Abstract: This document describes the enhanced implementation of the SCAMPI monitoring software and the applications. It is the successor of deliverable D2.2 “SCAMPI Prototype Implementation Report”, which focuses on the evolution of the different implemented SCAMPI prototypes. This deliverable elaborates on the detailed implementation of the architecture described in deliverable D1.3 “Final Architecture Design” and the optimizations that were considered. Next to the description of the internal structure of the different components, their integration and interconnection is carefully documented. The last chapter in this document describes the implementation of the various SCAMPI applications.

The SCAMPI Consortium consists of:

<table>
<thead>
<tr>
<th>Coordinator</th>
<th>Contractor</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERENA</td>
<td>Coordinating Member</td>
<td>The Netherlands</td>
</tr>
<tr>
<td>IMEC</td>
<td>Principal Contractor</td>
<td>Belgium</td>
</tr>
<tr>
<td>FORTH</td>
<td>Principal Contractor</td>
<td>Greece</td>
</tr>
<tr>
<td>LIACS</td>
<td>Principal Contractor</td>
<td>The Netherlands</td>
</tr>
<tr>
<td>NETikos</td>
<td>Principal Contractor</td>
<td>Italy</td>
</tr>
<tr>
<td>Uninet</td>
<td>Principal Contractor</td>
<td>Norway</td>
</tr>
<tr>
<td>CESNET</td>
<td>Principal Contractor</td>
<td>Czech Republic</td>
</tr>
<tr>
<td>FORTHnet</td>
<td>Principal Contractor</td>
<td>Greece</td>
</tr>
<tr>
<td>Masaryk University</td>
<td>Principal Contractor</td>
<td>Czech Republic</td>
</tr>
<tr>
<td>Siemens</td>
<td>Principal Contractor</td>
<td>Germany</td>
</tr>
</tbody>
</table>
Contents

1 Introduction .............................................. 7

2 Architectural Design .................................. 10

3 The Monitoring API (MAPI) ............................. 13
   3.1 Creating and Terminating Network Flows .............. 14
   3.2 Reading packets from a flow .......................... 14
   3.3 Applying functions to Network Flows .................. 14
   3.4 Flow Records .................................. 16
   3.5 MAPI example: monitoring FTP traffic ............. 16

4 The MAPI Daemon (MAPid) .............................. 19
   4.1 Inter Process Communication ....................... 19
   4.2 SCAMPI in a distributed environment ................ 20
   4.3 Handling MAPI function calls in the MAPI daemon .... 23
      4.3.1 Creating and closing online and offline flows .... 23
      4.3.2 Applying functions to a flow .................... 24
      4.3.3 Retrieving results from a function ................ 24
      4.3.4 Obtaining packets from a configured flow ....... 25
      4.3.5 Retrieving asynchronous events .................. 25
      4.3.6 Loading/unloading MAPI function libraries ...... 27
   4.4 The Standard MAPI Function Library ............... 27
      4.4.1 Design and implementation ...................... 27
      4.4.2 The Standard MAPI function library ............ 33
      4.4.3 BPF_FILTER - The BPF packet filter and optimizations 33
      4.4.4 BYTE_COUNTER and PKT_COUNTER .............. 37
      4.4.5 COOKING - Cooking a network flow ............. 37
      4.4.6 ETHEREAL - The Ethereal protocol filter .......... 38
      4.4.7 SAMPLE - Packet sampling ...................... 38
      4.4.8 STR_SEARCH - Pattern matcher .................. 40
      4.4.9 TO_FILE - Storing packets in a file ............ 41
      4.4.10 TO_BUFFER - Exporting packets to a buffer .... 44
   4.5 The MAPI Drivers ................................. 45
SCAMPI Deliverable D2.3
Enhanced SCAMPI Implementation and Applications

8.3.1 Description and Environment ......................... 89
8.3.2 Components ........................................ 90
8.4 Threshold Alerting for Traffic Engineering ............... 92
  8.4.1 Description and Environment ....................... 92
  8.4.2 Components ....................................... 93
8.5 Security Applications ................................ 93
  8.5.1 Intrusion Detection Application .................... 93
  8.5.2 NetFlow IDS ...................................... 95
  8.5.3 Denial of Service Attack Detection Application ..... 96

A Configuring, Compiling and Running the SCAMPI Software 100
  A.1 General configuration of the SCAMPI software ......... 100
  A.2 Configuring and compiling the SCAMPI software components ... 101
    A.2.1 The Ethereal Display Filters [optional] .......... 101
    A.2.2 The admission control daemon [optional] .......... 102
    A.2.3 Compiling the MAPI DAG driver .................. 104
    A.2.4 Compiling the SCAMPI daemon .................... 104
  A.3 Running the SCAMPI software .......................... 104
    A.3.1 The MAPI daemon .................................. 104
    A.3.2 A SCAMPI application ............................ 105

B Assertions ............................................. 106
List of Figures

2.1 MAPI components ............................................. 11
4.1 Remote MAPI using RPC ........................................ 21
4.2 Optimization of filter expressions ............................... 34
4.3 Processing time of expressions containing identical atoms .......... 35
4.4 Performance comparison ......................................... 36
4.5 The kernel-level ring buffer .................................... 50
4.6 Main components of the IXP1200 ................................. 53
5.1 Overview of admission/resource control .......................... 63
7.1 The COMBO6 card ............................................... 74
7.2 The COMBO-4MTX add-on card .................................. 75
7.3 The COMBO-4SFP add-on card ................................... 75
7.4 The COMBO-2XFP add-on card .................................. 76
7.5 The COMBO-PTM card ............................................ 77
7.6 Firmware of SCAMPI adapter ...................................... 78
8.1 Application that Counts Traffic Per Virtual Streaming Server ....... 86
8.2 Flowrep components ............................................. 86
8.3 Flowrep database design ......................................... 87
8.4 Flowrep web interface ............................................ 88
8.5 Architecture of the QoS monitoring application .................... 89
8.6 Architecture of the Database ..................................... 91
8.7 Comparison between Snort NIDS and MAPI-based NIDS. .......... 95
8.8 Adaptive threshold algorithm. An alarm is signaled when the mea-
urements exceed a threshold, for a number of consecutive intervals k. .......... 97
8.9 CUSUM algorithm. An alarm is signaled when the accumulated
volume of measurements $g_i$ that are above some traffic threshold,
 exceed some aggregate volume threshold $h$. ...................... 98
Chapter 1

Introduction

With the widespread use and deployment of the Internet, traffic monitoring has been increasingly used as a mechanism to improve the performance and security of computer networks. In today’s operational networks, network monitoring is needed by different kinds of users that would like to monitor different aspects of the network traffic. Some users may for example be interested in obtaining aggregate traffic statistics, other users may be interested in monitoring each and every packet that travels through their network (e.g. in order to detect cyberattacks). Current state-of-the-art tools focus on solving only one subclass of all network monitoring problems. Network administrators for example use tools that monitor the state of their networks in order to detect possible malfunctions and alert when such an event occurs. In addition to the detection of possible network failures, most of these tools can also monitor the amount of traffic that flows through the various segments of the network in order to inform administrators of the network usage and enable them to make informed traffic engineering decisions. Besides the traffic monitoring tools used by network administrators, there also exist monitoring environments specialized in discovering Intrusion attempts. These Intrusion Detection Systems (IDSs) examine every packet they capture from the network and try to detect one of the known intrusion threats. Similar to IDSs, Denial-of-Service attack detection tools monitor the network traffic in order to detect generalized network attacks. Finally, there exists a variety of traffic capture tools that focus on capturing (and optionally storing on persistent storage for offline analysis) all network packets (possibly along with their payloads) in real-time.

