Network Working Group M. Nossik, F. Welfeld, M. Richardson
INTERNET-DRAFT Solidum Systems Corporation
draft-nossik-pax-pdl-00.txt October 16, 1998
Expires in six months

PAX PDL - a non-procedural packet description language

Status of This memo

Abstract

This document describes PAX Pattern Description Language (PDL). PAX is a special purpose language for definitions of filters (recognizers) for sequential inputs. The language is suitable for describing pattern matching criteria in policy-based networking devices such as QoS routers and switches, packet filters, RMON probes, traffic shapers, etc. It provides consistent means of programming policy-based networking devices based on different hardware and software platforms. Programs written in PAX can be built incrementally, where elementary patterns can be used as building blocks for more complex ones. The language encourages modular and object-oriented design.

Table of Contents

1. Scope and purpose

This document provides a reference guide for using the PAX language. Numerous examples help illustrate the capabilities and expressive power of the language.

PAX PDL is a non-procedural Fourth Generation Language (4GL). This means that it describes the patterns that need to be recognized, but does not describe how to recognize them. The details of how it is being done are hidden in the compiler. The PAX PDL allows concise, flexible description

of headers and data contained in packets.

Although general enough for any kind of pattern recognition, such as database searches, PAX is intended for use primarily in data communications networks, such as data communication devices.

This document does not describe the compiler for the language, implementation details or the API used to classify the data packets.

Readers should be familiar with Backus Naur Form (BNF) notation, used to define the language grammar. PAX is defined in terms of a left-recursive context free grammar. HEADER-2

1.1. Specification of Requirements

The keywords "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", and "MAY" that appear in this document are to be interpreted as described in Bradner97.

2. Language structure

In terms of form and organization, PAX resembles the data structure definition part of the C programming language. Given its widespread acceptance, the authors have tried to borrow as much as possible from C. This results in the superior expressive power of PAX, compared to competing specifications (e.g. BPF, PATHFINDER and APF).

2.1. Characters, separators and tokens

The character set used in source programs consists of the whitespace plus the following 89 graphical characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9

_ { } [ ] ( ) # < > : ; . ? * + - / & | ~ ! = , \ " '
```

Whitespace characters (space, horizontal TAB, vertical TAB, form- feed and new-line) are used as separators between tokens, identifiers and language constructs.

2.2. Comments

PAX supports both C and C++ type comments.

C-style comments are any string of characters starting with the /*, up to the nearest */. C-style comments are treated as a whitespace.

C++ comments are any string of characters following //, to the nearest

new-line character. C++-style comments are treated as a whitespace.

2.3. Preprocessing

PAX provides support for some useful constructs through the use of a preprocessor. As discussed below, major useful preprocessor features include:

o source file inclusion,

o conditional compilation, and

o defines and macros.

2.3.1. Preprocessor directives

Lines beginning with # (perhaps preceded by whitespace) are interpreted by the preprocessor as preprocessing directives. Line boundaries are significant. The preprocessing directive extends to the next newline character not immediately preceded by a backslash ( ). The syntax of these lines is independent of the rest of the language. HEADER-3

2.3.2. Source file inclusion

The PAX preprocessor supports the #include directive. The directive specifies that the compiler must switch processing from the current file to the file specified on this line. This allows modular, hierarchical program design. The PAX preprocessor supports both conventional quoted ("file") and angle bracketed (<file>) notation, for example:

```
#include <system_file.pax>
#include "project_file.pax"
```

2.3.3. Conditional compilation

The PAX preprocessor supports the following conditional compilation directives:

