oTN-2008-08 Openlab Technical Note

CINBAD investigation of different packet filters

Natalia Basha
Milosz Marian Hulboj
Ryszard Erazm Jurga
22 August 2008
Version 1.3

Distribution:: **Public**

Introduction

This report presents my work for the 2008 openlab Summer Student Program at CERN. It is divided as follows. First, I present the usage of Berkeley Packet Filtering to filtering sFlow data. Next I report my findings in anomaly detection. And I end with a discussion of future work and conclusions.

Filtering is a deterministic selection of packets based on their content. The packet is selected if its content matches the specified mask. The selection decision is not biased by the packet position in the packet stream. This approach requires the packet content inspection, since packets have different formats and a fixed length mask is not applicable [3].

Network anomalies such as failures, attacks etc. are common-place in today's computer networks. Identifying, diagnosing and treating anomalies in a timely fashion are fundamental parts of day to day network operations. [5]

Berkeley packet filtering

The Berkeley Packet Filter (BPF) is the architecture for user-level packet capture. BPF provides a raw interface to data link layers in a protocol-independent fashion [1]. BPF uses a register-based 'filter machine' that can be implemented efficiently on today's register based CPUs. BPF uses a simple, non-shared buffer model made possible by today's larger address spaces [1].
BPF can be used to do virtually any matching. This part will be extended in 2.2 and 2.3.

The Packet Capture library (pcap)

The Packet Capture library (*pcap*) provides a high level interface to packet capture systems. *Pcap* consists of an application programming interface (API) for capturing the network traffic. Unix-like systems implement *pcap* in the *libpcap* library; Windows uses a port of *libpcap* known as *WinPcap*. Monitoring software may use *libpcap* or *WinPcap* to capture packets travelling over a network. The libraries also support saving captured packets to a file. Later on these files can be read by applications to perform the analysis.

Libpcap and *WinPcap* provide the packet-capture and filtering engines of many open-source and commercial network tools, including protocol analyzers (packet sniffers), network monitors, network intrusion detection systems, traffic-generators and network-testers.

In order to see how we can take advantage of the *libpcap* API, we wrote an application, that used the basic functionality of the library:

1. Captured packets from the network or read them from a file in tcpdump format
2. Printed basic information about packets:
   - packet number, timestamp, size.
3. Printed statistics about the number of packets:
   - captured (read), dropped, filtered at the end of the program or at the end of the file.
4. Applied a filter to the network traffic or a file and preserve original packet numbers

Below is a diagram that shows the structure of the application. (see code in appendices 6.1).

Diagram1. Structure of program.

Below there are definitions of functions that we used:

- *pcap_t *pcap_open_offline(const char *fname, char *errbuf)*
  the function is called to open a savefile for reading. *fname* specifies the name of the file to open. *errbuf* is used to return error or warning text. It returns a descriptor of opened file;

- *int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)*
  the function is used to compile the string *str* into a filter program; *program* is a pointer to a *bpf_program* struct and is filled in by *pcap_compile()**;** optimize* controls whether optimization on the resulting code is performed; *netmask* specifies the IPv4 netmask of the network on which packets are being captured; it is used only when checking for IPv4 broadcast addresses in the filter program;

- *int pcap_setfilter(pcap_t *p, struct bpf_program *fp)*
  the function is used to specify a filter program. *fcode* is a pointer to a *bpf_program* struct, usually the result of a call to ***pcap_compile()***. **-1** is returned on failure, in which case pcap_geterr() may be used to display the error text; **0** is returned on success;

- *int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)*
  the function is used to collect and process packets. It keeps reading packets until *cnt* packets are processed or an error occurs;
- *int pcap_stats(pcap_t *p, struct pcap_stat *ps)*
  The function returns 0 and fills in a *pcap_stat* structure. The values represent packet statistics from the start of the run to the time of the call. If there is an error or the underlying packet capture doesn't support packet statistics, -1 is returned and the error text can be obtained with *pcap_geterr()*

- *char *pcap_geterr(pcap_t *p)*
  the function returns the error text pertaining to the last *pcap* library error;

- *void pcap_close(pcap_t *p)*

The function closes the files associated with *fd* and deallocates resources.

