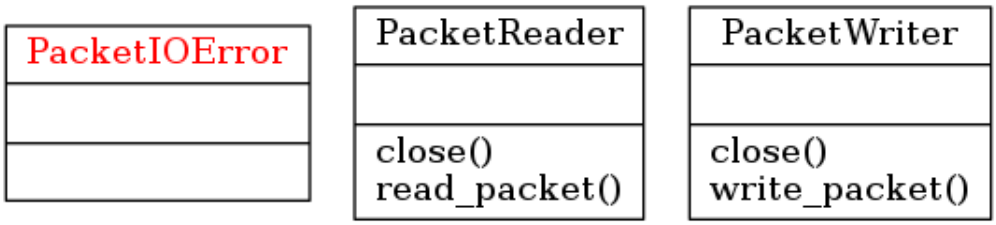

Figure 4.3: *The PacketReader and PacketWriter abstract classes, with the generic PacketIOError exception.*

The PacketReader and PacketWriter classes should also raise exceptions when they find a

problem reading or writing a packet, respectively. The exceptions raised should be derive from the PacketIOError class.

4.2.3 Protocols and Protocol Instances

Network protocols are represented by Protocol classes. Protocols are the most complex abstraction represented in this design.

Logically speaking, instances of communication over a network conform to a linear hierarchy of protocols. A *carrier protocol* encapsulates another protocol, which itself may be another carrier protocol. The system models a packet's identity around the concept of *protocol instances*, which are instances of communication adhering to a specific protocol.

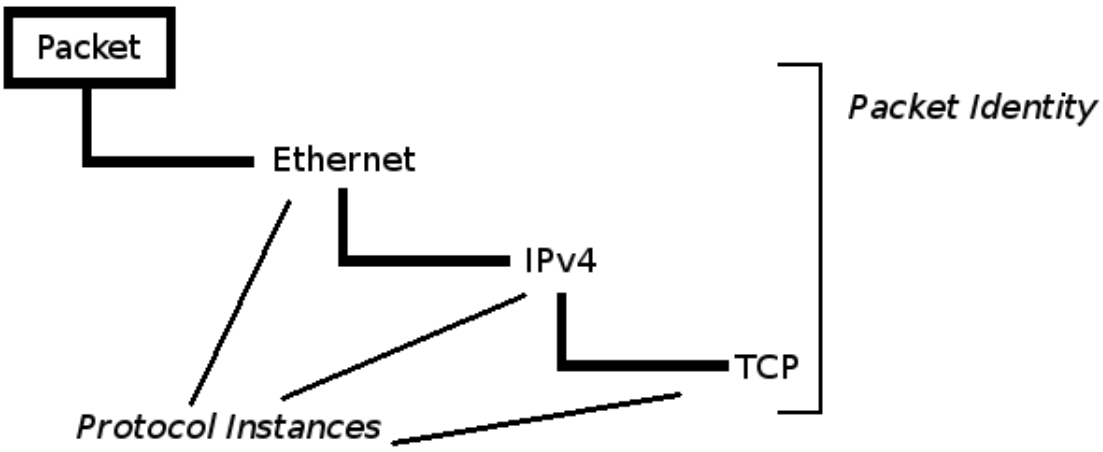

Figure 4.4: *A packet's identity and protocol instances.*

A carrier protocol will also have some kind of *route*, a known source and destination. This may or may not depend on the parent protocol. For instance, IP packets will typically be sent over Ethernet frames, but IP implements a logically independent network from an Ethernet network - these frames, when reaching a router, will be sent somewhere else (with altered frame data), potentially to entirely different routers over an entirely different carrier, all while preserving the IP frame. This is how internetworking works. Conversely, UDP packets rely on the logical route established by IP packets to transmit data, and are basically an extension of the *protocol* or *next* field in an IP header.

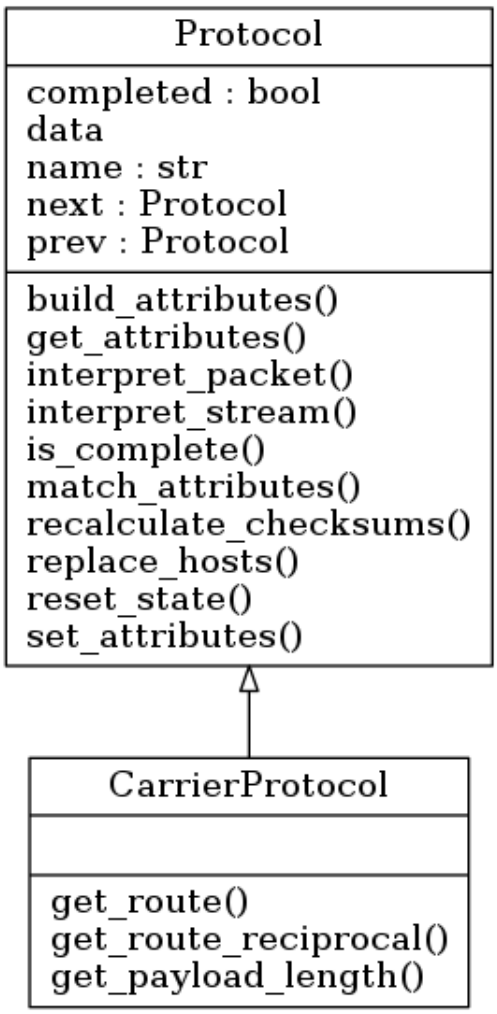

Figure 4.5: *The protocol and carrier protocol classes*

The diagram above shows the design of the Protocol and CarrierProtocol classes. For any protocol to be represented in the program, it must have a Protocol class. This class has a well defined form, and this is important as extensibility is a key requirement.

Protocol classes have a single *class variable* and three *unbound methods* (static methods), in addition to the per-instance fields and bound methods typical of most classes. These are used to identify the protocol, and construct *protocol instances* respectively. A more detailed description of these is given below.

Firstly, every protocol instance has a field called data. This is intended to be used as a quasi-internal field, holding either a Python 'memoryview' object or a 'memorymap' object (see 4.3). Python `memoryview` objects are effectively pointers to regions of memory held by objects supporting the *buffer* protocol. A `memoryview` of a `bytearray` provides a read-write view of that data without requiring a direct reference to it. Like most python collections, `memoryview` objects support being sliced - this is important.

Because packets have a linear hierarchy of protocols, Protocol instances have a structure not unlike a linked list - each instance has a parent protocol (called `prev`) and a child protocol (called `next`); these are set to none if this is the bottom most or top most protocol, respectively.

Additionally, there is a boolean value called `completed`, which is to signify whether this protocol instance has finished being dissected. Setting this value to false allows other parts of the code to know that a packet has had its identification deferred until more information is available.

As for bound methods, there are 6 defined, `get_attributes`, `set_attributes`, `match_attributes`, `is_complete`, `recalculate_checksums` and `replace_hosts`.

The `is_complete`, `recalculate_checksums` and `replace_hosts` methods are all propagating methods - they will call their own equivalent on the child protocol instance, if any. `is_complete` works on the aformentioned `completed` boolean, returning true if and only if this value is true for all child protocols.

As replacing host identity information is one of the goals of the project, a `replace_hosts` method was defined. This takes a single argument, a dictionary. This dictionary defines a mapping between host identifiers and their replacements, the idea being that implementations of this method will look for possible substitutions and alter their instance data accordingly, and then call `replace_hosts` on the child instance. This provides a very simple and reliable way of doing this substitution. In a similar vein, `recalculate_checksums` is for updating any checksum data. Actual recalculation should start from the top most protocol instance, as the lower level carriers may have checksums on their payload (i.e. UDP).

The `get_attributes`, `set_attributes`, `match_attributes` (and `build_attributes`) methods all work with or produce a set of *protocol attributes*. The precise format of said attributes is entirely up to the implementing class, but is suggested to be a *dictionary*.

`get_attributes` should to return 'attributes' of that protocol instance (for example, IPv4 attributes might contain the source/destination address and the protocol number). Conversely, `set_attributes` should to alter packet data such that the packet corresponds to the attributes provided. `match_attributes` compares a set of attributes to those of the protocol instance - if they match, this method returns true, else false.

Protocol classes are also required to have a few unbound methods. The most important one, is `interpret_packet`. This method takes two arguments, a `memoryview` or `memorymap` object, and a *parent* protocol instance. It is the job of this method to construct and return a protocol instance, possibly calling `interpret_packet` a subset of the data provided. This method is also responsible for maintaining state for completing deferred identification. `interpret_stream` effectively does the same thing, but for *streams*. Because both of these methods may have side effects, as a library, it was deemed appropriate to have some kind of mechanism for restoring the default set of state for these classes - this is the job of the `reset_state` method.

`build_attributes` is the last unbound method. This method is intended to return attributes not unlike those returned by `get_attributes`, and is used for turning human input (an 'attribute string') into a workable object.

Finally, all protocol classes need to have a human-readable name. This is for both output (such as when 'printing' a packet) but mainly as a unique identifier for a Protocol class. The intention is, that protocols are referred to by this name, and that a Protocol class can be replaced by one with the same name, should the user want to use an alternate implementation. This name is also used in constructing and comparing attributes.

4.3 Memory maps

In order to identify protocols, packet data needs to be examined. As some protocols can have a variety of carriers, and carrier protocols can have a variable size, the exact location of one protocol's header is not a constant offset.

The initial approach to this problem was just to copy the payload section of a protocol and present that to the next protocol interpreter. This works, but means any modifications to the protocol header would require the entire packet to be rebuilt - a time consuming operation.

Python *memoryview* objects provide a memory-efficient and simple way of presenting smaller sections of data from an object. They are probably best illustrated with an example:

Listing 4.1: *A demonstration of memoryviews. v is a view of data, and v1/v2 are subviews of data.*

```
>>> data = bytearray(b"abcdef")
>>> v = memoryview(data)
>>> bytes(v)
b'abcdef'
>>> v1 = v[0:3]
>>> v2 = v[3:6]
>>> bytes(v1)
b'abc'
>>> bytes(v2)
b'def'
>>> v2[1] = b"E"
>>> bytes(v)
b'abcdEf'
```

Here, some data object (called `data`) is being read and altered by some *memoryview* objects (called `v`, `v1` and `v2`).

Importantly for this project, alterations to *memoryview* objects affect their backing store - changes are reflected in the object the views are based on. These are sufficient to provide a linear tree of protocol instances with read/write access to packet data, but it was known from the start of the project that sometimes, parts of protocol data will exist over many packets.

For identification purposes, this can easily be solved by simply copying parts of the fragmented data and concatenating them. This solution is a different application of the first solution presented and suffers from all its downsides in addition to the problem of making alteration a very difficult task.

To solve this problem efficiently, a *memorymap* class is needed. This class would provide logically contiguous access to multiple, separate *memoryview* objects.

The ideal implementation would have a similar interface to that of *memoryview* objects, and function as a drop-in replacement. This means protocol classes need not worry over how many packets their data is spread.

4.4 User Interface

The project is more then just a library. The project also seeks to provide a useful user interface to manipulate packet captures. In terms of requirements, this section refers to the design of **FR6:** *merge program*. **FR7:** *timestamp alteration program*, **FR8:** *filter program* and **FR9:** *host identity replacement program*.

As is normal for batch data processing tools, these have a command line interface.

4.4.1 Common interface elements

The programs are to understand their command line arguments as a series of options with mostly unary or nullary parameters. For ease of use, there will be no specific order to command line options and no explicit options will be mandatory, each program having *default* behaviour.

Here, arity means how many arguments the option requires, i.e. a nullary option takes no arguments, a unary option requires one, a binary option would require 2 etc. Frequency is how often the option should appear. Singular indicates the option should appear at most once, arbitrary means the option can appear any number of times.

There will be commonalities between tools. These are detailed below.

Option Flag				
Short	**Long**	**Arity**	**Frequency**	**Description**
`-l`	`--load`	unary	arbitrary	Loads a external support code module.
`-h`	`--help`	nullary	singular	Prints out help text.
`-i`	`--input`	unary	singular	The capture file to read from (defaults to stdin).
`-o`	`--output`	unary	singular	The capture file to write to (defaults to stdout).

4.4.2 The List Tool

The list tool will take a packet capture, and print out a succinct, human-readable interpretation of each packet.

The list tool has no special parameters, and does *not* accept the `--output` option.

The output should be something along the lines of:

```
Mon Oct 20 15:51:29 2014 Linktype: 1, Identity: eth/arp, Length: 42
```

4.4.3 The Merge Tool

The merge tool will take an arbitrary number of packet capture files, and write out a single capture from these.

The merge tool was also seen as the appropriate place to do time alteration, so this program fulfils the role of **FR6:** *merge program* and **FR7:** *timestamp alteration program.*

The merge tool will take a slightly different `--input` option in that it will permit the option to occur multiple times.

Option Flag				
Short	**Long**	**Arity**	**Frequency**	**Description**
`-i`	`--input`	unary	arbitrary	The capture files to read from (if none specified, reads from stdin exclusively).
`-r`	`--relative`	nullary	singular	Makes all packet arrival times from a file relative to the time the first packet recorded in that file (this is on by default).
`-a`	`--absolute`	nullary	singular	Packets retain their original time (this is the opposite of the *relative* mode above).
`-t`	`--offset`	unary	singular	Time offset, this value is added to the arrival times of all packets. If this value is not provided, it is set to the *average arrival time of the first packet in each capture.*

```
[user@host ~]$ merge -i cap1.pcap -i cap2.pcap -o cap12.pcap
```

4.4.4 The Filter Tool

The filter tool will fulfil the role of **FR8:** *filter program.*

Option Flag				
Short	**Long**	**Arity**	**Frequency**	**Description**
`-k`	`--keep`	unary	arbitrary	A packet identity to keep.
`-d`	`--discard`	unary	arbitrary	A packet identity to discard.
`-p`	`--policy`	unary	singular	The policy to apply to any unaccounted for packet. Accepts either *keep* or *discard*.

The tool will operate by first checking if a packet's identity matches anything in the *keep* set. If so, the packet will be reproduced in output. If not in the *keep* set, the packet's identity is then checked against items in the *discard* set. If any match is sound, the packet will not be present in output. If a packet's identity is not present in either set, the value of *policy* is used - if keep, the packet is kept, if discard, the packet is discarded.

Here, an *identity* will be specified as a specially formatted string, consisting of *protocol names* and *protocol attributes*.

Valid protocol names are arbitrary strings that contain no colons or slashes. Valid protocol attributes share the same restrictions.

⟨*prototype*⟩ ::= ⟨*protocol name*⟩ '`:`' ⟨*protocol attributes*⟩
| ⟨*protocol name*⟩

⟨*identity*⟩ ::= ⟨*prototype*⟩ | ⟨*prototype*⟩ '`/`' ⟨*identity*⟩

The exact format of the protocol attributes string depends entirely on the Protocol class, this string will be parsed by the `build_attributes` method. However, a consistent format should be used (but not imposed) - to this end, the format used by all built-in Protocol classes will take the form of a simple key-equals-value semicolon separated list.

⟨*kvp*⟩ ::= ⟨*key*⟩ '`=`' ⟨*value*⟩

⟨*protocol attributes*⟩ ::= ⟨*kvp*⟩ | ⟨*kvp*⟩ '`;`' ⟨*protocol attributes*⟩

Invocation might look like:

```
[user@host ~]$ filter -i cap1.pcap -o cap1-noip6.pcap -k eth/ip6 -p discard
```

for permitting only IPv6 packets, or

```
[user@host ~]$ filter -i cap1.pcap -o cap12.pcap -d eth/arp:opcode=1
```

for removing all ARP requests, leaving ARP responses intact.

