PERIL:
Packet Extraction, Reporting, and Identification Language

Nathan Steinmann

18 December 2007

Table of Contents

1. Introduction

PERIL is a language for identifying network protocols and extracting fields and values, referred to as *metadata*, from a collection of IPv4 packets contained in a capture file. Internally, PERIL represents metadata from a single packet as an *event*, which consists of a protocol identification string, such as `POP3`, and a sequence of name/value pairs, such as `USERNAME=ALICE`. The output of a PERIL program is zero or more events, which can be thought of as a simplified textual representation of targeted network activity. Such functionality is desirable, for example, by network security personnel auditing activity at a network border device, such as a firewall or router.

To simplify the implementation of protocol detectors, PERIL only supports stateless identification algorithms, i.e., those algorithms that are able to completely determine the identity of a packet without referring to data contained in other packets. A consequence of this design decision is that PERIL will not be able to identify the first protocol in a packet, the data-link layer, as doing so typically requires locating consistent values at defined offsets across a large number of packets. As a work-around, PERIL requires that all packets in an input file begin with an IPv4 header. Moreover, PERIL input files must be saved in the widely used PCAP format, supported by many popular tools including tcpdump and Wireshark.

1.1. Motivation

Many tools already exist for viewing packet data. In general, these programs are designed to present a holistic view of the capture file, frequently displaying a table with a synopsis of each packet and another table with a dissected view of the fields and data in the currently selected packet. Although many of these programs permit the user to filter packets according to some specified criteria, the result is typically the entire packet and not user-specified fields within the packet.

PERIL's design is also motivated in part by the complexity of reading and manipulating packet data programmatically using common libraries. For instance, simply reading a packet using a popular library required no fewer than four variables and two function calls if no error handling were used. Additionally, many packet manipulation libraries only return packets as arrays of bytes and do not provide functionality to read multi-byte values or strings from a packet.

1.2. Goals

PERIL is designed to be an intuitive, yet simple language for identifying and extracting information from IPv4 packets.

1.2.1. Intuitive Syntax

PERIL is based on the C language, adding only three additional constructs: packet tests, conformance rules, and packet references. All of PERIL's mathematical and logical operators are identical to their C counterparts.

1.2.2. Simplified Packet Manipulation

PERIL provides features to ease packet access and manipulation. More specifically, the PERIL runtime handles opening the input file and iterating through packets without any explicit instruction in the PERIL program itself. Additionally, the runtime provides simple instructions to read a byte, multi-byte values, or strings directly from the packet.

2. Language Tutorial

A PERIL program is logically separated into four sections, which must appear in the following order: global variable declarations, packet tests, conformance rules, and function definitions. All sections must be present except for global variable declarations, which are optional.

The runtime automatically handles reading packet data from the capture file (supplied via a command-line parameter) and iterating over the packets. For each packet in the input, each section in the PERIL code is executed as if it were in a loop. Access to the currently active packet is also provided by the runtime through a *packet reference*, further described in section 3.5.2.

2.1. Computing the Length of IPv4 and TCP headers

The length (in octects) of an IPv4 header is easily calculated with a global variable declaration and a packet reference. The length of an IPv4 packet (in 32-bit words) is transmitted in the low-order nybble of the first byte of the header. We can therefore calculate the length in octects by:

```
int ipLength = (byte[0] & 0xf ) * 4;
```

In this example, `byte[0]` is a packet reference that simply returns the first byte of the current packet as an integer value. The TCP header length (in 32-bit words) is given in the high-order nybble of the 13th byte of the TCP header. Written as a global variable declaration, we have:

```
int tcpLength = ((byte[ipLength+12] & 0xf0) >> 4) * 4;
```

For convenience, let's also declare a third variable that contains the offset to the application layer protocol:

```
int dataOffset = ipLength + tcpLength;
```

2.2. Examining the Application Layer Protocol

Let's assume that we're interested in detecting HTTP GET packets. To do this, three conditions need to hold: the transport protocol must be TCP, the destination port must be 80, and the first three characters of the HTTP payload must be "GET". We'll begin by checking the type of the transport protocol by using a packet test. TCP is identified by the constant 6 in the IPv4 Next Protocol field at offset 9 of the IPv4 header. In packet test syntax:

```
tcp : byte[9] == 6;
```

The `:` denotes that this a packet test with the label `tcp`. In this case, the byte at offset 9 in the current packet is compared with 6. If true, `tcp` evaluates as true; otherwise, it evaluates as false.

A similar test is needed for the destination port number. The destination port is a two-byte quantity beginning two bytes into the TCP header. Expressed as a packet test, we have:

```
dstPort : bytes[ipLength+2, ipLength+3] == 80;
```

Although very similar to the previous example, the packet reference uses the `bytes` keyword to access a multi-byte value. To avoid the use of multi-precision arithmetic, `bytes` is limited to returning no more than four bytes in a single reference.

The third form of a packet reference accesses packet data as a string, which we'll use to verify that the first three bytes of the HTTP payload are "GET":

```
get : string[dataOffset, dataOffset+2] == "GET";
```

2.3. Combining Multiple Packet Tests

As mentioned in the previous section, all three of the `tcp`, `dstPort`, and `get` packet tests must be true to identify a HTTP GET packet. We express this using a conformance rule:

```
httpGet when tcp && dstPort && get;
```

The keyword `when` denotes that this statement is a conformance rule, and similar to a packet test, `http` is a label that uniquely identifies this conformance rule. When a conformance rule evaluates as true, two actions are triggered:

- A new event is automatically created by the runtime with its protocol identification string equal to the label of the conformance rule, and
- program execution is transferred to a function whose name matches the conformance rule, if one exists. Any unevaluated conformance rules are skipped.

2.4. Extracting Metadata

Assuming that a conformance rule has matched, then metadata can be extracted from the current packet by populating event fields in a function. Continuing with our HTTP

example, let's create one field called document that stores that path of the requested document. To do this, we'll define a function whose name matches the name of the conformance rule from the previous section:

```
httpGet()
{
    #document = string[SPACE+1, EOP];
}
```

Several things are happening in this example. Let's begin with the `#document` identifier. Identifiers beginning with a # are fields contained within the current event. The second new feature is the `SPACE` keyword. Each time `SPACE` is called, it returns the offset of the next space character in the packet. The last feature is the `EOP` keyword, which resolves to the last valid offset in the current packet. Since HTTP GET packets begin with GET, followed by a space and end with the document path, SPACE+1 returns the offset of the first character of the document path and EOP returns the offset of the last character of the document path. Thus, this statement assigns the requested document path to the document field in the current event.