In tcpdump savefile information is stored in *pcap_pkthdr* structure.

```
struct pcap_sf_pkthdr
{
        pcap_timeval ts;          /*!< time stamp */
        u_int32_t caplen;         /*!< length of portion present */
        u_int32_t len;            /*!< length this packet (off wire) */
};
```

where *pcap_timeval* is

```
struct pcap_timeval
{
        u_int32_t tv_sec;           /*!< seconds */
        u_int32_t tv_usec;          /*!< microseconds */
};
```

For making this code nicer we added a “wrapper” class based on design pattern RAII (Resource Acquisition Is Initialization). The technique combines acquisition and release of resources with initialization and initialization of objects. RAII is also used to ensure exception safety. RAII makes it possible to avoid resource leaks without extensive use of try/catch blocks and is widely used in the software industry. New class *pcap_file* has been attached in appendices 6.2.

Meta data filtering

For getting more information about each packet (for instance: the switch name and the interface number where we captured the packet) we used metadata files. These files were obtained by extracting appropriate information from the sFlow datagrams.

The sFlow is the standard that describes a mechanism to capture traffic data in switched or routed networks. It uses a sampling technology to collect statistics from the device. For this reason it is applicable to high speed networks (at gigabit speeds or higher) [7].

A flow sample consists of packet data and metadata. The packet data will typically contain a sampled header structure. If the agent is incapable of taking a sample of the whole packet, then either truncated version of the packet can be returned or special protocol dependent structures are being used (e.g. sampled IPv4 and sampled IPv6 structures) Each sample provides the input and output interface as well as the sampling rate for the given port. The metadata structure provides additional information about:

- sFlow datagram;
- flow header (interfaces, drops, sampling parameters, etc);

In the metadata we use the expanded datasource notation (meaning 2 u_int32_t are being used for interface information) [8].

We applied the filter to data and metadata at the same time.

When we filtered data, metadata should correspond to data. For example (see diagram 2), we filtered packets number 1, 3 and 7. After filtering, the first packet will be first, the third will be second, and the seventh – third. But if we want to add metadata, we should use packet numbers before filtering. Therefore in data structure we should have both of these numbers.

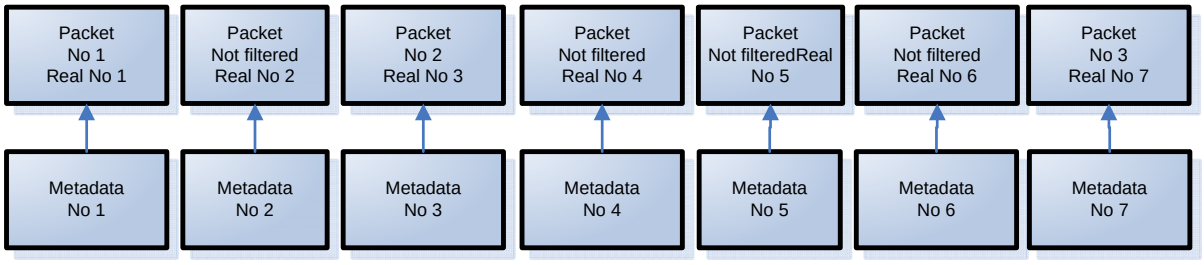


Diagram 2. Filtering data, add information from meta data.

The algorithm for filtering was:

1. Apply filter to data file
2. From each filtered packet get real packet number(number before filtering)
3. Using this packet number, find metadata in a metafile.

See code in appendices 6.3.