In the case of protocol names, ideally alphanumeric.

4.4.5 The Maphosts Tool

The 'maphosts' realises **FR9:** *host identity replacement program.*

Option Flag				
Short	**Long**	**Arity**	**Frequency**	**Description**
`-m`	`--mac`	unary	arbitrary	A mapping of an Ethernet MAC address to its replacement.
`-4`	`--ip4`	unary	arbitrary	A mapping of an IPv4 address to its replacement.
`-6`	`--ip6`	unary	arbitrary	A mapping of an IPv6 address to its replacement.

The `-m`, `-4` and `-6` understand address pairs as their argument. Address pairs have the form:

⟨*address pair*⟩ ::= ⟨*address*⟩ '`=`' ⟨*address*⟩

The address here, will take the form:

⟨*hexbyte*⟩ ::= 00 | 01 | 02 | 03 ... FF

⟨*hexpair*⟩ ::= ⟨*hexbyte*⟩ ⟨*hexbyte*⟩

⟨*decbyte*⟩ ::= 0 | 1 | 2 | 3 ... 255

⟨*MAC address*⟩ ::= ⟨*hexbyte*⟩ '`:`' ⟨*hexbyte*⟩ '`:`' ⟨*hexbyte*⟩ '`:`' ⟨*hexbyte*⟩ '`:`' ⟨*hexbyte*⟩ '`:`' ⟨*hexbyte*⟩

⟨*IPv4 address*⟩ ::= ⟨*decbyte*⟩ '`.`' ⟨*decbyte*⟩ '`.`' ⟨*decbyte*⟩ '`.`' ⟨*decbyte*⟩

⟨*IPv6 address*⟩ ::= ⟨*hexpair*⟩ | ⟨*hexpair*⟩ '`:`' ⟨*IPv6 address*⟩
| ⟨*IPv6 address*⟩ '`::`' ⟨*IPv6 address*⟩

Note that this particular grammar definition of an IPv6 address is rather vague. A more detailed explanation is given in RFC 5952[25].

Typical invocation might be:

```
[user@host ~]$ maphosts -i cap1.pcap -o cap12.pcap -4 192.168.0.20=172.16.0.20
```

for replacing occurences of `192.168.0.20` with `172.16.0.20`.

Chapter 5

Implementation

Implementation started concurrently with design. Challenges encountered during implementation would change the design, and vice versa.

Due to the sheer number of networked protocols and the complexities in supporting some of them, the project was never expected to approach 100% completion. As a result of this, much of the implementation time was actually spent improving the design sufficiently to correctly support a small subset of protocols, while considering other protocols being added in the future.

An interesting point to note is that no external dependencies or code were used. Only builtin Python classes and modules in the standard library were used. These were the `struct` module, the `sys` module, the `binascii` module and (for the commandline interface, the `argparse` module).

5.1 Meta-Implementation

While this section is not really about any specific implementation detail, it does cover noteworthy disciplines used by the author *while implementing*.

5.1.1 Coding Style

Some of the functional requirements (namely **NFR10:** *code structure* and **NFR11:** *code documentation*) effectively require code to be well-written and well-documented.

To this end, a variant of the standard Python code style was used. In essence:

i) Indentation is done with four spaces.

ii) *Private* fields in a class start with an underscore.

iii) Functions and methods are lower_case_with_underscores.

iv) Constants are CAPITALS_WITH_UNDERSCORES.

v) Class identifieirs are written in UpperCamelCase.

In contrast to normal Python style guidelines, this particular style makes classes easy to differentiate from functions.

In addition to following the aforementioned coding style, the code was also written to be mostly PEP8[26] compliant (this was checked with the `pep8` conformance checker, see [27]).

External dependencies being anything not present in the Python standard library.

5.1.2 Version Control

Any programming project will benefit form a form of version control. Version control helps keep a record of progress and allows mistakes to be easily reverted.

Version control is also useful as a kind of synchronisation system between multiple computers, permitting an individual to continue work while away from their normal work environment.

The author used *git* (`http://git-scm.com`) as the version control system (VCS), due to having prior knowledge on how to use it.

5.2 The `packet` super-package

The 'library' portion of the implementation is contained within its own package, the `packet` package. All of the modules have PyDoc docstrings, allowing for interactive documentation browsing with the pydoc utility. Should the reader want to do this, simply do:

```
$ pydoc -b packet
```

from the `src` directory. The same documentation is also included in the appendix (section 9.2).

5.3 The `memorymap` Module

The `memorymap` module implements the *memorymap* class as described in 4.3, and works by mapping multiple *memoryview* objects to a single *logical* 'index space'.

The `memorymap` class stores each constituent *memoryview* object (denoted V) as part of a three-tuple, with their *logical* starting and ending index (min and max, respectively). This tuple is henceforth referred to as a *segment* (or S). These segments are stored in a list (Q), in ascending order of ranges.

A note on notation: Slightly non-standard notation and concepts have been employed to make this mathematical model more readable and understandable. *Lists* are *tuples* with different notation.

For a tuple, superscript denotes a component, i.e. for tuple $T = (a,\, b,\, c\,...)$, $T^a = a$.

For a list, square brackets denote a specific element, i.e. for list $R = (\pi,\, \epsilon,\, \lambda\,...)$, $R[3] = \lambda$

$$S_i = (V_i,\, min_i,\, max_i)$$
$$Q = [S_0,\, S_1,\, S_2\,...\,S_i]$$

Given S_a and S_b as any two elements in Q, where $a < b$,

$$S_a^{max} \leq S_b^{min}$$

A *logical index* ($n_{logical}$) is translated to a pair of actual indices, the *segment index* ($n_{segment}$), which refers to one of the segments in the segment list, and a *local index* (n_{local}), which is the part of the segment.

This translation is done by a very simple algorithm. A binary search of the segment list is done, identifying the range encompassing a *logical index*.

$$n_{segment} = search(Q,\, n_{logical})$$
$$N = Q[n_{segment}]$$
$$n_{local} = n_{logical} - N^{min}$$

5.4 The `common` Module

The common module contains functions and classes which are useful in many different locations. This is where the packet class is defined.

This class contains the rather large `LinkType` enumeration, used in the identity package and the capfile package.

Also contained, is a handful of input parsing functions. A few basic ones, such as `parse_int` and `parse_hexbytes`.

There are functions for parsing string representations of MAC, IPv4 and IPv6 addresses, and turning binary representations into strings. these are named `*_str2bin` and `*_bin2str`, and take a single argument returning None if the format is erroneous.

5.5 The `capfile` Package

This package contains code for reading and writing abstract packets to packet capture files. Currently, only the *pcap* files are supported, but the author hopes that extending the library and programs to accept different formats to be relatively straightforward.

5.5.1 The `core` Module

This contains abstract classes intended to make adding new capture formats easier. Very simple PacketReader and PacketWriter classes are provided, with a PacketIOError exception class for abstracted error handling.

5.5.2 The `pcap` Module

This module contains two classes, a PcapReader class, which implements abstract methods in the PacketReader class, and a PcapWriter class, which implements abstract methods in the PacketWriter class.

These classes implement support for manipulating *pcap* files according to the pcap format specification[28], which is described below.

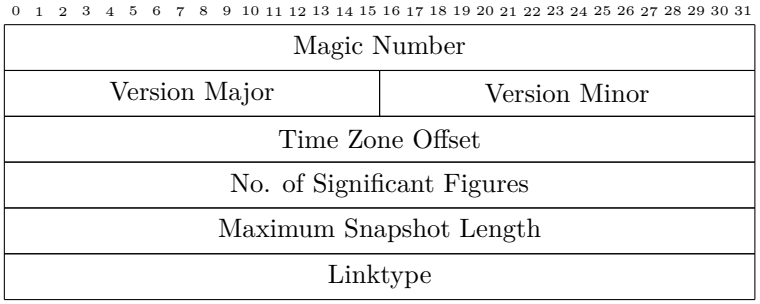

Figure 5.1: *pcap global header.*

The pcap format is quite simple, perhaps the reason for its ubiquity. A pcap file starts with a *magic number* field. This indicates the format specifics of that capture file. The magic number is $a1\ b2\ c3\ d4$ for *millisecond resolution* pcap files, and $a1\ b2\ 3c\ 4d$ for *nanosecond resolution* files. Additionally, this field is in the same byte ordering as the rest of the fields in the file, so $d4\,c3\,b2\,a1$ would indicate that the file stores little-endian millisecond resolution fields, rather then big-endian.

The rest of the fields are indicated in figure 5.1, and are fairly self explanatory. Linktype is an integer constant indicating the network media (such as Ethernet). These are the same as the values present in the `common` module.

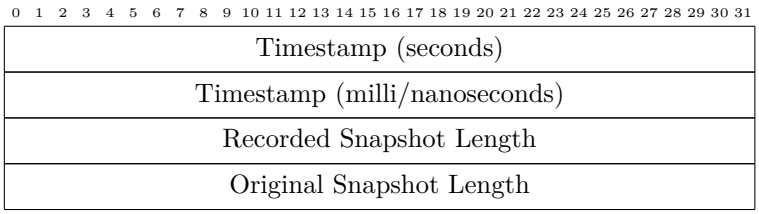

Figure 5.2: *pcap record header*

Pcap *records* consist of a record header and a packet snapshot. The packet data is presented verbatim, as it was on the network, but truncated to fit within the maximum snapshot length defined in the header.

The record header also has a simple layout. A 32-bit timestamp, in seconds since the UNIX epoch, and another 32-bit field of either millisecond or nanosecond resolution time, which serves as additional precision for this value.

Two more 32-bit fields are present, consisting of the recorded snapshot length and the original snapshot length. The recorded length indicates how many bytes after this header belong to a packet snapshot. Record headers immediately follow the global header and snapshot data.

These classes handle erroneous data in a robust and well defined manner - if the PcapReader encounters a truncated file, it will read and return all entries which are intact, raising a PcapFormatError exception when the truncated entry is reached.

If the PcapWriter is instructed to write a packet with a LinkType that differs from that of the network section of the file, it will raise a PcapRangeError exception.

5.6 The `identity` Package

This module is where most of the interesting code is and is the package where all 'builtin' protocol classes reside.

5.6.1 The `core` Module

The core module in the identity package is where the Protocol class is defined. This serves as step 0 in the packet identification system.

This module contains the registry of all supported protocols, and maintains a mapping of *protocol name* to *protocol class*. All Protocol classes should be registered with this module. Additionally, this module maintains a mapping of *linktypes* to *protocol names*, used in order to determine which *base protocol class* to use.

The Protocol registration and resolution is done using the `register_protocol` and `lookup_protocol`. The use of a string identifier instead of a direct class reference serves two purposes: Firstly, it allows for *alternate* implementations of builtin Protocol classes to be used, and provides an easy way for parts a program, particularly those that handle user input, to get access to the same Protocol class used in the identity of a packet.

More importantly, it provides the `root_identify` function, which starts the identification of a packet. This function does not pay attention to *deferred* identities, so applications that want a more complete understanding of packets should use the `identify` pipeline instead.

5.6.2 The `ip` Module

The `ip` module is part of the identity package, and is used by the the `ip4` and `ip6` modules.

The `ip` module provides a registry of Protocol classes mapped to IP protocol numbers. It also

contains a handful of constants for said numbers. Registration is done just as in the `core` module.

Aside from providing the IP protocol registry used by these modules, it provides an implementation of the checksum algorithm mentioned in 2.7.3. This is used by the IPv4, TCP and UDP Protocol classes, and should be sufficient for others using similar schemes.

Listing 5.1: *IP checksum algorithm, in Python.*

```
# The checksum algorithm is:
#       The checksum field is the 16 bit one's complement of the one's
#       complement sum of all 16 bit words in the header.  For purposes of
#       computing the checksum, the value of the checksum field is zero.

def checksum(data):
    # General implementation of the IP checksum algorithm.
    n16words, rem = divmod(len(data), 2)

    if rem != 0:
        # Argument is not in terms of 16-bit words.
        return None

    # generate list of 16-bit words.
    words = [word for word in struct.Struct("!"+"H"*n16words).unpack(data)]

    ocsum = 0
    for word in words:
        # One's complement addition of 16-bit words.
        acc = ocsum + word
        carry = acc >> 16
        ocsum = (acc + carry) & 0xFFFF

    b1, b2 = divmod(ocsum, 256)
    return bytes((~b1 & 0xFF, ~b2 & 0xFF))
```

5.6.3 The Protocol classes

There are quite a few Protocol classes implemented, enough to provide basic sanitisation and anonymisation features useful for the project.

In this project, each protocol class has its own Python module. The modules that currently exist are `eth`, `arp`, `ip4`, `ip6`, `tcp` and `udp`, `icmp` and `icmp6`. (Albeit some are incomplete, see the critical evaluation in chapter 8).

All of these classes expose the same interface as specified in section 4.2.3 of the design chapter. Because there are so many, this section will focus on one, the IPv4 class in the `ip4` module.

The IPv4 class is one of the more complex Protocol classes, namely because it specifies fragmentation behaviour and has a checksum.

Internally, the IPv4 class makes use of a '`FragmentTracker`' class. This class exists to store references to all packets that make up a fragmented datagram. Until such a packet is complete, the `completed` boolean field is set to false for all constituents. Once the composite datagram is known to be complete, the `completed` field is set to true, a `memorymap` is created from the payloads of all constituent packets, and presented to the next Protocol class.

Once a complete datagram is reassembled, the protocol field in the IP header (see figure 2.3) is used with the `ip.lookup_ip_protocol` function to determine the next Protocol class to use. The `next` field for all constituent IPv4 instances is set to that of the new Protocol class and the *last* fragment becomes the parent for that class.

see IANA for a complete list, http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

5.7 The `pipeline` Package

The pipeline package provides three packet sources, `merge`, `filter` and `identify`. These all behave like packet sources as described in 4.2.2, and accept packet sources as input.

5.7.1 The `merge` Module

The `merge` module contains the *merge* generator function.

This function takes a list of packet sources and two optional parameters, *relative* and *offset*. If relative is true, then the time stamp of the first packet from a source is subtracted from all packets from that source. If offset is set to any number, this value is added to the time of all packets, otherwise the average start time is used in place. This function returns a generator that produces the next chronological packet on each iteration.

5.7.2 The `filter` Module

The `filter` module contains the *filter* generator function.

The filter function takes a packet source, two sets of *prototype lists* and a policy.

A prototype list is simply a list of *prototypes*. A prototype is a 2-tuple, consisting of a protocol name and a set of protocol attributes. These protocol attributes are usually obtained from the `build_attributes` static method in the Protocol class with the corresponding name.

A prototype list matches a protocol identity if:

i) Each prototype in the prototype list has a protocol name matching that of the corresponding protocol instance in the identity.

ii) Each prototype in the prototype list has a set of protocol attributes which, when passed to the `match_attributes` method of the corresponding protocol instance, evaluates to true.

iii) The number of protocol instances in the protocol identity is equal to or greater than the number of prototypes in the prototype list.

There is an internal function, `identity_match`, that implements the matching algorithm described above.

