Nathan Steinmann
nsteinma@gmail.com

Programming Languages and Translators
Language Proposal

I. Introduction

This paper describes a language for identifying communication protocols and extracting fields and values, referred to as *metadata*, from a set of packets in a capture file. To simplify the language runtime, we shall restrict the input to packets beginning with an IPv4 header. This eliminates the need to identify data-link layer protocols, which typically require finding chains of packets that exhibit similar values at specific byte offsets in each packet. For the purposes of this project, such chaining serves only to complicate the runtime and does not enrich the language itself.

The output of a program written in the language will be a set of *events*, one for each packet that matches the identification algorithms contained in the program. An event may be thought of as a simplified textual representation of a packet. More formally, an event is defined as containing exactly one protocol identifier string, such as `POP3` or `SMTP`, and zero or more key/value fields, such as `HASH_ALGORITHM=SHA-256`, or `USERNAME=ALICE`. Keys are always strings, but values may be any data type supported by the language.

II. Language Description

Structurally, the language is similar to *awk* in that the runtime is responsible for reading and iterating over packets from a capture file without requiring explicit instructions in the program. This enables the language to concentrate on protocol identification and metadata extraction instead of file management.

a. Organization

The language is logically separated into four sections. The first section contains zero or more global variable declarations, as described by the grammar:

declaration → *type identifier*;
type → **int** | **string**
identifier → [A-Za-z][A-Za-z0-9_]*

Following the global variables are one or more packet tests of the form:

packet-test → *test-ident* : *condition*

test-ident → *identifer*
condition → *packet-reference relop literal*
relop → < | <= | = | != | >= | >
literal → [0-9]* | [A-Za-z]*

where

packet-reference refers to a subset of the current packet through one of three system variables: *bytes*, *byte*, or *string*. The *packet-reference* non-terminal is further described in section II.b.

literal is either a numeric or string literal depending on whether the value returned from *packet-reference* is numeric or a string.

The next section is a series of one or more conformance rules that combine packet tests using boolean operators. Rules are expressed as:

rule → *rule-ident* **?** *conformance-rule*
rule-ident → *identifier*
conformance-rule → *test-ident logicop conformance-rule*
conformance-rule → *test-ident*
logicop → && | '||'

Note: *conformance-rule* may also include parentheses to override *logicop* precedence or associativity.

Conformance rules are executed in the order in which they appear. Once a successful rule is found, any remaining rules will be skipped.

The final section of a program is zero or more functions used to extract metadata from the current packet. Functions are comprised of local variable declarations, metadata extraction/assignment statements, and calls to other functions.

Functions are categorized into one of three types:

1) *Built-in:* Built-in functions have a special significance to the language runtime. Only one built-in function is currently defined: *init()*, which is called immediately after a successful conformance test is executed and before any other functions are called.

2) *Entry-point:* An entry-point function is one that has the same name as a *rule-ident*. An entry-point function is called if the conformance test identified by the matching *rule-ident* is successful.

3) *Utility*: Utility functions are any function other than those defined in (1) and (2). Utility functions may be called by built-in, entry-point, or by other utility functions.

b. Accessing Packet Data

Packet data may be referenced through the reserved system variables *bytes*, *byte*, or *string*. All three variables may be thought of as an array, and are accessed using an array-like syntax. A brief description of each variable follows:

1) *bytes[x, y]*: returns a numeric value starting at offset *x* from the beginning of the packet and including *y* bytes. Offsets start at zero, and multi-byte words are interpreted in big-endian order. To eliminate the need for multi-precision arithmetic, we shall assume that programs written in the language will be executed on 32-bit processors and therefore restrict $1 <= y <= 4$.

2) byte*[x]*: returns the byte at offset *x*. Synonymous with *bytes[x, 1]*.

3) *string[x, y]*: returns *y* characters starting at offset *x* as a string.

c. Data Types

Two data types are supported – strings and integers. An integer is a four byte signed numeric type and is declared using the **int** keyword. A string is an arbitrarily long sequence of contiguous characters and is declared using the **string** keyword.

d. Arithmetic

The language supports basic arithmetic calculations, such as addition, subtraction, multiplication, division, and fundamental bitwise operations, including AND, OR, XOR, and NOT. The grammar, although not defined here, is expected to be similar to C language.

e. Events

If a packet matches any of the conformance rules, then an event is generated with a protocol identifier string equal to the *rule-ident* of the matching conformance rule. Any subsequent function may then add additional key/value fields to the event by using the *event* system variable as follows:

```
event.key = value;
```

The key field need not be declared, and takes the data type of the value being assigned to it. Subsequent assignments to the same field name overwrite previously assigned values.

f. Other System Variables

In addition to the three packet-referencing and event variables, the system also defines two additional variables, EOP (END OF PACKET) and SPACE. EOP is an integer variable that contains the offset of the last byte of the current packet. SPACE is also an integer, but is somewhat more complicated than EOP. The first time SPACE is referenced for a given packet, its value is the offset of the first space character in that packet. Subsequent references to SPACE on the same packet return the offset of the second space character, followed by the third, and so on. If SPACE is called and no space characters exist or remain in the current packet, -1 is returned.

g. Comments

Both single and multi-line comments are supported with syntax identical to that used in the C language.

III. Sample Program

The following program identifies POP3 traffic from a client to server by detecting any TCP traffic destined to port 110. If the packet contains a USER or PASS command, denoted by the string "USER" or "PASS" in the first four characters, then the username or password is extracted and stored in an event. For any other command, an event is created with the command string stored in the command field.

```
/*
Global variable declaration.  Initialize with a calculation
to determine the length of the IP header.
*/
int ipLength = (byte[0] & 0xf) * 4;

/*
Another global variable declaration.  Possibly used later
in the program by multiple functions.
*/
int space;

// Next protocol TCP?
tcp : byte[9] = 6;

// Client to server POP3?
dstport110 : bytes[ipLength, 2] = 110;
```

```
// Does the packet contain the user command?
user : string[0, 3] = "user";
USER : string[0, 3] = "USER";

// Does the packet contain the password command?
pass : string[0, 3] = "pass";
PASS : string[0, 3] = "PASS";

/*
Conformance rule - Is this a TCP packet going to the POP3
well-known port that contains a user command, either in all
uppercase or lowercase?
*/
pop3_user ? tcp & dstport110 && (user || USER);

/*
Similar to previous conformance rule, except looks for the
pass command.
/*
pop3_pass ? tcp & dstport110 && (pass || PASS);

// POP3 client-to-server catch-all conformance rule
pop3 ? tcp & dstport;

init()
{
     space = SPACE;
     event.command = string[0, space-1];
}

pop3_user()
{
     event.username = string[space+1, EOP];
}

pop3_pass()
{
     event.password = string[space+1, EOP];
}
```