

Vrije Universiteit Amsterdam
MPhil Parallel and Distributed Computer Systems
Compiler Construction Practical

Piffle

A Packet Filter Language

Jaap Weel

Version 0.1.1

Contents

1	Notices	3
1.1	Latest version of this document	3
1.2	Copyright notice	3
2	Introduction	4
3	Bounded packet filters	5
3.1	Dynamic bounds on packet filters	5
3.2	Static bounds on arbitrary packet filters	5
3.3	Proof carrying code	6
3.4	Bounds on restricted packet filters	6
4	Language definition	8
4.1	Vocabulary and representation	8
4.2	Declarations	9
4.3	Types	9
4.4	Expressions	9
4.5	Operators	10
4.6	The types of expressions	11
4.7	Constant expressions	13
4.8	Source files	13
4.9	The filter function	13
4.10	Preprocessor	13
5	The Piffle compiler, pfc	14
5.1	The pfc(1) manual page	14
6	The pcap boilerplate, pcap.c	17
6.1	The pfc.pcap(1) manual page	17
6.2	Writing programs for the pcap.c boilerplate	18
7	The test boilerplate, test.c	20
7.1	The pfc.test(1) manual page	20
8	An example of a packet filter	21
8.1	A packet filter in PFL	21
8.2	The packet filter translated to C by pfc	23

1 Notices

1.1 Latest version of this document

For up-to-date information on Piffle, and the latest version of this document, be sure to visit the following web site:

<http://code.google.com/p/piffle/wiki/PiffleWiki>

1.2 Copyright notice

Copyright (c) 2007, Jaap Weel. This work is licensed under the Creative Commons Attribution-Share-Alike 3.0 License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/3.0/>

or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

2 Introduction

Piffle is a pronounceable form of PFL, that is, Packet Filter Language. (“Piffle” is also an obscure word for “chatter”.) Piffle is a very simple programming language for writing network packet filters, with one special feature: the compiler can enforce limits on the processor time and memory used for processing each packet of input. This is accomplished by passing arguments to the `-C` and `-M` command line options. Memory bounds are in words, more or less, and CPU bounds are in arbitrary units.

In this document, I will first consider in general the problem of checking arbitrary programs for transgression of such resource bounds, and notice that it is very hard. I will conclude that in Piffle we must punt and restrict the class of programs that can be used as packet filters, which makes the problem comparatively easy.

After that, I will give the definition of the Piffle language, and some instructions for how to use the Piffle compiler.

3 Bounded packet filters

An important requirement in designing network packet processing code is that the amount of time and memory spent processing each packet be bounded. We do not want the entire network connection grinding to a halt because of an unforeseen infinite loop in a packet filter. There are several ways to prevent such havoc. In this chapter, I will discuss each of them, and explain which one I chose and why.

3.1 Dynamic bounds on packet filters

We can check bounds at runtime. For instance, we can run the packet filter on each packet for a specified quota of clock ticks and drop the packet if processing has not finished by the time the quota runs out.

One problem with this runtime approach is that a bug in the packet filters can remain hidden for a long time until some unusual packet is received that triggers the bug.

Another problem is that it is not clear exactly what to do when the bound is exceeded: Should we drop the packet? Should we pass it along unaltered? We would like to be able to give the packet filter author the opportunity to decide what to do with packets that take too long to analyze. But if we allow that sort of flexibility within the dynamic checking framework, we will inevitably have to allow a chunk of cleanup code to deal with “slow” packets, which in turn can only run for a restricted amount of time. If *that* time is exceeded, do we need a third program to deal with slow packet filters that have slow cleanup code?

3.2 Static bounds on arbitrary packet filters

Given the problems with checking bounds at run time, we might want to check at compile time. To do that, we need a language in which we can express exactly those packet filter programs that execute within given time and space bounds.

Determining whether a given program runs within these bounds sounds like it ought to be undecidable, but it is not. After all, we are not trying to evaluate whether a given packet filter will terminate *at all, in the end*; we merely want it to terminate within a given number of steps. (The space bound problem is analogous.)

The problem of determining whether a given Turing machine will terminate within a certain number of steps is, in fact, decidable. In a limited number N of steps the machine can read only a limited amount N of its tape, which means we could check whether the time bound by exhaustively running the machine for at most N steps on all tapes of length at most N .