Currently, most of the above monitoring tools and environments are based on different sets of primitives and functions. For example, traffic monitoring tools are based on the primitives provided by network routers such as NetFlow [21]. On the other hand, traffic capture tools, such as those used by NLANR, are build on top of a custom-made hardware-software infrastructure, collectively known as OC3MON [7]. Intrusion detection systems, such as Snort [19], have been implemented on top of the libpcap [17] packet capture library, while Denial-of-Service detection systems are being implemented on top of firewalls, which on Linux-based
systems are usually implemented on top of netfilter and iptables. To make matters worse, commercial vendors frequently use their own libraries and standards, contributing even more to the Babel of network monitoring tools and environments, making it even more difficult to write portable network monitoring applications.

Monitoring is also complicated by the fact that the network speed increases at a rate that exceeds Moore’s Law. For example, some research suggests that bandwidth doubles roughly every 9-18 months [2], an observation that is usually referred to as “Gilder’s Law”. At the same time, network monitoring applications tend to become more complex and demanding. Where early monitoring applications commonly required little information from the network (e.g. aggregated traffic or statistics), more recent tools may need a much more significant amount of information, possibly including both header and payload for each and every packet. Worse still, the amount of processing needed on this data tends to increase. For example, for detecting Internet worms and various other forms of cyberattacks, we may need such computationally intensive processing as string matching on the entire payload.

Addressing the challenges posed by the state-of-the-art tools in network monitoring, SCAMPI represents a step towards building an affordable network monitoring system for high-speed networks that will enable the development of portable applications. SCAMPI employs a three-pronged approach in order to achieve its goals:

- **Standard Monitoring API (MAPI).** SCAMPI has defined [26] and implemented a set of monitoring calls/primitives that are collectively called the MAPI. Monitoring applications are written using this MAPI. The MAPI is implemented on top of several platforms, decoupling the development of the monitoring applications from the monitoring environment on top of which the applications will be executed. Monitoring applications are written once, and are able to run on top of any monitoring environment without the need to re-write or re-compile the application.

- **Expressive power.** Current monitoring application programming interfaces provide little (if any) expressive power to application programmers. Application programmers are not able to communicate their monitoring requirements to the underlying network monitoring system. As a result, frustrated application programmers end up receiving all network packets in the address space of their application where they perform the operations they need. As a simple example of the poor expressive power of current network monitoring systems, consider a user that wants to sample one out of every 100 packets in order to find the most popular applications that use his/her network. Current network monitoring systems (like libpcap/Berkeley Packet Filters [17, 16], and Linux Socket Filters [12]) do not enable users to express such simple sampling requirements. Therefore, users that are interested in receiving just one out of every 100 packets are required to read all packets, and just discard 99 out of every 100 of them. To overcome these limitations, SCAMPI’s
MAPI will enable monitoring application programmers to express their requirements to the underlying monitoring system, which in turn will decide how these requirements are more efficiently implemented.

- **Scalability through special-purpose hardware and parallelism.** Although network monitoring can be performed on top of traditional network adapters, SCAMPI, wherever possible, will exploit specialized network adapters that provide some monitoring functionalities in hardware. These devices contain on-board processors and FPGAs that can be programmed to perform monitoring functions and off-load the host processor, memory system, and I/O bus from much of their load. An important part of the SCAMPI project is the design, development and manufacturing of our own specialized network adapter, called the SCAMPI adapter. The SCAMPI adapter is able to do packet filtering, sampling, payload inspection and produce traffic statistics in hardware at a rate of 10Gbit/s.

While the SCAMPI architecture is fully documented in Deliverable D1.3 “Final Architecture Design” [26], this document presents the enhanced implementation of the SCAMPI monitoring software and the proposed applications. It is the successor of deliverable D2.2 “SCAMPI Prototype Implementation Report” [27], which focused on the evolution of the different implemented SCAMPI prototypes. This deliverable elaborates on the detailed implementation of the architecture and the optimizations that were considered. Next to the description of the internal structure of the different components, their integration and interconnection is carefully documented. This document also describes the design, functionality and implementation of the various SCAMPI applications.

Chapter 2 focuses on the architectural design of the SCAMPI software. Chapter 3 describes the Monitoring API (MAPI) as described in Deliverable 1.3, with respect to the implementation. In Chapter 4 the implementation of the MAPI daemon is documented, while chapter 5 elaborates on the admission and resource control daemon. Chapter 6 describes the SCAMPI error handling subsystem and chapter 7 documents the developed SCAMPI adapters. Finally, chapter 8 handles the design and implementation of the various developed SCAMPI applications. Appendix A illustrates how a user can configure, compile and run the SCAMPI software.
Chapter 2
Architectural Design

The main goal when designing MAPI was to create an architecture that added as little overhead as possible on the actual processing of packets. MAPI is also a multiuser API where multiple users can run several monitoring jobs concurrently and it was important to create an architecture where global optimization based on all monitoring jobs from all users would be possible. Another important feature was to make MAPI extensible so that new packet processing functions could easily be added. It should also be possible to run the same applications on different hardware without having to make any changes to the application.

Figure 2.1 shows a general overview of the architecture of the current prototype. The basic components are applications using MAPI, the daemon MAPId, kernel drivers and the hardware adapters.

To solve the problem of global optimization, a daemon was used. All processing of packets is controlled by this daemon and the actual processing can be done inside the daemon, in hardware, in kernel or a combination of these. With a central location for controlling packet processing, it is easy to implement mechanisms that looks at all the monitoring jobs from all users and optimizes the processing of packets accordingly. For example, if two different applications both want only TCP traffic, this filtering will only be done once.

The applications uses a MAPI stub to communicate with the daemon using IPC. This IPC communication is however only used when new flows are being set up. When an application connects to a newly created flow and starts reading results, the MAPI stub called by the application reads the results directly from MAPId using shared memory or directly from hardware without having to rely on the relatively slow IPC.