Once the final statement in the function has been executed, the event is printed to standard output, the next packet in the capture file is read, and execution resumes by reinitializing the global variable declarations.

2.5. Running

All code should be placed in a single source file, say http.peril. Execute the translator from the project's `bin` directory with the command:

```
java –jar peril.jar http.peril http.c
```

This will produce the file http.c containing the translated C code. It can be built by running `make` in the same directory, which will compile and link the code with the libraries libpcap.a (for accessing the packet capture file) and libperil.a (the PERIL runtime). When executing the resulting binary, simply supply the path to the capture file as a command-line parameter.

3. Language Manual

3.1. Lexical Conventions

The PERIL language utilizes several types of tokens: comments, identifiers, keywords, constants, operators, and separators. Separators include tab, space, carriage return, and newline, and are ignored by PERIL except as needed to separate other token types.

3.1.1. Comments

The characters // denote the beginning of a single line comment, which terminates at the next encountered newline character. Similarly, the characters /* indicate the beginning of a multiline comment, which terminates upon the characters */. Comments do not nest, nor do they occur within string literals.

3.1.2. Identifiers

Identifiers are a sequence of letters and digits and must begin with a letter or a hash symbol (#). Identifiers beginning with a hash are events, and are used to store metadata from packets. Identifiers are case sensitive and may be of any length.

3.1.3. Keywords

Keywords are identifiers with special significance to PERIL and may not be used as regular identifiers. The PERIL keywords are: **int**, **string**, **byte**, **bytes**, **EOP**, **SPACE**, and **when**.

3.1.4. Constants

There are two types of constants: integers and string literals.

3.1.4.1. Integers

Integer constants may be specified as either decimal or hexadecimal values. An integer constant is hexadecimal if it begins with the characters 0x and is followed by a sequence of digits, including the hexadecimal digits a or A through f or F. If the integer constant does not begin with the characters 0x, then it is interpreted as decimal and must consist of a sequence of decimal digits.

3.1.4.2. String Literals

String literals consist of a sequence of characters enclosed in double quotes. Double quotes may be included as part of a string literal by using the escape sequence \".

3.2. Meaning of Identifiers

Variables, packet tests, conformance rules, functions, and variables are all named using identifiers. Variables have associated properties indicating their scope and the type of information referred to by the variable.

3.2.1. Scope

All identifiers have global scope with the exception of variables, which may have either global or local scope. Global variables are declared at the beginning of the program, outside of a function definition. Local variables are declared inside a function definition, and are visible only within that function. The value of a local variable does not persist between calls to a function.

3.2.2. Type

Variables may be of type integer or string. Integer variables store 32 bits of information, and are always interpreted as an unsigned value. Strings store a sequence of characters and may be any length.

3.3. Expressions

The subsections listed below are in order of the precedence of the operators. Operators defined within the same subsection have equal precedence.

3.3.1. Primary Expressions

Primary expressions are identifiers, constants, parenthesized expressions, subscripted identifiers, or function calls, and are associated left to right.

3.3.1.1. identifier

Identifiers (described in section 2.2) are primary expressions provided that they have been properly declared as described below.

3.3.1.2. constant

Constants, as defined in section 2.4, are primary expressions and may be either numeric integers or string literals.

3.3.1.3. (expression)

Parenthesized expressions are primary expressions with higher precedence than the equivalent expression without parenthesis. The type and value of the parenthesized expression is equivalent to the unadorned expression.

3.3.1.4. primaryExpression [expression secondaryIndex$_{opt}$]

Primary expressions followed by an expression and an optional secondary index in square brackets is a primary expression referred to as a subscripted expression. In PERIL, the primary expression refers to a collection of data elements and the indexes (i.e. the expression and optional secondary index) identify a particular data element or range of data elements within the collection.

3.3.1.5. identifier ( ) ;

An identifier followed by an open and close parenthesis and a semicolon is a primary expression denoting a function call.

3.3.2. Postfix Expression

A postfix expression is a primary expression followed by square brackets containing an expression or two expressions separated by a comma.

3.3.3. Multiplicative operators

The binary multiplicative operators *, /, and % group left to right. Both operands of a multiplicative operator must be integer.

*3.3.3.1. expression * expression*

The * operator denotes multiplication.

3.3.3.2. expression / expression

The / operator denotes integer division. Dividing by zero is a run-time error and will cause a PERIL program to terminate.

3.3.3.3. expression % expression

The % operator returns the remainder of the first operand divided by the second.

3.3.4. Additive operators

The binary operators + and – group left to right. Both operators require integer operands.

3.3.4.1. expression + expression

The + operator returns the sum of the operands.

3.3.4.2. expression – expression

The – operator returns the difference of the second operand subtracted from the first.

3.3.5. Shift operators

The binary shift operators << and >> group left to right. Both operands must be integer.

3.3.5.1. expression << expression

3.3.5.2. expression >> expression

The left shift operation is denoted by <<, and indicates shifting the binary representation of the first operand left by the number of bits specified by the second

operand modulo 32. All vacated bits are zero-filled. The right shift operator, >>, is defined similarly, differing only in the direction that bits are shifted.

3.3.6. **Relational operators**

The binary relational operators perform logical comparisons between the operands. The operands must be the same type, but that type may be either integer or string.

3.3.6.1. expression < expression

3.3.6.2. expression > expression

3.3.6.3. expression <= expression

3.3.6.4. expression >= expression

The operators < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) return 1 if the specified relation is true or 0 if it is false.

3.3.7. **Equality operators**

Similar to the relational operators, the binary equality operators perform logical comparisons between the operands. The operands must be the same type, but that type may be either integer or string.

3.3.7.1. expression == expression

3.3.7.2. expression != expression

The operators == (equal to) and != (not equal to) return 1 when the specified condition is true or 0 if it is false.

3.3.8. **Bitwise operators**

3.3.8.1. expression & expression

The binary & operator groups left-to-right and returns the bitwise logical `and` of the operands. Both operands must be of type integer.

3.3.8.2. expression ^ expression

The binary ^ operator groups left-to-right and returns the exclusive bitwise `or` of the operands. Both operands must be of type integer.

3.3.8.3. expression | expression

The binary | operator groups left-to-right and returns the bitwise inclusive `or` of the operands. Both operands must be of type integer.

