Extending Netsniff

Urs Keller[1], Jason But
Centre for Advanced Interent Architectures. Technical Report 050204B
Swinburne University of Technology
Melbourne, Australia
urs.keller@epfl.ch, jbut@swin.edu.au

Abstract– This technical report describes, how to extend netsniff with additional stream and packet level parser. It also describes how to extend the log file parser and database, that were built to do statistics on the data collected by Netsniff.

Keywords– Netsniff, Extension, Stream-parser, Packet-parser

I. Introduction

Netsniff [1] was developed as part of the ICE^3 project [2] to capture and analyze network traffic, producing application level information. It is currently deployed in a handful of home-based DSL installations of CAIA researchers. It will hopefully be deployed further in the near future. This report describes how to extend Netsniff to support more protocols and applications by adding additional software modules to the application. It also explains how to extend the Netsniff database used to generate statistical results on the collected data. The directory structure for the Netsniff project is described in more detail in Appendix B

II. Structure

Netsniff is based on an object-oriented design and implemented using C++. The underlying pcap library [3] is used to capture all network traffic which is then passed on to a class hierarchy for parsing and logging of relevant information. The hierarchical layout allows a packet header to be processed for information by a single class which then decides whether to pass the enclosed payload to another class for processing. This structured layout allows Netsniff to be easily extended to support:

- Different physical and link-layer protocols
- New Transport Layer protocols
- New Application Layer Protocols.

Netsniff works slightly differently with stream based protocols such as TCP Streams. Netsniff uses a TCPStream instance to reconstruct an entire TCP Stream - gathering TCP level statistics in the process - and passing the TCP bit-stream to an application level parser for further processing.

As such, captured packets are processed both at the packet level and at the stream level where appropriate. Current implementation of packet level processing is indicated in Figure 1 where an arrow indicates what packet types are currently checked for within a particular packet type. The PCAPDev class is not a packet in its own right, but contains the function called by the pcap library [3] and creates the first instance of the Packet class used to process the captured data.

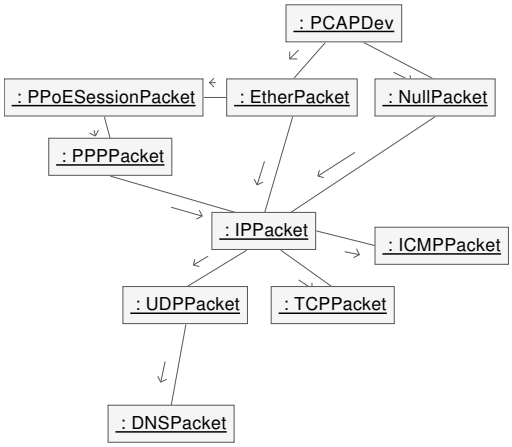

Figure 1: Collaboration Diagram of packet classes

For example, if an ICMP packet is captured on an Ethernet device, the PCAPDev class will construct an EtherPacket instance which will process the Ethernet headers and construct an IPPacket instance. This class processes the IP header and constructs an ICMPPacket instance which processes the remainder of the payload. This design allows parsing of ICMP packets over a variety of underlaying protocols as long as their parsers are complete.

Current Stream level processing is shown in Figure 2, indicating which applications running over a TCP session are currently processed and logged for information.

Any packets that are not processed as part of a supported application are automatically shortened to 68 bytes and written to a log file in tcpdump format.

III. Adding a physical and link-layer protocol

To add a different protocol at this level, we need to:

- Decide what data to collect and output.
- Subclass the Packet class to create a new class type to handle the required protocol.

[1] Urs Keller worked on Netsniff while visiting CAIA from the Swiss Federal Institute in Lausanne $EPFL$.

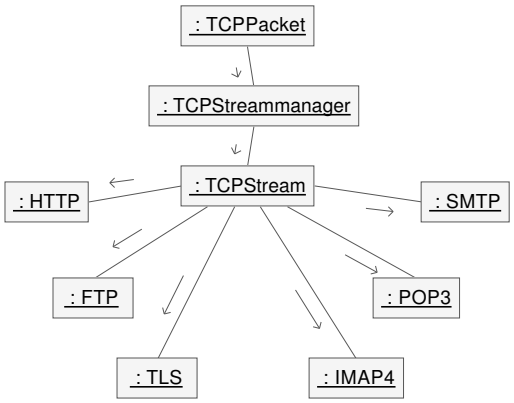

Figure 2: Stream level protocols

- Modify code to create an instance of the new class at the correct level (e.g. If implementing an ATM parser, perhaps the ATMPacket class would be created within PCAPDev, if implementing an ARP parser, we would perhaps create the ARPPacket class within EtherPacket.