It should be obvious that this procedure, while quite satisfactory as a constructive proof of decidability, is impractical, because the process of bounds checking itself would take lots of time (in fact, exponential in N). What we need is a bound on the process of bounds checking.

3.3 Proof carrying code

One way to design a language that would let us check bounds at compile time would be to annotate the program with a proof of the statement that it will terminate within a given number of steps.

Proofs could be provided either as annotations or as part of a type system.¹ The use of proof carrying code would allow all packet filters to be written that can be proven to execute within bounds, and it would allow for the bounds checking problem itself to be tractable. Depending on the details of the formal system used, proof (or type) checking can be polynomial (and often linear) in the length of the proof. Therefore, bound checking of annotated code complies with the rule of thumb that compilers should use only algorithms that are at most polynomial in the length of the program.

The practical liabilities of proof carrying code are well known. Programmers are unfamiliar with proof systems, and those who know their way around them could still have trouble coming up with the (possibly lengthy) proofs in individual cases. Therefore, we will not pursue proof carrying code any further now, and turn instead to a simpler solution.

3.4 Bounds on restricted packet filters

If we do not want to get bogged down in the depths of formal type theory, but still design a language that would let us check bounds at compile time, we arrive at what is, in fact, the traditional solution to the problem of bounded packet filters. We will construct a programming language out of familiar abstractions, and that language will allow us to express a *large but arbitrary* subset of the set of all packet filters that execute in bounded time.

Such a language is not Turing complete, but if it is designed intelligently, it can cover a large fraction of the problems that people may want to solve using packet filters.

An example of a packet filter language based on familiar programming abstractions is the traditional BSD packet filter *bpf*.² Its packet filters are written in a simple machine code with a little twist: the virtual machine allows jumps forward, but not backward. That way, the time spent executing any given packet filter program is at most proportional to the length of the program.

For Piffle, we take a similar approach. Piffle is not based on machine language, but on block structured, procedural programming languages such as C and Pascal. Just as the

¹Crary and Weirich. "Resource Bound Certification." 27th POPL, p.184.

²McCanne and Van Jacobson. "The BSD packet filter." USENIX 1993.

bpf language omits backward jumps from machine code, Piffle omits unbounded looping constructs and recursive procedure calls from block structured, procedural code. Piffle does provide some bounded looping constructs, and procedure calls are allowed as long as the call graph contains no cycles.

4 Language definition

The Piffle language is defined as a set of sentences, called source files, well formed according to the Piffle syntax. To describe the syntax, I will use a modified version of the Backus Normal Form (also known as Backus-Naur Formalism or BNF), using $x^?$ to indicate that x may appear either once or not at all, x^+ to indicate that x may appear at least once, and x^* to indicate that x may appear any number of times, including not at all.

I will not provide a formal definition of the semantics of Piffle. In general, when I do not specify the meaning or type of some Piffle construct, it can be assumed that it is analogous to the semantics of its closest analog in C.

4.1 Vocabulary and representation

A well formed Piffle program is composed of tokens, which are identifiers, literals, and reserved tokens. Each token is a sequence of characters, and may not contain whitespace unless this is explicitly specified. Any amount of whitespace may be inserted between tokens, and indeed must be inserted between any two tokens that would make a valid token when put together.

Whitespace is any sequence of blanks, tabs, vertical tabs, form feeds, line feeds, and carriage returns (in C notation, any character in the string "`\v\f\t\r\n`"). Comments may also appear wherever whitespace can appear. (Unlike in C, they may *not* appear inside of a token.) In-line comments may be inserted between any two tokens. They are delimited by `/*` and `*/` and may be nested. End-of-line comments may also be inserted between any two tokens, and they run from `//` to the next line end, or the end of the file, whichever comes first.

