A Tutorial Introduction to MAPI

July 16, 2009

1 Introduction

This document provides a tutorial introduction to the Monitoring Application Programming Interface (MAPI). It aims to give first-time users an overview of the basic functionality of MAPI for rapid development of advanced network monitoring applications.

1.1 What is MAPI?

MAPI is a highly *expressive* and flexible network monitoring API which enables users to clearly communicate their monitoring needs to the underlying traffic monitoring platform. MAPI has been designed as part of the SCAMPI network monitoring system[1]. Briefly, SCAMPI uses programmable hardware to perform computationally intensive operations, while the middleware offers support for running multiple monitoring applications simultaneously, and MAPI offers a standardized API to applications that is much more expressive than existing solutions. Furthermore, MAPI can also be used with commodity network interfaces or specialized network monitoring hardware (e.g., DAG cards[2]).

1.2 Why should I use MAPI?

MAPI builds on a generalized flow abstraction that allows users to tailor measurements to their own needs. MAPI elevates network flows to *first-class status*, enabling programmers to define and operate on flows in a flexible and efficient way. Where necessary and feasible, MAPI also allows the user to trigger custom processing routines not only on summarized data, but also on the packets themselves.

The expressiveness of MAPI enables the underlying monitoring system to make informed decisions in choosing the most efficient implementation, while providing a coherent interface on top of different lower-level elements, including intelligent switches, high-performance network processors, and special-purpose network interface cards. Thus, besides providing performance benefits, MAPI decouples the development of the monitoring applications from the environment on top of which they will be executed. Applications are written once, and are able to run on top of various monitoring environments without the need to alter or re-compile their code.

[1] `http://www.ist-scampi.org/`

[2] `http://www.endace.com/`

2 Basic Functions

This section gives an overview of the basic MAPI function calls. For more information about the available MAPI functions and their complete description please refer to `mapi(3)` and `mapi_stdflib(3)` man pages, also included in Appendices A and B, respectively.

2.1 Creating and Terminating Network Flows

Central to the operation of the MAPI is the action of creating a network flow:

```
int mapi_create_flow(char *dev)
```

This call creates a network flow and returns a flow descriptor `fd` that refers to it, or -1 on error. This network flow consists of all network packets which go through network device `dev`. The packets of this flow can be further reduced to those which satisfy an appropriate filter or other condition, as described in Section 2.2.

Besides creating a network flow, monitoring applications may also close the flow when they are no longer interested in monitoring:

```
int mapi_close_flow(int fd)
```

After closing a flow, all the structures that have been allocated for the flow are released. If the call fails, a value of -1 is returned.

2.2 Applying functions to Network Flows

Network flows allow users to treat packets that belong to different flows in different ways. For example, a user may be interested in *logging* all packets of a flow (e.g. to record an intrusion attempt), or in just *counting* the packets and their lengths (e.g. to count the bandwidth usage of an application), or in *sampling* the packets (e.g. to find the IP addresses that generate most of the traffic). The abstraction of the network flow allows the user to clearly communicate to the underlying monitoring system these different monitoring needs. To enable users to communicate these different requirements, MAPI enables users to associate functions with network flows:

```
int mapi_apply_function(int fd, char * funct, ...)
```

The above association applies the function `funct` to every packet of the network flow `fd`, and returns a relevant function descriptor `fid`. Depending on the applied function, additional arguments may be passed. Based on the header and payload of the packet, the function will perform some computation, and may optionally discard the packet.

MAPI provides several *predefined* functions that cover some standard monitoring needs through the MAPI Standard Library (`stdflib`). For example, applying the `BPF_FILTER` function with parameter `tcp and dst port 80` restricts the packets of the network flow denoted by the flow descriptor `fd` to the TCP packets destined to port `80`. Other example functions include: `PKT_COUNTER` which counts all packets in a flow, `SAMPLE`

which can be used to sample packets, etc. For a complete list of the available functions in `stdflib` and their description please refer to the `mapi_stdflib(3)` man page, also included in Appendix B.