The virtual methods within Packet that must be overloaded are:

A. Constructor

The constructor should call the base class constructor with a pointer to its encapsulating packet. This allows packet classes further down the protocol tree the ability to backtrack and request information from parent packets. If the packet type contains an encapsulated packet that is processed by another class, this class should be created and assigned to pcSubPacket. If any information in the header is to be processed, it must be done in the constructor. Once the packet is completely processed, the memory allocated to store the packet contents will be discarded. If the packet and/or header contents are required for other purposes they should either be copied or extracted in the class header.

The PCAPDev class maintains the global netsniff anonymisation flag, which can (and should) be passed to the Packet (and inherited) class constructors. The anonymisation flag can tell your constructor whether it should anonymise any potentially sensitive information. Your constructor should also pass the anonymise flag to any lower level packet parsers. Anonymisation services provided by netsniff are outlined in Section VIII.

B. Destructor

Free any resources allocated in the constructor.

C. ParseStream

If this packet forms part of a stream and is the lowest level of information before the stream is identified, then this method should be overloaded to support the processing of that stream. At the moment, this method is only implemented in the TCPPacket class to construct and maintain TCP Streams. Please see the existing TCPPacket implementation for an example on how to process other stream based protocols.

D. Output

If any information retrieved at this layer should be logged, this is where the work is done. Generally the packet header will be parsed in the constructor and relevant information stored in member variables. This method is automatically called with an output stream to output any information about this level of the protocol tree. Any implementation should also call the Output() method on any further encapsulated packets so that all processed information about the packet is output. If your packet parser will not log any information, the default implementation will automatically call the sub-packet Output() method.

IV. Adding a Packet-Based Application Layer Protocol

Applications that fall into this category perform all their communications at the Packet level rather than a Stream level. Examples include DNS, NFS and ARP. The same steps apply as creating a mid-layered parser, we need to subclass the Packet class, however there are a few other tasks to perform.

A. Constructor

Set the bParsed member variable to true. This will ensure that information is logged to a log file rather that the tcpdump format file. Do not set pcSubPacket to any value, leave it with its default NULL value.

B. DumpFileName

Overload this method to return the file name to dump packet information to. This should only return a file name, the application code will automatically prepend the correct directory location to this filename. The default implementation returns ”/dev/null” to direct logged traffic to.

V. Example of Packet-Based Application Layer Protocol

We try to explain the process described in the previous section by giving an example. We will discuss the implementation of the ARPPacket class by outlining the following steps:

- Decide what data to collect
- Subclass the Packet class
- Create an instance of the created Packet class from another packet or from PCAPDev itself.

A. Output definition

The data you want to collect can be conveniently specified by an output grammar. We will see later, that having a BNF grammar, is handy when building a parser for the log files produced by Netsniff. The grammar for ARPPacket is shown in Listing 1 and how the output eventually looks like in Listing 1.

Once we have decided on what data to include in the output file, we can start writing the code.

```
arp_output    := packet_output SP arp_log_line
arp_log_line := ar_hrd SP ar_pro SP
                ar_hln SP ar_pln SP
                ar_op SP
                ar_sha SP ar_spa SP
                ar_tha SP ar_tpa
```

Listing 1: Grammar for ARP log_line

B. Extend Packet

The Packet class defines the interface shown in Listing 2. What follows is a more detailed method description.

```
class ARPPacket : public Packet
{
 public:
  ARPPacket(u_char *pcPacket,
            Packet *pcParent,
            bool bAnonymise);
  virtual std::string DumpFileName();
  virtual void Output(std::ostream &osOutStream);
  virtual bool OmmitOutputFromParent();
};
```

Listing 2: ARPPacket definition (include/arppacket.h)