Listing 5.2: *Packet identity matching algorithm.*

```
def identity_match(protocol_instances, prototypes):
    # Compares a packet's identity list (protocol instances) to a list of
    # 'prototypes'.
    # Every prototype must match it's corresponding protocol instance.
    # Prototypes are a (name, attrs) tuple.

    prototype_iter = iter(prototypes)
    match = False
    for protocol_instance, prototype in\
      zip(protocol_instances, prototype_iter):
        if protocol_instance.name is not prototype[0] or \
          not protocol_instance.match_attributes(prototype[1]):
            break
    else:
        # This checks if the number of prototypes exceeds
        # the number of protocol_instances.
        try:
            next(prototype_iter)
        except StopIteration:
```

```
            match = True

    return match
```

These two sets of prototype lists are the *keep* set and *discard* set. Any packet matching something in the keep set is remitted as output, any packet matching something in the discard set is not. If a packet matches no prototype lists, the *policy* is used, either keeping or discarding the packet.

5.7.3 The `identify` Module

The `identify` module contains the *identify* generator function.

This function is very simple, and offers a nicer interface to the `root_identify` function in the `identity.core` module.

For every packet in the provided source, it will pass it to the `root_identify` function. If the identity is incomplete, it will store the packet in a buffer and read the *next* packet, buffering these until the first packet has it's identity completed.

If the packet source is exhausted before a packet has its identity completed, then the incompletely identified packets are returned.

5.8 The `pcpapu` Tool

During development, it was deemed easier to implement the utilities as *subcommands* in a larger utility.

PCPAPU stands for 'Packet Capture Processor and Publishing Utility', and at its basic level, implements the `-l` option, and provides a rudimentary way of loading support code.

This utility comes with seperate help text for all subcommands, see appendix (section 9.1).

5.8.1 The `list` Subcommand

The list subcommand implements the same interface as that of the list tool described in the design section 4.4.2.

The list command is a very simple tool. Input parsing aside, the list tool consists of a rather elegant two lines:

Listing 5.3: *Important section of the list tool.*

```
for packet in identify(source):
    print(packet)
```

5.8.2 The `merge` Subcommand

The merge subcommand implements the same interface as that of the merge tool described in the design section 4.4.3.

The merge subcommand exposes the same interface as the merge pipeline. Its input processing expects a date in a specific form, `YYYY/MM/DD HH:MM:SS`, which is converted into a UNIX timestamp.

Note that this does make for some rather non-intuitive behaviour. In order to merge capture files without altering their timestamps, it is necessary to specify "`1970/1/1 00:00:00`" as the time offset in addition to specifying the `-a` flag. This interaction should be reviewed at a later date.

5.8.3 The `filter` Subcommand

The filter subcommand implements the same interface as that of the filter tool described in the design section 4.4.4.

Just like the merge subcommand, this is implemented in terms of *pipelines*, first using the identify pipeline, then the filter pipeline.

5.8.4 The `maphosts` Subcommand

The maphosts subcommand implements the same interface as that of the maphosts tool described in the design section 4.4.5.

It utilises the identify pipeline, and will call the cascading `replace_hosts` method and `recalculate_checksums` method on the root protocol instance for each packet.

Chapter 6

Testing

The testing for this project was done throughout development. A rather informal mixture of *white box* and *black box* testing was employed throughout most of the development process. Typically, a small test run with known inputs and expected outputs was run after every major feature or re-factoring work was carrier out, and if erroneous or regressive behaviour was observed, code was reviewed and debugged.

6.1 Testing During Development

During development, a straightforward develop-test-debug approach was used to check for correctness in new code. Any newly implemented feature would be tested.

For simple modules, an interactive Python interpreter was started and the module imported and interacted with. Many errors in early modules were detected this way.

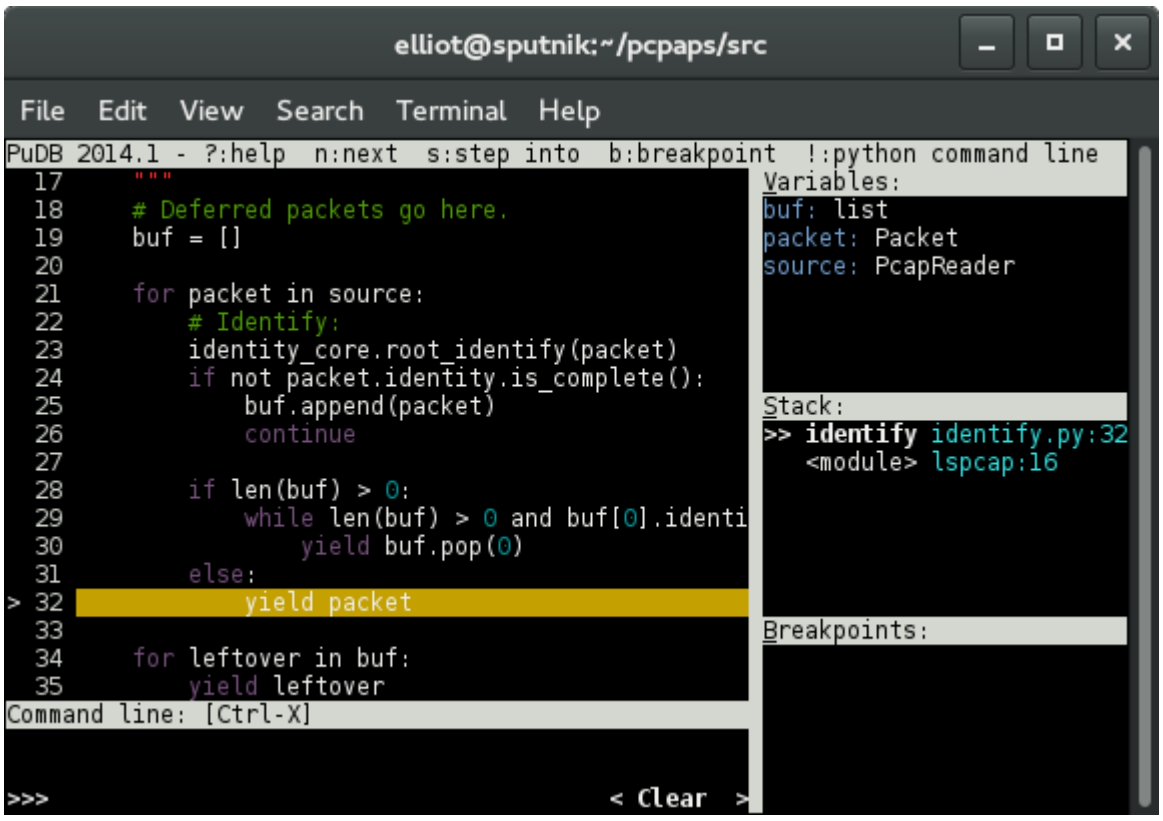

Figure 6.1: *Screenshot of pudb, the debugger used. (http://pypi.python.org/pypi/pudb)*

When the programs became more complex, a debugger was used. This permitted analysis of complex state at runtime without the (mis)use of 'print statements'.

6.1.1 Regression Testing

When any major feature stopped working, the use of version control and the regression itself (altered output, stack traces etc.) made it easy to determine what changes had caused the problem, and made fixing it simple.

Explicit regression testing was done whenever any code re-factoring was carried out.

6.1.2 Output verification

Output was generally verified using *Wireshark*, to see if the output was a valid pcap file and if packet data was modified correctly. Features of Wireshark such as checksum verification were employed to test wheter checksum recalculation algorithms worked.

6.2 Black-Box Testing

For these tests, a real network capture between two computers, containing examples of various protocols (in excess of the ones supported) was used. This packet capture starts from 2014/10/20 at 15:49:59.

6.2.1 Testing the List Tool

The list tool is a very straightforward tool, and was used mostly to verify the output of other tools. Nevertheless, this tool represents a kind of benchmark - if this tool does not work correctly, none of the others are likely to function.

Test ID	**Input**	**Expected Output**	**Actual Output**	**Comments**
LIS1	filepath specified with the `-i` option	Listing of packets.	Listing of packets.	Works as expected
LIS2	capture file piped to standard input	Listing of packets.	Listing of packets.	Works as expected

The exact output of this program was cross-referenced with that of *Wireshark*, with which it agreed with for all supported protocols.

6.2.2 Testing the Merge Tool

To verify the output of these tests, the *list* tool as tested above was used.

Test ID	**Input**	**Expected Output**	**Actual Output**	**Comments**
MER1	filepath specified with the `-i` option, different filepath specified with the `-o` option.	A duplicate of the capture file.	Almost a duplicate of the capture file.	See section 6.3.1.
MER2	like above, with the `-t` option set to `"1970/1/1 00:00:00"`	Capture file claiming to start from the 1st of January, 1970.	Capture file claiming to start from the 1st of January, 1970.	Works as expected

Test ID	Input	Expected Output	Actual Output	Comments
MER3	like above, with the `-t` option set to `"2014/10/30 15:49:59"`	Capture file claiming to start from the 30th of October, 2014.	Capture file claiming to start from the 30th of October, 2014.	Works as expected
MER4	Two files specified with `-i`, the other file being the result of the previous test.	Capture file consisting of packets interleaved from each file, claiming to start from the 15th of October, 2014.	Capture file consisting of packets interleaved from each file, claiming to start from the 15th of October, 2014.	Works as expected
MER5	The `-a` option enabled and the `-t` option set to `"1970/1/1 00:00:00"`, along with two files specified with `-i`, the other file being the result of MER3.	Capture file consisting of packets from each file neatly concatenated, claiming to start from the 10th of October, 2014.	Capture file consisting of packets from each file neatly concatenated, claiming to start from the 10th of October, 2014.	Works as expected

These captures were also opened in *Wireshark*, which reported no abnormalities.

6.2.3 Testing the Filter Tool

Thorough testing for this for the filter tool would require testing all possible Protocol classes. This would require a lot of time, so testing will focus on demonstrating tool usage scenarios using different protocol classes.

Additionally, not all protocols are supported by the library. *Unsupported* protocols are not identified, and packets that contain them will have a short identity chain, e.g. “eth/ip4”.

Test ID	Input	Expected Output	Actual Output	Comments
FIL1	filepath specified with the `-i` option, different filepath specified with the `-o` option.	A duplicate of the capture file.	Almost a duplicate of the capture file.	See section 6.3.1.
FIL2	As above, but with the `-d` option given `eth/arp` as an argument.	A similar capture file, stripped of all Ethernet/ARP packets.	A similar capture file, stripped of all Ethernet/ARP packets.	Works as expected.
FIL3	As above, but using `-k` instead of `-d` and with -p set to `discard`	A capture file, containing nothing but Ethernet/ARP packets.	A capture file, containing nothing but Ethernet/ARP packets.	Works as expected.
FIL4	Similar I/O flags, with `-k eth/ip4/tcp` and `-d eth/ip4`	A similar capture file, where all Ethernet/IPv4 packets are removed unless they carry TCP.	A similar capture file, where all Ethernet/IPv4 packets are removed unless they carry TCP, or use an unsupported protocol on top of IP.	This was discovered to be a bug in the filter pipeline, where more specific prototypes would apparently match less specific identities.

Test ID	Input	Expected Output	Actual Output	Comments
FIL4	Similar I/O flags, with `-k eth/ip4/tcp` and `-d eth/ip4`	A similar capture file, where all Ethernet/IPv4 packets are removed unless they carry TCP.	A similar capture file, where all Ethernet/IPv4 packets are removed unless they carry TCP.	Works as expected, error in filter pipeline fixed.

The filter program performed its job admirably, with only one error corrected in testing.

6.2.4 Testing the Maphosts Tool

Replacement is only expected to function for the protocols supported. This meant that occurrences in unsupported protocols such as DNS are *not* expected to be replaced.

Test ID	Input	Expected Output	Actual Output	Comments
FIL1	filepath specified with the `-i` option, different filepath specified with the `-o` option.	A duplicate of the capture file.	Almost a duplicate of the capture file, with some TCP checksums corrected.	See extended commentary in section 6.3.2.
FIL2	Similar, with the addition of `-m de:ad:be:ef:13:37 = bc:ee:7b:df:e8:08`	A variant of the capture file, with all references to the MAC address `de:ad:be:ef:13:37` replaced with `bc:ee:7b:df:e8:08`.	A variant of the capture file, with all references to the MAC address `de:ad:be:ef:13:37` replaced with `bc:ee:7b:df:e8:08`.	Worked as expected. References to the old MAC address persisted in the form of link-local IPv6 address. This is not considered a bug.
FIL3	Similar, but with `-4 10.128.128.1 = 192.168.0.20`	A variant of the capture file, with all references to the IP address `10.128.128.1` replaced with `192.168.0.20`.	A variant of the capture file, with all references to the IP address `10.128.128.1` replaced with `192.168.0.20`.	Works as expected.
FIL4	Similar, but with `-6 fe80::14fa:1cff:feaa:7cfa = fe80::f00f:bad0`	A variant of the capture file, with all references to the IP address `fe80::14fa:1cff:feaa:7cfa` replaced with `fe80::f00f:bad0`.	A variant of the capture file, with all references to the IP address `fe80::14fa:1cff:feaa:7cfa` replaced with `fe80::f00f:bad0`.	Works as expected.

6.3 Interesting results

6.3.1 Pcap Global Header Snapshot Length Difference

The maximum length field in the pcap global header differed, otherwise these captures were identical.

This is not to be considered an error, the PcapWriter class, by default, creates output files

To elaborate, IPv6 link local addresses are often based on the MAC address of the link hardware they are set on. These addresses are actually IPv6 addresses, and hence should *not* be replaced as a MAC address.

with a snapshot length of 65535. This is well in excess of any packet length actually sent over the network, and was suggested on the libpcap format description as a value. Wireshark itself uses a larger value.

6.3.2 TCP/UDP checksums

Much to the authors surprise, there were *less* TCP checksum errors in the reproduced packet capture. It would seem that all TCP PSH command packets sent *from the host conducting the packet capture* had incorrect checksums in the original sample capture, yet these packets were never resent. The same was true with UDP checksums from this host - incorrect.

This was determined to be because of TCP/UDP checksum offloading on said host, meaning that the capture software recorded TCP/UDP segments *before* the correct checksum had been computed and sent. This is supported by the fact correct checksums being present in segments sent *to the host*.

Chapter 7

Future Work

The author hopes the current system provides useful utilities and a suitable groundwork for further extension, but naturally, there are aspects that would make the current system better.

i) Adding an explicit notion of 'groups' to make filtering fragmented protocols easier.

ii) A more rigorous definition of what constitutes a set of protocol attributes.

iii) A tool to allow a more general form of packet manipulation. The `set_attributes` method for protocols, which was unused by all existing tools, was designed to allow this.

iv) More tools to help with *reducing* packet capture files to simpler states. A tool that, for instance, extracts TCP sessions into individual files would be beneficial.

v) A utility that maintains a database of reduced captures, that constructs captures according to some kind of abstract difficulty. This would provide a more *automated* solution to the original problem statement.

vi) A utility for introducing variance in packet captures, e.g. extracting IP addresses and randomising them. Support for replacement is present, so again, most of the groundwork is there for this.

vii) Altering the packet data model to support *adding* or *removing* data rather then just changing it.

Chapter 8

Critical Evaluation

A lot of the groundwork needed for future development of this rather large project has been covered.

8.1 Success

The author is generally pleased with the outcome of the project, a working suite of tools.

8.1.1 Protocol Support

The tool suite supports modifying a number of essential protocols fully, with falsification undetectable by *Wireshark* for all supported protocols. Packet captures falsified in this way maintain the same degree of legibility as that of the original, with few if any hints that the file has been tampered with.