Although these functions enable users to process packets, and compute the network traffic metrics they are interested in without receiving the packets in their own address space, they must somehow communicate their results to the interested users. For example, a user that will define that the function `PKT_COUNTER` will be applied to a flow, will be interested in reading what is the number of packets that have been counted so far. This can be achieved by allocating a small amount of memory for a data structure that contains the results. The functions that will be applied to the packets of the flow will write their results into this data structure. The user who is interested in reading the results will read the data structure using:

```
mapi_results_t * mapi_read_results(int fd, int fid)
```

The above call receives the results computed by the function denoted by the function descriptor `fid`, which has been applied to the network flow `fd`. It returns a pointer to the memory where the result's data structure is stored.

```
typedef struct mapi_results {
  void* res; //Pointer to function specific result data
  unsigned long long ts; //timestamp
  int size; //size of the result
} mapi_results_t;
```

The res field of this data structure is a pointer to the actual result. It also provides a 64-bit timestamp for the results, that is the number of microseconds since 00:00:00 UTC, January 1, 1970 (the number of seconds is the upper 32 bits). It refers to the time when the MAPI stub received the result from mapid. The memory for the results of each function is allocated from the stub once, during the instantiation of the flow.

2.3 Reading packets from a flow

Once a flow is established, the user will probably want to read packets from the flow. Packets can be read one-at-a-time using the following *blocking* call:

```
struct mapipkt * mapi_get_next_pkt(int fd, int fid)
```

which reads the next packet that belongs to flow `fd`. In order to read packets, the function `TO_BUFFER` (which returns the relevant `fid` parameter) must have previously been applied to the flow. If the user does not want to read one packet at-a-time and possibly block, (s)he may register a callback function that will be called when a packet to the specific flow is available:

```
int mapi_loop(int fd, int fid, int cnt, mapi_handler callback)
```

The above call makes sure that the handler `callback` will be invoked for each of the next `cnt` packets that will arrive in the flow `fd`.

3 Distributed Monitoring (DiMAPI)

The MAPI also offers capabilities for distributed passive network monitoring, using many remote and distributed monitoring sensors. This is achieved through an extension of the basic MAPI functionallity that we call DiMAPI. We describe in this section the basic functionality that DiMAPI offers to users in order to develop advanced distrbuted network monitoring applications.

3.1 What is DiMAPI?

DiMAPI is an Application Programming Interface for Distributed Network Monitoring that provides to users the same framework as MAPI. It enhances MAPI with the functionality of remote and distributed network monitoring. DiMAPI has been designed as part of the LOBSTER network monitoring system[3]. The applications that use DiMAPI can easily communicate with many remote monitoring sensors, properly configure them and retrieve results from every one.

3.2 When should I use DiMAPI?

DiMAPI offers the expressive and flexible framework that MAPI provides for applications that need to run remotely or use more than one monitoring sensors. All the applications that till now run locally (in the same computer where the monitoring interface is located), can also run remotely (the monitoring interface belongs to a remote host) in exactly the same way by using DiMAPI. Also, DiMAPI can be used for the development of applications that communicate with many distributed monitoring sensors, by using the notion of network scope.

3.3 Writing applications using DiMAPI

Writing applications using DiMAPI is done in exactly the same way as using MAPI. Firstly, using *mapi_create_flow* you can create network flows described from the flow descriptor that is returned. *mapi_create_flow* takes as argument the network scope that consists of all the monitoring sensors we want, including the monitoring interface for each one, e.g.

```
mapi_create_flow("host1:eth0, host1:eth1, host2:/dev/dag, host3:eth0");
```

For every network flow, you can apply the function you want using the *mapi_apply_function*. This function will be applied to all the remote monitoring sensors defined at the network scope. Before getting the results, the *mapi_connect* function have to be called. To get results from the remote sensors, the *mapi_read_results* function is used for the corresponding function that have been applied in exactly the same way as in local MAPI.

While in MAPI the *mapi_read_results* function returns a single instance of mapi_results_t struct, in DiMAPI it returns a vector of mapi_results_t structs, one for every remote monitoring sensor (int the same order that these sensors had been declared in *mapi_create_flow*. We remind that *mapi_read_results* returns the following data structure:

[3] `http://www.ist-lobster.org/`

```
typedef struct mapi_results {
  void* res; //Pointer to function specific result data
  unsigned long long ts; //timestamp
  int size; //size of the result
} mapi_results_t;
```

For flows associated with remote interfaces, the timestamp that is returned by *mapi_read_results* refers to the time when mapicommd received the result from its associated local mapid. mapiommd then just forwards this timestamp to the MAPI stub of the remote application. This avoids any interference with the network RTT. The necessary memory for these structs has been allocated, once per every function applied.