We printed information about

1. data:
   - Packet number after filtering(*reader->get_counter()*);
   - Real packet number(*reader->get_true_counter()*);
   - Time stamp(*hdr->ts.tv_sec and hdr->ts.tv_usec*);
   - Packet length(*hdr->len*);
2. metadata:
   - Time stamp (*meta.header.timestamp.tv_sec and meta.header.timestamp.tv_usec)*;
   - Agent IP(*meta.header.agent_ip*);
   - Datasorce index(*meta.ds_index*);
   - SNMP ifIndex of input interface. 0 if interface is not known(*meta.input*);
   - SNMP ifIndex of output interface (*meta.output*).

Below we attached an example output from the program (see Diagram 3).

```
Packet: #   39(142329) ts=1215700303.287007 (    60 bytes) Metadata:
ts=1215700303.287007 deviceIP:(172.21.32.195)  ds_index:         4 input:
1 output:          22

Packet: #   40(145179) ts=1215700304.479572 (    60 bytes) Metadata:
ts=1215700304.479572 deviceIP:(137.138.186.193)  ds_index:       131 input:
131 output:  1073741823

Packet: #   41(148769) ts=1215700305.991573 (    60 bytes) Metadata:
ts=1215700305.991573 deviceIP:(172.21.63.131)  ds_index:        26 input:
4 output:          15

Packet: #   42(151053) ts=1215700306.942924 (    60 bytes) Metadata:
ts=1215700306.942924 deviceIP:( 172.21.56.3)  ds_index:         2 input:
36 output:          18
```

```
Packet: #   43(154228) ts=1215700308.243805 (    60 bytes) Metadata:
ts=1215700308.243805 deviceIP:(172.21.32.195)  ds_index:        17 input:
1 output:          22

13:56.13.000 [1] printLogStats(): File: /home/nbasha/data/sample.raw.1215700243,
ps_recv: 155623, ps_drop: 0, ps_ifdrop: 3085809009, counter:43,
true_counter:155623
```

Diagram 3. Result of applying filer to data and getting corresponded meta data.

In the next step we wanted to find the way of combining filters for data and metadata. We wanted to take advantage of the same rules that *tcpdump* uses in *bpf* structure and construct a filter for metadata. In the next chapter we present rules for creating the filter for metadata.

The BSD Packet Filtering

We investigated the possibility of using Berkeley Packet Filter for the purpose of our project.

Berkeley filter program is an array of instructions, with all branches forwardly directed and terminated by a return instruction. Each instruction performs some action on the pseudo-machine state, which consists of an accumulator, index register, scratch memory store, and implicit program counter [2].

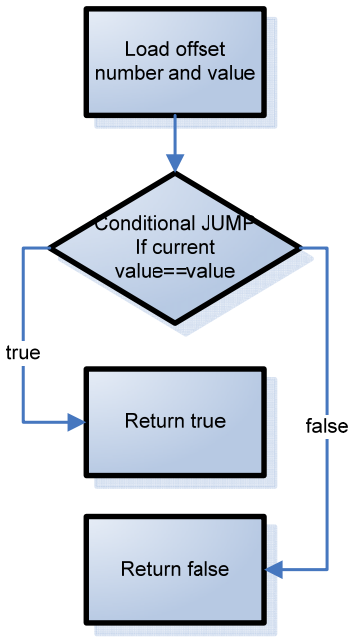
The following structure defines the instruction format:

```
struct bpf_insn
{
       u_short code;
       u_char  jt;
       u_char  jf;
       u_long k;
 };
```

k field is being used as an offset value, and the *jt* and *jf* fields are used as offsets by the branch instructions.

The opcodes are encoded in a semi-hierarchical fashion. There are eight classes of instructions: *BPF_LD, BPF_LDX, BPF_ST, BPF_STX, BPF_ALU, BPF_JMP, BPF_RET, and BPF_MISC*. Other various modes of operation can be obtained by using special modifier flags defined in <*net/bpf.h*>. [2] These instructions are flexible and can cover all necessary situations for data filtering. For example, in the load instruction data size can be word (32 bits), half word (16 bits) or byte. In the conditional jump, the instruction compares accumulator with a constant (value). This value can be equal, greater, greater or equal to accumulator.