Inside MAPId there is a module called mapidcom. This module is responsible for all IPC communication to and from user applications. When mapidcom receives an IPC message, it forwards it to the correct MAPI driver inside MAPId which does the actual work. Having drivers for different devices inside MAPId allows user applications to run on top of the different hardware without any change. Figure 2.1 only shows one driver, the mapidagdrv, but this is just for simplicity.
Figure 2.1: MAPI components
Multiple drivers can be loaded at the same time.

While a MAPI driver has complete freedom to implement its features any way it sees fit, the architecture is built up around drivers using a general purpose library, mapidlib, to handle flows and support for loadable functions. The library mapidlib keeps track of open flows and calls the functions that has been applied to flows when a new packet is captured.

The three last modules inside MAPId are all related to support for function libraries. These libraries can be loaded and unloaded dynamically while MAPId is running. The mapilibhandler loads new libraries and keeps track of which functions that are available. To make the implementation of new functions easier, a function helper library, fhelp, has been created.

The default method for returning results from function to the client is through shared memory. Support for this is already implemented in the fhelp module and is easy for functions to use. When other mechanisms are needed, functions specific code is needed on the application side as well as inside MAPId. This is why support for loading function libraries is included both in MAPId and the MAPI stub run by the client. For example say that a function want a client to read results directly from hardware. It must then return enough information to the application through IPC so that the MAPI stub can locate and read the correct results in hardware. The code that do the actual reading from hardware will be function specific and is therefor part of the function library that has been loaded.

So far tests show that the chosen architecture fulfills all the major design criteria. It is relatively fast with little overhead, it is easy to extend and there are very few restrictions on drivers so that when new more advanced hardware becomes available it will be easy to add support for them.
Chapter 3

The Monitoring API (MAPI)

The goal of an application programming interface is to provide a suitable abstraction which is both simple enough for programmers to use, and powerful enough for expressing complex application specifications. A good API should also relieve the programmer from the complexities of the underlying hardware while making sure that hardware features can be properly exploited.

MAPI builds on a simple and powerful abstraction, the network flow, but in a flexible and generalized way. In MAPI, a network flow is generally defined as a sequence of packets that satisfy a given set of conditions. These conditions can be arbitrary, ranging from simple header-based filters, to sophisticated protocol analysis and content inspection functions. For example, a very simple flow can be specified to include all packets, or all packets directed to a particular web server. A more complex flow may be composed of all TCP packets between a pair of subnets that contain the string “User-agent: Mozilla/5.0”.

Our approach to network flows is therefore fundamentally different from existing models that constrain the definition of a flow to the set of packets with the same source and destination IP address and port numbers within a given time-window. In contrast with existing models, MAPI gives the “network flow” a first-class status: flows are named entities that can be manipulated in similar ways to other programming abstractions such as sockets, pipes, and files. In particular, users may create or destroy (close) flows, read, sample or count packets of a flow, apply functions to flows, and retrieve other traffic statistics from a flow.

By using first-class flows, users can express a wide variety of new monitoring operations. For example, MAPI flows allow users to develop simple intrusion detection schemes that require content inspection [1]. In contrast, traditional approaches to traffic/network flows, such as NetFlow, IPFIX, and related systems and proposals do not have the means of providing the advanced functions required for this task.

In the remainder of this Section we present an overview of the main operations provided by MAPI. A complete specification of MAPI and a more detailed description of each function is provided in [25].
3.1 Creating and Terminating Network Flows

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

\[
fd = \text{mapi}_\text{create}_\text{flow}(\text{char }*\text{dev})
\]

This call creates a network flow, and returns a flow descriptor \( fd \) that points to it. This network flow consists of all network packets which go through network device \( \text{dev} \). The packets of this flow can be further reduced to those which satisfy an appropriate filter or other function, as described in Section 3.3.

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

\[
fd = \text{mapi}_\text{close}_\text{flow}(\text{flow}_\text{desc }fd)
\]

After closing a flow, all the structures that have been allocated for the flow are released.

3.2 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:

\[
\text{packet }* \text{mapi}_\text{get}_\text{next}_\text{packet}(\text{flow}_\text{desc }fd, \text{func}_\text{desc }fid)
\]

which reads the next packet that belongs to flow \( fd \). In order to read packets, the function \text{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:

\[
\text{mapi}_\text{loop}(\text{flow}_\text{desc }fd, \text{int }cnt, \text{mapi}_\text{handler }callback)
\]

The above call makes sure that the handler \( \text{callback} \) will be invoked for each of the next \( cnt \) packets that will arrive in the flow \( fd \).

3.3 Applying functions to Network Flows

Besides neatly arranging packets, network flows allow users to treat packets that belong to different flows in different ways. For example, a user may be interested in \textit{logging} all packets of a flow (e.g. to record an intrusion attempt), or in just \textit{counting} the packets and their lengths (e.g. to count the bandwidth usage of an

\footnote{Although the \text{mapi}_\text{loop} call is inspired from the \text{pcap}_\text{loop} call of the \text{libpcap} library \cite{note1}, in contrary to \text{pcap}_\text{loop}, \text{mapi}_\text{loop} is non-blocking.}

scampi@ist-scampi.org 19th April 2004
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 flows:

```
mapi_apply_function(flow_desc fd, function f, ...)
```

The above association applies function \( f \) to every packet of flow \( fd \). 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. For example, `mapi_apply_function(fd, "BPF_FILTER", "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. Moreover, function `PACKET_COUNT` counts all packets in a flow, function `SAMPLE_PACKETS` can be used to sample packets, etc. There also exist functions that count various traffic metrics, like bandwidth or fragmented packets.

More advanced functions include IP defragmentation and TCP reassembly (i.e., *cooking*), which enables the monitoring system to perform some pre-processing on the stream of packets. For example, when the `COOKING` function is applied to a flow, then the individual packets are pre-processed according to the TCP/IP protocol and concatenated into a data stream. This preprocessing will re-assemble fragmented packets, discard retransmitted packets, re-order out-of-order packets, etc.

MAPI also provides parameterized hashing functions that will take user defined arguments. Based on the value of the hashing function, the packet may be kept or discarded. Although these functions will 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 `packet_count` 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 or a data structure to each network flow. 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_read_results(flow_desc fd, function f, void * result)
```

After the return of the above call the result structure will be populated with the appropriate values computed by the function.
3.4 Flow Records

Although some applications may need to capture entire packets, several other applications may only need statistics about packets that belong to a network flow. For example, an application may be interested in counting only the size of incoming and outgoing traffic of a web server. MAPI can provide such statistics on a per-flow basis using Flow Records. This is achieved with two variables. FLOW_RECORD is a logical variable which when set to TRUE enables the collection of Flow Records. FLOW_RECORD_TYPE can take values from the set \{IPFIX, NETFLOW_V5, NETFLOW_V9\}, selecting the format of the records that MAPI will maintain.

To read the collected flow records, an application can use the read_flow_record function that returns a flow record for the specified flow.

3.5 MAPI example: monitoring FTP traffic

In this Section we present an example of using MAPI to monitor all FTP traffic in a system. The main difficulty with monitoring FTP traffic, as compared to applications like email or web traffic, is that FTP transfers may be performed over dynamically allocated ports, which are not known in advance. FTP uses a well-known port (i.e. 21) only as a control channel. When a file transfer is initiated, the FTP server informs the client about the dynamic port number to be used for the transfer. Therefore, in order to accurately account for all FTP traffic, a monitoring application needs to monitor port 21 to find new clients as well as the dynamic ports these new clients will use in order to transfer their data. Traditional monitoring systems, such as NetFlow, find it difficult to monitor traffic of applications that use dynamically generated ports. For example, although NetFlow and similar approaches, can report the amount of observed traffic per port, they do not know which applications these (dynamically generated) ports correspond to, and thus it is difficult to attribute network traffic to specific applications. On the contrary, MAPI is able to analyze packet payloads to find the dynamically generated ports and to associate those ports with the application that generated them.

The following code can be used to monitor all FTP traffic using MAPI:

```c
packet *p;
flow_descriptor fd, xfers[1024];
struct byte_count_results br;
int src_port, dst_port, count, total_ftp_traffic=0;
char new_flow[64];

/* Create a flow to monitor the control port of FTP: port 21 */
1: fd = mapi_create_flow("/dev/scampi");
2: mapi_apply_function(fd, BPF_FILTER, "tcp port 21");

/* Find packets that indicate the beginning of a new transfer */
/* such packets contain the string "227 Entering Passive Mode" */
```
3: mapi_apply_function(fd, SUBSTRING_SEARCH, "227 Entering Passive Mode");

    /* Track the next 100 transfers */
4: for(count=0; count<100; count++){
5:  p = mapi_get_next_packet(fd);
    /* extract_ports gets a packet which indicates the beginning */
    /* of a new transfer and extracts the dynamic data port */
6:  extract_ports(p, &src_port, &dst_port);
    /* Create a flow to track the forthcoming transfer according */
    /* to the information contained in the control packet */
7:  sprintf(new_flow_filter, "tcp src port %d and dest port %d",
             port[0], port[1]);
    /* Create a new flow for this data transfer */
8:  xfers[count] = mapi_create_flow(/dev/scampi);
9:  mapi_apply_function(xfers[count], BPF_FILTER, new_flow_filter);
10: mapi_apply_function(xfers[count], BYTE_COUNT);
}

/* summary */
9: for(count=0; count<100; count++){
10: mapi_read_results(xfers[count],BYTE_COUNT,&br);
11: total_ftp_traffic += br.bytes;
}

In order to monitor all FTP traffic, we initially define a network flow for capturing all FTP control packets that go through port 21 (line 1). We are interested only for packets indicating a file transfer initiation, thus substring search is applied to distinguish them among the rest (line 2). An example payload of such packet is the following:

227 Entering Passive Mode (147,52,17,51,146,226)

This packet is sent by the server and contains the IP address (147.52.17.51) and the port number (37602) of the forthcoming transfer. Therefore, all necessary information for the transfer session is known so a new flow can be defined for its accounting.

Peer-to-peer, multimedia conferencing and messaging applications usually operate in the same fashion, negotiating transfer ports through a control channel. It is straightforward to adapt the above code to monitor the traffic of any of these applications.

Although the above example demonstrates that MAPI can provide traffic information that traditional flow-level traffic summaries, such as NetFlow, cannot provide, one could have used a packet dumping facility, such as tcpdump or other libpcap-based tools, in order to find similar information. However, implementing the above application using libpcap would have resulted in longer code and higher overheads. For example, libpcap does not provide any string searching facility, and thus the programmer would have to provide a significant chunk of
code to substitute line 2 above. In addition, libpcap does not provide any facility to apply functions to packets, and thus the programmer would have to provide the code to read packets and count their bytes. Instead of forcing the programmer to provide all this mundane code, MAPI already provides this frequently used functionality.
Chapter 4

The MAPI Daemon (MAPId)

4.1 Inter Process Communication

Every application addresses the monitoring platform throughout MAPI. An application uses functions defined in MAPI interface to configure the MAPI Daemon and retrieve results from it. The MAPI Library is compiled into each application and encapsulates the communication with MAPI Daemon. This interprocess communication is implemented using sockets.

At the MAPI Daemon side, mapidcom is the module that handles all the communication with the user applications. During the daemon startup, this module creates a socket and binds it to a predefined address. mapidcom consists mainly of an infinite loop that receives and processes requests from applications through this socket. On the other side, MAPI functions send requests to the daemon through a temporary socket created by the MAPI stub during the application start-up. After sending a request, the function waits for a corresponding acknowledgement from the daemon indicating successful completion of the requested action, or a specific error in case of failure. Each request consists of a structure as follows:

```c
struct mapiipcbuf {
    mapiipcMsg cmd;
    int fd;
    char function[FUNCT_NAME_LENGTH];
    int pid;
    unsigned char data[DATA_SIZE];
};
```

The structure contains all the necessary arguments to implement this request:

- `cmd` contains the request sent by the application to the mapid through the corresponding MAPI function. Such requests are:

  CREATE_FLOW
For example the CREATE_FLOW request creates a flow, the APPLY_FUNCTION request applies a function to it, and so on.

- \texttt{fd} is the network flow descriptor.
- \texttt{function} is used in conjunction with the APPLY_FUNCTION request and denotes the function that must be invoked on each packet of the network flow.
- \texttt{pid} is the process id of the application.
- \texttt{data[DATA\_SIZE]} is a buffer that is being used to keep relevant information for each request. For example, for a CREATE_FLOW request, it holds the name of the monitoring device that will provide the packets to the network flow, or for an APPLY_FUNCTION request, it holds the arguments of the function being invoked.

## 4.2 SCAMPI in a distributed environment

So far, we only addressed applications that require monitoring information from a single observation point. Other applications require not only information from a single observation point, but they need monitoring data from the entire network. Therefore, the application needs to obtain information from multiple distributed points in the network. Consider a Quality of Service (QoS) application. This application needs to determine network characteristics such as throughput, loss, jitter, goodput, delay,... These characteristics can only be computed if we have information from at least both the ingress and egress nodes in the network. The MAPI supports these kinds of applications by creating multiple network flows, at least one for each observation point, with the remote MAPI.

### IPC versus RPC

The interfaces of the Remote MAPI and the “local” MAPI are very similar. The only remarkable difference is the location of the invocation of certain function-calls. In the case of the “local” MAPI, every function is executed on the local monitoring agent. There is no need to specify the location of the monitoring agent.
When the remote MAPI is used to initiate a monitoring job, the location and identification of the monitoring agent needs to be specified. The application has to specify which observation point should run the monitoring job. Obviously, each participating observation point has to run the MAPI Daemon.

An application that needs to monitor a remote observation point, needs to use the remote MAPI. In this case, we extend the MAPI in order to allow remote executions. The `mapi_create_flow` function of the MAPI has to be extended with a location/identity parameter to locate or identify the observation point. From an implementation point-of-view, instead of inter-process communication (IPC) to communicate with the MAPI Daemon, we use remote procedure calls (RPC).

The MAPI interface and the MAPId can communicate in two different ways. If the MAPI is local to the MAPId we use simple IPC for the communication. For instance, if we apply a function to a flow, the MAPI puts the function information and the flow descriptor in an IPC message. The MAPId retrieves this message, takes the necessary actions and puts the result of these actions into a new message. In turn, this message is retrieved by the MAPI, which returns the original calling function.

If the MAPI and the MAPId are not local to each other, simple IPC is no longer possible. In such a case we have to use RPC (Figure 4.1). The MAPId continues to use IPC for its communication, but in between the MAPI and the MAPId we put another component (mapid_rpc), running locally to the MAPId, that acts as a translator between the two protocols. So what happens is that the MAPI does a RPC function call on the mapid_rpc which in turn translates this function call to an IPC message which is retrieved by the MAPId. The MAPId puts the result of the requested action in another IPC message which is retrieved by the mapid_rpc and the information in this message is returned as the result of the RPC function.
call. The MAPId is in fact unaware whether the client application runs local or remote.

The MAPI can be compiled in three different ways:

- Compiling it with the IPC_ONLY option set, it only incorporates the functionality to use simple IPC message passing. Trying to use a remote MAPId with such a MAPI will not work.

- Compiling it with the RPC_ONLY option set, will result in a MAPI with only RPC functionality. Using RPC, a MAPI will be able to communicate with both a remote as well as a local MAPId. However, since it uses RPC, local communication will have a lower performance than the simple IPC variant.

- Without the RPC_ONLY or IPC_ONLY option, the MAPI will incorporate both IPC and RPC functionality and will decide at runtime which method of communication to use.

If a user is certain that the MAPI will always be local to the MAPId, he/she should compile it to the IPC_ONLY version. If the user is certain that only a remote MAPId will be used, the RPC_ONLY option should be used. If the user is uncertain or if both a remote and local MAPId will be used, the complete version should be compiled.

Creating multiple network flows

A second step in writing a distributed monitoring application is to open multiple network flows, at least one for each observation point. After retrieving monitoring results from these flows, the application can process and correlate them. The processing and correlation of the data obtained from the distributed sites is the responsibility of the applications. The MAPI only offers the programmer the opportunity to easily obtain distributed data.

The following example illustrates a basic application that requires 2 observation points.

```c
int fd1, fd2;
int counter1, counter2;
unsigned long long result1, result2;

fd1 = mapi_create_flow("eth1@10.0.0.1");
mapi_apply_function(fd1, BPF_FILTER, "SRC 10.0.0.1 and DST 10.0.0.2");
counter1 = mapi_apply_function(fd1, PKT_COUNTER);

fd2 = mapi_create_flow("eth1@10.0.0.2");
mapi_apply_function(fd2, BPF_FILTER, "SRC 10.0.0.1 and DST 10.0.0.2");
counter2 = mapi_apply_function(fd2, PKT_COUNTER);

mapi_connect(fd1);
mapi_connect(fd2);
```
sleep(60);
mapi_read_results(fd1, counter1,&result1);
mapi_read_results(fd2, counter2,&result2);

printf("There were %lld packets dropped\n", counter1, counter2);
mapi_close_flow(fd1);
mapi_close_flow(fd2);

In this application 2 flows are configured, one on the observation point with IP address 10.0.0.1 (the ingress) and one on the observation point with IP address 10.0.0.2 (the egress). Both the flows count all the packets with source address the first observation point and destinations address the second observation point. After one minute, the application prints the number of dropped packets, sent from observation point 1 to 2. This very basic example illustrates how an application can use both the remote MAPI and the creation of multiple flows in order to monitor distributes sites. After this configuration, the application only needs to correlate and process the retrieved data.

4.3 Handling MAPI function calls in the MAPI daemon

4.3.1 Creating and closing online and offline flows

The syntax of the MAPI call for creating a flow is:

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

where dev specifies the device that should be used for capturing packets for the flow.

When MAPId is started it reads the configuration file an loads all the drivers specified there. The configuration file states which devices should use which driver. When an application calls mapi_create_flow a message is sent to mapidcom inside MAPId through IPC. mapidcom will then look at the device that the flow wants to use, locate the correct driver and call the mapidrv_create_flow interface for that driver.

How mapidrv_create_flow is implemented, varies for each driver. Normally a driver will check to see if this device has already been used for monitoring jobs. If not, it will initialize the device and start capturing packets. The driver will also add the new flow to mapidlib which is responsible for keeping track of the open flows.

The syntax for the MAPI call to create an offline flow is:

```c
int mapi_create_offline_flow(char* path, int format, int speed)
```

where path specifies the full path and name of the file that should be read, format specifies the file format and speed specifies if the file should be read as fast
as possible or if timestamps of packets should be used to process the packets at the same time interval as they were captured.

When a mapi_create_offline_flow is called, mapidcom will again use the configuration file to see which driver should handle the request. Each file format is assigned to a driver. Before calling mapidrv_create_offline_flow, mapidcom will first create a new virtual device for the file that is being read. This means that each file opened by a mapi_create_offline_flow is treated by MAPI drivers as a separate device.

Internally, the mapidrv_create_offline_flow is very similar to mapidrv_create_flow. The only difference is that instead of reading packets from the device, the function mapidrv_create_offline_flow opens a file and reads packets from it.

### 4.3.2 Applying functions to a flow

Functions are applied to a flow through the mapi_apply_function call:

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

where `fd` is the function descriptor and `funct` is the name of the function. The rest of the arguments depends on the function being applied.

A request for applying the function is sent to mapidcom which uses `fd` to find out which driver is handling this flow. The request is then forwarded to this driver.

Most drivers simply forward an apply_function request to mapidlib. mapidlib uses the library handler to locate the correct function and then calls the instance interface of this function to create a new instance.

### 4.3.3 Retrieving results from a function

Results from a function are retrieved through the mapi_read_results function. The syntax of this function is:

```c
int mapi_read_results(int fd, int fid, void* result)
```

where `fd` is the flow descriptor, `fid` is the ID of the function and result is a pointer to structure where results are copied to. In case of a successful retrieval zero is returned, elsewhere -1.

Initially, a request for reading results is sent to the MAPI daemon. The daemon then calls the appropriate function of the corresponding driver, packs the result and sends it back to the client. The results returned through IPC is not the actual results but information needed by the client to get hold of the results either through shared memory or directly from hardware. This information is stored so subsequent calls to mapi_read_results can use it directly without any IPC communication.

When mapi_read_results receives the information from the daemon, it checks to see if the function whose results are being read, implements a function specific client_read_results. If so, the functions client_init is called and then client_read_results.
If the function do not implement a client_read_results function, a default method that reads the results from shared memory is used.

### 4.3.4 Obtaining packets from a configured flow

As described in chapter 3, after applying the TO_BUFFER function an application can get a packet from the flow with the function:

```c
packet* mapi_get_next_packet(flow_desc fd, func_desc fid)
```

or

```c
mapi_loop(flow_desc fd, int cnt, mapi_handler callback)
```

where fd is the flow descriptor and fid is the ID of the function. mapi_loop also needs the number of packets that need to be processed and a callback function that needs to be applied on each packet.

The request for reading a packet is sent to the MAPI daemon, the same way as done when calling the mapi_read_results function. After getting hold of the pointer to the shared memory, subsequent calls read the results directly from the shared memory. This shared memory contains a FIFO queue containing references to packets in the joint circular buffer. If a new packet is present in the buffer, a pointer to that packet is returned to the application. More information is given in section 4.4.10.

### 4.3.5 Retrieving asynchronous events

Asynchronous events are implemented as thresholds in MAPI. Since thresholds can be met while no packets are received by the monitoring application, this kind of function must be running outside of the normal packet flow. Thresholds are applied on another mapi function to get useful data from mapid. Three kinds of thresholds exist:

1. Boolean
2. Integer
3. Unsigned long long

There are several ways to use thresholds:

- Inside the packet flow: these thresholds are checked every time a packet is received. If a timeout value is added, no new events will be sent to the application after the first event during this timeout. The thresholds can be applied on the real value of the thresholded function, or on the difference with the previous packet.
Polling: these thresholds poll the value at a predefined rate (specified in the timeout parameter). This runs outside of the normal packet flow, so it is possible to generate events while no packets arrive. It is again possible to use the real value, or the difference between 2 polls.

Moving window: this threshold function uses 2 parameters. A timeout value, which represents the length of the moving window. And a divider, giving the number of poll-intervals within one window. This threshold is always based on the sum over the moving window of the differences of the thresholded value between polling-intervals.

The last configuration parameter represents the number of thresholds that should be generated before the function stops. If a value of 0 is supplied, the thresholding runs until it is removed. Prototype:

```
void mapi_apply_function(char *flow_desc, char *THRESHOLD, char *threshold_type, int function_id, float threshold, int threshold_boundary_type, int timeout, int threshold_count);
```

No special value is added to threshold_type. The timeout value is the interval while no new events are generated. If this is 0, then all events are forwarded.

```
void mapi_apply_function(char *flow_desc, char *THRESHOLD, char *threshold_type, int function_id, float threshold, int threshold_boundary_type, int timeout, int threshold_count);
```

If TR_POLL is added to the threshold_type flags, the polling version is used. The timeout value represents the polling interval.