In order to know the number of the remote monitoring hosts that our network scope consists of, and so the number of the mapi_results_t instances that *mapi_read_results* will return, we use the *mapi_get_scope_size* function.

```
int mapi_get_scope_size(int fd)
```

This function takes as a single argument the flow descriptor and returns the number of the corresponding monitoring sensors. In case of a local MAPI application, it returns 1. In this way we provide full compatibility between MAPI and DiMAPI applications.

The other MAPI function that returns data from the monitoring sensors is the *mapi_get_next_pkt* function. In DiMAPI, the *mapi_get_next_pkt* returns packets from the monitoring sensors in a round-robin way, if it is possible. Finally, in order to terminate, cleanup and close a network flow the *mapi_close_flow* fucntion is used.

4 Management function calls

MAPI contains various management function calls that provides information about a running MAPI daemon (MAPId). These function calls provides information about available devices and libraries as well as active flows and functions applied to flows.

The available function calls are:

```
int mapi_get_device_info(int devicenumber, mapi_device_info_t* info);
int mapi_get_next_device_info(int devicenumber, mapi_device_info_t* info);
int mapi_get_library_info(int libnum, mapi_lib_info_t *info);
int mapi_get_next_library_info(int libnum, mapi_lib_info_t* info);
int mapi_get_libfunct_info(int libnum, mapi_libfunct_info_t *info);
int mapi_get_libfunct_next_info(int libnum, mapi_libfunct_info_t *info);
int mapi_get_flow_info(int fd, mapi_flow_info_t *info);
int mapi_get_next_flow_info(int fd, mapi_flow_info_t *info);
int mapi_get_function_info(int fd, int fid, mapi_function_info_t *info);
int mapi_get_next_function_info(int fd, int fid, mapi_function_info_t *info);
```

The mapi_get_?_info calls, retrieves information about one specific resource identified by an integer ID. The mapi_get_next_?_info calls returns information about the next resource with a hight ID than the one specified. This is used for looping through all available resources.

The following code is an example on how the management functions can be used for listing all available libraries and the functions in each library:

```
1  int id=-1,fid;
2  mapi_lib_info_t info;
3  mapi_libfunct_info_t finfo;
4
5  printf("ID\tName\t# functions\n");
6  while(mapi_get_next_library_info(id++,&info)==0) {
7          printf("%d\t%s\t%d\n",info.id,info.libname,info.functs);
8          fid=-1;
9          while(mapi_get_next_libfunct_info(info.id,fid++,&finfo)==0)
10               printf("\t\t%s(%s)\n",finfo.name,finfo.argdescr);
11 }
```

This code uses mapi_get_next_library_info to loop through all available libraries and print out the id and name of the library and the number of functions. It then uses the mapi_get_next_libfunct_info to loop through and print information about all the available functions in each library.

5 Installation

MAPI is available from `http://mapi.uninett.no` as a source code distribution. Currently MAPI has been tested with the Linux OS and supports the following monitoring interfaces:

- Commodity Ethernet NICs
- Endace DAG cards
- Napatech capture cards

It is recommended to download the latest public source code release. You can also checkout the latest development version from the subversion repository using the following command:

```
svn co --username public --password public \
        https://svn.testnett.uninett.no/mapi/trunk
```

5.1 Software Compilation

After you have unpacked the sources, you first need to configure the distribution using the supplied `configure` script.[4] The following configure options are available for enabling support for optional features:

[4] In case you have checked out the sources from the subversion repository, you need to first run the supplied `bootstrap.sh` script in order to create the generated autoconf files. It requires the latest versions of the `autoconf`, `automake`, and `libtool` tools.

`--enable-dimapi`	Support for remote and distributed monitoring (cf. Section 3)
`--enable-ssl`	Enable encryption of DiMAPI traffic
`--enable-dag`	Support for Endace DAG packet capture cards
`--enable-napatech`	Support for Napatech packet capture cards
MAPI function libraries	
`--enable-trackflib`	Build the traffic characterization library
`--enable-anonflib`	Build the traffic anonymization library
`--enable-ipfixflib`	Build the NetFlow export library
`--enable-extraflib`	Build the Extra MAPI function library
Miscellaneous options	
`--enable-funcstats`	Enable function statistics. This option enables packet counters for each applied function

Follow these steps to compile and install the software:

```
./configure
make
make install
```

The default installation prefix is `/usr/local` and can be changed with the `--prefix` configure parameter. All files are installed in appropriate directories under the prefix path. For example, binaries are installed in `<prefix>/sbin`.

5.1.1 Library path

The `libmapi` library is installed by default into `/usr/local/lib`. Some Linux distributions do not scan this directory as part of the default library path, so this can cause problems to programs linked with the shared version of `libmapi`.