The simple filter compares concrete value of some characteristic with model value. It consists of 4 instructions that you can see on diagram 4. In *JUMP* instruction *"TRUE=0"* means if *value==cur_value* than we go to the next line ("*RETURN TRUE*"). Else we should jump one line and go to line number 3 ("*RETURN FALSE*").




```
0 LOAD offset
1 (value==cur_value)? JUMP TRUE=0; FALSE=1
2 RETURN TRUE
3 RETURN FALSE
```

Diagram 4. Simple BSD packet filter.

We wrote a simple example of using the BPF instruction for the metadata (see code in appendices 6.4.1).

Then, we wrote an example of applying filters for both data and metadata. First we applied a data filter and if the match was positive, we applied a metadata filter (see code in appendices 6.4.3). Then, we wrote a complex filter for more than one field combined using 'or' or 'and' operators. It seemed that the only difference would be in jump instructions (see code in appendices 6.4.2).

We analysed whether it would be feasible to automate the filter building process for complex filter combined with 'or' and 'and' expressions.

We tried few methods to build this filter

1. Build the filter reusing *libpcap* functions

   We tried to imitate the process of building the filter that is similar to one in function *pcap_compile()* (function that converts string into the filter program-array with *bpf* instructions).

   We generated new statement (with offset number and offset value) using *pcap* functions:

   - *new_stmt()* – generate new statement with offset number;
   - *sappend(s, s2)* – append statement s2 to s;
   - *new_block()* – generate new block with jump instruction;
   - *gen_and(b1,b2)* – generate 'and' block;
   - *gen_or(b1,b2)* – generate 'or' block;

   After a series of testing we have realized that this method worked correctly only for a single field. If we tried to combine two or more fields it returned wrong result. For example, we wanted to filter packets that had *"agent_ip==10.38.136.23 and input==2 or output==48"*. We marked *agent_ip==10.38.136.23* as *A*, *input==21* as *B* and *output==48* as *C*. Then built a "truth table" for this expression (see Table 1)

A	B	C	Result
0	0	0	**0**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Table 1. Truth table for expression “agent_ip==10.38.136.23 and input==21 or output==48”.

After this we saw the result of the program. (the second truth table, see Table 2). Using Karnaugh maps we built expression that this filter returned. For this table the result was:

(**NOT**(A) C) +(**NOT**(B)A)

A	B	C	Result
0	0	0	**0**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Table 2. Truth table 2 – result of filtering meta data.

We thought, a problem was in building instructions for operators AND and OR.

2. Build the expression tree

Next idea was to build the expression tree.
We took the similar idea of building the abstraction tree as used in BPF Model:

In the tree model each node represents a boolean operation while the leaves represent test predicates on packet fields. The edges represent operator-operand relationships [1] (see diagram5, left picture).

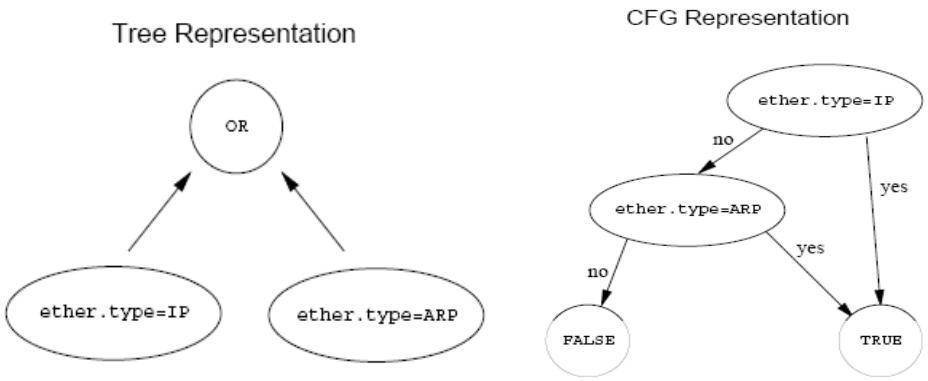


Diagram 5. Filter Function representation.
Left: expression tree; Right: Control Flow Graph.