Additionally, the author managed to implement core IPv6 support, and can correctly identify the protocols it carries. IPv6 address replacement also works correctly, allowing all typical address types to be changed.

8.1.2 Performance

While performance was never a primary goal, good performance is indicative of good software.

The order of growth (determined by code analysis) for most of the algorithms in the code is $O(n)$, where n is the number of packets to be processed. This is the best that can reasonably be expected from any kind of stream processor.

The tools seem to perform well, processing moderately sized captures (approx. 600 packets) in immeasurable time. Admittedly, no strict performance tests have been carried out, things like CPU time and I/O time have not been factored in, but the author nonetheless conducted some informal measurements made with the UNIX `time` command and a process memory monitor.

The merge tool seems to have a linear complexity. A merge between two large capture files, producing a packet capture around 154 megabytes containing 338 *thousand* packets, took only 6 seconds of real time to create (*Wireshark* took roughly 3 seconds to *open* the file, leading the author to believe that this may be disk I/O bound). When this file was then merged with itself, it took a predictable 12 seconds of real time.

The filter tool seems to be fairly memory efficient. When filtering the 308 megabyte file produced from the test above, it took 46 seconds to filter all ARP packets from the input and used around 6 megabytes of memory (the python interpreter itself seems to use approximately 3 megabytes). When given an additional criteria to filter IPv6 packets, 49 seconds were required, while retaining the same memory usage.

8.2 Criticism

8.2.1 Protocol Support

The project was never expected to be 100% complete in terms of supporting protocols, but some important protocol classes are incomplete. Most notably, the TCP class currently supports basic tests against source/destination ports and correcting the checksum - the TCP state machine is non-existent and TCP sessions cannot be tracked. As a result of this, it is currently not possible to distinguish between protocols using TCP as their transport.

The reason TCP support is incomplete is simply time. The author failed to account for the complexity of tracking TCP sessions, so the class was only half-implemented.

The author would like to point out that despite these limitations, the tools remain useful for their purpose; often *all* TCP traffic is interesting and as checksum recalculation is implemented properly, meaning packets *carrying* a TCP session can be manipulated without issue.

Additionally, ICMP (and ICMPv6) only exists as a stub Protocol class. This is an issue as ICMP, much like TCP and UDP, contains checksums that include carrier information.

8.2.2 The `set_attributes` method

The `set_attributes` method is not used by any code in the project. It was the author's hope that a `modify` tool could be produced, but this did not happen due to time constraints.

The `set_attributes` method remains unimplemented in most protocol classes, with only ARP and Ethernet fully supporting it. This should have been made clearer in the documentation.

8.3 Retrospective Design Successes and Criticisms

One of the biggest successes is the total lack of assumptions protocol classes make about their parent protocol. This renders each protocol class extremely flexible, for instance, IPv4 and IPv6 can both be passed over IP. Both protocol classes make no assumptions about the parent protocol, so just two classes can be used to support IPv4 over IPv4, IPv4 over IPv6, IPv6 over IPv4 and IPv6 over IPv6, in addition to both IPv4 and IPv6 over Ethernet.

Another positive aspect of the design is that of the exposed interface. Exposing a high level interface using the concept of a *packet source* means that tools and utilities making use of the library mainly have to concern themselves with sanitising user input - the main body of the `list` tool is just two lines long.

As for criticisms, the Protocol class is somewhat unwieldy. In retrospect, a clear distinction between *protocol information* and *dissection* could have been made, a so called *ProtocolDissector* class or function.

8.4 Retrospective Implementation Successes and Criticisms

The interface provided by the implementation seems fairly intuitive for a commandline application, and the documentation provided by the `--help` parameter provides a useful guide.

It is hard to accurately judge performance when there are neither real-time requirements nor comparable tools, but it does appear to have generally good performance, with simple operations almost always being bottlenecked by storage bandwidth.

Due to protocol specifications, TCP and UDP *do* check the parent for the purpose of checksum recomputation.

8.5 Insight

Research, design and implementation were all done concurrently. This methodology turned out to work quite well - the project is somewhat 'complete' and a (hopefully) decent report has been produced.

The author started off with a fairly strong knowledge of *what* made computer networks tick, but not exactly *how*.

This is no longer the case.

The author has new found respect and admiration for the protocols developed over 30 years ago, which despite significant changes in Internet culture and the way security is handled, still remain working.

8.6 Important Lessons

The author is the type of programmer who will write code first, then ask requirements later. While this has worked in the past for the author, it should be noted that in the case of this project, it did not work. The author has discovered that there *is* a substitute for planning - it is called sleep deprivation.

Elliot Thomas

June 28, 2015

References

[1] The Wireshark team (Accessed April, 2015)
The Wireshark network protocol analyser
https://www.wireshark.org/

[2] Richard Sharpe, Guy Harris, Ulf Lamping (4th of March, 2015)
EDITCAP (1), The Wireshark Network Analyzer

[3] Gerald Combs, numerous others (4th of March, 2015)
WIRESHARK (1), The Wireshark Network Analyzer

[4] Aaron Turner, Fred Klassen (Accessed April, 2015)
tcpreplay home page(s). http://tcpreplay.appneta.com
(see also: http://tcpreplay.synfin.net/)

[5] Addy Yeow Chin Heng (Accessed April, 2015)
libpcap-based Ethernet packet generator.
http://bittwist.sourceforge.net/

[6] Brian W. Kernighan, Dennis M. Ritchie (1988)
The C Programming Language, 2nd edition.

[7] Bjarne Stroustrup (1997)
The C++ Programming Language, 3rd edition.

[8] The GCC Team (30th of June, 2014)
The GNU Compiler for the Java Programming Language
https://gcc.gnu.org/java/

[9] Oracle Corporation (Accessed April, 2015)
'Learn about Java Technology'
https://www.java.com/en/about/

[10] Python Software Foundation (Accessed April, 2015)
Python
https://www.python.org/

[11] Simon Peyton Jones (December, 2014)
The Haskell 98 Report
https://www.haskell.org/onlinereport/intro.html

[12] Mono Project (Accessed April, 2015)
Cross platform, open source .NET framework
http://www.mono-project.com

[13] Various authors (13th of April, 2014)
Should I use Python 2 or Python 3 for my development activity?
https://wiki.python.org/moin/Python2orPython3

[14] Peter H. Salus. (1st of June, 1994)
A Quarter-Century of Unix

[15] Dr. Winston W. Royce (26th August, 1970)
MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS
http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

[16] IEEE Computer Society (28th December, 2012)
IEEE Standard for Ethernet (IEEE Std 802.3-2012)
http://standards.ieee.org/getieee802/download/802.3-2012_section1.pdf

[17] David C. Plummer (November, 1982)
An Ethernet Address Resolution Protocol, or
Converting Network Protocol Addresses
https://tools.ietf.org/html/rfc826

[18] Information Sciences Institute, University of Southern California (September, 1981)
INTERNET PROTOCOL
https://tools.ietf.org/html/rfc791

[19] S. Deering, R. Hinden, The Internet Society (December 1998)
Internet Protocol, Version 6 (IPv6)
https://tools.ietf.org/html/rfc2460

[20] Number Resources Organisation (3rd of February, 2011)
Free Pool of IPv4 Address Space Depleted
https://www.nro.net/news/ipv4-free-pool-depleted

[21] J. Postel (28th of August, 1980)
User Datagram Protocol
https://tools.ietf.org/html/rfc768

[22] Information Sciences Institute, University of Southern California (September, 1981)
TRANSMISSION CONTROL PROTOCOL
https://tools.ietf.org/html/rfc793

[23] P. Mockapetris (November, 1987)
DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION
https://tools.ietf.org/html/rfc1035

[24] T. Narten, E. Nordmark, W. Simpson, H. Soliman (September, 2007)
Neighbor Discovery for IP version 6 (IPv6)
https://tools.ietf.org/html/rfc4861

[25] S. Kawamura, M. Kawashima (August 2010)
A Recommendation for IPv6 Address Text Representation
https://tools.ietf.org/html/rfc5952

[26] Guido van Rossum, Barry Warsaw, Nick Coghlan (1st of August, 2013)
PEP 8 - Style Guide for Python Code
https://www.python.org/dev/peps/pep-0008/

[27] Johann C. Rocholl (last updated 18th of March, 2015)
pep8 - Python style guide checker
http://github.com/jcrocholl/pep8/

[28] Ulf Lamping, Guy Harris, Gerald Combs, numerous others (Last edited: 29th of July, 2013)
'Development/LibpcapFileFormat'
https://wiki.wireshark.org/Development/LibpcapFileFormat

Chapter 9

Appendix

9.1 Program help text

9.1.1 pcpapu --help

```
usage: ./pcpapu [-h] [-l LOAD] {filter,merge,maphosts,list} ...

Packet Caputre Processor And Publishing Utility.

positional arguments:
  {filter,merge,maphosts,list}
                        subcommand
  args                  subcommand arguments

optional arguments:
  -h, --help            show this help message and exit
  -l LOAD, --load LOAD
```

9.1.2 pcpapu list --help

```
usage: ./pcpapu list [-h] [-i filepath]

Print packet metadata to standard output.

optional arguments:
  -h, --help            show this help message and exit
  -i filepath, --in filepath
                        Input file
```

9.1.3 pcpapu merge --help

```
usage: ./pcpapu merge [-h] [-i filepath] [-o filepath] [-r] [-a] [-t time]

Merge packet captures and adjust their times.

optional arguments:
  -h, --help            show this help message and exit
  -i filepath, --in filepath
                        Input file (can occur multiple times)
  -o filepath, --out filepath
                        Output file
```

```
  -r, --relative        Use relative times. (Default)
  -a, --absolute        Do not use relative times.
  -t time, --time-offset time
                        Time offset, in Y/M/D H:M:S format.
```

9.1.4 pcpapu filter --help

```
usage: ./pcpapu filter [-h] [-i filepath] [-o filepath] [-k identity]
                       [-d identity] [-p keep/discard]

Filter packets from a capture.
Identities have the form:
<identity> ::= <prototype> | <prototype> / <identity>
<prototype> ::= <protocol name> | <protocol name> : <protocol attributes>
For example, to identify any ethernet packet carrying TCP over IP: /eth/ip4/tcp
More specific identities can be specified.
For example, /eth:dmac=30:21:af:42:73:30/ip4
will match all ethernet/IPv4 packets with that destination MAC address.

optional arguments:
  -h, --help            show this help message and exit
  -i filepath, --in filepath
                        Input file
  -o filepath, --out filepath
                        Output file
  -k identity, --keep identity
                        Packets matching this identity will be kept.
  -d identity, --discard identity
                        Packets matching this identity will be discarded.
  -p keep/discard, --policy keep/discard
                        The policy for any packet not matching an identity.
                        (Default is keep)
```

9.1.5 pcpapu maphosts --help

```
usage: ./pcpapu maphosts [-h] [-i filepath] [-o filepath] [-4 IP4-pair]
                         [-6 IP6-pair] [-m MAC-pair]

Replace host identities inside packets from a capture.

optional arguments:
  -h, --help            show this help message and exit
  -i filepath, --in filepath
                        Input file
  -o filepath, --out filepath
                        Output file
  -4 IP4-pair, --ip4 IP4-pair
                        IPv4 address find/replace pair.
  -6 IP6-pair, --ip6 IP6-pair
                        IPv6 address find/replace pair.
  -m MAC-pair, --mac MAC-pair
                        MAC address find/replace pair.
```

9.2 `packet` package API documentation

This documentation was generated from the documentation strings in the packet package written for this project using pydoc.

9.2.1 packet.capfile.core

```
Help on module packet.capfile.core in packet.capfile:

NAME
    packet.capfile.core

DESCRIPTION
    core:
     - Abstract classes for packet capture file readers/writers.

CLASSES
    builtins.Exception(builtins.BaseException)
        PacketIOError
    builtins.object
        PacketReader
        PacketWriter

    class PacketIOError(builtins.Exception)
     |  Exception raised by Packet(Reader|Writer)s.
     |
     |  Method resolution order:
     |      PacketIOError
     |      builtins.Exception
     |      builtins.BaseException
     |      builtins.object
     |
     |  Data descriptors defined here:
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Methods inherited from builtins.Exception:
     |
     |  __init__(self, /, *args, **kwargs)
     |      Initialize self.  See help(type(self)) for accurate signature.
     |
     |  __new__(*args, **kwargs) from builtins.type
     |      Create and return a new object.  See help(type) for accurate signature.
     |
     |  ----------------------------------------------------------------------
     |  Methods inherited from builtins.BaseException:
     |
     |  __delattr__(self, name, /)
     |      Implement delattr(self, name).
     |
     |  __getattribute__(self, name, /)
     |      Return getattr(self, name).
     |
     |  __reduce__(...)
     |
     |  __repr__(self, /)
     |      Return repr(self).
     |
     |  __setattr__(self, name, value, /)
     |      Implement setattr(self, name, value).
     |
     |  __setstate__(...)
     |
     |  __str__(self, /)
     |      Return str(self).
     |
     |  with_traceback(...)
     |      Exception.with_traceback(tb) --
     |      set self.__traceback__ to tb and return self.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors inherited from builtins.BaseException:
```

```
 |
 |  __cause__
 |      exception cause
 |
 |  __context__
 |      exception context
 |
 |  __dict__
 |
 |  __suppress_context__
 |
 |  __traceback__
 |
 |  args

class PacketReader(builtins.object)
 |  Abstract class for packet readers.
 |  Implementatons must provide the 'read_packet' method.
 |  This class implements the __next__ method based on read_packet()
 |
 |  Methods defined here:
 |
 |  __iter__(self)
 |      This makes the object an iterable.
 |      A 'self iterable' as it simply returns self.
 |
 |  __next__(self)
 |      Iterator protocol interface.
 |
 |  close(self)
 |      Abstract method close. Should close filesystem resources.
 |
 |  read_packet(self)
 |      Abstract method read_packet.
 |      Should return the 'Packet' object, or None if there are no packets left.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class PacketWriter(builtins.object)
 |  Abstract class for packet writers.
 |  Implementations must provide the 'write_packet' method.
 |
 |  Methods defined here:
 |
 |  close(self)
 |      Abstract method close. Should cleanup any filesystem resources.
 |
 |  write_packet(self, packet)
 |      stact method write_packet
 |      ould take one argument, the 'Packet' object.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

9.2.2 packet.capfile.pcap

```
Help on module packet.capfile.pcap in packet.capfile:

NAME
    packet.capfile.pcap

CLASSES
```

```
packet.capfile.core.PacketIOError(builtins.Exception)
    PcapFormatError
    PcapRangeError
packet.capfile.core.PacketReader(builtins.object)
    PcapReader
packet.capfile.core.PacketWriter(builtins.object)
    PcapWriter

class PcapFormatError(packet.capfile.core.PacketIOError)
 |  Exception raised when a file format error occurs.
 |
 |  Method resolution order:
 |      PcapFormatError
 |      packet.capfile.core.PacketIOError
 |      builtins.Exception
 |      builtins.BaseException
 |      builtins.object
 |
 |  Data descriptors inherited from packet.capfile.core.PacketIOError:
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.Exception:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.BaseException:
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __reduce__(...)
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __setstate__(...)
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  with_traceback(...)
 |      Exception.with_traceback(tb) --
 |      set self.__traceback__ to tb and return self.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from builtins.BaseException:
 |
 |  __cause__
 |      exception cause
 |
 |  __context__
 |      exception context
 |
 |  __dict__
 |
 |  __suppress_context__
 |
 |  __traceback__
 |
 |  args
```