ARPPacket(u_char, Packet*, bool)*

Most of the important code for a packet parser goes into the constructor. Usually you cast the incoming buffer pcPacket to a structure, which corresponds to the packet format you are parsing. You usually find definitions for those structures in the system header filese. Note that this has the disadvantage of creating a dependency on the system header files which may cause issues when porting Netsniff to other platforms. For ARP you find the structure definition in /usr/include/net/if_arp.h. The structure we are interested in is shown in Listing 3.

```
struct arphdr {
 u_short ar_hrd; /* format of hardware address */
 u_short ar_pro; /* format of protocol address */
 u_char ar_hln;  /* length of hardware address */
 u_char ar_pln;  /* length of protocol address */
 u_short ar_op;  /* one of: */
#ifdef COMMENT_ONLY
 u_char ar_sha[]; /* sender hardware address */
 u_char ar_spa[]; /* sender protocol address */
 u_char ar_tha[]; /* target hardware address */
 u_char ar_tpa[]; /* target protocol address */
#endif
};
```

Listing 3: ARP structure as defined in <net/if_arp.h>

string DumpFileName()

This method must return the filename of the file, you want Netsniff to write the log to. This will correspond to the ostream passed into the method Output described below.

void Output(ostream)

In this method you output the information parsed and stored by the packet's constructor. For the ARP case this is probably exactly the information in the packet, so you could make it look like this:

```
2004-09-29 10:42:46.814500 Headers(14) Payload(46)
  1 2048 6 4 1 0004dd38
a402 136.186.229.2 000000000000 136.186.229.117
```

Listing 4: example output in arp.log

bool OmmitOutputFromParent()

As obvious from Figure 1 Netsniff has a chain of protocol packets, which each has in term its own Output function. These are called in the order the packets are nested in each other, for instance Ethernet, IP, UDP output in that order. Sometimes for a particular protocol you want to prevent the immediate parent from outputting its information. If OmmitOutputFromParent() returns true, the Output function in the parent packet won't output it's information. In most cases you don't need to override this method, though.

C. Construct a sub packet from another packet

Each packet from a particular protocol has to be created somewhere in the chain of protocol packets. For instance if it is a packet protocol residing at the Data Link layer it would most probably be created from pcapdev.cpp, or if it is a protocol running on top of Ethernet, it would be created from the EthernetPacket class and thus be placed in the ethernet.cpp file. This is the case for ARP. Note, that ARP, to our knowledge, is used by the Ethernet/IP pair only, although the specification would allow for other link layers. This means that it's only placed in etherpacket.cpp. A snippet from etherpacket.cpp in Listing 5 shows where ARPPacket has to be added.

```
EtherPacket::EtherPacket(u_char *pcPacket,
                         Packet *pcParent,
                         bool bAnonymise)
  : Packet(pcParent)
{
 const struct ether_header *psEtherHdr
   = (const struct ether_header *) pcPacket;
 lHeaderLen = ETHER_HDR_LEN;
 switch (ntohs(psEtherHdr->ether_type))
 {
  case ETHERTYPE_IP:
   pcSubPacket=new IPPacket(pcPacket+lHeaderLen,
                            this,
                            bAnonymise);
   break;
  case ETHERTYPE_ARP:
   pcSubPacket=new ARPPacket(pcPacket+lHeaderLen,
                             this,
                             bAnonymise);
   break;
 }
}
```

Listing 5: Adding ARPPacket to EtherPacket (src/etherpacket.cpp)

VI. Adding a Stream-Based Application Layer Protocol

Typically we are concerned with processing an Application that runs over TCP. The existing netsniff code will automatically reconstruct the TCP Stream and extract TCP level information for us. This allows us to construct a parser that is concerned only with the application

layer information. Currently netsniff supports HTTP, FTP, HTTPS (over TLS), POP3, SMTP and IMAP4. This section describes how to extend netsniff to parse other application layer protocols.