Identifiers are sequences of letters, digits, and underscores. The first character may not be a digit.

$$\begin{aligned} \textit{ident} &= (\textit{letter} \mid \boxed{_}) (\textit{letter} \mid \boxed{_} \mid \textit{digit})^* \\ \textit{letter} &= \boxed{\text{A}} \mid \cdots \mid \boxed{\text{Z}} \mid \boxed{\text{a}} \mid \cdots \mid \boxed{\text{z}} \\ \textit{digit} &= \boxed{0} \mid \cdots \mid \boxed{9} \end{aligned}$$

The literals available in Piffle, are numbers, the `unit` literal, which is the canonical value of type `void`, `true`, which means the same as `1 : bool`, and `false`, which means the same as `0 : bool`. The only numbers available are integers. They can be represented as decimal, octal, or hexadecimal, the same way as in Haskell 98. Note that this differs

from C syntax in that leading 0s are simply ignored, and the prefix for octal is 0o or 0O rather than simply 0.

$$\begin{aligned}
 \textit{literal} &= \boxed{\text{unit}} \mid \boxed{\text{true}} \mid \boxed{\text{false}} \mid \textit{integer} \\
 \textit{integer} &= \textit{digit}^+ \mid (\boxed{0\text{o}} \mid \boxed{0\text{O}}) \textit{octit}^+ \mid (\boxed{0\text{x}} \mid \boxed{0\text{X}}) \textit{hexit}^+ \\
 \textit{hexit} &= \boxed{0} \mid \dots \mid \boxed{9} \mid \boxed{\text{A}} \mid \dots \mid \boxed{\text{F}} \mid \boxed{\text{a}} \mid \dots \mid \boxed{\text{f}} \\
 \textit{octit} &= \boxed{0} \mid \dots \mid \boxed{7}
 \end{aligned}$$

4.2 Declarations

Piffle allows no recursion, and all functions must be defined before they are used. Therefore, there is no need for forward declarations. If the return type of a function is omitted, `void` is implied. The meaning of duplicate definitions is undefined.

$$\begin{aligned}
 \textit{declaration} &= \textit{vardec} \mid \textit{fundec} \\
 \textit{vardec} &= \boxed{\text{var}} \textit{ident} \boxed{:} \textit{type} \\
 \textit{fundec} &= \boxed{\text{fun}} \textit{ident} \boxed{(} \textit{argl}^? \boxed{)} \boxed{(} \boxed{:} \textit{type} \boxed{)}^? \boxed{=} \textit{expression} \\
 \textit{argl} &= \textit{ident} \boxed{:} \textit{type} \boxed{(} \boxed{,} \textit{ident} \boxed{:} \textit{type} \boxed{)}^*
 \end{aligned}$$

4.3 Types

A type in Piffle is `void`, an atomic type, or an array. Atomic types come in signed and unsigned, and have 8, 16, or 32 bits. Each array type encompasses only arrays of a specific length; you can declare an “array of 1024 integers”, but not an “array of integers”. This is an important feature, because it is what enables the calculation of resource limits. It is at present *not* actually used for anything else; in particular, there are no bounds checks.

$$\begin{aligned}
 \textit{type} &= \boxed{\text{void}} \mid \textit{atomic} \mid \textit{atomic} \boxed{[} \textit{integer} \boxed{]} \\
 \textit{atomic} &= \boxed{\text{bool}} \mid \boxed{\text{u8}} \mid \boxed{\text{u16}} \mid \boxed{\text{u32}} \mid \boxed{\text{s8}} \mid \boxed{\text{s16}} \mid \boxed{\text{s32}}
 \end{aligned}$$

Whenever a value of type of `void` is expected, a value of any other type will do as well.

4.4 Expressions

Unlike in C or Pascal, in Piffle there is no syntactic distinction between expressions and statements; every expression can be used as a statement and vice versa.

```

expression = expression2 (binop expression2)*
expression2 = unop* expression3 ( [ expression ] )? ( [ : ] atomic )?
expression3 = literal
| ident ( [ ( expression ( [ , ] expression )*)? ] )?
| [ ( expression ) ]
| [ { ( declaration ; )* ( expression ; )* } ]
| if expression then expression ( else expression )?
| for ident in expression ( while expression )? do expression
| for ident from integer to integer
  ( while expression )? do expression

```

Cast expressions (like `x : u32`) allow the use of a value of one atomic type with another atomic type. The exact meaning of such conversions is undefined but expected to correspond to the behavior of the C compiler on the platform in use.

Blocks have the value of the last expression in them (which is why there is no `return` statement); if a block contains no expressions, `unit` is implied.