To resolve this issue, you can add `/usr/local/lib` to the default system library path by adding the line `/usr/local/lib` into `/etc/ld.so.conf`. After saving it, run `ldconfig` to update the system library cache.

Another possible option, however not recommended, is to set the environment variable `LD_LIBRARY_PATH` as follows:

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

5.2 Monitoring Sensor Configuration

In order to setup a monitoring sensor, the MAPI daemon (*mapid*) has to be configured and run to the monitoring machine. *mapid* is configured via the *mapi.conf* configuration file, located in the installation directory (usually */usr/local/etc/mapi/mapi.conf*). In this file we can configure the network interfaces that can provide *mapid* with network packets, their corresponding MAPI drivers, the MAPI function libraries that we want to support in this *mapid*, the libraries and drivers path and other. A typical example of this file is given below:

```
libpath=/usr/local/share/mapi
drvpath=/usr/local/share/mapi

libs=stdflib.so:extraflib.so

dimapi_port=2233
syslog_level=1
logfile=

[driver]
device=eth0
driver=mapinicdrv.so

[driver]
device=eth1
driver=mapinicdrv.so

[driver]
device=lo
driver=mapinicdrv.so
description=This is a driver for local capture

[format]
format=MFF_PCAP
driver=mapinicdrv.so
description=Offline pcap-capture
```

For DiMAPI, the MAPI communication daemon (*mapicommd*) must also run to the monitoring machine. *mapicommd* is responsible for accepting requests for all the MAPI calls from remote hosts, forward them to the local *mapid* and return the answers and results from the local *mapid* back to the remote application. *mapicommd* uses TCP sockets with optional SSL encryption for the communication. The port number that *mapicommd* uses to listen for incoming connections is defined in *mapi.conf* located in the installation directory.

Both mapi deamons, *mapid* and *mapicommd*, support loggism mechanism to files and to syslog. For logging to file, the filename is defined in *mapi.conf* in the `logfile` field. In order to log messages to syslog, mapid and/or mapicommd should be executed with the flag -s (mapid -s and/or mapicommd -s). Two different types of messages are sent to syslog, debug messages and general information messages. *mapi.conf* the syslog level can be configured:

- **syslog_level=0**: Log only general information and not debugging messages.
- **syslog_level=1**: Log general information plus debugging information. Debug messages are printed to stdout and syslog.

- **syslog_level=2**: Log general information plus debugging information. Debug messages are printed only to syslog.

The default level is 2.

5.3 Compiling new MAPI applications

Any user is able to write its own MAPI applications and programs, using the basic functions described above and the several function libraries that is also provided. In order to compile a MAPI program, the flag *-lmapi* should be used. For example:

```
gcc my_mapi_program.c -o my_mapi_program -lmapi
```

6 Examples

The following sections present some example programs that introduce the concept of the network flow as defined in MAPI, and demonstrate the ease of programming simple applications that perform complex monitoring operations using MAPI.

6.1 Getting Started: Simple Packet Count

This first simple program demonstrates the basic steps that must be taken in order to create and use a network flow. In this example, a network flow is used for counting the number of packets destined to a web server in a time period of 10 seconds.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <mapi.h>
5
6  int main() {
7
8      int fd, fid;
9      mapi_results_t *result;
10
11     /* create a flow using the eth0 interface */
12     fd = mapi_create_flow("eth0");
13     if (fd < 0) {
14         printf("Could not create flow\n");
15         exit(EXIT_FAILURE);
16     }
17
18     /* keep only the packets directed to the web server */
19     mapi_apply_function(fd, "BPF_FILTER", "tcp and dst port 80");
20
21     /* and just count them */
22     fid = mapi_apply_function(fd, "PKT_COUNTER");
```