```
class PcapRangeError(packet.capfile.core.PacketIOError)
 |  Exception raised when a unrepresentable value is encountered.
 |
 |  Method resolution order:
 |      PcapRangeError
 |      packet.capfile.core.PacketIOError
 |      builtins.Exception
 |      builtins.BaseException
 |      builtins.object
 |
 |  Data descriptors inherited from packet.capfile.core.PacketIOError:
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.Exception:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.BaseException:
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __reduce__(...)
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __setstate__(...)
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  with_traceback(...)
 |      Exception.with_traceback(tb) --
 |      set self.__traceback__ to tb and return self.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from builtins.BaseException:
 |
 |  __cause__
 |      exception cause
 |
 |  __context__
 |      exception context
 |
 |  __dict__
 |
 |  __suppress_context__
 |
 |  __traceback__
 |
 |  args

class PcapReader(packet.capfile.core.PacketReader)
 |  PcapReader: reader for pcap files.
 |
 |  Method resolution order:
 |      PcapReader
 |      packet.capfile.core.PacketReader
 |      builtins.object
 |
```

```
 |  Methods defined here:
 |
 |  __init__(self, fstream, magic=None)
 |      Creates a PcapReader from an open stream.
 |      Takes one mandatory and one optional argument,
 |       - fstream: The readable stream to use.
 |       - magic: The first four bytes of the file.
 |      If this is None, four bytes are read from the stream first.
 |
 |  close(self)
 |      Closes the stream.
 |
 |  read_packet(self)
 |      Reads a packet record header and it's data from the stream.
 |      Returns a Packet object, or None on EOF.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from packet.capfile.core.PacketReader:
 |
 |  __iter__(self)
 |      This makes the object an iterable.
 |      A 'self iterable' as it simply returns self.
 |
 |  __next__(self)
 |      Iterator protocol interface.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from packet.capfile.core.PacketReader:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class PcapWriter(packet.capfile.core.PacketWriter)
 |  PcapWriter: writer for pcap files.
 |
 |  Method resolution order:
 |      PcapWriter
 |      packet.capfile.core.PacketWriter
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, stream, magic=b'\xd4\xc3\xb2\xa1', thiszone=0, snaplen=65535, network=1)
 |      Setup a new PcapWriter object, and write a global header to the stream.
 |
 |  close(self)
 |      Closes the stream.
 |
 |  write_packet(self, packet)
 |      Writes a packet record header and data to the stream.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from packet.capfile.core.PacketWriter:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

FUNCTIONS
    pcap_magic_resolve(magic)
        Function that resolves pcap's magic number into an appropriate struct/scale.
        Raises PcapFormatError on invalid magic.

DATA
    PCAP_BE_GLOB_HDR = <Struct object>
    PCAP_BE_NANOSEC = b'\xa1\xb2<M'
    PCAP_BE_PKT_HDR = <Struct object>
    PCAP_BE_REGULAR = b'\xa1\xb2\xc3\xd4'
    PCAP_LE_GLOB_HDR = <Struct object>
    PCAP_LE_NANOSEC = b'M<\xb2\xa1'
```

```
    PCAP_LE_PKT_HDR = <Struct object>
    PCAP_LE_REGULAR = b'\xd4\xc3\xb2\xa1'
    PCAP_MAJOR_VER = 2
    PCAP_MINOR_VER = 4
```

9.2.3 packet.identity.arp

```
Help on module packet.identity.arp in packet.identity:

NAME
    packet.identity.arp

CLASSES
    packet.identity.core.Protocol(builtins.object)
        ARP

    class ARP(packet.identity.core.Protocol)
     |  Method resolution order:
     |      ARP
     |      packet.identity.core.Protocol
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  __init__(self, data, prev)
     |
     |  get_attributes(self)
     |      Retrieve a set of attributes describing fields in this protocol.
     |
     |  replace_hosts(self, hostmap)
     |      This method replaces MAC and IP addresses of both the sender and
     |      target based on the given mapping.
     |
     |  set_attributes(self, attrs)
     |      Alter packet data to match a set of protocol attributes.
     |
     |  ----------------------------------------------------------------------
     |  Static methods defined here:
     |
     |  build_attributes(attrstr)
     |      Creates a set of ARP attributes from an attribute string.
     |
     |  interpret_packet(data, parent)
     |      Interpret packet data for this protocol.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  hardware_is_ethernet
     |
     |  protocol_is_ipv4
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  name = 'arp'
     |
     |  ----------------------------------------------------------------------
     |  Methods inherited from packet.identity.core.Protocol:
     |
     |  __iter__(self)
     |      Generator method for accessing this/child protocol instances.
     |
     |  is_complete(self)
     |      Returns True if this and all child protocols are complete.
     |
     |  match_attributes(self, tattrs)
     |      Tests wether this ProtocolIdentity matches a set of attributes.
     |      This is the default implementation and compares attributes of this
     |      instance (from get_attributes) to the provided attributes (tattrs).
     |      This is not commutative - all of the target keys MUST be in
     |      this ProtocolIdentity's keys. Additionally, if the target's key is
     |      equal to None, it is treated as a wildcard and matches.
     |
     |  recalculate_checksums(self)
```

```
 |      This method should recalculate any kind of checksum used by this
 |      protocol, after any 'child' checksums have been recomputed.
 |      That is to say, the recalculation should propagate up, from the
 |      highest-level protocol to the lowest. The default implementation does
 |      nothing other then this propogation and should suffice for
 |      protocols without validation.
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from packet.identity.core.Protocol:
 |
 |  interpret_stream(stream, parent)
 |      Abstract static method interpret_stream.
 |      Like interpret_packet, this takes two arguments, a 'parent' protocol
 |      instance (which can be none) and a 'stream' object.
 |      This method should create an instance of it's class, determine the next
 |      (if any) protocol to interpret, (setting the next field), then
 |      return said instance.
 |
 |  reset_state()
 |      This static method should restore the initial state to any kind of
 |      state tracker this class uses. This means, any data held by the class
 |      to associate multiple bits of data (think IP fragmentation or TCP)
 |      should be forgotten.
 |      The default implementation does nothing and should suffice for simple protocols.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from packet.identity.core.Protocol:
 |
 |  completed
 |
 |  data
 |
 |  next
 |
 |  prev

DATA
    ARP_HARDWARE_ETHERNET = 1
    ARP_MIN_SIZE = 8
    ARP_OPCODE_REPLY = 2
    ARP_OPCODE_REQUEST = 1
    ARP_PROTOCOL_IPV4 = 2048
```

9.2.4 packet.identity.core

```
Help on module packet.identity.core in packet.identity:

NAME
    packet.identity.core

DESCRIPTION
    core: root module of identification system
    Contains class definition for a 'Stream'.
    Contains abstract class definition for Protocol and CarrierProtocol.

CLASSES
    builtins.Exception(builtins.BaseException)
        ProtocolFormatError
    builtins.object
        Protocol
            CarrierProtocol
            ProtocolStub
        Stream
    enum.Enum(builtins.object)
        AddrType

    class AddrType(enum.Enum)
     |  Method resolution order:
     |      AddrType
     |      enum.Enum
     |      builtins.object
     |
     |  Data and other attributes defined here:
     |
     |  IP4 = <AddrType.IP4: 'ip4'>
```

```
 |
 |  IP6 = <AddrType.IP6: 'ip6'>
 |
 |  MAC = <AddrType.MAC: 'mac'>
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from enum.Enum:
 |
 |  name
 |      The name of the Enum member.
 |
 |  value
 |      The value of the Enum member.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from enum.EnumMeta:
 |
 |  __members__
 |      Returns a mapping of member name->value.
 |
 |      This mapping lists all enum members, including aliases. Note that this
 |      is a read-only view of the internal mapping.

class CarrierProtocol(Protocol)
 |  This class represents a protocol that carries other protocols.
 |  In addition to the Protocol methods, a CarrierProtocol needs to implement
 |  the get_route and get_route_reciprocal methods.
 |
 |  Method resolution order:
 |      CarrierProtocol
 |      Protocol
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  get_payload_length(self)
 |      Abstract method returning the lengh of the payload,
 |      if indicated in the protocol.
 |
 |  get_route(self)
 |      Abstract method returning a hashable object that represents
 |      the 'route' (the destination, source and direction) a carrier
 |      protocol would direct a packet.
 |
 |  get_route_reciprocal(self)
 |      Abstract method returning a hashable object not unlike the above,
 |      that returns the opposite route.
 |      E.g. if get_route returned A -> B, this should return B -> A.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Protocol:
 |
 |  __init__(self, data, prev)
 |      Constructor for protocol instances.
 |
 |  __iter__(self)
 |      Generator method for accessing this/child protocol instances.
 |
 |  get_attributes(self)
 |      Abstract method get_attributes.
 |      Should return a dict of useful attributes.
 |
 |  is_complete(self)
 |      Returns True if this and all child protocols are complete.
 |
 |  match_attributes(self, tattrs)
```

```
 |      Tests wether this ProtocolIdentity matches a set of attributes.
 |      This is the default implementation and compares attributes of this
 |      instance (from get_attributes) to the provided attributes (tattrs).
 |      This is not commutative - all of the target keys MUST be in
 |      this ProtocolIdentity's keys. Additionally, if the target's key is
 |      equal to None, it is treated as a wildcard and matches.
 |
 |  recalculate_checksums(self)
 |      This method should recalculate any kind of checksum used by this
 |      protocol, after any 'child' checksums have been recomputed.
 |      That is to say, the recalculation should propagate up, from the
 |      highest-level protocol to the lowest. The default implementation does
 |      nothing other then this propogation and should suffice for
 |      protocols without validation.
 |
 |  replace_hosts(self, hostmap)
 |      This method should replace instances of host identification,
 |      namely IP addresses and MAC addresses.
 |      This operation should propagate to child protocols.
 |      The default implementation does nothing other then this propogation
 |      and should suffice for protocols without any kind of host identification.
 |
 |  set_attributes(self, attrs)
 |      Abstract method set_attributes.
 |      Should accept a dict of attributes and update data accordingly.
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from Protocol:
 |
 |  build_attributes(attrstr)
 |      Abstract static method build_attributes.
 |      Should return a dict of attributes based on a human-readable
 |      expression string.
 |
 |  interpret_packet(data, parent)
 |      Abstract static method interpret_packet.
 |      This takes two arguments, a 'parent' protocol instance
 |      (which can be none) and data.
 |      This should create an instance of it's class, determine the next
 |      (if any) protocol to interpret (setting the next field), then
 |      return said instance.
 |
 |  interpret_stream(stream, parent)
 |      Abstract static method interpret_stream.
 |      Like interpret_packet, this takes two arguments, a 'parent' protocol
 |      instance (which can be none) and a 'stream' object.
 |      This method should create an instance of it's class, determine the next
 |      (if any) protocol to interpret, (setting the next field), then
 |      return said instance.
 |
 |  reset_state()
 |      This static method should restore the initial state to any kind of
 |      state tracker this class uses. This means, any data held by the class
 |      to associate multiple bits of data (think IP fragmentation or TCP)
 |      should be forgotten.
 |      The default implementation does nothing and should suffice for simple protocols.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Protocol:
 |
 |  completed
 |
 |  data
 |
 |  next
 |
 |  prev
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from Protocol:
 |
 |  name = None

class Protocol(builtins.object)
 |  This class represents a protocol.
```

```
|  An instance of a protocol has a set of 'attributes', such as
|  fields in the header of a packet. An attribute is a smaller piece
|  of variable data in a protocol, for instance, a sender's IP address
|  is an attribute of an IP header. A protocol also has a 'next' and 'prev' -
|  child and parent protocols, respectively.
|  All protocol instances have a 'completed' flag.
|
|  Methods defined here:
|
|  __init__(self, data, prev)
|      Constructor for protocol instances.
|
|  __iter__(self)
|      Generator method for accessing this/child protocol instances.
|
|  get_attributes(self)
|      Abstract method get_attributes.
|      Should return a dict of useful attributes.
|
|  is_complete(self)
|      Returns True if this and all child protocols are complete.
|
|  match_attributes(self, tattrs)
|      Tests wether this ProtocolIdentity matches a set of attributes.
|      This is the default implementation and compares attributes of this
|      instance (from get_attributes) to the provided attributes (tattrs).
|      This is not commutative - all of the target keys MUST be in
|      this ProtocolIdentity's keys. Additionally, if the target's key is
|      equal to None, it is treated as a wildcard and matches.
|
|  recalculate_checksums(self)
|      This method should recalculate any kind of checksum used by this
|      protocol, after any 'child' checksums have been recomputed.
|      That is to say, the recalculation should propagate up, from the
|      highest-level protocol to the lowest. The default implementation does
|      nothing other then this propogation and should suffice for
|      protocols without validation.
|
|  replace_hosts(self, hostmap)
|      This method should replace instances of host identification,
|      namely IP addresses and MAC addresses.
|      This operation should propagate to child protocols.
|      The default implementation does nothing other then this propogation
|      and should suffice for protocols without any kind of host identification.
|
|  set_attributes(self, attrs)
|      Abstract method set_attributes.
|      Should accept a dict of attributes and update data accordingly.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  build_attributes(attrstr)
|      Abstract static method build_attributes.
|      Should return a dict of attributes based on a human-readable
|      expression string.
|
|  interpret_packet(data, parent)
|      Abstract static method interpret_packet.
|      This takes two arguments, a 'parent' protocol instance
|      (which can be none) and data.
|      This should create an instance of it's class, determine the next
|      (if any) protocol to interpret (setting the next field), then
|      return said instance.
|
|  interpret_stream(stream, parent)
|      Abstract static method interpret_stream.
|      Like interpret_packet, this takes two arguments, a 'parent' protocol
|      instance (which can be none) and a 'stream' object.
|      This method should create an instance of it's class, determine the next
|      (if any) protocol to interpret, (setting the next field), then
|      return said instance.
|
|  reset_state()
|      This static method should restore the initial state to any kind of
```

```
 |      state tracker this class uses. This means, any data held by the class
 |      to associate multiple bits of data (think IP fragmentation or TCP)
 |      should be forgotten.
 |      The default implementation does nothing and should suffice for simple protocols.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  completed
 |
 |  data
 |
 |  next
 |
 |  prev
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  name = None

class ProtocolFormatError(builtins.Exception)
 |  Exception raised when a protocol class encounters an error dissecting a packet.
 |
 |  Method resolution order:
 |      ProtocolFormatError
 |      builtins.Exception
 |      builtins.BaseException
 |      builtins.object
 |
 |  Data descriptors defined here:
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.Exception:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.BaseException:
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __reduce__(...)
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __setstate__(...)
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  with_traceback(...)
 |      Exception.with_traceback(tb) --
 |      set self.__traceback__ to tb and return self.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from builtins.BaseException:
 |
 |  __cause__
 |      exception cause
```