If-then-else statements have values, and correspond more closely to the C ternary operator (`?:`) than to the C if-else statement. If an `else` clause is omitted, `unit` is implied.

Loops in Piffle are somewhat peculiar, in order to accommodate resource bound checking. There are two types of loops: one will iterate with a variable sequentially bound to each number in a range (which includes both bounds), and the other will iterate with a variable sequentially bound to each element in an array. Both allow the use of a `while` clause, which will be evaluated before each iteration; if it evaluates to 0, the loop is aborted immediately. The iterator variable is bound within the while clause, so it is perfectly valid to say something like `for i from 0 to 10 while is_useful(i) do ...`

4.5 Operators

The operators are given by

```

binop = [ * ] | [ / ] | [ % ] | [ + ] | [ - ] | [ << ] | [ >> ] | [ < ]
| [ > ] | [ <= ] | [ >= ] | [ == ] | [ != ] | [ & ] | [ ^ ]
| [ | ] | [ && ] | [ || ] | [ = ] | [ += ] | [ -= ] | [ *= ]
| [ %= ] | [ /= ] | [ |= ] | [ &= ] | [ ^= ] | [ >>= ] | [ <<= ]
unop = [ + ] | [ - ] | [ ~ ] | [ ! ]

```

The grammar as specified so far correctly and completely specifies the language, but it is misleading and ambiguous, because it does not indicate the meaning of certain compound expressions. In fact, the actual Piffle parser will respect the standard C precedence rules, as summarized in the following table.

Operators (ordered from tight to loose)	Associativity
* / %	left-to-right
+ -	left-to-right
<< >>	left-to-right
< > <= >=	left-to-right
== !=	left-to-right
&	left-to-right
^	left-to-right
	left-to-right
&&	left-to-right
	left-to-right
= += -= *= %= /= = &= ^= >>= <<=	right-to-left

The keywords have the loosest possible preference, and thus the expressions at the end of `if` and `for` expressions extend to the right as far as possible.

Even though any expression may be used, according to the grammar, as the left-hand side of an assignment, assignment is only a meaningful operation when the left-hand side is an “lvalue” of the form `ident ([[expression]]) ?`.

4.6 The types of expressions

The binary operators `*`, `/`, `%`, `+`, `-`, `&`, `|`, `^`, `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `|=`, `^=`, all take two operands of the same, atomic, type, and return a value of that type, that is, formally,

$$*, /, %, +, -, \&, |, \wedge, =, *=, /=, \% =, +=, -=, \& =, |=, \wedge = : \forall \tau \in \{\text{bool}, \text{u8}, \text{u16}, \text{u32}, \text{s8}, \text{s16}, \text{s32}\}. \tau \times \tau \rightarrow \tau.$$

The unary operators `+`, `-`, and `~` all take one operand of any integer type and return a value of that type, that is, formally,

$$+, -, \sim : \forall \tau \in \{\text{bool}, \text{u8}, \text{u16}, \text{u32}, \text{s8}, \text{s16}, \text{s32}\}. \tau \rightarrow \tau.$$

The binary operators `<`, `>`, `<=`, `>=`, `==`, `!=`, `&&`, and `||` all take two operands of arbitrary atomic types, and return a value of type `bool`, that is, formally

$$<, >, <=, >=, ==, !=, \&\&, || : \forall \tau, v \in \{\text{bool}, \text{u8}, \text{u16}, \text{u32}, \text{s8}, \text{s16}, \text{s32}\}. \tau \times v \rightarrow \text{bool}.$$

The unary operator `!` takes one operand of any integer types and returns a value of type `bool`, that is, formally,

$$! : \forall \tau \in \{\text{bool}, \text{u8}, \text{u16}, \text{u32}, \text{s8}, \text{s16}, \text{s32}\}. \tau \rightarrow \text{bool}.$$

For the binary operators `<<`, `>>`, `<<=`, and `>>=`, the value being shifted and the value indicating the number of bits to shift need not be of the same type, but both types must be atomic, and the entire expression gets the type of the value being shifted. Formally,

$$\ll, \gg, \ll=, \gg= : \forall \sigma, \tau \in \{\text{bool}, \text{u8}, \text{u16}, \text{u32}, \text{s8}, \text{s16}, \text{s32}\}. \tau \times \sigma \rightarrow \tau.$$

A sequence `{...}` of expressions has the value and type of the last expression in the sequence, or `void` if there are no expressions. All expressions preceding the last are expected to have type `void`, but as stated above, any other type will do when `void` is expected.