We built this tree easily, but found some problems in building a filter:

- To start building the filter, we chose start-node. It could be:
  - any leaf;
  - a node whose children are leaves;
  - root;
  - deepest leaf, etc.;
- In the end we added return blocks. But we wanted to know the number of lines for each the block before we started building filter (unfortunately *bpf* virtual machine supports only positive offsets for jump instructions – thus we needed to know at which place the return blocks);
- If in the jump instruction we skipped some blocks, how we could know the number of line, where we wanted to jump to next?
- Sometimes we had loops in filter. It happened when we followed not the longest path between two nodes. To solve this problem, for example, we could add instructions one more time.

After these experiments with this tree we decided to modify its structure and convert it into a state diagram.

3. Convert the tree to the state diagram form

The CFG (Control Flow Graph) is a model where each node represents a packet field predicate while the edges represent control transfers. The right hand branch is traversed if the predicate is true, the left hand branch if false. There are two terminating leaves which represent true and false for the entire filter [1](see diagram5, right picture).

We started building tree from the root node. And for each node we had a pointer to *next jump if true* and *next jump if false* nodes. For example, we wanted to build filter: *"agent_ip==10.38.136.23 and input==21 or output==48"*

Algorithm was as follows.

- Build the expression tree (see diagram 6.1);
- Recursively traverse the tree (explore as far as possible along each branch before backtracing) and give number to every leaf. Leaf number= offset number of concrete node(See these numbers on diagram 6.1) ;
- Generate a state machine using expression tree(see diagram 6.2):
  - Start with "left most"(or "rightmost") child of the tree and move to the right(left) side of the plot;
  - According to the tree structure, an internal node represents an operation. Check parent's operation. If it is:

        - AND operation:
            - Generate TRUE branch that goes to brother leaf
            - Generate FALSE branch that goes to leaf in next operation
        - OR operation
            - Generate TRUE branch that goes to leaf in next operation
            - Generate FALSE branch that goes to brother leaf
    - Go to next leaf and repeat previous steps;
- Generate code using the state machine:
    - Start with root node;
    - For every node
        - Check that we didn't visit this node before (in order not to do useless calculations);
        - In recursion go down to subtree return *next_false* and *next_true* nodes ;
        - Build state with pointers to *next_false* and *next_true* nodes;
        - Generate filter code.
    - Add return blocks.
    -

You can see the result on diagram 6.3. Expression *"TRUE=0"* means that we should go to the next line (or jump 0 lines) and *"FALSE=2"* means jump through 2 lines and go to the third line.

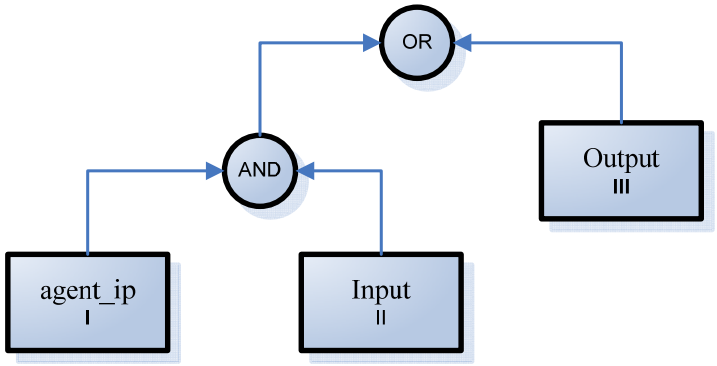


Diagram 6.1.Expression tree.