3.3.9. Logical operators

3.3.9.1. expression && expression

The binary && operator returns 1 if both operands are non-zero, 0 otherwise. Both operands must be of type integer.

3.3.9.2. expression || expression

The binary || operator returns 1 if either operand is non-zero, 0 otherwise. Both operands must be of type integer.

3.3.10. Assignment operators

3.3.10.1. expression = expression

The binary = operator stores the second operand into the memory location referred to by the first operand. The type of both operands must be identical.

3.4. Declarations

Declarations are used to specify names and data types of variables. Declarations have the form:

```
declaration:
        typeSpecifier declaratorList ;
```

3.4.1. Type Specifiers

The type specifiers are

typeSpecifier:
int
string

and are described further in section 3.1.4.1 and 3.1.4.2, respectively.

3.4.2. Declarators

As described previously, declarations consist of a type specifier followed by a declarator list, which is a sequence of comma-separated declarators.

declaratorList:
declarator
declarator , *declarator-list*

The type given in a declaration applies to all declarators in the declarator list. Declarators have the form:

declarator:
identifier $initializer_{opt}$;

3.4.3. Initializers

Declarators may optionally initialize a variable by including an initializer. If present, the initializer follows the identifier and consists of an assignment operator followed by an expression.

initializer:
= *expression*

3.5. Statements

A statement is defined as an expression followed by a semicolon. Related to a statement is a compound statement, which is an open brace, zero or more local variable declarations, zero or more statements, and a closing brace. Compound statements are used only in the body of a function definition.

3.5.1. Packet Tests

At least one packet test must follow the optional global variable declarators. Packet tests consist of an identifier, a colon, and an expression. The expression will typically refer to packet data through a packet reference and compare it with a constant or computed value. The result of a packet test will be interpreted by one or more conformance rules as a boolean value where zero is considered to be false and any non-zero value is true.

3.5.2. Packet References

Packet data is represented as a sequence of bytes and is referred to using an array-like syntax via the **byte**, **bytes**, or **string** keywords. **byte** is accessed using a single subscript, so that **byte**[x] returns the byte at offset x in the current packet. **bytes** can return up to four contiguous bytes, and is referenced as **bytes**[x, y], where x is the starting offset in the current packet and y is the ending offset, inclusive. **string** uses similar notation as **bytes**, but returns the data as a string and therefore has no length restriction on the amount of data that can be returned.

3.5.3. Conformance Rules

Packet tests are followed by at least one conformance rule. Conformance rules are syntactically similar to packet tests, differing only by the replacement of the colon with the keyword **when**. Conformance rules are used evaluate the results of packet tests through boolean algebra.

3.6. Functions

One of more function definitions follow the conformance rules. Functions may have the same name as a conformance rule, a unique identifier, or may have the special identifier `init`. If a conformance rule evaluates as true, then the function with the matching name is called. Function names without matching conformance rules are utility functions, and may be called by other functions. If defined, the `init` function will be called before any other functions are executed.

3.6.1. Definitions

Following the conformance rules are one or more function definitions. A definition consists of an identifier suffixed by an open and close parenthesis followed by a compound statement.

3.7. Syntax Summary

3.7.1. Expressions

```
expression:
     primaryExpression
     expression binop expression

primaryExpression:
     identifier
     constant
     ( expression )
     primaryExpression [ expression secondaryIndex_opt]
     functionCall

secondaryIndex:
     , expression

functionCall:
     identifier ( ) ;
```

The primaryExpresion operators

() []

have highest priority and associate left-to-right. The binary operators all associate left-to-right, and are listed in decreasing precedence as indicated:

```
binop:
     *    /    %
     +    -
     >>   <<
     <    >    <=   >=
     ==   !=
     &
     ^
     |
     &&
     ||
     =
```

3.7.2. Declarations

```
declaration:
     typeSpecifier declaratorList ;
```

```
typeSpecifier:
     int
     string

declaratorList:
     declarator
     declarator , declarator-list

declarator:
     identifier initializer_opt ;

initializer:
     = expression
```

3.7.3. Statements

```
statement:
     expression ;
     packetTest ;
     conformanceRule ;

statementList:
     statement
     statement statementList

packetTest:
     identifier : expression ;

conformanceRule:
     identifier when expression;
```

3.7.4. Functions

```
functionDefinition:
     identifier ( ) functionBody

functionBody:
     { declaratorList statementList }
```

4. Project Plan

4.1. Project Timeline

Antipicated Date	Actual Date	Milestone
25 September	25 September	Project Whitepaper
11 October	16 October	Lexer / Parser Complete
18 October	18 October	Language Reference Manual
1 November	10 November	Abstract Syntax Tree Complete
15 November	24 November	Tree Walker Complete
4 December	8 December	Initial Version
13 December	16 December	Final Version
18 December	18 December	Project Report

4.2. Development Environment

The project was developed entirely on an Apple MacBook Pro running MacOS 10.4. All code was written in Eclipse 3.2.2 using ANTLR 2.7.7 and Java 1.5.0_07. The PERIL runtime was written in C using Eclipse and compiled with gcc 4.0.1. The build process for Java was managed using Apache Ant 1.6.5, and C was managed using GNU Make 3.80. All code and documentation was maintained in a remote CVS repository.

5. Architectural Design

PERIL consists of standard compiler units, such as the lexer, parser, and an abstract syntax tree, as well as several units specific to PERIL. These units include the Translator, Scope, and PERILExpression classes. The relationship between the classes is shown in Figure 5.1:

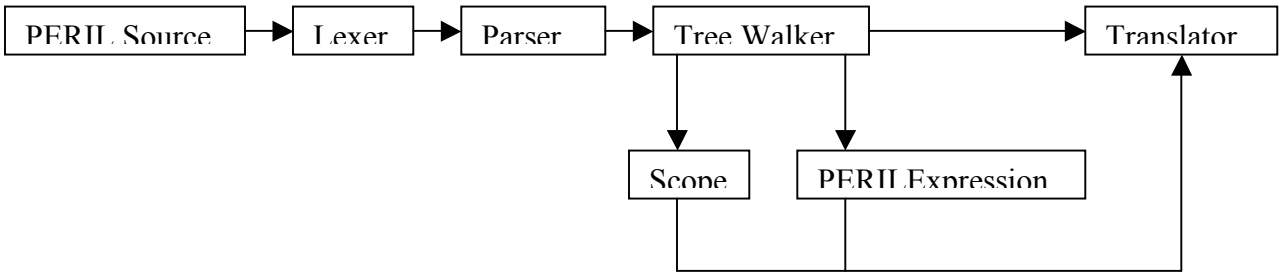

Figure 5.1: PERIL Block Diagram

Not represented in the diagram is the overall driver of PERIL, the Main class. Main is responsible for instantiating the lexer, parser, and tree walker, and opening and passing the contents of the PERIL input file to the lexer. The parser is relatively self-contained, requesting tokens from the lexer as needed and returning an abstract syntax tree (AST) representing the source program. Main passes the AST to the tree walker, which analyzes the AST, creating Scope and PERILExpression objects as necessary and storing them in the Translator object.