An `if` expression consists of a condition, a consequent, and an optional alternative. If the alternative is omitted, `unit` is implied. The condition may be of any atomic type. The types of the consequent and the alternative must match, but note again that because any type can be used instead of `void`, if the type of either branch is `void`, the other branch can have any type. The alternative is evaluated only if the condition evaluates to 0; otherwise, the consequent is evaluated. In either case, the entire expression gets the value of the evaluated clause. To clarify:

```
fun frob () : void = {
  var x, y : u32;
  if 3 < 5 then do_something(); /* do_something() is executed */
  x = if 3 < 5 then 0 else 1;    /* x is set to 0 */
  x = if 3 > 5 then 0 else 1;    /* x is set to 1 */
  y = if 3 < 5 then 0;          /* This won't type check!! */
};
```

There are two iteration constructs in Piffle. Both are different from iteration constructs in comparable languages because of the need for computing resource bounds.

An expression of the form

$$\boxed{\text{for}} \alpha \boxed{\text{in}} \beta (\boxed{\text{while}} \gamma)^? \boxed{\text{do}} \delta \boxed{\text{od}}$$

has type `void`. The range β must be of type `array i τ` , for some integer i and some atomic type τ , and the variable α will be of type τ and scope over γ and δ . The condition γ may have any atomic type, and the body δ must be of type `void` (with any other type allowed).

An expression of the form

$$\boxed{\text{for}} \alpha \boxed{\text{from}} \beta_0 \boxed{\text{to}} \beta_1 (\boxed{\text{while}} \gamma)^? \boxed{\text{do}} \delta \boxed{\text{od}}$$

has type `void`. The bounds $\beta_{0,1}$ must be constant expressions of any atomic type, and α will be of type `u32` and scope over γ and δ . The condition γ may have any atomic type, and the body δ must be of type `void` (with any other type allowed).

4.7 Constant expressions

Integer literals are assigned a special type (`sliteral`) by the compiler. This type is considered atomic, and values of type `sliteral` can always be used when any atomic type is expected.

4.8 Source files

A source file consists of a number of declarations separated by semicolons.

$$file = (declaration \boxed{;})^*$$

4.9 The filter function

The function named `filter` has a special meaning. It is used as the root of the call graph (which is actually a tree, in the Piffle case) by the compiler when checking for memory and CPU usage bounds. It is also the function that gets called on every incoming packet.

Strictly speaking, this is dependent on which boilerplate is being used. Boilerplates are a concept we will get to in the chapter about the compiler, but rest assured that all currently available boilerplates will cause `filter` to be called on each incoming packet.

4.10 Preprocessor

It should be possible to run a Piffle program through a regular C preprocessor (`cpp`). This is not done by default; you would have to do it manually. Also make sure not to use nested comments if you want to use `cpp`. The compiler does not interpret `cpp` line directives, so any error messages you get about preprocessed code will refer to positions in the preprocessed source.

5 The Piffle compiler, pfc

The Piffle compiler, `pfc`, is a fairly simple compiler, written in the Haskell programming language, that outputs C code. It is documented in the `pfc(1)` man page, which is reproduced here in its entirety.¹

5.1 The `pfc(1)` manual page

PFC(1)

PFC(1)

NAME

`pfc` - Piffle Compiler

SYNOPSIS

`pfc` [OPTIONS] file.pfl

INTRODUCTION

Piffle is a pronounceable form of PFL, that is, Packet Filter Language. Piffle is a very simple programming language for writing network packet filters, with one special feature: the compiler can enforce limits on the processor time and memory used for processing each packet of input. This is accomplished by passing arguments to the `-C` and `-M` command line options. Memory bounds are in words, more or less, and CPU bounds are in arbitrary units.