The necessary tasks are:

- Subclass the APPParser class defined in appparser.h (Listing 6).
- Register the port numbers of your Application Layer protocol with the class factory APPParserFactory defined in appparser.h (Listing 7). This is done by calling the method registerAPPParserCreate defined in the APPParserFactory class.

```
class APPParser
{
 public:
  APPParser(bool bAnonymiseData,
            TCPStream* pTCPStream)
    : bAnonymise(bAnonymiseData);

  virtual ~APPParser();

  virtual bool isThisApp();

  virtual void ParseClient
  (std::basic_string<u_char> strData);

  virtual void ParseServer
  (std::basic_string<u_char> strData);

  virtual bool Parsed();

  virtual std::string DumpFileName();

  virtual void Output(std::ostream &osOutStream);

  static APPParser* create
  (bool bAnonymiseData, TCPStream* pTCPStream);

protected:
  bool bAnonymise;
};
```

Listing 6: APPParser definition (include/appparser.h)

```
class APPParserFactory
{
 public:
  typedef APPParser* (*APPParserCreate)
    (bool bAnonymous, TCPStream* pTCPStream);

  static APPParserFactory* Instance();

  static bool registerAPPParserCreate
  (u_short port,
   APPParserCreate creator);

  static APPParser* getAPPParser
  (u_short usPort,
   bool bAnonymous,
   TCPStream* pTCPStream);

 private:
  APPParserFactory();
  std::map<u_short, APPParserCreate> appparserMap;
};
```

Listing 7: APPParserFactory definition (include/appparser.h)

The virtual methods you should overload in your APPParser class are:

A. ParseClient

Since data from the TCP Stream is collected a portion at a time, we will never have the entire bit-stream to process. The underlying TCPStream class will reconstruct the bit-stream in the correct order and pass portions to your class in the ParseClient() and ParseServer() methods. Your implementation should parse this data block in restartable blocks for any relevant information and store this information for later output. Your constructor will be called with an Anonymisation flag which you should honour when either processing or outputting information. The data passed to this method is the bit-stream passed from the Client to the Server.

B. ParseServer

Should perform the same task as ParseClient() except that the bit-stream is that passed from the Server to the Client.

C. Parsed

Returns whether this TCP Stream is parsed or not, this should be overloaded to return true. The base class returns false to ensure that underlying packets are dumped to the not-parsed file.

D. DumpFileName

Returns the file name to dump all logged information too. As per a Packet-type application, netsniff will ensure the file is created in the correct directory. This should be overloaded based on the application being parsed. The base class ensures all TCP Stream information that is not specifically parsed is dumped to "tcpstream.txt"

E. Output

As per the Packet-type parsers, you should output any logged information to the provided output stream in this method. This method is automatically called when the TCP Stream has terminated.

F. Create

This is the function called by the APPParserFactory. You want to create a dynamically allocated object (with new) by calling the constructor that you have implemented and that has the same arguments and return it. Netsniff will worry about the deallocation of that object.

VII. Example of Stream-Based Application Layer Protocol

A. FTP a simple example

In order to clarify the concepts summarized above, we will outline how the FTP parser in Netsniff is implemented. FTP in the form most people use, is not very complicated. RFC959 [4] describes its basic version, but a lot of extensions have been added in the years since its writing. In case you're interested, [5] is a valuable summary of many extensions added to FTP.

We assume, that Netsniff monitors the control connection between server and client. This protocol is usually a client command followed by a server reply either complete or temporary. Which means, that the server and client

are taking turns. FTP can also open additional connections via the use of the PORT, PASV or EPSV commands. These TCP connections are established to transfer larger ammounts of data such as file transfers and directory listings. The data we collect in the FTP parser is listed in the output grammar shown in Listing 8. If we look at the ftp_log_line2 production, we see that this is used to associate the control connection with the data connections used. An example output is shown in Listing 9.

B. FTP parser implementation

Although the implementation of FTP is quite straight forward, it is not shown in this document, because it is still quite big. We recommend using a hash map to parse protocols that are textual, since this is not only more efficient when having many commands, but makes the code mostly automatic to generate.

```
ftp_output            ::= tcp_output
                          ftp_log_line1
                          ftp_log_line2
ftp_log_line1         ::= (ftp_username |
                          ftp_anonymized_user) SP
                          '"' ftp_osstring '"'
                          SP
                          ftp_fileoperations SP
                          ftp_error_fileoperations
                          SP
                          ftp_unimplemented SP
                          ftp_error_responses SP
                          ftp_unknown_cmds SP
                          ftp_anonymous SP
                          ftp_login CRLF
ftp_log_line2         ::= *( ("LIST"|"RETR"|"SEND")
                            SP
                            ftp_data_stream_port
                            CRLF
                           )
ftp_username          ::= STRING
ftp_anonymized_user   ::= STRING
ftp_osstring          ::= STRING
ftp_data_stream_port  ::= NUMBER
ftp_fileoperations    ::= NUMBER
ftp_error_fileops     ::= NUMBER
ftp_unimplemented     ::= NUMBER
ftp_error_responses   ::= NUMBER
ftp_unknown_cmds      ::= NUMBER
ftp_anonymous         ::= BOOL
ftp_login             ::= BOOL
```