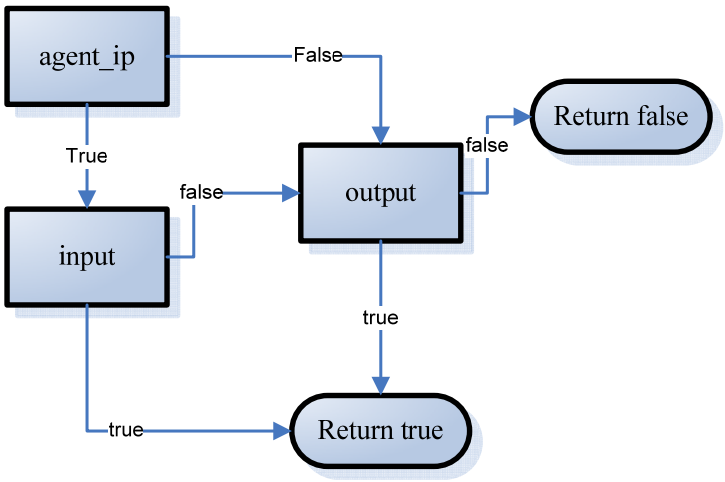


Diagram 6.2. State machine.

```
0 LOAD agent_ip.offset
1 JUMP agent_ip.value TRUE=0; FALSE=2
2 LOAD input.offset
3 JUMP input.value TRUE=2; FALSE=0
4 LOAD output.offset
5 JUMP output.value TRUE=0; FALSE=1
```

```
6 RETURN 1
7 RETURN 0
```

Diagram 6.3. filter code.

This algorithm passed all tests and is already being used in Cinbad tcpdump implementation.
The class hierarchy that we have developed for this state diagram is shown on diagram 7.

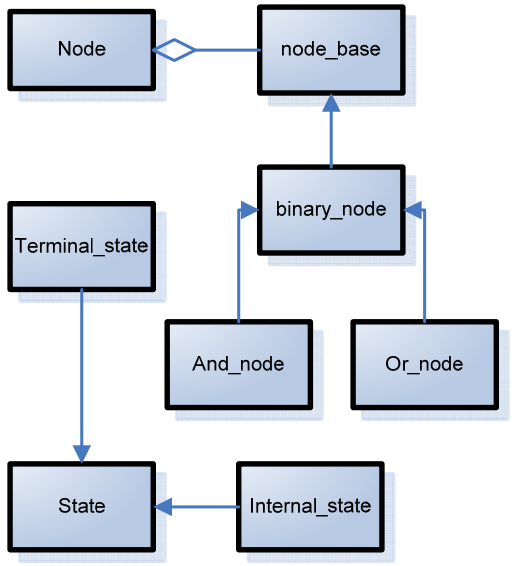


Diagram 7. Class hierarchy for filter generation.

Summary

During my work for Cinbad project I improved my knowledge of C++ and Boost libraries, studied the *pcap* packet capture library. I became familiar with mechanisms used for packet filtering and extended my knowledge about different network protocols. In addition to that, I have learned some methods of network anomalies detection.

My findings from Berkeley Packet Filter studies are already being used in Cinbad tcpdump implementation.

References

[1] Steven McCanney, Van Jacobsony :The BSD Packet Filter: A New Architecture for User-level Packet Capture. Lawrence Berkeley Laboratory, December 19, 1992,
[2] Steven McCanne: Man BPF. Lawrence Berkeley Laboratory, Summer 1990,
[3] Ryszard Erazm Jurga, Milosz Marian Hulboj: Packet Sampling for Network Monitoring. CERN — HP Procurve openlab project, 2007.
[4] IEEE 1044-1993: Standard Classification for Software Anomalies., page 1, The Institute of Electrical and Electronics Engineers, Inc., New York, USA, 1994, ISBN 1-55937-383-0
[5] Paul Barford, Jeffery Kline, David Plonka and Amos Ron: A Signal Analysis of Network Traffic Anomalies
[6] Ryszard Jurga: Networking overview. Lecture for CERN Openlab Summer Students2008, CERN 2008.
[7] Elisa Jasinska, sFlow - I can feel your traffic. AMS-IX use of sFlow, Amsterdam Internet Exchange, December 2006.
Institute of Standards and Technology, Gaithersburg, MD 20899-8930, 2007
[8] sFlow, http://sflow.org .