DESCRIPTION

`pfc` is a compiler that compiles programs written in Piffle. It produces a C file, which must be further processed with the GNU C compiler. Typically, you would use `pfc` like this:

```
pfc -Bpcap.c foo.pfl
```

That would compile `foo.pfl` into a new file called `foo.c`, and it would embed the `pcap.c` boilerplate, which is the one you should use when you want to use the `pcap` library. Next, you would compile the resulting C program with

```
cc -o foo foo.pfl -lpcap
```

¹The man pages included in this manual have been converted to plain text using `nroff` and `colcrt`, which strips out most of the formatting. If you know a better way of inserting a man page into a \LaTeX document, please tell me.

OPTIONS

`-h` or `--help`

Prints a short usage message.

`-a` or `--ansicolors`

If you specify this option, error and debug messages will be in color, using standard ANSI escape sequences. This is very useful for debugging, and looks pretty.

`-d` [`<N>`] or `--debug`[`=<N>`]

Sets the debug level. When `<N>` is not specified, default is 0. When `-d` is not specified at all, default is 99. A debug level setting of `N` causes all debug messages to be printed to standard error that are marked with an importance greater than `<N>`.

`-C` `<N>` or `--cpu`=`<N>`

This will cause the compiler to abort with an error if the main function (`filter()`) would run for longer than `<N>` arbitrary CPU usage units on at least some inputs.

`-M` `<N>` or `--mem`=`<N>`

This will cause the compiler to abort with an error if the main function (`filter()`) would consume more than approximately `<N>` words of memory on at least some inputs.

`-B` `<file>` or `--boilerplate`=`<file>`

This will cause the contents of the specified `<file>` to be inserted into the output. This argument is not mandatory, but the output of the compiler is not very useful without one of the boilerplate files inserted into it. The compiler will produce `cpp` line directives so that `gcc` will reference errors in the boilerplate to the appropriate lines in the boilerplate file.

FILES

The boilerplate files used by `pfc` must be in the boilerplate search path, which consists of the contents of the `PIFFLEBOILER` environment variable appended to the standard path,

```
./usr/share/piffle:/usr/local/share/piffle
```

BUGS

When a command line option is specified more than once, all occurrences but the last are silently ignored. I plan to write a grand unified program configuration library for Haskell one day, which is going to read command lines, environment variables, and various sorts of configuration files. Obviously, even though the grand unified library will prob-

ably never materialize, I am reluctant now to overly complicate the command line parsing in pfc.

The C code produced by pfc is not standard ANSI C. It uses GNU extensions and can only be compiled by gcc.

AUTHOR

Jaap Weel <weel@ugcs.caltech.edu>

pfc-0.1

PFC(1)

6 The pcap boilerplate, pcap.c

As I explained in the pfc man page, you usually want to tell pfc to embed a certain “boilerplate” into its output. The “boilerplate” is a file that typically contains a `main()` function that provides some means of gathering network packets, repeatedly calling `filter()` on them, and spitting them back out.

There is a second man page, `pfc.pcap(1)`, that explains how to use a Piffle program that has been compiled using the `pcap.c` boilerplate. Enjoy.

6.1 The pfc.pcap(1) manual page

PFC.PCAP(1)

PFC.PCAP(1)

NAME

`pfc.pcap` - Any Piffle program using the `pcap.c` boilerplate

SYNOPSIS

`pfc.pcap` [OPTIONS]

DESCRIPTION

This manual page applies to any Piffle program that has been compiled with the `-Bpcap.c` option.

OPTIONS

If no input device or file is specified, the program will try to read from the standard network device. If no output file is specified, and `-v` is not specified, the program will most likely produce no output at all, which is probably not what you want.

`--help` or `-h`

Show a help text.

`--device=<dev>` or `-d <dev>`

Use `<dev>` as monitoring device. The device notation is the same as for `tcpdump`, so you can use `tcpdump -D` to get a list of valid devices.

`--input=<file>` or `-f <file>`

Read the input from capture file `<file>`. (Use `-` for `stdin`.)

`--output=<file> or -o <file>`

Write the output packets to file `<file>`. The resulting file is a tcpdump-compatible capture file. (Use `-` for stdout.)