```
 |
 |  __context__
 |      exception context
 |
 |  __dict__
 |
 |  __suppress_context__
 |
 |  __traceback__
 |
 |  args

class ProtocolStub(Protocol)
 |  Protocol class for 'stub' handlers - protocols in name only.
 |  This class doubles as the definition for the 'unknown' protocol.
 |
 |  Method resolution order:
 |      ProtocolStub
 |      Protocol
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  get_attributes(self)
 |      Stub. Returns an empty dictionary.
 |
 |  match_attributes(self, attrs)
 |      Stub. Always returns true.
 |
 |  set_attributes(self, attrs)
 |      Stub. Does nothing.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  interpret_packet(data, parent) from builtins.type
 |      Stub. Returns an instance of this class.
 |
 |  interpret_stream(stream, parent) from builtins.type
 |      Stub. Returns an instance of this class.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  build_attributes(attrstr)
 |      Stub. Returns an empty dictionary.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  name = 'unknown'
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Protocol:
 |
 |  __init__(self, data, prev)
 |      Constructor for protocol instances.
 |
 |  __iter__(self)
 |      Generator method for accessing this/child protocol instances.
 |
 |  is_complete(self)
 |      Returns True if this and all child protocols are complete.
 |
 |  recalculate_checksums(self)
 |      This method should recalculate any kind of checksum used by this
```

```
 |      protocol, after any 'child' checksums have been recomputed.
 |      That is to say, the recalculation should propagate up, from the
 |      highest-level protocol to the lowest. The default implementation does
 |      nothing other then this propogation and should suffice for
 |      protocols without validation.
 |
 |  replace_hosts(self, hostmap)
 |      This method should replace instances of host identification,
 |      namely IP addresses and MAC addresses.
 |      This operation should propagate to child protocols.
 |      The default implementation does nothing other then this propogation
 |      and should suffice for protocols without any kind of host identification.
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from Protocol:
 |
 |  reset_state()
 |      This static method should restore the initial state to any kind of
 |      state tracker this class uses. This means, any data held by the class
 |      to associate multiple bits of data (think IP fragmentation or TCP)
 |      should be forgotten.
 |      The default implementation does nothing and should suffice for simple protocols.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Protocol:
 |
 |  completed
 |
 |  data
 |
 |  next
 |
 |  prev

class Stream(builtins.object)
 |  This class represents a 'stream'.
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

FUNCTIONS
    lookup_protocol(protoname)
        This finds a protocol class by it's name.
        All protocol classes should be referred to by name. This allows them to be
        overridden by simply registering a different class with the same name.

    register_linktype(protoname, linktype)
        This function adds a protocol to the linktype registry.
        The linktype registry determines what protocol class to use first.
        Protocols in this registry would correspond to the valus of linktypes
        defined in packet.common.LinkType

    register_protocol(protocol)
        This function adds a protocol to the protocol registry.
        The protocol registry maps a protocol's name to it's class.

    root_identify(packet)
        Identify a packet.
        This function may have side effects.
        This function will set the packet.identity field to a protocol instance.
        This function may return packets with an incomplete identity (is_complete() == False).
        Packets with an incomplete identity can and will have their identities
        updated whenever a protocol class deems suitable.

    uint16pack(i)
        Converts a 16-bit int into bytes (big endian)

    uint16unpack(b)
        Converts bytes into a 16-bit int (big endian)
```

```
    uint32pack(i)
        Converts a 32-bit int into bytes (big endian)

    uint32unpack(b)
        Converts bytes into a 32-bit int (big endian)

DATA
    ATTRIBUTE_WILDCARD = None
    linktype_registry = {1: 'eth', 228: 'ip4', 229: 'ip6'}
    protocol_registry = {'arp': <class 'packet.identity.arp.ARP'>, 'eth': ...
    uint16 = <Struct object>
    uint32 = <Struct object>
```

9.2.5 packet.identity.eth

```
Help on module packet.identity.eth in packet.identity:

NAME
    packet.identity.eth

DESCRIPTION
    Ethernet dissection module.
    Contains the Ethernet (eth) Protocol class, and a handful of constants.

CLASSES
    packet.identity.core.CarrierProtocol(packet.identity.core.Protocol)
        Ethernet

    class Ethernet(packet.identity.core.CarrierProtocol)
     |  Class representing the Ethernet II (IEEE 802.3/1Q/1AD) protocol.
     |
     |  Method resolution order:
     |      Ethernet
     |      packet.identity.core.CarrierProtocol
     |      packet.identity.core.Protocol
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  __init__(self, data, prev)
     |      Constructor.
     |
     |  get_attributes(self)
     |      Returns the fields in this packet as a attribute dict.
     |
     |  get_ethertype(self)
     |      Returns the ethertype as a number.
     |
     |  get_route(self)
     |      Returns the route of this ethernet header, as a 12-byte string.
     |
     |  get_route_reciprocal(self)
     |      Returns the reciprocal route of this ethernet header.
     |
     |  replace_hosts(self, hostmap)
     |      Replaces the source and destination mac addresses with the corresponding
     |      value in the MAC section of the hostmap argument (if present).
     |
     |  set_attributes(self, attrs)
     |      Updates the fields in this header to represent the contents of an attribute dict.
     |
     |  ----------------------------------------------------------------------
     |  Static methods defined here:
     |
     |  build_attributes(attrstr)
     |      Creates a set of attributes from an attribute string.
     |
     |  interpret_packet(data, parent)
     |      Creates a protocol instance and determines the next protocol to use.
     |      This makes use of a registry of ethertype -> protocol names, updated
     |      with the 'register_ethertype' function in this module.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
```

```
|  payload_offset
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  name = 'eth'
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.CarrierProtocol:
|
|  get_payload_length(self)
|      Abstract method returning the lengh of the payload,
|      if indicated in the protocol.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from packet.identity.core.CarrierProtocol:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.Protocol:
|
|  __iter__(self)
|      Generator method for accessing this/child protocol instances.
|
|  is_complete(self)
|      Returns True if this and all child protocols are complete.
|
|  match_attributes(self, tattrs)
|      Tests wether this ProtocolIdentity matches a set of attributes.
|      This is the default implementation and compares attributes of this
|      instance (from get_attributes) to the provided attributes (tattrs).
|      This is not commutative - all of the target keys MUST be in
|      this ProtocolIdentity's keys. Additionally, if the target's key is
|      equal to None, it is treated as a wildcard and matches.
|
|  recalculate_checksums(self)
|      This method should recalculate any kind of checksum used by this
|      protocol, after any 'child' checksums have been recomputed.
|      That is to say, the recalculation should propagate up, from the
|      highest-level protocol to the lowest. The default implementation does
|      nothing other then this propogation and should suffice for
|      protocols without validation.
|
|  ----------------------------------------------------------------------
|  Static methods inherited from packet.identity.core.Protocol:
|
|  interpret_stream(stream, parent)
|      Abstract static method interpret_stream.
|      Like interpret_packet, this takes two arguments, a 'parent' protocol
|      instance (which can be none) and a 'stream' object.
|      This method should create an instance of it's class, determine the next
|      (if any) protocol to interpret, (setting the next field), then
|      return said instance.
|
|  reset_state()
|      This static method should restore the initial state to any kind of
|      state tracker this class uses. This means, any data held by the class
|      to associate multiple bits of data (think IP fragmentation or TCP)
|      should be forgotten.
|      The default implementation does nothing and should suffice for simple protocols.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from packet.identity.core.Protocol:
|
|  completed
|
|  data
|
|  next
|
```

```
     |  prev

FUNCTIONS
    find_ethertype_offset(data)
        Finds the offset of the true ethertype of a frame.

    register_ethertype(protocol, ethertype)
        Associates a protocol name with an ethertype.

DATA
    ETHERNET_MIN_FRAME_SIZE = 16
    ETHERTYPE_ARP = 2054
    ETHERTYPE_IEEE802_1AD = 34984
    ETHERTYPE_IEEE802_1Q = 33024
    ETHERTYPE_IP4 = 2048
    ETHERTYPE_IP6 = 34525
    ETHERTYPE_WOL = 2114
    ethertype_registry = {2048: 'ip4', 2054: 'arp', 34525: 'ip6'}
```

9.2.6 packet.identity.icmp6

```
Help on module packet.identity.icmp6 in packet.identity:

NAME
    packet.identity.icmp6

CLASSES
    packet.identity.core.ProtocolStub(packet.identity.core.Protocol)
        ICMPv6

    class ICMPv6(packet.identity.core.ProtocolStub)
     |  ICMPv6 stub.
     |
     |  Method resolution order:
     |      ICMPv6
     |      packet.identity.core.ProtocolStub
     |      packet.identity.core.Protocol
     |      builtins.object
     |
     |  Data and other attributes defined here:
     |
     |  name = 'icmp6'
     |
     |  ----------------------------------------------------------------------
     |  Methods inherited from packet.identity.core.ProtocolStub:
     |
     |  get_attributes(self)
     |      Stub. Returns an empty dictionary.
     |
     |  match_attributes(self, attrs)
     |      Stub. Always returns true.
     |
     |  set_attributes(self, attrs)
     |      Stub. Does nothing.
     |
     |  ----------------------------------------------------------------------
     |  Class methods inherited from packet.identity.core.ProtocolStub:
     |
     |  interpret_packet(data, parent) from builtins.type
     |      Stub. Returns an instance of this class.
     |
     |  interpret_stream(stream, parent) from builtins.type
     |      Stub. Returns an instance of this class.
     |
     |  ----------------------------------------------------------------------
     |  Static methods inherited from packet.identity.core.ProtocolStub:
     |
     |  build_attributes(attrstr)
     |      Stub. Returns an empty dictionary.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors inherited from packet.identity.core.ProtocolStub:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
```

```
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from packet.identity.core.Protocol:
 |
 |  __init__(self, data, prev)
 |      Constructor for protocol instances.
 |
 |  __iter__(self)
 |      Generator method for accessing this/child protocol instances.
 |
 |  is_complete(self)
 |      Returns True if this and all child protocols are complete.
 |
 |  recalculate_checksums(self)
 |      This method should recalculate any kind of checksum used by this
 |      protocol, after any 'child' checksums have been recomputed.
 |      That is to say, the recalculation should propagate up, from the
 |      highest-level protocol to the lowest. The default implementation does
 |      nothing other then this propogation and should suffice for
 |      protocols without validation.
 |
 |  replace_hosts(self, hostmap)
 |      This method should replace instances of host identification,
 |      namely IP addresses and MAC addresses.
 |      This operation should propagate to child protocols.
 |      The default implementation does nothing other then this propogation
 |      and should suffice for protocols without any kind of host identification.
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from packet.identity.core.Protocol:
 |
 |  reset_state()
 |      This static method should restore the initial state to any kind of
 |      state tracker this class uses. This means, any data held by the class
 |      to associate multiple bits of data (think IP fragmentation or TCP)
 |      should be forgotten.
 |      The default implementation does nothing and should suffice for simple protocols.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from packet.identity.core.Protocol:
 |
 |  completed
 |
 |  data
 |
 |  next
 |
 |  prev
```

9.2.7 packet.identity.icmp

```
Help on module packet.identity.icmp in packet.identity:

NAME
    packet.identity.icmp

CLASSES
    packet.identity.core.ProtocolStub(packet.identity.core.Protocol)
        ICMP

    class ICMP(packet.identity.core.ProtocolStub)
     |  ICMP stub.
     |
     |  Method resolution order:
     |      ICMP
     |      packet.identity.core.ProtocolStub
     |      packet.identity.core.Protocol
     |      builtins.object
     |
     |  Data and other attributes defined here:
     |
     |  name = 'icmp'
```

```
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.ProtocolStub:
|
|  get_attributes(self)
|      Stub. Returns an empty dictionary.
|
|  match_attributes(self, attrs)
|      Stub. Always returns true.
|
|  set_attributes(self, attrs)
|      Stub. Does nothing.
|
|  ----------------------------------------------------------------------
|  Class methods inherited from packet.identity.core.ProtocolStub:
|
|  interpret_packet(data, parent) from builtins.type
|      Stub. Returns an instance of this class.
|
|  interpret_stream(stream, parent) from builtins.type
|      Stub. Returns an instance of this class.
|
|  ----------------------------------------------------------------------
|  Static methods inherited from packet.identity.core.ProtocolStub:
|
|  build_attributes(attrstr)
|      Stub. Returns an empty dictionary.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from packet.identity.core.ProtocolStub:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.Protocol:
|
|  __init__(self, data, prev)
|      Constructor for protocol instances.
|
|  __iter__(self)
|      Generator method for accessing this/child protocol instances.
|
|  is_complete(self)
|      Returns True if this and all child protocols are complete.
|
|  recalculate_checksums(self)
|      This method should recalculate any kind of checksum used by this
|      protocol, after any 'child' checksums have been recomputed.
|      That is to say, the recalculation should propagate up, from the
|      highest-level protocol to the lowest. The default implementation does
|      nothing other then this propogation and should suffice for
|      protocols without validation.
|
|  replace_hosts(self, hostmap)
|      This method should replace instances of host identification,
|      namely IP addresses and MAC addresses.
|      This operation should propagate to child protocols.
|      The default implementation does nothing other then this propogation
|      and should suffice for protocols without any kind of host identification.
|
|  ----------------------------------------------------------------------
|  Static methods inherited from packet.identity.core.Protocol:
|
|  reset_state()
|      This static method should restore the initial state to any kind of
|      state tracker this class uses. This means, any data held by the class
|      to associate multiple bits of data (think IP fragmentation or TCP)
|      should be forgotten.
|      The default implementation does nothing and should suffice for simple protocols.
|
|  ----------------------------------------------------------------------
```

```
 |  Data descriptors inherited from packet.identity.core.Protocol:
 |
 |  completed
 |
 |  data
 |
 |  next
 |
 |  prev
```

9.2.8 packet.identity.ip4

```
Help on module packet.identity.ip4 in packet.identity:

NAME
    packet.identity.ip4

CLASSES
    builtins.object
        FragmentTracker
    packet.identity.core.CarrierProtocol(packet.identity.core.Protocol)
        IPv4

    class FragmentTracker(builtins.object)
     |  Used to track fragmented IP packets.
     |
     |  Methods defined here:
     |
     |  __init__(self)
     |
     |  add_fragment(self, frag)
     |      Returns True if packet is complete, False if otherwise.
     |
     |  try_insert(self, frag)
     |      Try to add a fragment to the frags list.
     |      Tristate return:
     |       - None  = last fragment added.
     |       - True  = fragment added, more needed.
     |       - False = fragment deferred.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  deferred_frags
     |
     |  frags
     |
     |  next_offset

    class IPv4(packet.identity.core.CarrierProtocol)
     |  Method resolution order:
     |      IPv4
     |      packet.identity.core.CarrierProtocol
     |      packet.identity.core.Protocol
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  __init__(self, data, prev)
     |
     |  get_attributes(self)
     |      Retrieve a set of attributes describing fields in this protocol.
     |
     |  get_fraginfo(self)
     |      Returns the flags, fragment offset and fragment ident.
     |
     |  get_payload_length(self)
     |      Returns the (possibly logical) payload length.
     |
     |  get_protocol(self)
     |      Returns the protocl number of this IP header.
     |
     |  get_route(self)
     |      Returns the route defined by this IP header.
     |
```