Listing 8: Output grammar FTPParser

VIII. Anonymisation Tools

Netsniff is expected to be able to anonymise potentially sensitive information if it is executed with the correct flags. However, to allow for correlation between applications and sessions, we would like to be able to determine if usernames, email addresses and/or IP addresses are repeated. To help ensure this is the case, netsniff includes a series of classes that perform these functions. If you need to anonymise any of the given data types, you can do so using the provided classes.

A. IP Address Anonymisation

The IPAddressMap class anonymises addresses using a similar algorithm to that used by tcpdpriv running with the -A 50 flag. The class maintains a map of seen IP addresses

```
2004-11-16 14:37:41.561382 45.139418 127.0.0.1
:4499 127.0.0.1:21 140 1360 653 1100 1 50 36 0 0
Hops Client:,0,26,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0
Hops Server:,0,21,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0
RTT:,50,64,64,46,46,31
Jitter:,36,29,29,41,41,41
ukeller "UNIX Type: L8 Version: BSD-199506"
 2 1 0 1 1
LIST 64466
RETR 50831
SEND 54099
```

Listing 9: Example output from FTP

to anonymised IP addresses so that the same IP address will be mapped to the same anonymised IP address. The implementation also maintains network locality information between two IP addresses in their anonymised form. If a new IP address is seen, the implementation will allocate a new anonymised address for the mapping.

If your parser processes IP addresses in its log output, you should use IPAddressMap in the case of the anonymisation flag being set. To use IPAddressMap you can call:

```
IPPacket::AddressAnonymiser()\
 ->Map(in_addr_t ipAddress)
```

on the IP address that is to be anonymised, where ipAddress is a 32 bit integer representing an IP address. Also make sure to pass in the address in network byte-order.

B. Hashing of strings

Strings are anonymised using secure hashing [6]. Strings are anonymised with a random key, which is regenerated at each start of netsniff or with a key read from file, which is specified on the command-line. The output of the string anonymisation is a hex string which (nearly) uniquely represents the original input string. Comparison of similar input strings is not possible since the hash function has the required property of equally dispersing strings in the input-space into the output-space. A solution to this is to have a more intelligent function, which actually breaks down the string to sub strings which characterize the original string. For an URL such as:

```
http://www.mozilla.org/products/index.html
```

this could be something like:

```
http:// +
hash('www') + '.' +
hash('mozilla') + '.' +
hash('org') + '/' +
hash('products') + '/' +
hash('index.html')
```

For a simple user-name, for which we currently use string anonymisation the current scheme is certainly sufficient.

IX. Adding to the logfileparser

For each packet type or application layer protocol supported, Netsniff creates an output file, which is in text format. These files are usually kept with as little overhead in them as necessary. In order to create useful statistics on the data Netsniff collects, we decided to write a log file parser, that writes the data into a database. The back ends currently supported are Sqlite and Sqlite3 [7], it should be a small effort to add other more powerful database back ends when needed.

If Netsniff is extended with additional protocols, it is also necessary to adopt the database and the log file parser. The following subsections will describe how this is done for FTP.

A. Extending the Database

The database Netsniff uses was designed with the DBDesigner4 tool [8]. It is released under GPL and runs on Windows, Linux and the Linux emulation under FreeBSD. More information on how to install DBDesigner4 is provided in Appendix A. Its use is quite straight forward and very similar to other DBM tools.

If we go back to Listing 8, which describes the output grammar for FTP, we can identify how data should be split into tables. Figure 3 shows how the database schema has been extended for FTP. Note that the table FTPDataConnections corresponds to ftp_log_line2 and the tables FTPOS and FTP to ftp_log_line1. We see also, that an FTP control connection, to which tuples in the table FTP correspond, are uniquely defined by its underlying TCP stream and can therefore inherit their keys. This is called generalization. The table FTPDataConnections relates TCP streams to an FTP control connection. These streams are, as described above, used to transfer files or directory listings. Further the table FTPOS has been introduced to save space on assumption that the number of distinct operating system strings is inferior to the number of FTP connections.