```
#ifdef
#if
#else
#endif
```

Conditional compilation may be used in "standard header" files to prevent multiple inclusion.

2.3.4. Defines and macros

The PAX preprocessor supports the #define directive which specifies a macro/alias for a specific string. This feature is useful for improving code layout and readability, by providing an elegant way to configure PAX files using literals in PAX header files.

ANSI-C preprocessor capabilities SHOULD NOT be used beyond those

described in this document.

2.4. Keywords and names

The following keywords are used in the language constructs:

AND
ANYOF
BIT
EXPORT
IMPORT
NOT
OR
PATTERN
UINT
WHERE

Note: These values are reserved and cannot be used as identifiers of any kind. They are not case sensitive, but are by convention shown in this document as being in upper case.

Identifiers are used within the language to convey pattern names, parameter names and field names.

2.5. Program structure

A PAX program is a non-empty sequence of statements. There are three types of statements available in PAX:

PATTERN
IMPORT
EXPORT

2.5.1. PATTERN statement

The PATTERN statement is the main building block of a PAX program. It describes both the prototype and the layout of the pattern.

The name of the pattern defined by the PATTERN statement has a global scope and can be used to reference this pattern at any point in the program past this PATTERN statement. Names of the parameters used in the prototype have a scope of the pattern body.

2.5.2. IMPORT statement

The IMPORT statement specifies a prototype for a pattern residing in an external library. IMPORT statement describes the pattern prototype (name and parameters) allowing it to be referenced.

Note:Pattern conditionals cannot be used on imported patterns. Use #include statements for conditionals.

The name of the pattern defined by the IMPORT statement has a global scope and can be used to reference this pattern at any point in the program past this IMPORT statement. Names of the parameters used in the

prototype are meaningless, as actual parameters are passed in pattern references by position.

2.5.3. EXPORT statement

The EXPORT statement specifies a pattern(s) to be written to the output file (pattern library). Any number of EXPORT statements can appear anywhere in the program. Pattern can be named in the EXPORT statement even before it is defined.

3. Patterns

Pattern is a basic PAX concept.

Unlike C, where you can see the variable as a basic low level construct, and the function as the basic high level construct, pattern is both the low level and high level construct.

3.1. Elementary patterns

The following describes the basic elementary patterns:

```
<pattern>:=<builtin>
<builtin>:=<var_builtin_type> <length>
<var_builtin_type>:= BIT | UINT
<length>:=<number>
```

The simplest pattern looks like:

```
BIT 16
```

This matches any 16 bits in the input. BIT is a builtin field type.

Here we can as well use:

```
UINT 16
```

UINT, another builtin field type, is used for unsigned numeric fields.

```
<pattern> := <builtin_type> <length> <atomic_conditional_expr>
```

Matching anything is not very useful. Here is more valuable pattern:

```
UINT 4 >= 5
```

This pattern matches a 4 bit field when its numerical value is greater than 4 (e.g. ihl field in the IP header).

>= is a relational operator and 5 is a literal value. Both the relational operator and the literal value are appropriate for the UINT

field type.

3.2. Field concatenation

The most common way of building more complex patterns is by concatenating simpler ones, such as those below:

```
<concatenation> := <concatenation_head> } |
                   <concatenation_head> ; }

<concatenation_head>  := { <field> |
              <concatenation_head> ; <field> |
              <concatenation_head> ; <cond_field> |
              <concatenation_head> ; <anyof_field>

<cond_field>     := <field_name> <pattern> WHEN <pattern_cond_expr>
<field>          := <field_name> <pattern>
```

A sample concatenation follows:

```
          {
            version UINT 4 == 4;
            ihl UINT 4 == 5;            /* length == 5 : no options */
            typeOfService UINT 8;
            totalLength UINT 16;
            identification UINT 16;
            flagReserved BIT 1 == 0;
            flagDontFragment BIT 1;
            flagMoreFragments BIT 1 == 0;    /* last fragments only */
            fragmentOffset UINT 13 == 0;    /* first fragments only */
            timeToLive UINT 8;
            protocol BIT 8 = 6;                              /* TCP */
            headerChecksum BIT 16;
            sourceAddress BIT 32;
            destinationAddress BIT 32;/* this semicolon is optional */
          }
```

This pattern matches the IP version 4 headers of TCP/IP non- fragmented packets without IP options.

When patterns are put in sequence on the <field_list> list, the resulting pattern matches the input only when the first list element matches the beginning of the input; the next element matches some subsequent bits and so on. Elements of the above list are not simple patterns, but fields - patterns with the names attached, according to the following rule of the language grammar:

```
          <field> := <field_name> <pattern>
```

Note: Naming fields allows you to reference them later. See Pattern Conditional Expressions on page.

3.3. Field combination

Compare the use of the opening and closing square brackets, ( ) in the

following to the brackets used in Field Concatenation, in 3.2.

```
<combination>  := <combination_head> ] |
                  <combination_head> ; ]

<combination_head>  := [ <field> |
                       <combination_head> ; <field>
```

Often data has alternative layouts, as discussed following the sample IEEE 802.2 LLC header code below:

```
{
  dsap BIT 8 <> 0xFF;
  ssap BIT 8 <> 0xFF;
  control [
    longControl {
      control1 BIT 2 <> 0b11;
      control2 BIT 14
    };
    shortControl {
      control1 BIT 2 == 0b11;
      control2 BIT 6
    }
  ]
}
```

The control field of this pattern can be either 8 or 16 bits long. This is represented by the combination of two patterns : longControl and shortControl.

Input is accepted if it matches either of these patterns. The operands of the combination must be mutually exclusive; that is, no input can match more than one of them. In the example code, this is guaranteed by the distinct conditions on longControl.control1 and shortControl.control1.

The C aficionados may notice that syntactical similarity between concatenation and combination in the PAX is somewhat analogous to the similarity between struct and union. Do not carry this too far: semantics of the concatenation resembles struct much more than combination resembles union.

3.4. Naming patterns

As shown in the preceding example, any pattern may be used to build more complex patterns. This is a powerful concept: PAX patterns can be created in hierarchical fashion. Although possible, it is infeasible to retype component patterns verbatim each time it is used. Similar to typedefs in C, PAX allows you to assign a name to a pattern and subsequently refer to that pattern by its name:

```
<pattern_name>:= <identifier>
<pattern_prototype>:= PATTERN <pattern_name>
```

```
<pattern_def>:= <pattern_prototype> <pattern>
```

Any identifier can be used as pattern name, as long as all names within compilation unit are unique. The simplest prototype consists just of the keyword and the name. For more complex patterns, see 3.6, Parametric patterns. The pattern definition is just the prototype followed by the pattern itself, as shown below:

```
/* IP */
#define   ETHERTYPE_IP   0x0800

/* Internet Control Message Protocol */
#define   IPPROTO_ICMP    1

PATTERN  Ethernet_Hdr_For_IP {
  destination BIT 48;
  source BIT 48;
  type UINT 16 == ETHERTYPE_IP
}

PATTERN  IP_v4_Hdr_For_ICMP {
  version UINT 4 == 4;
  ihl UINT 4 == 5;            /* length == 5 : no options */
  typeOfService UINT 8;
  totalLength UINT 16;
  identification UINT 16;
  flagReserved BIT 1 == 0;
  flagDontFragment BIT 1;
  flagMoreFragments BIT 1 == 0;    /* last fragments only */
  fragmentOffset UINT 13 == 0;    /* first fragments only */
  timeToLive UINT 8;
  protocol BIT 8 == IPPROTO_ICMP;                /* ICMP */
  headerChecksum BIT 16;
  sourceAddress BIT 32;
  destinationAddress BIT 32
 }
```

3.5. Pattern references

Pattern reference (i.e. the name of the pattern) can be used instead of the pattern itself:

```
<pattern_ref>:= <pattern_name>
<pattern>:= <pattern_ref>

PATTERN  ICMP_Over_IP_Over_Ethernet_Hdr {
  eh  Ethernet_Hdr_For_IP;
  iph IP_v4_Hdr_For_ICMP;
  type UINT 8;
  code UINT 8;
```

```
  checksum BIT 16;
}
```

3.5.1. Specialized fields

You do not have to build and name all the patterns you are going to use later. While packet layouts are defined by protocols and are mostly known in advance, conditions applied to data in the packets could vary widely. PAX allows you to define a pattern and to apply some conditions to its data only when it is referenced:

```
<pattern>:= <pattern_ref> WHERE <pattern_conditional_expr>
```

Thus, you may define more generic patterns and customize them as required, as shown below:

```
PATTERN  Ethernet_Hdr {
  destination BIT 48;
  source BIT 48;
  type UINT 16
}

PATTERN  IP_v4_Hdr {
  version UINT 4 == 4;
  ihl UINT 4 == 5;            /* length == 5 : no options */
  typeOfService UINT 8;
  totalLength UINT 16;
  identification UINT 16;
  flagReserved BIT 1 == 0;
  flagDontFragment BIT 1;
  flagMoreFragments BIT 1 == 0;    /* last fragments only */
  fragmentOffset UINT 13 == 0;    /* first fragments only */
  timeToLive UINT 8;
  protocol BIT 8;
  headerChecksum BIT 16;
  sourceAddress BIT 32;
  destinationAddress BIT 32

}

PATTERN  ICMP_Over_IP_Over_Ethernet_Hdr {
  eh  Ethernet_Hdr WHERE type == ETHERTYPE_IP;
  iph IP_v4_Hdr WHERE protocol == IPPROTO_ICMP;
  type UINT 8;
  code UINT 8;
  checksum BIT 16;
}
```

3.5.2. Conditional fields

Conditional field is used to describe patterns that are have different layouts depending on certain conditions.

```
<cond_field> := <field_name>
                <pattern> WHEN <pattern_cond_expr>
```

Consider the pattern for matching the MAC LLC header:

```
PATTERN IEEE_802_2_LLC {
   DSAP BIT 8 <> 0xFF;
   SSAP BIT 8 <> 0xFF;
   Control1 BIT 2;
   LongControl  BIT 14 WHEN Control1 <> 0b11;
   ShortControl BIT  6 WHEN Control1 == 0b11;
 }
```

Without resorting to conditional field, we would have to express the same pattern in less elegant way through the use of combination.