```
|  get_route_reciprocal(self)
|      Returns the reciprocal route of this IP header.
|
|  recalculate_checksums(self)
|      Recalculate the checksum for this IP header.
|
|  replace_hosts(self, hostmap)
|      Replace source/destination addresses.
|
|  set_attributes(self, attrs)
|      Alter packet data to match a set of protocol attributes.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  build_attributes(attrstr)
|      Creates a set of attributes from an attribute string.
|
|  interpret_packet(data, parent)
|      Interpret packet data for this protocol.
|
|  reset_state()
|      Resets the fragment_trackers dict.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  logical_payload_length
|
|  payload_end_ver_ihl
|
|  payload_length
|
|  payload_offset
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  name = 'ip4'
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from packet.identity.core.CarrierProtocol:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.Protocol:
|
|  __iter__(self)
|      Generator method for accessing this/child protocol instances.
|
|  is_complete(self)
|      Returns True if this and all child protocols are complete.
|
|  match_attributes(self, tattrs)
|      Tests wether this ProtocolIdentity matches a set of attributes.
|      This is the default implementation and compares attributes of this
|      instance (from get_attributes) to the provided attributes (tattrs).
|      This is not commutative - all of the target keys MUST be in
|      this ProtocolIdentity's keys. Additionally, if the target's key is
|      equal to None, it is treated as a wildcard and matches.
|
|  ----------------------------------------------------------------------
|  Static methods inherited from packet.identity.core.Protocol:
|
|  interpret_stream(stream, parent)
|      Abstract static method interpret_stream.
|      Like interpret_packet, this takes two arguments, a 'parent' protocol
|      instance (which can be none) and a 'stream' object.
|      This method should create an instance of it's class, determine the next
|      (if any) protocol to interpret, (setting the next field), then
```

```
     |      return said instance.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors inherited from packet.identity.core.Protocol:
     |
     |  completed
     |
     |  data
     |
     |  next
     |
     |  prev

FUNCTIONS
    ip4_extract_fragment_info(fragdat)

DATA
    IP4_FLAG_DONT_FRAGMENT = 2
    IP4_FLAG_EVIL_BIT = 4
    IP4_FLAG_MORE_FRAGMENTS = 1
    fragment_trackers = {}
```

9.2.9 packet.identity.ip6

```
Help on module packet.identity.ip6 in packet.identity:

NAME
    packet.identity.ip6

CLASSES
    packet.identity.core.CarrierProtocol(packet.identity.core.Protocol)
        IPv6

    class IPv6(packet.identity.core.CarrierProtocol)
     |  Method resolution order:
     |      IPv6
     |      packet.identity.core.CarrierProtocol
     |      packet.identity.core.Protocol
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  __init__(self, data, prev)
     |
     |  get_attributes(self)
     |      Retrieve a set of attributes describing fields in this protocol.
     |
     |  get_payload_length(self)
     |      Returns the payload length.
     |
     |  get_protocol(self)
     |      Returns the protocl number (next header) of this IPv6 header.
     |
     |  get_route(self)
     |      Returns the route defined by this IPv6 header.
     |
     |  get_route_reciprocal(self)
     |      Returns the reciprocal route of this IPv6 header.
     |
     |  replace_hosts(self, hostmap)
     |      Replace source/destination addresses.
     |
     |  set_attributes(self, attrs)
     |      Alter packet data to match a set of protocol attributes.
     |
     |  ----------------------------------------------------------------------
     |  Static methods defined here:
     |
     |  build_attributes(attrstr)
     |      Creates a set of attributes from an attribute string.
     |
     |  interpret_packet(data, parent)
     |      Interpret packet data for this protocol.
     |
     |  ----------------------------------------------------------------------
```

```
|  Data descriptors defined here:
|
|  payload_length
|
|  payload_offset
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  name = 'ip6'
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from packet.identity.core.CarrierProtocol:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.Protocol:
|
|  __iter__(self)
|      Generator method for accessing this/child protocol instances.
|
|  is_complete(self)
|      Returns True if this and all child protocols are complete.
|
|  match_attributes(self, tattrs)
|      Tests wether this ProtocolIdentity matches a set of attributes.
|      This is the default implementation and compares attributes of this
|      instance (from get_attributes) to the provided attributes (tattrs).
|      This is not commutative - all of the target keys MUST be in
|      this ProtocolIdentity's keys. Additionally, if the target's key is
|      equal to None, it is treated as a wildcard and matches.
|
|  recalculate_checksums(self)
|      This method should recalculate any kind of checksum used by this
|      protocol, after any 'child' checksums have been recomputed.
|      That is to say, the recalculation should propagate up, from the
|      highest-level protocol to the lowest. The default implementation does
|      nothing other then this propogation and should suffice for
|      protocols without validation.
|
|  ----------------------------------------------------------------------
|  Static methods inherited from packet.identity.core.Protocol:
|
|  interpret_stream(stream, parent)
|      Abstract static method interpret_stream.
|      Like interpret_packet, this takes two arguments, a 'parent' protocol
|      instance (which can be none) and a 'stream' object.
|      This method should create an instance of it's class, determine the next
|      (if any) protocol to interpret, (setting the next field), then
|      return said instance.
|
|  reset_state()
|      This static method should restore the initial state to any kind of
|      state tracker this class uses. This means, any data held by the class
|      to associate multiple bits of data (think IP fragmentation or TCP)
|      should be forgotten.
|      The default implementation does nothing and should suffice for simple protocols.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from packet.identity.core.Protocol:
|
|  completed
|
|  data
|
|  next
|
|  prev
```

9.2.10 packet.identity.ip

```
Help on module packet.identity.ip in packet.identity:

NAME
    packet.identity.ip

DESCRIPTION
    ip:
    Internet Protocol Number registry and checksum algorithm.

FUNCTIONS
    checksum(data)
        General implementation of the IP checksum algorithm.

    lookup_ip_protocol(protonum)
        Looks up a protocol name from number.

    register_ip_protocol(protoname, protonum)
        Add a protocol number <-> name mapping.

DATA
    PROTO_ETHERIP = 97
    PROTO_ICMP = 1
    PROTO_IPV4 = 4
    PROTO_IPV6 = 41
    PROTO_IPV6_ICMP = 58
    PROTO_TCP = 6
    PROTO_UDP = 17
    ip_protocol_registry = {1: 'icmp', 6: 'tcp', 17: 'udp', 58: 'icmp6'}
```

9.2.11 packet.identity.tcp

```
Help on module packet.identity.tcp in packet.identity:

NAME
    packet.identity.tcp

CLASSES
    builtins.object
        TCPStateMachine
    packet.identity.core.CarrierProtocol(packet.identity.core.Protocol)
        TCP

    class TCP(packet.identity.core.CarrierProtocol)
     |  Method resolution order:
     |      TCP
     |      packet.identity.core.CarrierProtocol
     |      packet.identity.core.Protocol
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  __init__(self, data, prev)
     |
     |  get_attributes(self)
     |      Retrieve a set of attributes describing fields in this protocol.
     |
     |  recalculate_checksums(self)
     |      Recalculate the checksum for this TCP header.
     |
     |  set_attributes(self, attrs)
     |      Alter packet data to match a set of protocol attributes.
     |
     |  ----------------------------------------------------------------------
     |  Static methods defined here:
     |
     |  build_attributes(attrstr)
     |      Creates a set of attributes from an attribute string.
     |
     |  interpret_packet(data, parent)
     |      Interpret packet data for this protocol.
     |
     |  reset_state()
     |      Resets the tcp_routes dict.
```

```
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  payload_offset
|
|  port
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  name = 'tcp'
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.CarrierProtocol:
|
|  get_payload_length(self)
|      Abstract method returning the lengh of the payload,
|      if indicated in the protocol.
|
|  get_route(self)
|      Abstract method returning a hashable object that represents
|      the 'route' (the destination, source and direction) a carrier
|      protocol would direct a packet.
|
|  get_route_reciprocal(self)
|      Abstract method returning a hashable object not unlike the above,
|      that returns the opposite route.
|      E.g. if get_route returned A -> B, this should return B -> A.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from packet.identity.core.CarrierProtocol:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.Protocol:
|
|  __iter__(self)
|      Generator method for accessing this/child protocol instances.
|
|  is_complete(self)
|      Returns True if this and all child protocols are complete.
|
|  match_attributes(self, tattrs)
|      Tests wether this ProtocolIdentity matches a set of attributes.
|      This is the default implementation and compares attributes of this
|      instance (from get_attributes) to the provided attributes (tattrs).
|      This is not commutative - all of the target keys MUST be in
|      this ProtocolIdentity's keys. Additionally, if the target's key is
|      equal to None, it is treated as a wildcard and matches.
|
|  replace_hosts(self, hostmap)
|      This method should replace instances of host identification,
|      namely IP addresses and MAC addresses.
|      This operation should propagate to child protocols.
|      The default implementation does nothing other then this propogation
|      and should suffice for protocols without any kind of host identification.
|
|  ----------------------------------------------------------------------
|  Static methods inherited from packet.identity.core.Protocol:
|
|  interpret_stream(stream, parent)
|      Abstract static method interpret_stream.
|      Like interpret_packet, this takes two arguments, a 'parent' protocol
|      instance (which can be none) and a 'stream' object.
|      This method should create an instance of it's class, determine the next
|      (if any) protocol to interpret, (setting the next field), then
|      return said instance.
|
|  ----------------------------------------------------------------------
```

```
 |  Data descriptors inherited from packet.identity.core.Protocol:
 |
 |  completed
 |
 |  data
 |
 |  next
 |
 |  prev

class TCPStateMachine(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, ir, rr)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  initiator_route
 |
 |  responder_route

DATA
    tcp_routes = {}
```

9.2.12 packet.identity.udp

```
Help on module packet.identity.udp in packet.identity:

NAME
    packet.identity.udp

CLASSES
    packet.identity.core.CarrierProtocol(packet.identity.core.Protocol)
        UDP

    class UDP(packet.identity.core.CarrierProtocol)
     |  Method resolution order:
     |      UDP
     |      packet.identity.core.CarrierProtocol
     |      packet.identity.core.Protocol
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  __init__(self, data, prev)
     |
     |  get_attributes(self)
     |      Retrieve a set of attributes describing fields in this protocol.
     |
     |  recalculate_checksums(self)
     |      Recalculate the checksum for this UDP header.
     |
     |  set_attributes(self, attrs)
     |      Alter packet data to match a set of protocol attributes.
     |
     |  ----------------------------------------------------------------------
     |  Static methods defined here:
     |
     |  build_attributes(attrstr)
     |      Creates a set of attributes from an attribute string.
     |
     |  interpret_packet(data, parent)
     |      Interpret packet data for this protocol.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  payload_length
     |
     |  payload_offset
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
```

```
|  name = 'udp'
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.CarrierProtocol:
|
|  get_payload_length(self)
|      Abstract method returning the lengh of the payload,
|      if indicated in the protocol.
|
|  get_route(self)
|      Abstract method returning a hashable object that represents
|      the 'route' (the destination, source and direction) a carrier
|      protocol would direct a packet.
|
|  get_route_reciprocal(self)
|      Abstract method returning a hashable object not unlike the above,
|      that returns the opposite route.
|      E.g. if get_route returned A -> B, this should return B -> A.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from packet.identity.core.CarrierProtocol:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from packet.identity.core.Protocol:
|
|  __iter__(self)
|      Generator method for accessing this/child protocol instances.
|
|  is_complete(self)
|      Returns True if this and all child protocols are complete.
|
|  match_attributes(self, tattrs)
|      Tests wether this ProtocolIdentity matches a set of attributes.
|      This is the default implementation and compares attributes of this
|      instance (from get_attributes) to the provided attributes (tattrs).
|      This is not commutative - all of the target keys MUST be in
|      this ProtocolIdentity's keys. Additionally, if the target's key is
|      equal to None, it is treated as a wildcard and matches.
|
|  replace_hosts(self, hostmap)
|      This method should replace instances of host identification,
|      namely IP addresses and MAC addresses.
|      This operation should propagate to child protocols.
|      The default implementation does nothing other then this propogation
|      and should suffice for protocols without any kind of host identification.
|
|  ----------------------------------------------------------------------
|  Static methods inherited from packet.identity.core.Protocol:
|
|  interpret_stream(stream, parent)
|      Abstract static method interpret_stream.
|      Like interpret_packet, this takes two arguments, a 'parent' protocol
|      instance (which can be none) and a 'stream' object.
|      This method should create an instance of it's class, determine the next
|      (if any) protocol to interpret, (setting the next field), then
|      return said instance.
|
|  reset_state()
|      This static method should restore the initial state to any kind of
|      state tracker this class uses. This means, any data held by the class
|      to associate multiple bits of data (think IP fragmentation or TCP)
|      should be forgotten.
|      The default implementation does nothing and should suffice for simple protocols.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from packet.identity.core.Protocol:
|
|  completed
|
```

```
     |  data
     |
     |  next
     |
     |  prev

DATA
    UDP_MIN_SIZE = 8
```

9.2.13 packet.pipeline.merge

```
Help on module packet.pipeline.merge in packet.pipeline:

NAME
    packet.pipeline.merge - merge: contains the definition for the merge function.

FUNCTIONS
    merge(preaders, relative=True, offset=None)
        Generator function that takes a list of packet sources and two parameters.
        If relative is true, then the time of the -first- packet from a source is
        subtracted from all packets from that source.
        If offset is not none, this value is added to the time of all packets,
        otherwise the average start time is used.
        Returns a generator that produces the next chronological packet on each iteration.
```

9.2.14 packet.pipeline.filter

```
Help on module packet.pipeline.filter in packet.pipeline:

NAME
    packet.pipeline.filter - filter: contains the filter function.

FUNCTIONS
    filter(source, keep=None, discard=None, policy=True)
        Selectively keeps or discards packets based on their identity.
        This pipeline assumes that packets have been identified prior to use.
        The keep and discard sets are collections of 'identities'
        Identities are lists of 'prototypes'.
        A prototype is a combination of a protocol name and a set of attributes.
        The name must match that of the corresponding protocol instance, and the
        match_attributes method of the corresponding protocol instance must
        return true when presented with the prototype attributes.

    identity_match(protocol_instances, prototypes)
        Compares a packet's identity list (protocol instances) to a list of
        'prototypes'.
        Every prototype must match it's corresponding protocol instance.
        Prototypes are a (name, attrs) tuple.

DATA
    DISCARD = False
    KEEP = True
```

9.2.15 packet.pipeline.identify

```
Help on module packet.pipeline.identify in packet.pipeline:

NAME
    packet.pipeline.identify - identify: Contains the identify function.

FUNCTIONS
    identify(source)
        Returns each packet in the source, with the packet identity set.
        This generator will defer packets until their identity is complete.
        As a result, this generator can consume a lot of memory, up to the total
        number of packets produced by source.

        If the source is exhausted before a packet's identity is complete,
        the packets will then be returned regardless of status.
```

9.2.16 packet.common

```
Help on module packet.common in packet:
```

```
NAME
    packet.common

DESCRIPTION
    common:
     - Definition for the 'Packet' class
     - Functions for operating on the 'Packet' class.
     - Functions for handling generic user input, i.e. parse_int.
     - Functions for converting a ip4/ip6/mac address to string form and back.
    Contains two constants, PACKET_MINAGE and PACKET_MAXAGE, two
    psuedopackets being older then or newer then all other packets, respectively.