`--filter=<exp> or -F <exp>`

Tell the the pcap library to compile the expression `<exp>` into a Berkeley Packet Filter and apply it to incoming packets before feeding them to the Piffle program. For instance, you can use `-Ftcp` if you want to select only TCP packets. The syntax for `<exp>` is described in the `tcpdump(8)` man page.

`-v or --verbose`

Be more verbose during monitoring.

RETURN VALUES

The program will print error messages to `stderr`, and return the following error codes:

- 0 Normal exit, no problems.
- 1 Problem parsing command line.
- 2 Problem opening file or device.
- 3 Problem with a pcap library call.

SEE ALSO

`pfc(1)`, `pcap(3)`, `tcpdump(8)`

BUGS

Please report to the author.

AUTHOR

Jaap Weel <weel@ugcs.caltech.edu>

pfc-0.1

PFC.PCAP(1)

6.2 Writing programs for the pcap.c boilerplate

The `pcap.c` boilerplate defines a function prototype `filter()` in C as follows:

```
uint32_t filter(uint8_t inp[], uint32_t in_sz, uint8_t out[], uint32_t out_sz);
```

This function is supposed to be declared in Piffle as follows:

```
fun filter( inp : u8[...], inp_sz : u32,
           out : u8[...], out_sz : u32 ) : u32 = ...;
```

The `inp` and `out` arrays are the input and output packets, and the `inp_sz` and `out_sz` integers contain the size of those arrays. The function is supposed to write things into the `out` array, and return the actual length of the outgoing packet.

For instance, a `filter` function that simply copies the incoming packet would look like this:

```
fun filter( inp : u8[65536], inp_sz : u32,
           out : u8[65536], out_sz : u32 ) : u32 = {
  var i : u32;
  var out_sz_new : u32;

  out_sz_new = if inp_sz < out_sz then inp_sz else out_sz;

  for i from 0 to 65536 while i < out_sz_new do {
    out[i] = inp[i];
  };

  out_sz_new;
};
```

7 The test boilerplate, test.c

There is another boilerplate called `text.c`, which exists mainly for the benefit of the Piffle test suite, and is probably not useful for anything else. It is nonetheless documented on its own manpage, `pfc.test(1)`.

7.1 The `pfc.test(1)` manual page

PFC.TEST(1)

PFC.TEST(1)

NAME

`pfc.test` - Any Piffle program using the `test.c` boilerplate

SYNOPSIS

`pfc.test`

DESCRIPTION

This manual page applies to any Piffle program that has been compiled with the `-Btest.c` option. A Piffle program compiled that way takes no command line arguments. It reads newline-separated lines from `stdin` until EOF, and sends each line to the `filter()` function as a null-terminated character array. Empty lines are ignored. The `filter()` function is expected to return a null-terminated character array. Output is printed to `stdout`, separated by newlines. Error checking is minimal, and error messages are cryptic. The `test.c` boilerplate exists mainly just to support the Piffle test suite, and probably should not be used for anything else.

SEE ALSO

`pfc(1)`

BUGS

Please report to the author.

AUTHOR

Jaap Weel <weel@ugcs.caltech.edu>

`pfc-0.1`

PFC.TEST(1)