```
PATTERN IEEE_802_2_LLC {
   DSAP BIT 8 <> 0xFF;
   SSAP BIT 8 <> 0xFF;
   Control [
      LongControl {
         Control1 BIT 2 <> 0b11;
         Control2 BIT 14
      };
      ShortControl {
         Control1 BIT 2 == 0b11;
         Control2 BIT 6
      }
   ]
}
```

3.5.3. Variant Fields

Variant fields serve to describe variants of the pattern layout depending on the value of the other field or on the value of arbitrarily complex conditional expression involving multiple fields.

```
<anyof_field> := <field_name> <anyof_head> } |
                 <field_name> <anyof_head> ; }

<anyof_head>  := ANYOF {<case> |
                        <anyof_head> ; <case>

    <case>    := <case_selector> : <case_name> <pattern>

    <case_selector> := <pattern_cond_expr>
```

The <case_name> must be different from the names of all cases already included in <anyof_head>.

Scope for the <pattern_conditional_expr> is the pattern for which an ANYOF field is a member. Only the fields preceding this ANYOF field in the definition and definition subfields may be referenced in the

<pattern_conditional_expr>. The ANYOF is a convenient shorthand for the combination operator. The following demonstrates the semantics:

```
PATTERN original { pre_field_1 ...;
                     ...;
                   pre_field_K ...;
                   anyOfField ANYOF {
                      case_selector_1 : pattern_1;
                        ...;
                      case_selector_M : pattern_M
                   }
                   post_field_1 ...;
                     ...;
                   post_field_N ...
                 }

PATTERN equivalent [
            case1 {
               /* conditions corresponding
                  to case_selector_1 */ ;
               pre_field_1 ... WHERE ... ;
                 ...;
               /* conditions corresponding
                  to case_selector_K */ ;
               pre_field_1 ... WHERE ... ;
               anyOfField pattern_1;
                  post_field_1 ...;
                     ...;
                  post_field_N ...
            }
             ...;
            caseM {
               /* conditions corresponding to
                  case_selector_M */
               pre_field_1 ... WHERE ... ;
                 ...;
               /* conditions corresponding to
                  case_selector_M */
               pre_field_K ... WHERE ... ;
               anyOfField pattern_M;
               post_field_1 ...;
                  ...;
               post_field_N ...
            };
]
```

Note: Patterns corresponding to different case selectors MUST be mutually exclusive.

3.6. Pattern length adjustment

Patterns of variable length can be adjusted to a user-specified length.

```
      <pattern>   := <pattern_ref> <length>
```

The pattern adjusted to the particular length n will accept the packet:

1. if the unadjusted pattern will accept the evaluated data within the the first n bits; 2. if the evaluated data would not be rejected by the unadjusted pattern within the n bits.

In other words, in the case of matching before the adjusted length is exhausted, the remaining bits are skipped (ignored, treated as padding). In case of non-matching after the adjusted length is exhausted, the remaining packet bits are not evaluated (truncated).

Let's consider the options field in IP header.

```
OptionsAndPadding ANYOF {
  IHL == 5  : opt1 IP_4_16Options 0;
  IHL == 6  : opt2 IP_4_16Options 32;
  IHL == 7  : opt3 IP_4_16Options 64;
  IHL == 8  : opt4 IP_4_16Options 96;
  IHL == 9  : opt5 IP_4_16Options 128
      WHERE FirstOption.type == SECURITY_OPT;
  IHL == 10 : opt6 IP_4_16Options 160;
  IHL == 11 : opt7 IP_4_16Options 192;
  IHL == 12 : opt7 IP_4_16Options 224;
  IHL == 13 : opt8 IP_4_16Options 256;
  IHL == 14 : opt9 IP_4_16Options 288;
  IHL == 15 : opt10 IP_4_16Options 320
}
```

Here the options field is adjusted depending on the total header length defined by the IHL field. For the IHL of 9 words, if the first option is the security option, the packet will be accepted, otherwise it will be rejected.

3.7. Parametric patterns

Parametric patterns have the following structure:

```
<formal_parameter_list> :=
 <parameter_name> <builtin> | <formal_parameter_list> ,
                              <parameter_name> <builtin>