Once one has designed the table and it matches the output generated from Netsniff, the create script can be exported using the export function in DBDesigner4. Because DBDesigner4 is optimized for MySQL the create script needs to be massaged a bit. For Sqlite the following modifications will do:

- **NO ACTION:** (when using ON DELETE or ON UPDATE) Sqlite does not support NO ACTION, which means that strings ON UPDATE NO ACTION and ON DELETE NO ACTION have to be removed.
- **NOT NULL:** since Sqlite doesn't support AUTO_INCREMENT, it works around it, by inserting a NULL value into a primary key defined as (SIGNED) INTEGER. This will generate an auto increment value. It has the drawback, that you can't specify a primary key as NOT NULL. Note that a primary key can never be NULL and so this check is redundant, but DBDesigner4 and other Design tools add them nevertheless, which means a bit more of Find/Replace for us.

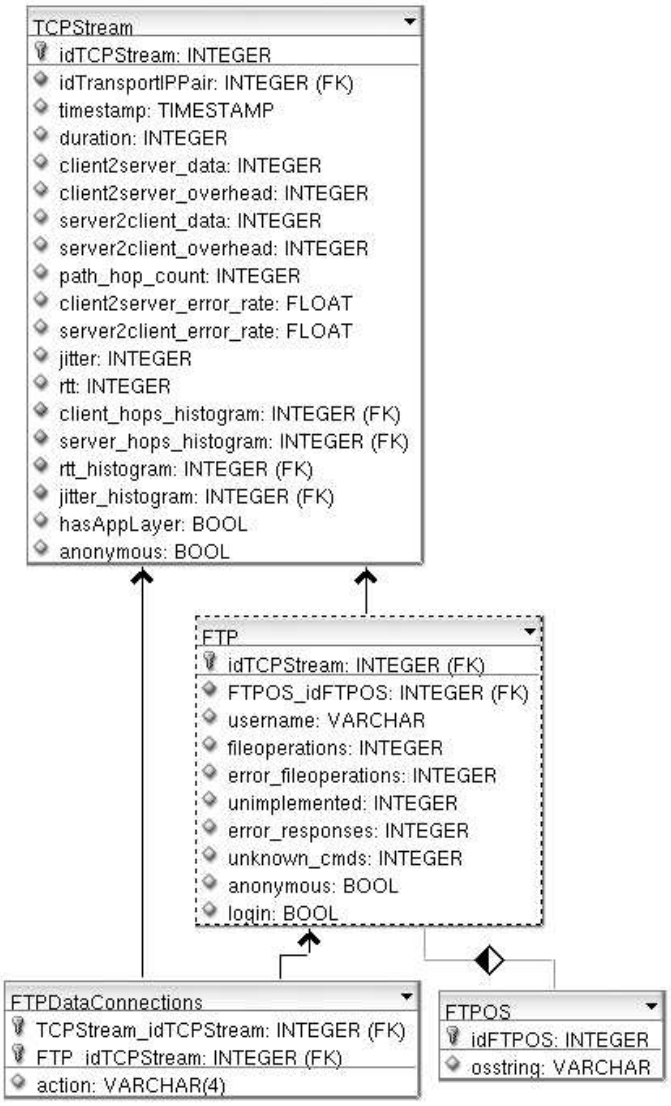

Figure 3: TCP/FTP SQL Tables

- **UNSIGNED:** in keys (see NOT NULL)
- **AUTO_INCREMENT:** (see NOT NULL)

The DBDesigner4 forum states, that in the next couple of months full Sqlite support will be added, which hopefully means a direct export to Sqlite.

Once the SQL create script has been massaged, it is recommended to test it with Sqlite's command line utility. On a terminal type:

```
sqlite test.db
```

And paste the script into the terminal. There must not be any errors when doing this with an empty database (passing a non existing file as argument to Sqlite). You can also pipe your database create script to Sqlite:

```
sqlite test.db < DBCreate.sql
```

If this hasn't produced errors the SQL commands have to be added to the sqlstrings.cpp file in the create_table string. Note that currently only Sqlite(3) is supported and therefore the SQL statements are hard coded in the source code.

B. Adding a log file parser

The actual parser is implemented by implementing one ore more classes extending from the Parser class and adding an entry function to the class ParserEntries. This process is described in the following paragraphs.