Appendices

Application one

```
#include <stdio.h>
#include <pcap.h>
int main(int argc, char *argv[])
{
pcap_t *fd;
char *dev, errbuf[PCAP_ERRBUF_SIZE];
struct bpf_program fcode;
struct pcap_stat stat_all ;

const char* filename="/home/nbasha/data/sample.raw.1215700243";
const char* filter="udp";

fd=pcap_open_offline( filename, ebuf);
NetMask=0xffffff;
      if (pcap_compile(fd, &fcode, filter, 1, NetMask)<0)
{
            fprintf(stderr,"\n Err in compile filter\n");
            return;
      }
      if(pcap_setfilter(fd, &fcode)<0)
{
            fprintf(stderr,"\n Err setfilter\n");
            return;
      }
printf("\nDETAILS ABOUT PACKETS \n");
pcap_loop(fd, 0, dispatcher_handler, NULL);
pcap_stats(fd, &stat_all);
printf("packets recieved:%s\n", stat_all.recv);
      printf("packets droped:%s\n",stat_all.ps_drop);
pcap_close(fd);
}
void dispatcher_handler(u_char *stuff,const struct pcap_pkthdr *header, const
u_char *pkt_data)
{
      printf("Packet #%d timestamp %d (%d bytes) \n",
            ++cnt,header->ts.tv_usec, header->len);
}
```

Application two

```
class  pcap_file
{
public:
      pcap_file(const char* filename, char *ebuf): fd(std::pcap_open_offline(
filename, ebuf))
      {
           if( !fd )
            throw std::runtime_error("input_file open failure");
      }
      ~pcap_file()
      {
            if( std::pcap_close(fd)!= 0 )
            {
            // deal with filesystem errors, fclose() may fail when flushing
      latest changes
            }
      }
      int compile_filter ( struct bpf_program *fp, char *str, int optimize,
      bpf_u_int32 *netmask)
      {
            if( std::pcap_compile(fd, fp, str, optimize, netmask) == -1 )
                  throw std::runtime_error("\n Err in compile filter\n");
      }
      int set_filter ( struct bpf_program *fp)
      {
           if( std::pcap_setfilter(fd, fp) == -1 )
            throw std::runtime_error("\n Err setfilter\n");
      }
      int loop ( int cnt, pcap_handler callback, u_char *user)
      {
            if( std::pcap_loop(fd, 0, callback, NULL) == -1 )
                  throw std::runtime_error(pcap_geterr(p));
     }
private:
std::pcap_t* fd ;
}
void dispatcher_handler(u_char *stuff,const struct pcap_pkthdr *header, const
u_char *pkt_data)
{
```

```
     printf("Packet #%d timestamp %d (%d bytes) \n", ++cnt,header->ts.tv_usec,
header->len);
}
```

Application three

```
#include <iostream>
#include <cstdio>
#include <stdexcept>
#include <boost/format.hpp>
#include <utils/fileutils.h>
#include <pcap/packet_source.h>
#include <pcap/pcap_include.h>
#include <sflow/sflow_packet.h>
#include <structure/ip/operations_ip4.h>

int main()
{
     const char* read_filename="/home/nbasha/data/sample.raw.1215700243";
     const char* metadata_name="/home/nbasha/data/sample.mraw.1215700243";
     const char* filter="udp";
     init_library_logs();
     pcap_pkthdr *hdr;
     const u_char* data;
     cern::pcap::packet_source_ptr reader =
     cern::pcap::packet_source_factory::get_file_reader(read_filename,filter);
     cern::utils::fileutils::file metadate(metadata_name,"r");
     cern::sflow::sflow_flow_metadata meta;
     while((reader->read(hdr,data)==1))
     {
      metadate.seek((reader->get_true_counter()-
      1)*sizeof(cern::sflow::sflow_flow_metadata));
      metadate.read(&meta,sizeof(cern::sflow::sflow_flow_metadata),1);
     //reinterpret_cast<*void>()


      std::cout << boost::format("Packet: #%|5d|(%|5d|) ts=%|d|.%|d| (%|6d|
     bytes) Metadata: ts=%|d|.%|d| deviceIP: (%|12d|) ds_index:%10d input:%12d
     output:%12d\n")
       % reader->get_counter()
       % reader->get_true_counter()
       % hdr->ts.tv_sec
       % hdr->ts.tv_usec
       % hdr->len
       % meta.header.timestamp.tv_sec
       % meta.header.timestamp.tv_usec
       % meta.header.agent_ip
       % meta.ds_index
```

```
        % meta.input
        % meta.output;
      }
}
```