<formal_parameter_list> :=
  PATTERN <pattern_name>    ( <formal_parameter_list> )
```

The WHERE clauses could be applied only to the patterns which are defined by the PATTERN statements in the same compilation unit.

Internal structure of the patterns IMPORT'ed from pattern libraries is not visible by the compiler. The following sample prototype has two parameters:

```
PATTERN IP_4_Hdr ( srcAddr BIT 32, dstAddr BIT 32 )
```

3.7.1. Parameters declared in the formal parameter

The definition of the parametric pattern consists of the prototype with formal parameter list and the pattern (the body).

Parameters declared in the formal parameter list can be used within the body of the pattern wherever the literal of the same type and length can occur, as shown in the following:

```
PATTERN IP_4_Hdr ( srcAddr BIT 32, dstAddr BIT 32 ) {
   version UINT 4 == 4;
   ihl UNIT 4 == 5; /* length == 5 : no options */
   typeOfService UINT 8;
   totalLength UINT 16;
   flagReserved BIT 1 == 0;
   flagDontFragment Bit 1;
   fragmentOffset UINT 13;
   timeToLive UINT 8;
   protocol BIT 8;
   headerChecksum BIT 16;
   sourceAddress BIT 32 == srcAddr;
   destinationAddress BIT 32 == dstAddr;
}
```

3.7.2. Parameters declared by reference

When parametric patterns are referenced, the actual parameters corresponding to the formal parameters have to be supplied. They can be literals (or the parameter references from the outer scope) of the appropriate type:

```
<operand>          := <numeric_literal> | <parameter_ref>
<actual_parmeter_list> :=
        <operand> | <actual_parameter_list> , <operand>
<pattern_ref>      := <pattern_name>(<actual_parameter_list)

e.g.

IMPORT TcpHeader ( sourcePort UINT 16, destinPort UINT 16 )

PATTERN Tip4TcpConn (srcIpAddress BIT 32, dstIpAddress BIT 32,
                     srcPort UINT 16,
                     dstPort UINT 16 ) {
  eh       Ethernet_Hdr WHERE Tyupe == ETHERTYPE_IP;
  iph      IP_4_Hdr( srcIpAddress, dstIpAddress )
                         WHERE protocol == IPPROTO_TCP AND
                                  fragmentOffset == 0;
  tcph     TcpHeader( srcPort, dstPort );
}

PATTERN ftp_attack {
         f1 ip4TcpConn( 209.67.1.1, 10.10.1.1, 1, 21 ) WHERE
```

```
                tcp.ctrl.syn == 1
        }
```

Note: 209.67.1.1 is a special notation for IPv4 addresses. It matches the BIT 32 data type.

4. Elementary patterns and Atomic conditional expressions HEADER-2

4.1. Literals

Literals allow representation of constants in various formats:

```
<binary_number>          := 0b0 | 0b1 | 0B0 | 0B1 |
<binary_number> 0
                                      <binary_number> 1

<octal_digit>            := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<octal_number>           := 0 | <octal_number> <octal_digit>

<decimal_digit>          := <octal_digit> | 8 | 9

<decimal_number>         := 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
                            <decimal_number> <decimal_digit>

<hex_digit>              := <decimal_digit> | a | b | c | d | e |
f |
                            A | B | C | D | E | F

<hex_number>             := 0x <hex_digit> | 0X <hex_digit> |
                            <hex_number> <hex_digit>

<number>                 := <binary_number> | <octal_number> |
                            <decimal_number> | <hex_number>

<numeric_literal>        := <number>
```

For UINT fields, literals can be expressed as binary numbers, octal numbers, decimal numbers or hexadecimal numbers.

In the above notation:

- a binary number is any sequence of 0s and 1s preceded by 0b or 0B,
- an octal number is any sequence of octal digits starting with 0,
- a decimal number is any sequence of decimal digits not starting with 0, and
- a hexadecimal number is any sequence of hexadecimal digits preceded by 0x or 0X.

```
  <binary_masked_string>   := 0b0 | 0b1 | 0b* | 0B0 | 0B1 | 0B* |
```

```
                                <binary_masked_string> 0 |
                                <binary_masked_string> 1 |
                                <binary_masked_string> *
```

```
     <octal_masked_string>    := 0 | <octal_number> <octal_digit> |
                                      <octal_number> *

     <hex_masked_string>      := 0x <hex_digit> | 0X <hex_digit> | 0X
     |
                                   <hex_number> <hex_digit> |
                                   <hex_number> *
     <masked_bitstring>        := <binary_masked_string> |
                                  <octal_masked_string> |
                                  <hex_masked_string>