```
23
24      /* connect to the flow */
25      if(mapi_connect(fd) < 0) {
26          printf("Could not connect to flow %d\n", fd);
27          exit(EXIT_FAILURE);
28      }
29
30      sleep(10);
31
32      /* read the results of the applied PKT_COUNTER function */
33      result = (mapi_results_t *)mapi_read_results(fd, fid);
34      printf("pkts: %llu\n", *((unsigned long long*)result->res) );
35
36      mapi_close_flow(fd);
37      return 0;
38  }
```

The flow of the code is as follows: We begin by creating a network flow (line 12) that will receive the packets we are interested in. We specify that we are going to use the `eth0` network interface for monitoring the traffic. For a different monitoring adapter we would use something like `/dev/scampi/0` for a Scampi adapter, or `/dev/dag0` for a DAG card, depending on the configuration. We store the returned flow descriptor in the variable `fd` for future reference to the flow.

In the next step we restrict the packets of the newly created flow to those destined to our web server by applying the function `BPF_FILTER` (line 19) using the filter `tcp and dst port 80`. The filtering expression is written using the `tcpdump(8)` syntax. Since we are interested in just counting the packets, we also apply the `PKT_COUNTER` function (line 22). In order to later read the results of that function, we store the returned function descriptor in `fid`.

The final step is to start the operation of the network flow by connecting to it (line 25). The call to `mapi_connect()` actually activates the flow inside the MAPI daemon (`mapid`), which then starts processing the monitored traffic according to the specifications of the flow. In our case, it just keeps a count of the packets that match the filtering condition.

After 10 seconds, we read the packet count by passing the relevant flow descriptor `fd` and function descriptor `fid` to `mapi_read_results()` (line 33). Our work is done, so we close the network flow in order to free the resources allocated in `mapid` (line 36).

6.2 Link Utilization

Our next example presents an application that periodically reports the utilization of a network link. It uses two network flows to separate the incoming from the outgoing traffic, and demonstrates how to retrieve the results of an applied function by reference.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <signal.h>
```

```
5  #include <mapi.h>
6
7  static void terminate();
8  int in_fd, out_fd;
9
10 int main() {
11
12     int in_fid, out_fid;
13     mapi_results_t *result1, *result2;
14     unsigned long long *in_cnt, *out_cnt;
15     unsigned long long in_prev=0, out_prev=0;
16
17     signal(SIGINT, terminate);
18     signal(SIGQUIT, terminate);
19     signal(SIGTERM, terminate);
20
21     /* create two flows, one for each traffic direction */
22     in_fd = mapi_create_flow("eth0");
23     out_fd = mapi_create_flow("eth0");
24     if ((in_fd < 0) || (out_fd < 0)) {
25         printf("Could not create flow\n");
26         exit(EXIT_FAILURE);
27     }
28
29     /* separate incoming from outgoing packets */
30     mapi_apply_function(in_fd, "BPF_FILTER",
31             "dst host 139.91.145.84");
32     mapi_apply_function(out_fd, "BPF_FILTER",
33             "src host 139.91.145.84");
34
35     /* count the bytes of each flow */
36     in_fid = mapi_apply_function(in_fd, "BYTE_COUNTER");
37     out_fid = mapi_apply_function(out_fd, "BYTE_COUNTER");
38
39     /* connect to the flows */
40     if(mapi_connect(in_fd) < 0) {
41         printf("Could not connect to flow %d\n", in_fd);
42         exit(EXIT_FAILURE);
43     }
44     if(mapi_connect(out_fd) < 0) {
45         printf("Could not connect to flow %d\n", out_fd);
46         exit(EXIT_FAILURE);
47     }
48
49     while(1) {      /* forever, report the load */
50
51         sleep(1);
52
53         result1 = mapi_read_results(
```

```
54              in_fd, in_fid);
55          result2 = mapi_read_results(
56              out_fd, out_fid);
57          in_cnt = result1->res;
58          out_cnt = result2->res;
59
60          printf("incoming: %.2f Mbit/s (%llu bytes)\n",
61                  (*in_cnt-in_prev)*8/1000000.0, (*in_cnt-in_prev));
62          printf("outgoing: %.2f Mbit/s (%llu bytes)\n\n",
63                  (*out_cnt-out_prev)*8/1000000.0, (*out_cnt-out_prev));
64
65          in_prev = *in_cnt;
66          out_prev = *out_cnt;
67      }
68
69      return 0;
70  }
71
72  void terminate() {
73      mapi_close_flow(in_fd);
74      mapi_close_flow(out_fd);
75      exit(EXIT_SUCCESS);
76  }
```

The basic initial steps are similar to those in the previous example, with the main difference of manipulating two network flows instead of one. We begin by creating two network flows with flow descriptors `in_fd` and `out_fd` (lines 22 and 23) for the incoming and outgoing traffic, respectively, and then we apply the filters that will differentiate the traffic captured by each flow (lines 30– 33). In our case, we monitor the link that connects the host 139.91.145.84 to the Internet. All incoming packets will then have 139.91.145.84 as destination address, while all outgoing packets will have this IP as source address. In case that we would monitor a link that connects a whole subnet to the Internet, the host in the filtering conditions should be replaced by that subnet. For instance, for the subnet 139.91/16, we would define the filter `dst net 139.91.0.0` for the incoming traffic.