Example of using bpf instruction for meta data

1. This filter load agent_ip (IPv4 address of the sampling agent) and compare it with model value 0x800 (concrete ip value in network byte order)

   ```
   struct bpf_insn insns[] =
   {
         BPF_STMT(BPF_LD|BPF_H|BPF_ABS, 12),
         BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 0x800, 0, 1),
         BPF_STMT(BPF_RET|BPF_K,1),
         BPF_STMT(BPF_RET|BPF_K, 0),
   };
   ```

2. This filter accepts only IP packets between host 128.3.112.15 and 28.3.112.35.

   ```
   struct bpf_insn insns[] =
   {
         BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
         BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 8),
         BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 26),
         BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 2),
         BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
         BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 3, 4),
         BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 0, 3),
         BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
         BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 1),
         BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
         BPF_STMT(BPF_RET+BPF_K, 0),
   };
   ```

3. This example shows how apply filter for both data and meta data. Data filter can be “icmp” and meta data “agent_ip==1.2.3.4”. We print information about result of applying both filters

   ```
   #include <iostream>
   #include <cstdio>
   #include <stdexcept>
   #include <boost/format.hpp>
   #include <utils/fileutils.h>
   #include <pcap/packet_source.h>
   #include <pcap/pcap_include.h>
   #include <sflow/sflow_packet.h>
   #include <structure/ip/operations_ip4.h>
   #include <sys/types.h>
   #include <sys/time.h>
   #include <sys/ioctl.h>
   extern static struct block *gen_uncond(int);

   int main()
   {
     const char* read_filename="/home/nbasha/data/sample.raw.1215700243";
     const char* metadata_name="/home/nbasha/data/sample.mraw.1215700243";
     struct bpf_insn insns[] =
     {
               BPF_STMT(BPF_LD|BPF_H|BPF_ABS, 12),
               BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 0x800, 0, 1),
               BPF_STMT(BPF_RET|BPF_K,1),
   ```

```
            BPF_STMT(BPF_RET|BPF_K, 0),
  };
  struct bpf_insn metafilter[] = {
    BPF_STMT(BPF_LD|BPF_W|BPF_ABS, 12),
    BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 0xac1520c3, 0, 1),
    BPF_STMT(BPF_RET|BPF_K, 0),
    BPF_STMT(BPF_RET|BPF_K, 1),
  };
  init_library_logs();
  pcap_pkthdr *hdr;
  const u_char* data;

  cern::pcap::packet_source_ptr reader =
cern::pcap::packet_source_factory::get_file_reader(read_filename);

  cern::utils::fileutils::file metadate(metadata_name,"r");

  cern::sflow::sflow_flow_metadata meta;
  while((reader->read(hdr,data)==1))
  {
    metadate.seek((reader->get_true_counter()-
1)*sizeof(cern::sflow::sflow_flow_metadata));

    metadate.read(&meta,sizeof(cern::sflow::sflow_flow_metadata),1);
//reinterpret_cast<*void>()\
    std::cout << boost::format("Meta filtered %d  and  filtered data %d
k=%d\n ")
      % bpf_filter(metafilter, reinterpret_cast<u_char*>(&meta),
      sizeof(cern::sflow::sflow_flow_metadata),
      sizeof(cern::sflow::sflow_flow_metadata))
                % bpf_filter(insns,const_cast<u_char*>(data), hdr->len,
hdr->caplen)
                % insns->k;

  }
}
```