     <numeric_literal>         := <masked_bitstring>
```

For BIT fields, literals can be expressed as binary, octal or hexadecimal masked strings. Notation is very similar to binary, octal or hexadecimal numbers, with addition of * as a don't care digit.

* represents one don't care bit in binary string, one don't care octal nibble (3 bits) in octal string, and one don't care hexadecimal nibble (4 bits) in hexadecimal string.

Any valid number is also a valid masked string of the same type.

A binary masked string is any sequence of 0s, 1s, and *s preceded by 0b or 0B.

An octal masked string is any sequence of octal digits and *s starting with 0.

A hexadecimal masked string is any sequence of hexadecimal digits and *s preceded by 0x or 0X.

4.2. Relations

The following shows basic PAX relations:

```
<operand>                 := <numeric_literal> | <parameter_ref>
<rel_op>                  := == | <> | <= | >= | < | >
<atomic_conditional_expr> := <rel_op> <operand>
<pattern>                 := <builtin> <atomic_conditional_expr>
```

Conditions applied to the elementary (builtin) fields depend on the type of the field.

For UINT fields, <rel_op> can be any of:

o <

o <=

- >
- >=

o ==

o <>

<operand> can be a number or reference to the UINT parameter. ,

For BIT fields, <rel_op> can be only == or <>; <operand> has to be a masked string or reference to the BIT parameter. Any valid number is also a valid masked string.

4.3. Atomic conditional expressions

Several conditions can be applied to the same field using more complex atomic conditional expressions.

```
<pattern> := <builtin> ! <atomic_conditional_expr>
```

If <builtin> <atomic_conditional_expr> accepts the input, then <pattern> rejects it, and vice versa.

If the input is too short to be either accepted or rejected by <builtin> <atomic_conditional_expr> , then it is too short to be either accepted or rejected by <pattern>.

```
<pattern> := <builtin> <atomic_conditional_expr1> &&
             <atomic_conditional_expr2>
```

Input is accepted by <pattern> if it is accepted by both <builtin> <atomic_conditional_expr1> and <builtin> <atomic_conditional_expr2> .

```
<pattern> := <builtin> <atomic_conditional_expr1> ||
             <atomic_conditional_expr2>.
```

Input is accepted by <pattern> if it is accepted either by <builtin> <atomic_conditional_expr1> or <builtin> <atomic_conditional_expr2>.

```
<atomic_conditional_expr> := ( <atomic_conditional_expr1> )
```

Parentheses can be used to control the order of operations in usual way. In their absence, the precedence of the operators is:

1. !

2. &&

3. ||

5. Pattern Conditional Expressions

The following pattern was used as an example in 3.5, Pattern References:

```
PATTERN  ICMP_Over_IP_Over_Ethernet_Hdr {
  eh  Ethernet_Hdr WHERE type == ETHERTYPE_IP;
```

```
  iph IP_v4_Hdr WHERE protocol == IPPROTO_ICMP;
  type UINT 8;
  code UINT 8;
  checksum BIT 16;
}
```

In the above, the eh and iph fields are qualified by pattern conditional expressions in its simplest form. The above pattern is equivalent to:

```
PATTERN  ICMP_Over_IP_Over_Ethernet_Hdr {
  eh  Ethernet_Hdr_For_IP;
  iph IP_v4_Hdr_For_ICMP;
  type UINT 8;
  code UINT 8;
  checksum BIT 16;
}
```

Note: In the above example, the patterns Ethernet_Hdr_For_IP and IP_v4_Hdr_For_ICMP are defined as:

```
PATTERN  Ethernet_HdrFor_IP {
  ...
  type UINT 16 == ETHERTYPE_IP
}

PATTERN  IP_v4_Hdr_For_ICMP {
  ...
  protocol BIT 8 == IPPROTO_ICMP;
  ...
}
```

5.1. Field references

The examples shown in Pattern Conditional Expressions, above, put conditions only on the pattern fields directly referred to in the field definition. PAX also allows you to address embedded fields through any level of nesting, as shown below:

```
<field_reference>  := <field_name> |
                <field_reference>.<field_name>
<pattern_conditional_expr> :=
                <field_reference><atomic_conditional_expr>
```

If the <field_reference> points to the non-elementary pattern, then any of its fields can be addressed by <field_reference> . <field_name>.