If we have a look at the output grammar specification from Listing 8 and the database schema in Figure 3, we can identify a simple way to create a parser. We create:

- A class for the FTPDataconnections, corresponding to ftp_output_line2.
- A class for FTP, which corresponds to ftp_output_line1
- A class for ftp_output, which encapsulates the above two classes.

A parser for the TCPStream table to process the tcp output is already implemented.

If we have a look at the output grammar specification from Listing 8 and the database schema in Figure 3, we can identify a simple way, to create a parser. We create a class for the FTPDataconnections, which corresponds to ftp_output_line2, a class for FTP, which corresponds to ftp_output_line1 and a class, which sort of encapsulates the two of them, which is ftp_output. A parser for the TCPStream table respectively the tcp_output is already implemented, so we can just call this functionality.

The interface to the three classes is shown in Listing 10. The entry point to our parser is in the ftp output class. This means that we have to add an entry to that function in the ParserEntries class. Listing 11 shows how this is done, defining a function creating the parser object, calling transact on it and adding an entry to a map (which maps filename to function). Entries are kept in topological order, meaning an entry that depends on another has to be placed after the one it depends on.

Now we have registered the entry to the parser and defined the classes representing the data, we need to implement the parser functionality. Many pieces of this functionality have already been built for other log file parsers and are available in the Lexer class from which Parser is a direct subclass. Listing 13 and 13 show ftp_output and ftp_log_line1 respectively. The implementation from Listing 13 looks very similar to its corresponding grammar in Listing 8.

The last step is to implement the toSql function in each subclass of Parser. In case of ftp_log_line1 this is quite simple, since all except osstring goes into the same table. Listing 14 shows a simplified version on how this is done.

The entry point to the toSql function in ftp_ output is the method transact, this can be seen in Listing 11. Transact calls ftp_output::toSql() which in turn calls the toSql() functions of ftp_log_line1 and ftp_log_line2 (Listing 15).

Having implemented all these steps, the parser can be tested by:

```
logfileparser -sqlite test.db ftp.log
```

Where ftp.log is a log file generated by Netsniff, when parsing FTP traffic and test.db is either a new or pre-existing Netsniff database. This will parse ftp.log and create a database in test.db. The database can then be queried in the Sqlite console, which is accessed by:

```
sqlite test.db
```

```
class ftp_log_line1 : public Parser
{
public:
  ftp_log_line1(Parser &parser);
  virtual long long toSql(long long pk);
  std::string username;
  std::string ftp_osstring;
  long long fileoperations;
  long long error_fileoperations;
  long long unimplemented;
  long long error_responses;
  long long unknown_commands;
  bool anonymous;
  bool login;
};

class ftp_log_line2 : public Parser
{
public:
  ftp_log_line2(Parser &parser);
  virtual long long toSql(long long pk);
  std::multimap<std::string, unsigned int>
                                connectionList;
};

class ftp_output : public Parser
{
public:
  ftp_output(Parser &parser);
  virtual ~ftp_output();
  virtual long long toSql();
  tcp_output * tpoutput;
  ftp_log_line1 * l1;
  ftp_log_line2 * l2;
};
```

Listing 10: Class definitions (statistic/include/ftp_output.hpp)

```
class ParserEntries
{
  static bool staticInit()
  {
....
      fmap.push_back(fpair_t("ftp.log", ftp));
....
  }
....
  static void ftp(Parser& parser) {
    ftp_output(parser).transact();
  }
};
```

Listing 11: Additions in class parser entries (statistic/include/parserentries.hpp)

```
ftp_output::ftp_output(Parser &parser)
 : Parser(parser)
{
  tpoutput = new tcp_output(parser);
  l1 = new ftp_log_line1(parser);
  l2 = new ftp_log_line2(parser);
  skipLine();
}
```

Listing 12: ftp_output (statistic/src/ftp_output.cpp)

```
ftp_log_line1::ftp_log_line1(Parser &parser)
  : Parser(parser)
{
 username             = getString();       getSP();
 osstring             = getQuotedString(); getSP();
 fileoperations       = getInteger();      getSP();
 error_fileoperations = getInteger();      getSP();
 unimplemented        = getInteger();      getSP();
 error_responses      = getInteger();      getSP();
 unknown_commands     = getInteger();      getSP();
 anonymous            = getInteger();      getSP();
 login                = getInteger();      getSP();
 skipLine();
}
```