5.1. Scope

The Scope class stores the symbol table and any statements associated with a particular scope. There is always a default scope called “global” that stores global variable declarations and initializers. New Scope objects are created by the tree walker when a function definition is encountered in the AST.

5.2. PERILExpression

The PERILExpression class builds string representations of expressions. It includes methods corresponding to each operator documented in section 3.3.3 through 3.3.10.1. Among its notable responsibilities are recognizing and translating the string equality operator to the C strcmp function and translating event field references to a function call into the PERIL runtime.

5.3. Translator

The Translator class contains data structures to store the packet tests, conformance rules, and function definitions as well as Scope objects. Translator also contains methods for outputting the various PERIL constructs as C code.

5.4. Runtime Environment

A significant portion of PERIL's functionality is provided by its runtime library, which is implemented in C. The runtime API is:

```
void checkArguments(int, char *[]);
void clearEvent();
unsigned int getBytes(unsigned int, unsigned int);
char *getString(unsigned int, unsigned int);
void openCaptureFile(char *);
void printEvent();
int readPacket();
void setEventName(char *);
void setField(char *, char *);
unsigned int SPACE();
```

6. Test Plan

6.1. Sample Program: HTTP GET Requests

This section presents the complete program developed in the Tutorial. It identifies HTTP GET packets and extracts the path of the request document from the packet.

6.1.1. Source Code

```
int ipLength = (byte[0] & 0xf) * 4;
int tcpLength = ((byte[ipLength+12] & 0xf0) >> 4) * 4;
int dataOffset = ipLength = tcpLength;

tcp : byte[9] == 6;
dstPort : bytes[ipLength+2, ipLength+3] == 80;
get : string[dataOffset, dataOffset+2] == "GET";

httpGet when tcp && dstPort && get;

init()
{
}

httpGet()
{
      #document = string[SPACE+1, EOP];
}
```

6.1.2. Output

```
#include <peril.h>

void init();
void httpGet();
int ipLength;
int tcpLength;
int dataOffset;

int main(int argc, char *argv[])
{
```

```
	char get;
	char dstPort;
	char tcp;

	checkArguments(argc, argv);
	openCaptureFile(argv[1]);

	while (readPacket())
	{
		ipLength = (getBytes(0, 0) & 0xf) * 4;
		tcpLength = ((getBytes(ipLength + 12, ipLength + 12) &
0xf0) >> 4) * 4;
		dataOffset = ipLength = tcpLength;
		get = !strcmp(getString(dataOffset, dataOffset + 2),
"GET");
		dstPort = getBytes(ipLength + 2, ipLength + 3) == 80;
		tcp = getBytes(9, 9) == 6;

		if (tcp && dstPort && get)
		{
			httpGet();
			continue;
		}
	}

	return 0;
}

void init()
{
}

void httpGet()
{
	clearEvent();
	setEventName("httpGet");
	init();
	setField("document", getString(SPACE() + 1, EOP));
	printEvent();
}
```

6.2. Sample Program: POP3 Username/Password Extraction

This program identifies POP3 client to server traffic and extracts usernames and passwords.

6.2.1. Source Code

```
int ipLength = (byte[0] & 0xf) * 4;
int tcpLength = ((byte[ipLength+12] & 0xf0) >> 4) * 4;
int dataOffset = ipLength + tcpLength;
int space;

tcp : byte[9] == 6;
dstport110 : bytes[ipLength+2, ipLength+3] == 110;
user : string[dataOffset, dataOffset+3] == "user";
USER : string[dataOffset, dataOffset+3] == "USER";
pass : string[dataOffset, dataOffset+3] == "pass";
PASS : string[dataOffset, dataOffset+3] == "PASS";

pop3User when tcp && dstport110 && (user || USER);
pop3Pass when tcp && dstport110 && (pass || PASS);

init()
{
      space = SPACE;
      #command = string[dataOffset, space-1];
}

pop3User()
{
      #username = string[space+1, EOP];
}

pop3Pass()
{
      #password = string[space+1, EOP];
}
```

6.2.2. Output

```
#include <peril.h>

void init();
void pop3User();
```

```
void pop3Pass();
int ipLength;
int tcpLength;
int space;
int dataOffset;

int main(int argc, char *argv[])
{
      char USER;
      char PASS;
      char user;
      char pass;
      char tcp;
      char dstport110;

      checkArguments(argc, argv);
      openCaptureFile(argv[1]);

      while (readPacket())
      {
            ipLength = (getBytes(0, 0) & 0xf) * 4;
            tcpLength = ((getBytes(ipLength + 12, ipLength + 12) &
0xf0) >> 4) * 4;
            dataOffset = ipLength + tcpLength;
            USER = !strcmp(getString(dataOffset, dataOffset + 3),
"USER");
            PASS = !strcmp(getString(dataOffset, dataOffset + 3),
"PASS");
            user = !strcmp(getString(dataOffset, dataOffset + 3),
"user");
            pass = !strcmp(getString(dataOffset, dataOffset + 3),
"pass");
            tcp = getBytes(9, 9) == 6;
            dstport110 = getBytes(ipLength + 2, ipLength + 3) ==
110;

            if (tcp && dstport110 && (user || USER))
            {
                  pop3User();
                  continue;
            }

            if (tcp && dstport110 && (pass || PASS))
            {
```

```
                pop3Pass();
                continue;
            }

        }

        return 0;
}

void init()
{
        space = SPACE();
        setField("command", getString(dataOffset, space - 1));
}

void pop3User()
{
        clearEvent();
        setEventName("pop3User");
        init();
        setField("username", getString(space + 1, EOP));
        printEvent();
}

void pop3Pass()
{
        clearEvent();
        setEventName("pop3Pass");
        init();
        setField("password", getString(space + 1, EOP));
        printEvent();
}
```

6.3. Testing Strategy

Time didn't allow for development of a robust automated test suite. Once initial development was complete, testing was accomplished by executing the Unix `diff` command on the translator output against a known good sample and manually reconciling any unexpected differences.