```
PATTERN ftp_header {
  f1 ip4TcpConn( 209.67.1.1, 10.10.1.1, 1, 21 ) WHERE
tcph.ctrl.syn == 1
}
```

tcph is a field of ip4TcpConn, ctrl is a field of TcpHeader, and syn is a field of Control. See the pattern definitions in 3.6, Parametric Patterns.

5.2. Pattern Conditional Expressions

You may combine the pattern conditional expressions using the usual logical operators shown below:

```
<pattern_conditional_expr> := <field_reference>
                              <atomic_conditional_expr> |
                              ( <pattern_conditional_expr> ) |
                              NOT <pattern_conditional_expr> |
                              <pattern_conditional_expr> AND
                              <pattern_conditional_expr> |
                              <pattern_conditional_expr> OR
                              <pattern_conditional_expr>
```

6. Pattern Libraries

Some patterns may be written using their names to a single output file, the Pattern Library.

Pattern libraries serve as the input for further steps in the classification process, but they can be also used as a source of patterns for the compiler through IMPORT statements.

6.1. EXPORT statement

```
<export_def> :=  EXPORT <pattern_name> | <export_def>
<pattern_name>
```

Pattern construction is hierarchical. Many patterns may be named in a program which serve only as the intermediate steps towards an ultimate pattern. Files may also be #include'd which contain many more unneeded definitions.

To prevent littering of the result, the compiler writes to the output file only the patterns explicitly named in the EXPORT statements, such as:

```
EXPORT  Ethernet_Hdr IP_v4_Hdr
```

Any number of EXPORT statements can appear anywhere in the program. <pattern_name>s used in EXPORT statements do not need to be defined before, as processing of all EXPORT statements takes place only at the end of the compiler run.

6.2. IMPORT statement

```
<import_def> := IMPORT <pattern_name> |
                IMPORT <pattern_name> ( <formal_parameter_list> )
```

IMPORT statement declares the pattern to be available in one of the

import libraries. Such patterns can be referenced in the following statements, and the prototype provides enough information for the compiler to process references to parametric library patterns. WHERE

clauses cannot be applied to the library patterns, whereas the length adjustment can.

The patterns named in the IMPORT statements are retrieved from the libraries (and the agreement between prototypes from the library and from the IMPORT statement is checked) only at the end of the compiler run, and only if such patterns are (directly or indirectly) used by one of the patterns to be exported.

```
PATTERN ip4TcpConn ( srcIpAddress BIT 32,
                     dstIpAddress BIT 32,
                     srcPort UINT 16,
                     dstPort UINT 16 )
```

7. References

Bradner97
S. Bradner, "Key words for use in RFCs to indicate Requirement Levels", RFC2119, March 1997

APF96
H. Dan Lambright and Saumya K. Debray, "APF : A Modular language for Fast Packet Classification". Dept of Computer Science, University of Arizona, Tucson, August 30, 1996.

BPF93
S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture". In USENIX Technical Conference Proceedings, pages 259- 269, San Diego, CA, Jan. 1993.

CSPF87
J. C. Mogul, R. F. Rashid and M. J. Accetta, "The packet filter: An efficient mechanism for user-level network code". In Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, pages 39-- 51, Nov. 1987.

DPF96
Dawson R. Engler and M. Frans Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation", Proc. of ACM Sigcomm, August 1996.

MPF94
M. Yuhara, B. N. Bershad, C. Maeda, and J. E. Moss, "Efficient packet demultiplexing for multiple endpoints and large messages". Proc. Winter 1994 USENIX Conference, Jan. 1994.

PATHFINDER94
M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "PATHFINDER: A pattern-based packet classifier". In Proceedings of the First Symposium on Operating Systems Design and

Implementation, pages 115-123, November 1994.

8. Editors' Addresses

Misha Nossik
mnossik@solidum.com
Solidum Systems Corp
+1 (613) 244-4804

Michael Richardson
mcr@solidum.com

Feliks J. Welfeld
feliks@solidum.com

9. Appendix A. Formal Grammar

```
<program>           := <statement> | <program> < statement>

<statement>         := <export_def> | <import_def> |
                          <pattern_def>

<export_def>        := EXPORT <pattern_name> | <export_def>
                          <pattern_name>

<pattern_name>      := <identifier>

<identifier>        := <letter> | <identifier> <character>

<letter>            : = a | b | c | d | e | f | g | h | i | j | k |
                        l | m | n | o | p | q | r | s | t | u | v |
                        w | x | y | z | A | B | C | D | E | F | G |
                        H | I | J | K | L | M | N | O | P | Q | R |
                        S | T | U | V | W | X | Y | Z

<character>         := <letter> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
                        8 | 9 | _

<import_def>        := IMPORT <pattern_name> |
                       IMPORT <pattern_name> (
                          <formal_parameter_list> )