Listing 13: ftp_log_line1 (statistic/src/ftp_output.cpp)

```
long long ftp_log_line1::toSql(long long pk)
{

 ....

  ostringstream qs2;
  qs2 << "insert into FTP values("
      << pk << ","
      << ospk << ","
      << "'" << username << "',"
      << fileoperations << ","
      << error_fileoperations << ","
      << unimplemented << ","
      << error_responses << ","
      << unknown_commands << ","
      << anonymous << ","
      << login << ");";
  Parser::toSql(qs2.str());
  return 0;
}
```

Listing 14: ftp_log_line1 toSql() (statistic/src/ftp_output.cpp)

```
long long ftp_output::toSql()
{
  long long tcp_key = tpoutput->toSql();
  l1->toSql(tcp_key);
  l2->toSql(tcp_key);
  return 0;
}
```

Listing 15: ftp_output toSql() (statistic/src/ftp_output.cpp)

X. Conclusion

In this technical report we described how to extend Netsniff with an additional packet level and application level parser. We further described how to adopt the log file parser tool and its database, in order to reflect the changes made in Netsniff. This document hopefully serves as a guide line for future improvements, additions and development on Netsniff.

References

[1] Keller U. and But J. "Netsniff - Design and Implementation Concepts". Technical Report 050204A, CAIA, February 2005. http://caia.swin.edu.au/reports/050204A/CAIA-TR050204A.pdf.

[2] Centre for Advanced Internet Architecture. "ICE^3 Inverted Capacity Extended Engineering Experiment", January 2005. http://caia.swin.edu.au/ice/.

[3] "TCPDUMP / PCAP", January 2005. http://www.tcpdump.org.

[4] J. Postel and J.K. Reynolds. "File Transfer Protocol". RFC 0959, IETF, October 1985. http://www.ietf.org/rfc/rfc0959.txt.

[5] Network Sorcery. "FTP, File Transfer Protocol", January 2005. http://www.networksorcery.com/enp/protocol/ftp.htm.

[6] OpenSSL Project. "HMAC". http://www.openssl.org/docs/crypto/hmac.html.

[7] Hipp, D. R. "SQLite", October 2004. http://www.sqlite.org.

[8] fabFORCE.net. "Fabulouse Force Database Tools, DBDesigner4", January 2005. http://www.fabforce.net/dbdesigner4/.

[9] Borland Software Corporation. "Kylix 3", January 2005. http://www.borland.com/kylix/.

[10] Brovienas, L.J. and Lopez-Cabanillas, P. "kylixlibs", January 2005. http://kylixlibs.sourceforge.net/.

XI. Appendix

A. Installing DBDesigner4 under FreeBSD

DBDesigner4 can be obtained from [8]. It works well on linux_base-8-8.0_4. Other versions of linux_base haven't been tested. DBDesigner4 is a Kylix [9] application and depends on the Kylix libraries. They can be obtained from [10]. You will probably only need libborqt. Copy it to /compat/linux, chroot to /compat/linux and install the package. For the Redhat based linux_base this looks like this:

```
chroot /compat/linux bash
rpm -Uhv --nodeps \
libborqt-6.9.0-1.i386.rpm
```

You can now unpack the DBDesigner4 tar ball obtained from [8] and start with ./DBDesigner4 from the directory.

B. Netsniff's directory structure

- **ice/netsniff** Netsniff's base directory.
- **ice/netsniff/src** contains the header and source file for Netsniff.
- **ice/netsniff/scripts** scripts, which are used in gateway boxes, installed with Netsniff.
- **ice/netsniff/statistic** contains the Netsniff log file parser.
- **ice/netsniff/statistic/DOC** documentation and database diagrams for the log file parser
- **ice/netsniff/statistic/src** source code for the log file parser.

C. Checking out Netsniff from CVS

Netsniff lives in the CVS repository on mordor. To check it out from CVS you need an account on mordor and the necessary rights on the /home/cvs/ice/. In order to checkout Netsniff type in a (t)csh shell:

```
setenv SSH_RSH ssh
cvs -d :ext:mordor co ice/netsniff
```

This will checkout the HEAD of Netsniff from CVS. If you are not doing development, you might want to consider to get the latest tagged version instead.