7. Lessons Learned

Designing a programming language and translator was a significant challenge. One of more difficult aspects for me was learning and using ANTLR, as I found its documentation to be somewhat lacking. I also found certain ANTLR errors and warning to be hard (and time-consuming) to resolve. Despite this, I came to appreciate ANTLR's flexibility and power once I become comfortable with it.

While working on the translator, I became aware of several problems in the PERIL language. For example, the `SPACE` keyword, which returns the offset of the first space in the packet, should have accepted a parameter that indicated where in the packet to begin the search. Unfortunately, by the time I found the problem, it was too late in the project to modify the parser to accommodate the parameter.

I also realized after writing several test programs that every PERIL program shared common operations, such as computing the length of IP and TCP headers, and establishing a pointer to the application layer protocol payload. Had I conceptualized more example programs prior to writing the PERIL white paper, I would have designed features into the language to handle these repetitive tasks.

Appendix

7.1. Main.java

```
package peril;

import antlr.collections.*;
import java.io.*;
import peril.antlr.*;

public class Main
{
	private AST ast;
	private PERILLexer lexer;
	private PERILParser parser;
	private PERILWalker walker;
	private Translator t;

	public void parse(String filename) throws Exception
	{
		lexer = new PERILLexer(new
FileInputStream(filename));
		parser = new PERILParser(lexer);
		walker = new PERILWalker();
		parser.file();
		ast = parser.getAST();

		t = walker.file(ast);
	}

	public void translate(String outputFile) throws
IOException
	{
		BufferedWriter bw = new BufferedWriter(new
FileWriter(outputFile));

		bw.write("#include <peril.h>\n\n");
		t.writeFunctionPrototypes(bw);
		t.writeDeclarations(bw, "global");
		bw.write("\nint main(int argc, char *argv[])\n{\n");
```

```
            // Packet test variable declarations
            t.writePacketTestVariables(bw);

            bw.write("\n\tcheckArguments(argc,
argv);\n\topenCaptureFile(argv[1]);\n\n\twhile
(readPacket())\n\t{\n");
            t.writeInitializers(bw, "global");
            t.writePacketTestStatements(bw);
            t.writeConformanceRules(bw);

            // Close while loop
            bw.write("\t}\n");

            bw.write("\n\treturn 0;\n}\n");

            t.writeFunctions(bw);

            bw.close();
      }

      public static void main(String[] args)
      {
            Main main = new Main();

            try
            {
                  if (args.length != 2)
                  {
                        System.out.println("Usage peril <source-
file> <output-file>");
                        System.exit(1);
                  }

                  main.parse(args[0]);
                  main.translate(args[1]);
            }
            catch (Exception e)
            {
                  e.printStackTrace();
            }
      }
}
```

7.2. PERILExpression.java

```
package peril;

public class PERILExpression
{
	private StringBuilder expression;

	public PERILExpression(String s)
	{
		expression = new StringBuilder(s);
	}

	public PERILExpression assign(PERILExpression pe)
	{
		if (expression.toString().startsWith("#"))
			expression = new StringBuilder("setField(\"" +
expression.substring(1) + "\", " + pe +")");
		else if (pe.toString().compareTo("SPACE") == 0)
			expression.append(" = " + "SPACE()");
		else
			expression.append(" = " + pe);

		return this;
	}

	public PERILExpression bitwiseAnd(PERILExpression pe)
	{
		expression = new StringBuilder("(" + expression + " &
" + pe + ")");

		return this;
	}

	public PERILExpression bitwiseOr(PERILExpression pe)
	{
		expression.append(" | " + pe);

		return this;
	}

	public PERILExpression divide(PERILExpression pe)
	{
```

```
            expression.append(" / " + pe);

            return this;
      }

      public PERILExpression equal(PERILExpression pe)
      {
            if (pe.toString().startsWith("string") ||
pe.toString().charAt(0) == '"')
                  expression = new StringBuilder("!strcmp(" + this
+ ", " + pe +")");
            else
                  expression.append(" == " + pe);

            return this;
      }

      public PERILExpression greater(PERILExpression pe)
      {
            expression.append(" > " + pe);

            return this;
      }


      public PERILExpression greaterEqual(PERILExpression pe)
      {
            expression.append(" >= " + pe);

            return this;
      }

      public PERILExpression less(PERILExpression pe)
      {
            expression.append(" < " + pe);

            return this;
      }

      public PERILExpression lessEqual(PERILExpression pe)
      {
            expression.append(" <= " + pe);

            return this;
```

```
	}

	public PERILExpression logicalAnd(PERILExpression pe)
	{
		expression.append(" && " + pe);

		return this;
	}

	public PERILExpression logicalOr(PERILExpression pe)
	{
		expression = new StringBuilder("(" + expression + " ||
" + pe + ")");

		return this;
	}

	public PERILExpression leftShift(PERILExpression pe)
	{
		expression.append(" << " + pe);

		return this;
	}

	public PERILExpression minus(PERILExpression pe)
	{
		expression.append(" - " + pe);

		return this;
	}

	public PERILExpression modulus(PERILExpression pe)
	{
		expression.append(" % " + pe);

		return this;
	}

	public PERILExpression notEqual(PERILExpression pe)
	{
		expression.append(" != " + pe);

		return this;
	}
```

```
	public PERILExpression add(PERILExpression pe)
	{
		expression.append(" + " + pe);

		return this;
	}

	public PERILExpression rightShift(PERILExpression pe)
	{
		expression = new StringBuilder("(" + expression + " >>
" + pe + ")");

		return this;
	}

	public PERILExpression multiply(PERILExpression pe)
	{
		expression.append(" * " + pe);

		return this;
	}

	public PERILExpression xor(PERILExpression pe)
	{
		expression.append(" ^ " + pe);

		return this;
	}

	public String toString()
	{
		return expression.toString();
	}
}
```