Since we are interested in counting the amount of traffic passing through the monitored link, we apply the `BYTE_COUNTER` function to both flows (lines 36 and 37), and save the relevant function descriptors in `in_fid` and `out_fid` for future reference.

After activating the flows (lines 40–47), we enter the main program loop, which periodically calls the `mapi_read_results()` for each flow (lines 53–56) and prints the incoming and outgoing traffic in Mbit/s, and the number of bytes seen in each one second interval (lines 60–63). In each iteration, the current value of each `BYTE_COUNTER` function result is retrieved by dereferencing `in_cnt` and `out_cnt`.

In order to ensure a graceful termination of the program, we have initially registered the signals `SIGINT`, `SIGTERM`, and `SIGQUIT` with the function `terminate()` (lines 16–18), which closes the two flows and terminates the process.

6.3 Worm Detection

This example demonstrates how MAPI can be used for the detection of an Internet worm—a rather complicated task that requires deep packet inspection. The simplified application presented here constantly inspects the monitored traffic and prints any packets that match the *signature* of the Slapper worm.

A signature describes an intrusion threat by matching characteristic parts of the attack packet(s) against the packets of the traffic stream. Such signatures are commonly used in Network Intrusion Detection Systems (NIDSes), which constantly examine the network traffic and determine whether any signatures indicating intrusion attempts are matched. For example, a packet directed to port 80 that contains the string `/bin/perl.exe` is probably an indication of a malicious user attacking a web server. This attack can be detected by a signature which checks the destination port number of each captured packet, and defines a string search for `/bin/perl.exe` in the packet payload.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <signal.h>
5  #include <net/ethernet.h>
6  #include <netinet/ip.h>
7  /* inet_ntoa() */
8  #include <sys/socket.h>
9  #include <netinet/in.h>
10 #include <arpa/inet.h>
11
12 #include <mapi.h>
13
14 static void terminate();
15 void print_IP_pkt(struct mapipkt *pkt);
16
17 int fd;
18
19 int main() {
20
21     int fid;
22     struct mapipkt *pkt;
23
24     signal(SIGINT, terminate);
25     signal(SIGQUIT, terminate);
26     signal(SIGTERM, terminate);
27
28     /* create a flow using the eth0 interface */
29     fd = mapi_create_flow("eth0");
30     if (fd < 0) {
31         printf("Could not create flow\n");
32         exit(EXIT_FAILURE);
33     }
```

```
34
35      /* the bpf part of the signature */
36      mapi_apply_function(fd, "BPF_FILTER",
37              "udp and src port 2002 and dst net 139.91.23 and dst port 80");
38
39      /* the content search part of the signature */
40      mapi_apply_function(fd, "STR_SEARCH",
41              "|00 00|E|00 00|E|00 00|@|00|", 0, 100);
42
43      /* must use TO_BUFFER in order to read full packet records */
44      fid = mapi_apply_function(fd, "TO_BUFFER");
45
46      /* connect to the flow */
47      if(mapi_connect(fd) < 0) {
48          printf("Could not connect to flow %d\n", fd);
49          exit(EXIT_FAILURE);
50      }
51
52      while(1) {      /* forever, wait for matching packets */
53
54          pkt = mapi_get_next_pkt(fd, fid);
55          printf("\nSlapper worm packet!\n");
56          print_IP_pkt(pkt);
57      }
58
59      return 0;
60  }
61
62  void terminate() {
63      mapi_close_flow(fd);
64      exit(EXIT_SUCCESS);
65  }
```

In the same fashion as with the previous examples, the program starts by creating a network flow using the `eth0` network interface (line 29). We then configure the network flow according to the worm signature (lines 36–41). For the identification of the Slapper worm, we use the following signature taken from the default ruleset of the popular Snort Intrusion Detection System[5]:

```
alert udp $EXTERNAL_NET 2002 -> $HTTP_SERVERS $HTTP_PORTS
(msg:"MISC slapper worm admin traffic";
content:"|00 00|E|00 00|E|00 00|@|00|"; depth:100;
reference:url,isc.incidents.org/analysis.html?id=167;
reference:url,www.cert.org/advisories/CA-2002-27.html;
classtype:trojan-activity; sid:1889; rev:5;)
```

We presume that `$EXTERNAL_NET` is set to `any` IP address, `$HTTP_SERVERS` is set to the subnet 139.91.23, and `$HTTP_PORTS` is set to 80. Given these assumptions, packets

[5] `http://www.snort.org/`

that match this signature can also be returned by a network flow, after the application of the appropriate MAPI functions, as follows:

- the condition `$EXTERNAL_NET 2002 -> $HTTP_SERVERS $HTTP_PORTS` is fulfilled by applying the `BPF_FILTER` function with the filter `udp and src port 2002 and dst net 139.91.23 and dst port 80` (line 36).
- the content search condition `content:"|00 00|E|00 00|E|00 00|@|00|"; depth:100;` is fulfilled by applying the `STR_SEARCH` function with the same `content` and `depth` parameters, and `0` for the `offset` parameter (line 40).

In order to print specific information about each attack packet, we have to receive the full records of the matching packets to the address space of the application. This is accomplished by applying the function `TO_BUFFER` (line 43), which instructs `mapid` to store the packets that match the conditions of the flow into a shared memory segment. The application can then retrieve the stored records using `mapi_get_next_pkt()`.

In the main execution loop, the application blocks into `mapi_get_next_pkt()` (line 54) until a matching packet is available. When such a packet is captured, the application prints its source and destination MAC and IP addresses by calling `print_IP_pkt()`. The listing of the `print_IP_pkt()` function is included in Appendix G. Since the application has access to the full packet record, `print_IP_pkt()` can be altered as needed to print any other part of the packet, even the whole packet payload.

6.4 Using DiMAPI

This is a simple application that demonstrates the use of DiMAPI for distributed network monitoring. In this example we just count all the web packets in every monitoring sensor.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <unistd.h>
5  #include <mapi.h>
6
7  static void terminate();
8  int fd;
9
10 int main() {
11
12   int fid;
13   mapi_results_t *dres;
14   unsigned long long *count, total_count=0;
15   int i, loop;
16   int number_of_sensors;
17
18   signal(SIGINT, terminate);
19   signal(SIGQUIT, terminate);
20   signal(SIGTERM, terminate);
```

```
21
22    /* create a flow using a scope of three monitoring sensors */
23    fd = mapi_create_flow("sensor.uninett.no:/dev/dag0,
24                  mon1.ics.forth.gr:eth0, 123.45.6.7:eth2");
25    if (fd < 0) {
26      printf("Could not create flow\n");
27      exit(EXIT_FAILURE);
28    }
29
30    /* keep only the web packets */
31    if (mapi_apply_function(fd, "BPF_FILTER", "tcp and port 80") < 0) {
32        printf("Could not apply BPF_FILTER function\n");
33        exit(EXIT_FAILURE);
34    }
35
36    /* count them in every monitoring sensor */
37    fid = mapi_apply_function(fd, "PKT_COUNTER");
38    if (fid < 0) {
39      printf("Could not apply PKT_COUNTER function\n");
40      exit(EXIT_FAILURE);
41    }
42
43    /* connect to the flow */
44    if(mapi_connect(fd) < 0) {
45      printf("Could not connect to flow %d\n", fd);
46      exit(EXIT_FAILURE);
47    }
48
49    /* get the number of the monitoring sensors */
50    number_of_sensors = mapi_get_scope_size(fd);
51
52    /* read the results of the applied PKT_COUNTER function from all
53       hosts every 1 sec */
54    while(loop--){
55      sleep(1);
56
57      dres = mapi_read_results(fd, fid);
58
59      for (i=0; i<number_of_sensors; i++) {
60        count = (unsigned long long*) dres[i].res;
61        printf("web pkts in host %d: %llu\n",i, *count);
62        total_count += *count;
63      }
64      printf("Total web packets: %llu\n",total_count);
65    }
66
67    return 0;
68  }
69
```

```
70 void terminate() {
71   mapi_close_flow(fd);
72   exit(EXIT_SUCCESS);
73 }
```

We create a network flow with a scope of three monitoring sensors (line 23). Then we apply to this network flow the functions *BPF_FILTER* and *PKT_COUNTER*. These functions will be applied in all monitoring sensors we declared. Then we connect to this flow and start receiving results from *PKT_COUNTER* every one second. The *mapi_read_results* function returns a *mapi_results_t* element. The *mapi_get_scope_size* function gives up the number of the monitoring hosts that should give results, one *mapi_results_t* instance (in a table) per every monitoring sensor (in our example this will be equal to three). The actual result of the *PKR_COUNTER* function for the monitoring sensor *i* is retrieved from *dres[i].res* field. Finally, we close the network flow in order to free all the resources allocated in every monitoring sensor. For this example, MAPI should be configured with the *–enable-dimapi* configuration option for DiMAPI support.