<formal_parameter_list> := <parameter_name> <builtin> |
                           <formal_parameter_list> ,
                        <parameter_name> <builtin>

<parameter_name>    := <identifier>

<builtin>               := <var_builtin_type> <length>

<var_builtin_type>      := BIT | UINT
```

```
<length>                := <number>

<number>                := <binary_number> | <octal_number> |
```

```
<decimal_number> |
                          <hex_number>

<binary_number>         := 0b0 | 0b1 | 0B0 | 0B1 |
                           <binary_number> 0 | <binary_number> 1

<octal_number>          := 0 | <octal_number> <octal_digit>

<octal_digit>           := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<decimal_number>        := 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
                           <decimal_number> <decimal_digit>

<decimal_digit>         := <octal_digit> | 8 | 9

<hex_number>            := 0x <hex_digit> | 0X <hex_digit> |
<hex_number>
                           <hex_digit>

<hex_digit>             := <decimal_digit> | a | b | c | d | e | f |
                            A | B | C | D | E | F

<pattern_def>           := <pattern_prototype> <pattern>

<pattern_prototype>     := PATTERN <pattern_name> |
                            PATTERN <pattern_name> (
                            <formal_parameter_list> )

<pattern>               := <builtin> |
                      <builtin> <atomic_conditional_expr> |
                      <concatenation> |
                      <combination> |
                      <pattern_ref> |
                      <pattern_ref> WHERE <pattern_conditional_expr> |
                      <pattern_ref> <length> |
                <pattern_ref> <length> WHERE <pattern_conditional_expr>

<combination>           := <combination_head> ] |
                             <combination_head> ; ]

<combination_head>      := [ <field> |
                            <combination_head> ; <field>

<concatenation>          := <concatenation_head> } |
                               <concatenation_head> ; }

<concatenation_head>     := { <field> |
                            <concatenation_head> ; <field> |
                            <concatenation_head> ; <cond_field> |
                            <concatenation_head> ; <anyof_field>
```

```
<cond_field>            := <field_name> <pattern> WHEN
                                    <pattern_cond_expr>
```

```
<field>                 := <field_name> <pattern>

<atomic_conditional_expr>:= <rel_op> <operand> | (
                        <atomic_conditional_expr> ) |
                        ! <atomic_conditional_expr> |
                        <atomic_conditional_expr> &&
                               <atomic_conditional_expr> |
                        <atomic_conditional_expr> ||
                               <atomic_conditional_expr>

<rel_op>                 := == | <> | <= | >= | < | >

<operand>                := <numeric_literal> | <parameter_ref>

<numeric_literal>        := <number> | <masked_bitstring>

<masked_bitstring>       := <binary_masked_string> |
                            <octal_masked_string> |
                            <hex_masked_string>

<binary_masked_string>   :=  0b0 | 0b1 | 0b* | 0B0 | 0B1 | 0B* |
                             <binary_masked_string> 0 |
                             <binary_masked_string> 1 |
                             <binary_masked_string> *
<octal_masked_string>     := 0 | <octal_number> <octal_digit> |
                             <octal_number> *

<hex_number>              := 0x <hex_digit> | 0X <hex_digit> | 0X*
                             | <hex_number> <hex_digit> |
                             <hex_number> *

<parameter_ref>           := <parameter_name>

<field_list>              := <field> | <field_list> ; <field> |
<field_list> ;

<field>                   := <field_name> <pattern> |
                                <anyof_field>

<anyof_field>             := <field_name> <anyof_head> } |
                                <field_name> <anyof_head> ; }

<anyof_head>              := ANYOF {<case> |
                                <anyof_head> ; <case>

<case>                    := <case_selector> : <case_name>
                                <pattern>

<case_selector>           := <pattern_cond_expr>
```

```
<field_name>                := <identifier>

<case_list>                 := <case> | <case_list> ; <case> |
```

```
                                <case_list> ;

<case>                      := <case_selector> : <pattern>

<case_selector>             := <pattern_conditional_expr>

<pattern_conditional_expr> := <field_reference>
                              <atomic_conditional_expr> |
                              ( <pattern_conditional_expr> ) |
                              NOT <pattern_conditional_expr> |
                              <pattern_conditional_expr> AND
                              <pattern_conditional_expr> |
                              <pattern_conditional_expr> OR
                              <pattern_conditional_expr>

<field_reference>           := <field_name> |
                              <field_reference> . <field_name>

<pattern_ref>               := <pattern_name> |
                              <pattern_name> ( <actual_parameter_list> )

<actual_parameter_list>     := <operand> |
                               <actual_parameter_list> , <operand>
```