7.3. Scope.java

```
package peril;

import java.io.*;
import java.util.*;

class Scope
{
	/**
	 * Maps the symbol names to a <code>SymbolTableEntry</code>
containing their type and initializer, if provided.
	 */
	private Map<String, SymbolTableEntry> symbolTable = new
HashMap<String, SymbolTableEntry>();

	/**
	 * Stores the statements contained in this scope.
	 */
	private List<String> statements = new
LinkedList<String>();

	/**
	 * Adds the specified statement to the statement list for
this scope.
	 *
	 * @param statement The statement to add.
	 */
	public void addStatement(String statement)
	{
		statements.add(statement);
	}

	/**
	 * Adds a symbol to this scope's symbol table.
	 *
	 * @param type The type of the symbol being added.
	 * @param name The name of the symbol being added.
	 *
	 * @return true if the symbol was successfully added; false
if the symbol already exists.
	 */
	public boolean addSymbol(String type, String name,
```

```
PERILExpression initializer)
      {
            if (!symbolTable.containsKey(name))
            {
                  symbolTable.put(name, new
SymbolTableEntry(initializer, type));

                  return true;
            }
            else
                  return false;
      }

      public void writeDeclarations(Writer w) throws
IOException
      {
            SymbolTableEntry ste;

            for (Map.Entry me : symbolTable.entrySet())
            {
                  ste = (SymbolTableEntry)me.getValue();

                  if (ste.getType().compareTo("string") == 0)
                        w.write("char *" + me.getKey());
                  else
                        w.write(ste.getType() + " " + me.getKey());

                  w.write(";\n");
            }
      }

      public void writeInitializers(Writer w) throws
IOException
      {
            SymbolTableEntry ste;

            for (Map.Entry me : symbolTable.entrySet())
            {
                  ste = (SymbolTableEntry)me.getValue();

                  if (ste.getInitializer() != null)
                  w.write("\t" + me.getKey() + " = " +
ste.getInitializer() + ";\n");
            }
```

```
	}

	public void writeStatements(Writer w) throws IOException
	{
		for (String s : statements)
			w.write("\t" + s + ";\n");
	}

	private class SymbolTableEntry
	{
		private PERILExpression initializer;
		private String type;

		public SymbolTableEntry(PERILExpression initializer,
String type)
		{
			this.initializer = initializer;
			this.type = type;
		}

		public PERILExpression getInitializer()
		{
			return initializer;
		}

		public String getType()
		{
			return type;
		}
	}
}
```

7.4. Translator.java

```
package peril;

import java.io.*;
import java.util.*;

public class Translator
{
	private Map<String, String> conformanceRules = new
LinkedHashMap<String, String>();
	private Map<String, String> packetTests = new
HashMap<String, String>();
	private Map<String, Scope> scopes = new HashMap<String,
Scope>();

	/**
	 * Adds a new conformance rule.
	 *
	 * @param name The name of the conformance rule to add.
	 * @param expression The expression bound to the name.
	 *
	 * @return true if the symbol was successfully added; false
if the rule name already exists.
	 */
	public boolean addConformanceRule(String name,
PERILExpression expression)
	{
		if (!conformanceRules.containsKey(name))
		{
			conformanceRules.put(name,
expression.toString());

			return true;
		}
		else
			return false;
	}

	/**
	 * Adds a new packet test.
	 *
	 * @param name The name of the packet test to add.
```

```
	 * @param expression The expression bound to the name.
	 *
	 * @return true if the test was successfully added; false
if the test name already exists.
	 */
	public boolean addPacketTest(String name, PERILExpression
expression)
	{
		if (!packetTests.containsKey(name))
		{
			packetTests.put(name, expression.toString());

			return true;
		}
		else
			return false;
	}

	/**
	 * Adds the specified statement to the given scope.
	 *
	 * @param scopeName The name of the scope to add the
statement to.
	 * @param statement The statement to add.
	 */
	public void addStatement(String scopeName, PERILExpression
statement)
	{
		Scope scope;

		if ((scope = scopes.get(scopeName)) == null)
		{
			scope = new Scope();

			scopes.put(scopeName, scope);
		}

		if (statement != null)
			scope.addStatement(statement.toString());
	}

	/**
	 * Adds a symbol to the specified scope.
	 *
```

```
	 * @param scope The name of the scope containing the
specified symbol.
	 * @param type The type of the symbol.
	 * @param name The name of the symbol.
	 *
	 * @return true if the symbol was successfully added; false
if the symbol already exists.
	 */
	public boolean addSymbol(String scopeName, String type,
String name, PERILExpression initializer)
	{
		Scope scope;

		if ((scope = scopes.get(scopeName)) == null)
		{
			scope = new Scope();

			scopes.put(scopeName, scope);
		}

		return scope.addSymbol(type, name, initializer);
	}

	public void writeConformanceRules(Writer w) throws
IOException
	{
		for (Map.Entry me : conformanceRules.entrySet())
		{
			w.write("\t\tif (" + me.getValue() +
")\n\t\t{\n");
			w.write("\t\t\t" + me.getKey() +
"();\n\t\t\tcontinue;\n\t\t}\n\n");
		}
	}

	public void writeDeclarations(Writer w, String scopeName)
throws IOException
	{
		scopes.get(scopeName).writeDeclarations(w);
	}

	public void writeFunctionPrototypes(Writer w) throws
IOException
	{
```

```
            String scopeName;

            for (Map.Entry me : scopes.entrySet())
            {
                  scopeName = (String)me.getKey();

                  if (scopeName.compareTo("global") != 0)
                        w.write("void " + scopeName + "();\n");
            }
      }

      public void writeFunctions(Writer w) throws IOException
      {
            boolean isInit;
            String scopeName;

            for (Map.Entry me : scopes.entrySet())
            {
                  scopeName = (String)me.getKey();

                  if (scopeName.compareTo("global") != 0)
                  {
                        isInit = scopeName.compareTo("init") == 0 ?
true : false;

                        w.write("\nvoid " + scopeName + "()\n{\n");

                        if (!isInit)
                        {
                              w.write("\tclearEvent();\n");
                              w.write("\tsetEventName(\"" +
scopeName + "\");\n");
                              w.write("\tinit();\n");
                        }

                        ((Scope)me.getValue()).writeStatements(w);

                        if (!isInit)
                              w.write("\tprintEvent();\n");

                        w.write("}\n");
                  }
            }
      }
```

```
	public void writeInitializers(Writer w, String scopeName)
throws IOException
	{
		scopes.get(scopeName).writeInitializers(w);
	}

	public void writePacketTestStatements(Writer w) throws
IOException
	{
		for (Map.Entry me : packetTests.entrySet())
			w.write("\t\t" + me.getKey() + " = " +
me.getValue() + ";\n");

		w.write("\n");
	}

	public void writePacketTestVariables(Writer w) throws
IOException
	{
		for (String s : packetTests.keySet())
			w.write("\tchar " + s + ";\n");
	}
}
```