6.5 Using Anonymization

This is a simple application that shows some basic anonymization features of MAPI.

```
1  #include <stdio.h>
2  #include <mapi.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <net/ethernet.h>
6  #include <netinet/ip.h>
7  #include <sys/socket.h>
8  #include <netinet/in.h>
9  #include <arpa/inet.h>
10
11 void print_IP_pkt(struct mapipkt *rec);
12
13 int main(int argc, char *argv[]) {
14
15         int fd;
16         int fid;
17         int connect_status;
18         struct mapipkt *pkt;
19
20         fd=mapi_create_flow("eth0");
21         if(fd==-1) {
22             printf("Flow cannot be created. Exiting..\n");
23             exit(-1);
24         }
25
26         //Anonymization of TCP packets
27
```

```
28          mapi_apply_function(fd,"BPF_FILTER","tcp");
29
30          //map IP addresses to sequential integers (1-to-1 mapping)
31          mapi_apply_function(fd,"ANONYMIZE", "IP,SRC_IP,MAP");
32          mapi_apply_function(fd,"ANONYMIZE", "IP,DST_IP,MAP");
33
34          //replace with zero, tcp and ip options
35          mapi_apply_function(fd,"ANONYMIZE", "IP,OPTIONS,ZERO");
36          mapi_apply_function(fd,"ANONYMIZE", "TCP,TCP_OPTIONS,ZERO");
37
38          //remove payload
39          mapi_apply_function(fd,"ANONYMIZE", "TCP,PAYLOAD,STRIP,0");
40          //checksum fix in IP fixes checksums in TCP and UDP as well
41          mapi_apply_function(fd,"ANONYMIZE", "IP,CHECKSUM,CHECKSUM_ADJUST");
42          fid = mapi_apply_function(fd, "TO_BUFFER");
43
44          /* connect to the flow */
45          connect_status = mapi_connect(fd);
46          if(connect_status < 0) {
47                  printf("Connect failed");
48                  exit(0);
49          }
50
51          while(1) {      /* forever, wait for matching packets */
52
53                  pkt = mapi_get_next_pkt(fd, fid);
54                  printf("\nAnonymized tcp packet captured!\n");
55                  print_IP_pkt(pkt);
56          }
57
58          return 0;
59  }
```

In the above example, we create a network flow that captures only tcp packets. Then we apply anonymization on IP addresses, TCP/IP options, TCP payload and finally we fix TCP/IP checksums. A complete list of protocols and anonymization functions supported is provided in the `anonflib` man page, included in Appendix F. The listing of the `print_IP_pkt()` function is included in Appendix G. For this example, the *anonflib* library should be used. So, MAPI should be configured with *–enable-anonflib*.

A MAPI man page

B MAPI `stdflib` man page

C MAPI `dagflib` man page

D MAPI `extraflib` man page

E MAPI `trackflib` man page

F MAPI `anonflib` man page

G `print_IP_pkt()` listing

```
1  void print_IP_pkt(struct mapipkt *rec) {
2
3      int i;
4      unsigned char *p;
5      struct ether_header *eth;
6      struct iphdr *iph;
7
8      p = &(rec->pkt);
9      eth = (struct ether_header *)p;
10
11     /* print MAC addresses */
12     for(i=0; i<ETH_ALEN; i++) {
13         printf("%.2X", eth->ether_shost[i]);
14         if(i != 5) printf(":");
15     }
16     printf(" -> ");
17     for(i=0; i<ETH_ALEN; i++){
18         printf("%.2X", eth->ether_dhost[i]);
19         if(i != 5) printf(":");
20     }
21
22     /* make sure that this is indeed an IP packet */
23     if (ntohs(eth->ether_type) != ETHERTYPE_IP) {
24         printf("print_IP_pkt(): Not an IP packet\n");
25         return;
26     }
27
28     /* lay the IP header struct over the packet data */
29     iph = (struct iphdr *)(p + ETH_HLEN);
30
31     printf("\n%s -> ", inet_ntoa(*(struct in_addr *)&(iph->saddr)));
32     printf("%s\n", inet_ntoa(*(struct in_addr *)&(iph->daddr)));
33 }
```