CLASSES
    builtins.object
        Packet
    enum.Enum(builtins.object)
        LinkType

    class LinkType(enum.Enum)
     |  Linktype Enumerations matching those used in tcpdump and pcap files.
     |
     |  Method resolution order:
     |      LinkType
     |      enum.Enum
     |      builtins.object
     |
     |  Data and other attributes defined here:
     |
     |  APPLE_IP_OVER_IEEE1394 = <LinkType.APPLE_IP_OVER_IEEE1394: 138>
     |
     |  ARCNET_BSD = <LinkType.ARCNET_BSD: 7>
     |
     |  ARCNET_LINUX = <LinkType.ARCNET_LINUX: 129>
     |
     |  ATM_RFC1483 = <LinkType.ATM_RFC1483: 100>
     |
     |  AX25 = <LinkType.AX25: 3>
     |
     |  AX25_KISS = <LinkType.AX25_KISS: 202>
     |
     |  BACNET_MS_TP = <LinkType.BACNET_MS_TP: 165>
     |
     |  BLUETOOTH_BREDR_BB = <LinkType.BLUETOOTH_BREDR_BB: 255>
     |
     |  BLUETOOTH_HCI_H4 = <LinkType.BLUETOOTH_HCI_H4: 187>
     |
     |  BLUETOOTH_HCI_H4_WITH_PHDR = <LinkType.BLUETOOTH_HCI_H4_WITH_PHDR: 201...
     |
     |  BLUETOOTH_LE_LL = <LinkType.BLUETOOTH_LE_LL: 251>
     |
     |  BLUETOOTH_LE_LL_WITH_PHDR = <LinkType.BLUETOOTH_LE_LL_WITH_PHDR: 256>
     |
     |  BLUETOOTH_LINUX_MONITOR = <LinkType.BLUETOOTH_LINUX_MONITOR: 254>
     |
     |  CAN_SOCKETCAN = <LinkType.CAN_SOCKETCAN: 227>
     |
     |  C_HDLC = <LinkType.C_HDLC: 104>
     |
     |  C_HDLC_WITH_DIR = <LinkType.C_HDLC_WITH_DIR: 205>
     |
     |  DBUS = <LinkType.DBUS: 231>
     |
     |  DOCSIS = <LinkType.DOCSIS: 143>
     |
     |  DVB_CI = <LinkType.DVB_CI: 235>
     |
     |  EPON = <LinkType.EPON: 259>
     |
     |  ERF = <LinkType.ERF: 197>
     |
     |  ETHERNET = <LinkType.ETHERNET: 1>
     |
     |  FC_2 = <LinkType.FC_2: 224>
     |
     |  FC_2_WITH_FRAME_DELIMS = <LinkType.FC_2_WITH_FRAME_DELIMS: 225>
```

```
|
|  FDDI = <LinkType.FDDI: 10>
|
|  FRELAY = <LinkType.FRELAY: 107>
|
|  FRELAY_WITH_DIR = <LinkType.FRELAY_WITH_DIR: 206>
|
|  GPRS_LLC = <LinkType.GPRS_LLC: 169>
|
|  IEEE802_11 = <LinkType.IEEE802_11: 105>
|
|  IEEE802_11_AVS = <LinkType.IEEE802_11_AVS: 163>
|
|  IEEE802_11_PRISM = <LinkType.IEEE802_11_PRISM: 119>
|
|  IEEE802_11_RADIOTAP = <LinkType.IEEE802_11_RADIOTAP: 127>
|
|  IEEE802_15_4 = <LinkType.IEEE802_15_4: 195>
|
|  IEEE802_15_4_NOFCS = <LinkType.IEEE802_15_4_NOFCS: 230>
|
|  IEEE802_15_4_NONASK_PHY = <LinkType.IEEE802_15_4_NONASK_PHY: 215>
|
|  IEEE802_5 = <LinkType.IEEE802_5: 6>
|
|  INFINIBAND = <LinkType.INFINIBAND: 247>
|
|  IPMB_LINUX = <LinkType.IPMB_LINUX: 209>
|
|  IPMI_HPM_2 = <LinkType.IPMI_HPM_2: 260>
|
|  IPNET = <LinkType.IPNET: 226>
|
|  IPOIB = <LinkType.IPOIB: 242>
|
|  IPV4 = <LinkType.IPV4: 228>
|
|  IPV6 = <LinkType.IPV6: 229>
|
|  IP_OVER_FC = <LinkType.IP_OVER_FC: 122>
|
|  LAPD = <LinkType.LAPD: 203>
|
|  LINUX_IRDA = <LinkType.LINUX_IRDA: 144>
|
|  LINUX_LAPD = <LinkType.LINUX_LAPD: 177>
|
|  LINUX_SLL = <LinkType.LINUX_SLL: 113>
|
|  LOOP = <LinkType.LOOP: 108>
|
|  LTALK = <LinkType.LTALK: 114>
|
|  MPEG_2_TS = <LinkType.MPEG_2_TS: 243>
|
|  MTP2 = <LinkType.MTP2: 140>
|
|  MTP2_WITH_PHDR = <LinkType.MTP2_WITH_PHDR: 139>
|
|  MTP3 = <LinkType.MTP3: 141>
|
|  MUX27010 = <LinkType.MUX27010: 236>
|
|  NETANALYZER = <LinkType.NETANALYZER: 240>
|
|  NETANALYZER_TRANSPARENT = <LinkType.NETANALYZER_TRANSPARENT: 241>
|
|  NETLINK = <LinkType.NETLINK: 253>
|
|  NFC_LLCP = <LinkType.NFC_LLCP: 245>
|
|  NFLOG = <LinkType.NFLOG: 239>
|
|  NG40 = <LinkType.NG40: 244>
|
```

```
|  NULL = <LinkType.NULL: 0>
|
|  PFLOG = <LinkType.PFLOG: 117>
|
|  PKTAP = <LinkType.PKTAP: 258>
|
|  PPI = <LinkType.PPI: 192>
|
|  PPP = <LinkType.PPP: 9>
|
|  PPP_ETHER = <LinkType.PPP_ETHER: 51>
|
|  PPP_HDLC = <LinkType.PPP_HDLC: 50>
|
|  PPP_PPPD = <LinkType.PPP_PPPD: 166>
|
|  PPP_WITH_DIR = <LinkType.PPP_WITH_DIR: 204>
|
|  PROFIBUS_DL = <LinkType.PROFIBUS_DL: 257>
|
|  RAW = <LinkType.RAW: 101>
|
|  RTAC_SERIAL = <LinkType.RTAC_SERIAL: 250>
|
|  SCCP = <LinkType.SCCP: 142>
|
|  SCTP = <LinkType.SCTP: 248>
|
|  SITA = <LinkType.SITA: 196>
|
|  SLIP = <LinkType.SLIP: 8>
|
|  STANAG_5066_D_PDU = <LinkType.STANAG_5066_D_PDU: 237>
|
|  SUNATM = <LinkType.SUNATM: 123>
|
|  USBPCAP = <LinkType.USBPCAP: 249>
|
|  USB_LINUX = <LinkType.USB_LINUX: 189>
|
|  USB_LINUX_MMAPPED = <LinkType.USB_LINUX_MMAPPED: 220>
|
|  USER0 = <LinkType.USER0: 147>
|
|  USER1 = <LinkType.USER1: 148>
|
|  USER10 = <LinkType.USER10: 157>
|
|  USER11 = <LinkType.USER11: 158>
|
|  USER12 = <LinkType.USER12: 159>
|
|  USER13 = <LinkType.USER13: 160>
|
|  USER14 = <LinkType.USER14: 161>
|
|  USER15 = <LinkType.USER15: 162>
|
|  USER2 = <LinkType.USER2: 149>
|
|  USER3 = <LinkType.USER3: 150>
|
|  USER4 = <LinkType.USER4: 151>
|
|  USER5 = <LinkType.USER5: 152>
|
|  USER6 = <LinkType.USER6: 153>
|
|  USER7 = <LinkType.USER7: 154>
|
|  USER8 = <LinkType.USER8: 155>
|
|  USER9 = <LinkType.USER9: 156>
|
|  WATTSTOPPER_DLM = <LinkType.WATTSTOPPER_DLM: 263>
```

```
 |
 |  ZWAVE_R1_R2 = <LinkType.ZWAVE_R1_R2: 261>
 |
 |  ZWAVE_R3 = <LinkType.ZWAVE_R3: 262>
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from enum.Enum:
 |
 |  name
 |      The name of the Enum member.
 |
 |  value
 |      The value of the Enum member.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from enum.EnumMeta:
 |
 |  __members__
 |      Returns a mapping of member name->value.
 |
 |      This mapping lists all enum members, including aliases. Note that this
 |      is a read-only view of the internal mapping.

class Packet(builtins.object)
 |  Class consisting of five fields,
 |   - unixtime: Floating point number, time in seconds since 1st Jan, 1970.
 |   - linktype: Integer constant representing the root format of the
 |               packet as specified by the source.
 |   - origlen: Original length of the 'data' field.
 |   - data: Packet data as a bytearray.
 |   - identity: The root protocol instance.
 |
 |  Comparison operator methods and the length method are implemented.
 |  Comparisons work on the value of unixtime, so a < b means a is older then b.
 |  The length is 'origlen', so len(a) < len(b) means a was shorter then b.
 |
 |  Methods defined here:
 |
 |  __eq__(self, other)
 |
 |  __ge__(self, other)
 |
 |  __gt__(self, other)
 |
 |  __init__(self, ut, lt, ol, dat)
 |
 |  __le__(self, other)
 |
 |  __len__(self)
 |      # Implement length func based on 'origlen', such that:
 |      # len(a) < len(b) means a has an original length shorter then b.
 |
 |  __lt__(self, other)
 |      # Implement comprison operations based on 'unixtime', such that:
 |      # a < b means a is older then b.
 |
 |  __ne__(self, other)
 |
 |  __str__(self)
 |      Creates a human readable string summary of this packet.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  data
 |
 |  identity
 |
 |  linktype
 |
 |  origlen
 |
 |  unixtime
 |
 |  ----------------------------------------------------------------------
```

```
 |  Data and other attributes defined here:
 |
 |  __hash__ = None

FUNCTIONS
    ip4_bin2str(ip4b)
        Converts a IPv4 address in big-endian byte form to a
        string form not unlike the one accepted by the function above.
        Returns None on a format error.

    ip4_str2bin(ip4s)
        Converts a IPv4 address in dotted octet string form to a big-endian
        byte representation.
        Returns None on a format error.

    ip6_bin2str(ip6b)
        Converts a IPv6 address in big-endian byte form to a
        string form not unlike the one accepted by the function above.
        Returns None on a format error.

    ip6_str2bin(ip6s)
        Converts a IPv6 address in RFC-5952 format to a big-endian
        byte representation.
        Returns None on a format error.

    mac_bin2str(macb)
        Converts a MAC address in big-endian byte form to a
        string form not unlike the one accepted by the function above.
        Returns None on a format error.

    mac_str2bin(macs)
        Converts a MAC address in string representation (six hex numbers,
        delimited by colons) to a big-endian byte representation.
        Returns None on a format error.

    parse_hexbytes(hexstr)
        Simple hex string -> bytes parsing function.
        Uses binascii to do actual conversion, returns None on error.

    parse_int(intstr)
        Simple string -> integer parsing/guessing function.
        Uses 'int' to do actual conversion, returns None on error.

DATA
    PACKET_MAXAGE = <packet.common.Packet object>
    PACKET_MINAGE = <packet.common.Packet object>
```

9.2.17 packet.memorymap

```
Help on module packet.memorymap in packet:

NAME
    packet.memorymap - memorymap: Providing a logically contiguous mapping of several memoryviews

CLASSES
    builtins.tuple(builtins.object)
        segment
    collections.abc.MutableSequence(collections.abc.Sequence)
        memorymap

    class memorymap(collections.abc.MutableSequence)
     |  Maps a series of memoryview objects (or 'segments') into a single
     |  logical address space.
     |
     |  Method resolution order:
     |      memorymap
     |      collections.abc.MutableSequence
     |      collections.abc.Sequence
     |      collections.abc.Sized
     |      collections.abc.Iterable
     |      collections.abc.Container
     |      builtins.object
     |
     |  Methods defined here:
     |
```

```
|  __delitem__(self, idx)
|      Deleting items is not allowed.
|
|  __getitem__(self, idx)
|      Returns the item(s) at the specified index,
|      or returns a memorymap in the given range.
|
|  __init__(self, obj, slc=slice(None, None, None))
|
|  __len__(self)
|      Return the length of the logical memory map.
|
|  __setitem__(self, idx, val)
|      Sets the item(s) ad the specified index or slice.
|
|  add_segment(self, memview)
|      Takes a memorymap, memoryview or object supporting the
|      buffer interface, and adds it to a logical combination of buffers.
|
|  insert(self, idx, val)
|      Inserting items is not allowed.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  segments
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __abstractmethods__ = frozenset([])
|
|  ----------------------------------------------------------------------
|  Methods inherited from collections.abc.MutableSequence:
|
|  __iadd__(self, values)
|
|  append(self, value)
|      S.append(value) -- append value to the end of the sequence
|
|  clear(self)
|      S.clear() -> None -- remove all items from S
|
|  extend(self, values)
|      S.extend(iterable) -- extend sequence by appending elements from the iterable
|
|  pop(self, index=-1)
|      S.pop([index]) -> item -- remove and return item at index (default last).
|      Raise IndexError if list is empty or index is out of range.
|
|  remove(self, value)
|      S.remove(value) -- remove first occurrence of value.
|      Raise ValueError if the value is not present.
|
|  reverse(self)
|      S.reverse() -- reverse *IN PLACE*
|
|  ----------------------------------------------------------------------
|  Methods inherited from collections.abc.Sequence:
|
|  __contains__(self, value)
|
|  __iter__(self)
|
|  __reversed__(self)
|
|  count(self, value)
|      S.count(value) -> integer -- return number of occurrences of value
|
|  index(self, value)
|      S.index(value) -> integer -- return first index of value.
|      Raises ValueError if the value is not present.
|
|  ----------------------------------------------------------------------
|  Class methods inherited from collections.abc.Sized:
```

```
 |
 |  __subclasshook__(C) from abc.ABCMeta

class segment(builtins.tuple)
 |  segment(start, end, mem)
 |
 |  Method resolution order:
 |      segment
 |      builtins.tuple
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __getnewargs__(self)
 |      Return self as a plain tuple.  Used by copy and pickle.
 |
 |  __getstate__(self)
 |      Exclude the OrderedDict from pickling
 |
 |  __repr__(self)
 |      Return a nicely formatted representation string
 |
 |  _asdict(self)
 |      Return a new OrderedDict which maps field names to their values.
 |
 |  _replace(_self, **kwds)
 |      Return a new segment object replacing specified fields with new values
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  _make(iterable, new=<built-in method __new__ of type object at 0x7fb20369f000>, len=<built-in function len>) from builtins
 |      Make a new segment object from a sequence or iterable
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(_cls, start, end, mem)
 |      Create new instance of segment(start, end, mem)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      A new OrderedDict mapping field names to their values
 |
 |  end
 |      Alias for field number 1
 |
 |  mem
 |      Alias for field number 2
 |
 |  start
 |      Alias for field number 0
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  _fields = ('start', 'end', 'mem')
 |
 |  _source = "from builtins import property as _property, tupl..._itemget...
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
```

```
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.n
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return self*value.
 |
 |  __sizeof__(...)
 |      T.__sizeof__() -- size of T in memory, in bytes
 |
 |  count(...)
 |      T.count(value) -> integer -- return number of occurrences of value
 |
 |  index(...)
 |      T.index(value, [start, [stop]]) -> integer -- return first index of value.
 |      Raises ValueError if the value is not present.

FUNCTIONS
    clamp_end(idx, length)
        Clamps an index from None/negative to length (inclusive).

    clamp_start(idx, length)
        Clamps an index from 0 to length (exclusive).
```