8 An example of a packet filter

8.1 A packet filter in PFL

```
/*
 * This is an actual packet filter that should work with the pcap
 * boilerplate. It only lets through TCP packets. In particular, it
 * drops everything that is not IP (such as ARP), and everything that
 * is IP but not TCP (such as UDP). For instance,
 *
 * pcap_tcp -v -Fudp
 *
 * should give no output at all, while
 *
 * pcap_tcp -v -Ftcp
 *
 * should give output. (If there's any TCP traffic going on.)
 *
 * Note that, on my machine, which has a run-off-the-mill ethernet
 * adapter, the link level header that sits in front of the IP packet
 * is 16 bytes long, and has an EtherType in bytes 14 and 15 (counting
 * from 0). I have no idea if this code would work for 802.11*
 * wireless networks, FDDI, Myrinet &c.
 */

/*
 * Get the EtherType protocol number, or -1 if the packet is too small.
 */
fun ethertype( inp : u8[65536], inp_sz : u32 ) : s32 = {
    if inp_sz < 16 then -1 else ((inp[14] : u16 << 8) | (inp[15] : u16)) : s32;
};

/*
 * Is the packet an IP packet?
 */
fun ip_p ( inp : u8[65536], inp_sz : u32 ) : bool = {
    var ethertype_s : s32;

    ethertype_s = ethertype(inp, inp_sz);
    if ethertype_s < 0 then false else ethertype_s == 0x0800;
};

/*
 * Get the IP protocol number, or -1 if the packet is too small.
 */
fun ip_protocol( inp : u8[65536], inp_sz : u32 ) : s32 = {
```

```

    if inp_sz < 26 then -1 else inp[25] : s32;
};

/*
 * Is the packet a TCP packet?
 */
fun tcp_p ( inp : u8[65536], inp_sz : u32 ) : bool = {
    var ip_protocol_s : s32;

    ip_protocol_s = ip_protocol(inp, inp_sz);
    if ip_protocol_s < 0 then false else ip_protocol_s == 0x06;
};

/*
 * Is the packet a UDP packet?
 */
fun udp_p ( inp : u8[65536], inp_sz : u32 ) : bool = {
    var ip_protocol_s : s32;

    ip_protocol_s = ip_protocol(inp, inp_sz);
    if ip_protocol_s < 0 then false else ip_protocol_s == 0x11;
};

/*
 * This is the main filter function.
 */
fun filter( inp : u8[65536], inp_sz : u32,
            out : u8[65536], out_sz : u32 ) : u32 = {
    var out_sz_new : u32;

    /* If the input array does not fit in the output array, we'll
     * have to truncate. */
    out_sz_new = if inp_sz < out_sz then inp_sz else out_sz;

    /* Copy the input array to the output array. */
    for i from 0 to 65536 while i < out_sz_new do {
        out[i] = inp[i];
    };

    /* If the packet is not IP or not TCP, set the output size to
     * 0. */
    if !ip_p(inp, inp_sz) then out_sz_new = 0;
    if !tcp_p(inp, inp_sz) then out_sz_new = 0;

    out_sz_new;
};

```

8.2 The packet filter translated to C by pfc

```
/*
 * THIS FILE WAS GENERATED BY PFC, THE PIFFLE COMPILER
 * Command line arguments were: pcap_tcp.pfl
 */
#line 4 "pcap_tcp.c"

int32_t ethertype(uint8_t inp[65536], uint32_t inp_sz)
{
    if (inp_sz < 16) {
        return -1;
    } else {
        return (int32_t) (((uint16_t) (inp[14]) << 8) | (uint16_t) (inp[15]));
    }
}

int ip_p(uint8_t inp[65536], uint32_t inp_sz)
{
    int32_t ethertype_s;

    ethertype_s = ethertype(inp, inp_sz);
    if (ethertype_s < 0) {
        return 0;
    } else {
        return ethertype_s == 2048;
    }
}

int32_t ip_protocol(uint8_t inp[65536], uint32_t inp_sz)
{
    if (inp_sz < 26) {
        return -1;
    } else {
        return (int32_t) (inp[25]);
    }
}

int tcp_p(uint8_t inp[65536], uint32_t inp_sz)
{
    int32_t ip_protocol_s;

    ip_protocol_s = ip_protocol(inp, inp_sz);
    if (ip_protocol_s < 0) {
        return 0;
    } else {
        return ip_protocol_s == 6;
    }
}

int udp_p(uint8_t inp[65536], uint32_t inp_sz)
{
```

```

int32_t ip_protocol_s;

ip_protocol_s = ip_protocol(inp, inp_sz);
if (ip_protocol_s < 0) {
    return 0;
} else {
    return ip_protocol_s == 17;
}
}

uint32_t filter(uint8_t inp[65536], uint32_t inp_sz, uint8_t out[65536],
               uint32_t out_sz)
{
    uint32_t out_sz_new;

    out_sz_new = ((inp_sz < out_sz) ? inp_sz : out_sz);
    {
        uint32_t i;

        for (i = 0; (i <= 65536) && (i < out_sz_new); i++) {
            out[i] = inp[i];
        }
    }
    if (!ip_p(inp, inp_sz)) {
        out_sz_new = 0;
    }
    if (!tcp_p(inp, inp_sz)) {
        out_sz_new = 0;
    }
    return out_sz_new;
}

```