7.5. peril.g

```
header
{
package peril.antlr;
}

class PERILParser extends Parser;
options
{
buildAST = true;
exportVocab = PERIL;
k = 2;
}

tokens
{
DECL;
FILE;
FUNCDEF;
PKTREF;
}

file : (declarator)* (packetTest)+ (conformanceRule)+
(functionDefinition)+ EOF! { #file = #([FILE, "FILE"], file); };

/************
* Expressions
*************/
primaryExpression : ID | NUMBER | STRING | packetReference |
(LPAREN! expression RPAREN!) | functionCall;
//postfixExpression : primaryExpression (LBRACKET expression
(COMMA! expression)? RBRACKET)?;
multiplicativeExpression : primaryExpression ((STAR^ | DIVIDE^ |
MODULUS^) primaryExpression)*;
additiveExpression : multiplicativeExpression ((PLUS^ | MINUS^)
multiplicativeExpression)*;
shiftExpression : additiveExpression ((LSHIFT^ | RSHIFT^)
additiveExpression)*;
relationalExpression : shiftExpression ((LESS^ | GREATER^ |
LESSEQUAL^ | GREATEREQUAL^) shiftExpression)*;
equalityExpression : relationalExpression ((EQUAL^ | NOTEQUAL^)
relationalExpression)*;
```

```
bitwiseAndExpression : equalityExpression (BITWISEAND^
equalityExpression)*;
exclusiveOrExpression : bitwiseAndExpression (XOR^
bitwiseAndExpression)*;
bitwiseOrExpression : exclusiveOrExpression (BITWISEOR^
exclusiveOrExpression)*;
logicalAndExpression : bitwiseOrExpression (LOGICALAND^
bitwiseOrExpression)*;
logicalOrExpression : logicalAndExpression (LOGICALOR^
logicalAndExpression)*;
assignmentExpression : logicalOrExpression (ASSIGN^
logicalOrExpression)*;
expression : assignmentExpression;
packetReference : ("byte" | "bytes" | "string") LBRACKET!
expression (COMMA! expression)? RBRACKET! {#packetReference =
#([#PKTREF, "PKTREF"], packetReference); };

/***********
* Statements
************/
statement : expression SEMICOLON!;
compoundStatement : LBRACE! (declarator)* (statement)* RBRACE!;

/*******************************
* Packet test / Conformance rule
*******************************/
packetTest : ID COLON^ statement;
conformanceRule : ID "when"^ statement;

/***********
* Declarator
************/
declarator : ("int" | "string") ID (ASSIGN! expression)?
SEMICOLON! { #declarator = #([DECL, "DECL"], declarator); };

/**********
* Functions
***********/
functionCall : ID LPAREN RPAREN;
functionDefinition : ID LPAREN! RPAREN! compoundStatement {
#functionDefinition = #([FUNCDEF, "FUNCDEF"],
functionDefinition); };

class PERILLexer extends Lexer;
```

```
options
{
charVocabulary = '\3'..'\377';
exportVocab = PERIL;
k = 2;
testLiterals=false;
}

ASSIGN : '=';
BITWISEAND : '&';
BITWISEOR : '|';
COLON : ':';
COMMA : ',';
DIVIDE : '/';
EQUAL : "==";
GREATER : '>';
GREATEREQUAL : ">=";
ID options {testLiterals=true;} : (HASH)? LETTER (LETTER |
DIGIT)*;
LBRACE : '{';
LBRACKET : '[';
LESS : '<';
LESSEQUAL : "<=";
LOGICALAND : "&&";
LOGICALOR : "||";
LPAREN : '(';
LSHIFT : "<<";
MINUS : '-';
MODULUS : '%';
MULTILINECOMMENT : "/*" (options {greedy=false;}: (.))* "*/"
{$setType(Token.SKIP);};
NEWLINE : ('\n' | "\r\n" | '\r') {$setType(Token.SKIP);
newline();};
NOTEQUL : "!=";
NUMBER : (("0x") (DIGIT | 'a'..'f' | 'A'..'F')+) | (DIGIT)+;
PLUS : '+';
RBRACE : '}';
RBRACKET : ']';
RPAREN : ')';
RSHIFT : ">>";
SEMICOLON : ';';
SINGLELINECOMMENT : "//" (options {greedy=false;}: (.))* '\n'
{$setType(Token.SKIP); newline();};
STAR : '*';
```

```
STRING : '"' (('\\'! '"') | (~('\\' | '"')))* '"';
WS : (' ' | '\t') {$setType(Token.SKIP);};
XOR : '^';

protected DIGIT : '0'..'9';
protected HASH : '#';
protected LETTER : ('A'..'Z') | ('a'..'z');
```

7.6. walker.g

```
header
{
package peril.antlr;
}

{import peril.*;}

class PERILWalker extends TreeParser;

options
{
importVocab=PERIL;
}

{
      String scope = "global";
      Translator t = new Translator();
}

file returns [Translator trans]
{
      String type, name;
      trans = t;
}
      : #(FILE (declaration)* (packetTest)+ (conformanceRule)+
(functionDefinition)+);


declaration
{
      PERILExpression pe;
      String type;
}
      : #(DECL type=dataType name:ID pe=expression
{t.addSymbol(scope, type, name.getText(), pe);});

packetTest
{
      PERILExpression pe;
}
      : #(COLON id:ID pe=expression)
```

```
{t.addPacketTest(id.getText(), pe);};

conformanceRule
{
      PERILExpression pe;
}
      : #("when" id:ID pe=expression)
{t.addConformanceRule(id.getText(), pe);};

expression returns [PERILExpression pe]
{
      PERILExpression a, b;
      String s;
      pe = null;
}
      : #(ASSIGN a=expression b=expression {pe = a.assign(b);})
      | #(BITWISEAND a=expression b=expression {pe =
a.bitwiseAnd(b);})
      | #(BITWISEOR a=expression b=expression {pe =
a.bitwiseOr(b);})
      | #(DIVIDE a=expression b=expression {pe = a.divide(b);})
      | #(EQUAL a=expression b=expression {pe = a.equal(b);})
      | #(GREATER a=expression b=expression {pe = a.greater(b);})
      | #(GREATEREQUAL a=expression b=expression {pe =
a.greaterEqual(b);})
      | #(LESS a=expression b=expression {pe = a.less(b);})
      | #(LESSEQUAL a=expression b=expression {pe =
a.lessEqual(b);})
      | #(LOGICALAND a=expression b=expression {pe =
a.logicalAnd(b);})
      | #(LOGICALOR a=expression b=expression {pe =
a.logicalOr(b);})
      | #(LSHIFT a=expression b=expression {pe =
a.leftShift(b);})
      | #(MINUS a=expression b=expression {pe = a.minus(b);})
      | #(MODULUS a=expression b=expression {pe = a.modulus(b);})
      | #(NOTEQUAL a=expression b=expression {pe =
a.notEqual(b);})
      | #(PKTREF s=packetReference {pe = new
PERILExpression(s);})
      | #(PLUS a=expression b=expression {pe = a.add(b);})
      | #(RSHIFT a=expression b=expression {pe =
a.rightShift(b);})
      | #(STAR a=expression b=expression {pe = a.multiply(b);})
```

```
      | #(XOR a=expression b=expression {pe = a.xor(b);})
      | num:NUMBER {pe = new PERILExpression(num.getText());}
      | str:STRING {pe = new PERILExpression(str.getText());}
      | id:ID {pe = new PERILExpression(id.getText());};

dataType returns [String s]
{
      s = null;
}
      : #("int" {s = new String("int");})
      | #("string" {s = new String("string");});

functionDefinition
{
      PERILExpression pe = null;
}
      : #(FUNCDEF name:ID {scope = name.getText();
t.addStatement(scope, null);} (declaration)* (pe=expression
{t.addStatement(scope, pe);})* {scope = "global";});

packetReference returns [String s]
{
      PERILExpression a, b;
      s = null;
}
      : ("byte" a=expression) {s = new String("getBytes(" + a +
", " + a + ")");}
      | ("bytes" a=expression b=expression) {s = new
String("getBytes(" + a + ", " + b + ")");}
      | ("string" a=expression b=expression) {s = new
String("getString(" + a + ", " + b + ")");};
```

7.7. peril.c

```
#include <peril.h>

pcap_t *pcap;
struct pcap_pkthdr pkthdr;
const u_char *data;

void checkArguments(int argc, char *argv[])
{
      if (argc != 2)
      {
            fprintf(stderr, "Usage %s <input-file>\n", argv[0]);
            exit(1);
      }
}

void clearEvent()
{
      NODEPTR p = event.data, q;

      event.name = NULL;

      while (p != NULL)
      {
            q = p->next;
            free(p);
            p = q;
      }

      event.data = NULL;
}

unsigned int getBytes(unsigned int startOffset, unsigned int
stopOffset)
{
      unsigned int retval;

      switch (stopOffset - startOffset + 1)
      {
            case 4:
                  retval = ntohl(*(unsigned int
*)(data+startOffset));
```

```
                break;
          case 3:
                retval = ntohl(*(unsigned int
*)(data+startOffset)) & 0xffffff;

                break;
          case 2:
                retval = ntohs(*(unsigned short
*)(data+startOffset));

                break;
          case 1:
                retval = data[startOffset];

                break;
          default:
                retval = 0;
     }

     return retval;
}

char *getString(unsigned int startOffset, unsigned int
stopOffset)
{
     char *buffer;
     unsigned int length = stopOffset - startOffset + 1;

     if ((buffer = malloc(sizeof(char) * (length + 1))) ==
NULL)
     {
          fprintf(stderr, "Error allocating memory.\n");
          exit(1);
     }

     strncpy(buffer, (char *)(data+startOffset), length);
     buffer[length+1] = '\0';

     return buffer;
}

void openCaptureFile(char *filename)
{
```

```
	char errbuf[PCAP_ERRBUF_SIZE];

	if ((pcap = pcap_open_offline(filename, errbuf)) == NULL)
	{
		fprintf(stderr, "Error reading input file: %s\n",
errbuf);
		exit(1);
	}
}

void printEvent()
{
	NODEPTR temp = event.data;

	printf("Event: %s\n", event.name);

	while (temp != NULL)
	{
		printf("\t%s = %s\n", temp->field, temp->value);
		temp = temp->next;
	}

	printf("\n");
}

int readPacket()
{
	int retval;

	retval = (data = pcap_next(pcap, &pkthdr)) != NULL ? 1 : 0;

	EOP = pkthdr.caplen - 1;

	return retval;
}

void setEventName(char *name)
{
	event.name = name;
}

void setField(char *field, char *value)
{
	NODEPTR temp, p, q = p = event.data;
```

```
	// Search for an existing node with an identical field name
	while (p != NULL)
	{
		q = p;

		if (!strcmp(p->field, field))
		{
			// Field names match, update existing value
			p->value = value;

			break;
		}

		p = p->next;
	}

	if (p == NULL)
	{
		// Didn't find existing node with matching field name.
Insert new node at end of list.
		if ((temp = malloc(sizeof(NODE))) != NULL)
		{
			temp->field = field;
			temp->value = value;
		}
		else
		{
			fprintf(stderr, "Error allocating memory.\n");
			exit(1);
		}

		if (q == NULL)
			// Empty list
			event.data = temp;
		else
			q->next = temp;

		temp->next = NULL;
	}
}

unsigned int SPACE()
{
```

```
        char *pos = (char *)data;

        while (*pos != ' ' && (pos - (char *)data < EOP))
                pos++;

        return (pos - (char *)data < EOP) ? pos - (char *)data :
0;
}
```

7.8. peril.h

```
#ifndef PERIL_H
#define PERIL_H

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct _NODE
{
     char *field;
     char *value;
     struct _NODE *next;
} NODE;

typedef NODE *NODEPTR;

typedef struct
{
     char *name;
     NODEPTR data;
} EVENT;

EVENT event;
unsigned int EOP;

void checkArguments(int, char *[]);
void clearEvent();
unsigned int getBytes(unsigned int, unsigned int);
char *getString(unsigned int, unsigned int);
void openCaptureFile(char *);
void printEvent();
int readPacket();
void setEventName(char *);
void setField(char *, char *);
unsigned int SPACE();

#endif
```