```
void mapi_apply_function(char *flow_desc, char *THRESHOLD, char *threshold_type, int function_id, float threshold, int threshold_boundary_type, int timeout, int divider, int threshold_count);
```

If TR_MW is added to the threshold_type flags, this moving window version is used.

The possible values for threshold_boundary_type are:

- TR_B_POS: if the thresholded value gets lower than the measured value, the threshold is met.
- TR_B_EQ: if the thresholded value equals the measured value, the threshold is met.
- TR_B_NEG: if the thresholded value gets higher than the measured value, the threshold is met.
- TR_B_D_POS: the threshold is met when the difference with the previous measurement gets higher than the threshold.
The application waits with a blocking call on a semaphore. When a threshold is met, mapid increases this semaphore, and the application immediately decreases it again, since it is waiting on it. This is implemented in the mapi_read_results function of mapi. This function implements the blocking semaphore operation. When it returns, the threshold was reached.

4.3.6 Loading/unloading MAPI function libraries

Users can load and unload libraries using the mapi_load_library and mapi_unload_library functions. Both takes the file name of the library as a parameter.

When mapidcom receives a request to load a library, all loaded drivers are called and asked to load the library. The code for the library is in a shared library and is only loaded once, but each driver has to be told to load the library so that they can all keep track of the functions that are available.

Requests to unload a library is also sent to all drivers but a library can only be removed if all the functions inside the library is unused by any active flows. If a function is in use, an error message is returned to the user.

4.4 The Standard MAPI Function Library

4.4.1 Design and implementation

Being able to create libraries of functions that can be applied to packets and that can be loaded dynamically while MAPId is still running is a key technology in MAPI. This makes it possible to extend MAPI with new functionality without interrupting existing monitoring jobs.

A default set of libraries can be loaded automatically when MAPId is started. The configuration file has two entries that controls this:

libpath specifies where library files are located

libs list of libraries that should be loaded at startup. Multiple library file names are separated by a :

All users can also load new libraries at runtime by issuing a mapid_load_library command. The libpath entry in the configuration file prevents users to load unauthorized libraries. Since all libraries that can be loaded has to be located in the directory pointed to by libpath, this directory can be write protected and only an administrator can add new libraries.
Device OID

Since MAPI can support multiple types of hardware, there has to be support for implementing highly optimized functions that only runs on a specific type of hardware. To do this, each type of device supported by MAPI is assigned a device OID and all functions tell the system which device OIDs they are compatible with.

A device OID is similar to an OID in SNMP and is simply a list of integers separated by a “.” which points to a node in a tree. When a function states that it supports a specific device OID, it means that it can support this OID and all its children.

When a function is applied to a flow, the library handler inside MAPId first locates the most optimized version of the function. The most optimized version is the version that specifies the longest device OID that fits the device the flow is running on. If it is not possible to initialize this function, for example because the hardware resources are not available, then the library handler will locate the generic software version and use that instead. Generic software functions specify a device OID of “1” since that it the root node and all devices are children of this node.

Example Assume that there is a device with a device OID of “1.2.3.4”. A function is being applied to a flow running on this device and there are three different versions of this function. Each version specifies different device OIDs:

function1  devoid=”1”
function2  devoid=”1.2.3”
function3  devoid=”1.2.3.5”

In this example both function 1 and 2 can be used on the device while function 3 is not compatible. Of function 1 and 2, function 2 is the one with the longest OID so this means that it is the most optimized version and it should be the one that is used first.

Functions

All functions have ten different although many of them are optional to implement.

instance The instance interface creates a new instance of the function. It should do a rudimentary syntax check of the arguments passed to the function and store them for future references. No resources should be allocated here. If a device specific function can not be applied, it can return the error message

MFUNCT_COULD_NOT_APPLY_FUNC T,

and mapidlib will instead try to use the generic software version. This interface is mandatory to implement.
Arguments

`int fd` flow descriptor

`int fid` function ID

`mapid_hw_info_t *hwinfo` pointer to a structure that contains hardware related information

`flist_t *flist` list of functions that has already been applied to this flow

`mapiFunctArg *fargs` string containing the arguments of the function

`mapidflib_function_instance_t **instance` the newly created instance should be assigned to this argument

`init` The init interface initializes the function. It allocates necessary resources like shared memory or hardware based on the function arguments that was given to the instance interface. This interface is mandatory to implement.

Arguments

`mapidflib_function_instance_t *instance` pointer to the instance of the function

`process` The process interface do the actual processing of a packet and is called for each new packet that needs processing. This interface is optional to implement. Some functions will never do any actual processing in software since everything will be done in hardware and can the skip implementing this interface.

Arguments

`mapidflib_function_instance_t *instance` pointer to the instance of the function

`char *dev_pkt` pointer to the packet as returned by the device. This may include any device specific headers

`char *link_pkt` pointer to the link layer packet

`mapid_pkthdr_t pkt_head` pointer to the MAPI packet header. This contains timestamp information, the length of the captured packet and the actual length of the packet on the wire.

`get_result` This interface returns the function results. It is optional to implement since some functions do not return any results.

Arguments

`mapidflib_function_instance_t *instance` pointer to the instance of the function
change_args  This interface changes the arguments of a running function. It is optional to implement.

Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function
mapiFunctArg *fargs  new function arguments

reset  This interface resets the results of a function. It is optional to implement.

Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function

cleanup  This interface frees the resources used by the function and is called when the flow is closed. Mandatory to implement.

Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function

get_args  This interface returns the current arguments of a running function.

Arguments

mapidflib_function_instance_t *instance  pointer to the instance of the function
mapiFunctArg *fargs  pointer that is set by the interface so that it points to the arguments

client_init  This interface initializes the client side of the function.

Arguments

int fd  flow descriptor
int fid  function ID
void *res  pointer to the result information that was returned by the get_result interface

mapidflib_function_instance_t **instance  the newly created instance should be assigned to this argument
**client_read_result**  This interface returns function results to the mapi_read_result call which can then return it to the user application.

**Arguments**

*mapidflib_function_instance_t* *instance*  pointer to the instance of the function

*mapid_result_t* *res*  pointer to a structure where information about the results should be stored

**client_cleanup**  This interface frees all resources used by the client side of the function and is called when the function is closed.

**Arguments**

*mapidflib_function_instance_t* *instance*  pointer to the instance of the function

**Function instances**

The actual code for a function is only loaded once so each function must be able to handle multiple instances. The static definition of a function that is common to all instances are defined in the structure mapidflib_function_def:

- **libname**  name of library that the function belongs to
- **name**  name of function
- **descr**  textual description of the function
- **argdescr**  description of number and type of arguments that this function needs
- **devoid**  list of device OIDs that this function supports
- **instance**  pointer to the instance interface of the function
- **init**  pointer to the init interface of the function
- **process**  pointer to the process interface of the function
- **get_result**  pointer to the get_result interface of the function
- **change_args**  pointer to the change_args interface of the function
- **reset**  pointer to the reset interface of the function
- **cleanup**  pointer to the cleanup interface of the function
- **get_args**  pointer to the get_args interface of the function
As we can see, this structure contains the pointers to where in memory the code that implements the various interfaces are located. This is used by mapidlib so that it can call the various interfaces as needed.

In addition to this static information each function instance is defined by the structure mapidlib_function_instance:

- **fd**: flow descriptor
- **fid**: function identifier
- **name**: name of function
- **args**: string containing the arguments of the function
- **result**: information about function results that is sent back to the user
- **internal_data**: pointer to internal data used by the function instance
- **hwinfo**: pointer to hardware related information about the device the function is running on

**fhlp**

The library fhlp contains various common functions that MAPI functions can take advantage of to make the implementation easier. The following functions have been defined:

- **fhlp_add**: creates a new instance of a function.
- **fhlp_remove**: deletes an existing instance of a function
- **fhlp_copy_args**: makes a copy of the function arguments
- **fhlp_get_args**: returns the current function arguments
- **fhlp_set_shm_res**: generic function that can be used by functions who return simple results through shared memory.
- **fhlp_get**: returns an instance of a function
- **fhlp_check_software_funct**: returns true if there exists any software only functions already applied to a flow.
4.4.2 The Standard MAPI function library

The standard MAPI function library contains all the functions that should be available in any MAPI installation. Most of the functions are generic software functions that can run on any type of hardware. The following functions are currently part of the library:

- **BPF_FILTER**  bpf filter implementation
- **BYTE_COUNTER** counts the number of bytes captured by the flow
- **COOKING** defragmentation and reassemble of TCP/IP packets
- **ETHEREAL** ethereal filter implementation
- **PKT_COUNTER** counts the number of packets captured by the flow
- **SAMPLE** sampling of packets
- **STR_SEARCH** searcher for a string inside the packet
- **TO_FILE** stores the packets of a flow to file.
- **TO_BUFFER** stores the packets of a flow in a buffer that can be read by user applications

4.4.3 BPF_FILTER - The BPF packet filter and optimizations

In order to filter packets based on header fields of layer 2 (datalink layer), 3 (network layer) and 4 (transport layer) protocols, the SCAMPI library offers the BPF (Berkley Packet Filter) filter as provided by the libpcap library [17]. The syntax of BPF expressions is documented in [13].

**Implementation**

In the MAPI a BPF function can be applied by calling the following MAPI function:

```c
mapi_apply_function(int fd, "BPF_FILTER", char* exp);
```

where fd is the flow descriptor and exp the BPF expression. In case we use a regular NIC or DAG driver, the MAPI Daemon will fetch the BPF function from the standard MAPI library, attach and compile the BPF filter and link it in the packet processing chain. When using the SCAMPI adapter, the BPF function will be passed to the SCAMPI adapter driver.
Filter optimizations

The key idea behind the optimization is to minimize the number of evaluations by eliminating duplicate evaluations of the same (sub-)expression on a single packet. Consider for instance the situation where one application needs to count TCP packets and another application needs to count TCP packets from port 80. In this case the optimization algorithm will transform the left-most expression in Figure 4.2 to the right-most expression.

In the next few paragraphs, we will introduce the term atomic expression. An atomic expression or atom is a filter expression that cannot be subdivided in other expressions, e.g. “IP=10.10.10.5.”, “PROTOCOL = 6” or “TCP Port = 10”. An expression on the other hand is a logical combination of atoms by using the logical operators AND, OR, NOT, XOR and parentheses. We can write an expression in the following “Backus Naur Form” (BNF):

\[
e ::= e \text{ Operator } e | \text{ not } e | (e) | \text{ Atom}
\]

```
Operator ::= \text{ and} | \text{ or} | \text{ xor}
```

In a first step of the basic optimization algorithm, all configured expressions are parsed and stored in an expression tree, consisting of logical operators and references to the atomic expressions. E.g.

\[
(\text{host X or host Y}) \text{ and} (\text{port 1 or port 2})
\]

\[
\text{host Z and} (\text{port 1 or port 2})
\]

\[
\text{host Y}
\]

After the first parsing this leads to the following prefix notation:

```
AND
(OR
  port 1 refcount:2
  port 2 refcount:2)
(OR
  host Y refcount:2)
```
In the second step, based on the original tree, an overlap-tree is created. Nodes from different, overlapping paths are joined, so they point to the same node in the nodelist. Nodes with the same reference count in all paths and using the same operator are joined again to form larger sub-expressions. All nodes with a reference count equal to 1 within the same operator are joined to form larger expressions. In order to create the optimal form, redundant parentheses are removed.

Finally, all expressions are compiled and stored in the expression tree.

![Figure 4.3: Processing time of expressions containing identical atoms](image)

The evaluation of the aggregated expression relies on the fact that nodes with the same sub-expression point to the same node. If the current packet ID matches the one saved in the node, there is no need to evaluate the sub-expression another time. The saved result of the previous evaluation will be returned. When an expression is evaluated, the packet ID and the result are saved in the node. The evaluation
of the operators in an expression is also short-circuit. This means that the evaluation of an AND-operator is terminated when a sub-expression returns “false” and an OR-operator when a sub-expression returns “true”.

Figure 4.3 shows the packet processing time of a series of expressions compared to the optimized form. In both cases the processing time increases linearly with the number of (identical) atoms in the expressions. Because the optimized form will reuse already evaluated atoms, the processing time increases much slower.

Performance measurements

To validate the proposed optimization algorithms, we have done some experiments. We used a slimmed-down version of the DEFCON packet traces to generate traffic. To do the required measurements, 20 applications that each configure a BPF filter consisting of 5 atoms were installed. These five atoms are uniformly selected out of a set of 15 random atoms that are present in the source trace.

![Performance comparison](image)

Figure 4.4: Performance comparison

Figure 4.4 depicts the comparison of the performance of the various optimization techniques. Without any optimization the system needs about 37,100 clock cycles to process a packet. When we short-circuit an expression if the result of the evaluation is known, we can reduce the processing time to 34,700 clock cycles. If we eliminate all duplicate evaluations, the original processing time can be radically brought down to 21,300 cycles. The combination of both of the previous optimization techniques results in a packet processing time of about 20,400 cycles. Because the elimination technique results in the biggest performance boost, the additional improvement due to the short-circuit technique is negligible in our example. This is due to the fact that, in the last case, we primary short-circuit expressions where all remaining atoms are already evaluated once. The elimination technique reduces the re-evaluation of these atoms significantly. The probabilistic optimization technique is out of the scope of this document. More information about the probabilistic optimization and a theoretical study of the other optimization techniques...
4.4.4 BYTE_COUNTER and PKT_COUNTER

These are two simple functions that counts the number of packets in a flow or the number of bytes. The packet counter function is called PKT_COUNTER and the bandwidth meter is called BYTE_COUNTER.

Neither functions take any arguments and both of them returns an unsigned long long as a result.

4.4.5 COOKING - Cooking a network flow

Cooking function performs defragmentation of packets in IP layer and packet re-assembly in TCP layer. Cooking function keeps track of the open sessions as re-assemble demands stateful inspection. A session is defined as a unique quadruplet of source/destination IP addresses and source/destination ports.

In IP layer packets are fragmented due to network MTU (Maximum Transfer Unit) size. Cooking function receives all the fragments and assembles them into a single IP packet. In TCP layer packets may not arrive to the receiver in the order the sender transmitted them or in some cases some TCP fragments are overlapped. These packets are reordered in the correct way according to their initial sequence. The packets received are stripped from their TCP/IP headers and assembled into a single, cooked, packet. The cooked packet has an attached pseudo TCP/IP header whose only valid fields are the size of the cooked packet and session’s information (source/destination IP addresses and source/destination ports). A cooked packet is considered to be ready for processing under three cases:

- its size exceeds a specified threshold
- a timeout since the arrival of the first fragment is reached
- the session is closed (all fragments have arrived)

The size threshold is by default 32KB and the timeout is set to 30 sec. Cooking function is called through

`mapi_apply_function(fd,"COOKING",int size_threshold, int timeout)`

where size_threshold and timeout parameters set the size threshold and timeout respectively. In order to use the default value for a certain parameter, then user has to set its value to -1.
4.4.6 ETHEREAL - The Ethereal protocol filter

Ethereal is a free network protocol analyzer for Unix and Windows [23]. It allows the user to examine data from a live network or from a capture file on disk. The user can interactively browse the capture data and view summary and detailed information of each packet. Ethereal has several powerful features, including a rich display filter language and the ability to view the reconstructed stream of a TCP session.

The display filter language is used for the Ethereal filter in the MAPI. Display filters in Ethereal are very powerful. More fields are filterable in Ethereal than in any other protocol analyzer and the syntax to create filters is richer. Some examples of the Ethereal Display Filter syntax:

\[
\text{'tcp.port != 80 and http.request'}
\]
\[
\text{'eth.src[:4] == 00:00:83:00'}
\]
\[
\text{'tftp.source_file == C0A80014.SUN4M'}
\]

The Ethereal filter in MAPI has all the functionality of the Ethereal display filters. However, it does not include the capture filters, because these are BPF-filters and they are applied in the pcap-library. The Ethereal filters are implemented with a shared library, based on the Ethereal-sources. Because of the large number of supported protocols, this shared library is rather large (~13MB). The patch for the Ethereal-sources does not depend heavily on the Ethereal-sources itself, only the autoconf and automake files are adjusted, and 2 sourcefiles are added. That way, it shouldn’t be difficult to adapt the patch to future releases of Ethereal, as long as the program structure of Ethereal remains the same. The interface is very simple, only 2 functions are exported, one to initialize and compile the Ethereal filter and one to execute the filter on a packet. The Ethereal filter in MAPI uses one initialization parameter, a string containing the filter. The implementation checks every packet if it passes the filter, and drops them on failure. If successful, other functions after the Ethereal filter can process the packet, the Ethereal filter itself does not alter the packet or keeps information about it.

4.4.7 SAMPLE - Packet sampling

Probabilistic sampling

Probabilistic sampling function allows users to define the percentage of packets to be selected for further processing. Probabilistic sampling works as follows. User applies the function

\[
\text{mapi_apply_function}(fd,"SAMPLE_PACKETS",20,\text{PROBABILISTIC})
\]

The last argument defines the percentage of the packets to be accepted, in our example we want only the 20% of packets. Percentage cannot be negative or above one hundred. In case of incorrect arguments, probabilistic sampling functions maps
them in the nearest number limit, e.g. -1 becomes 0 or 120 becomes 100. In every call of the function, a random number between 0 and 100 is chosen. If this number is less or equal to the percentage defined by the user, then the packet is given for further processing, elsewhere processing of the proceeding functions of the flow is not committed.

**Periodic sampling**

This mode of sampling is deterministic. User defines the sampling period, for example setting the sampling period to 20 means that user wants one per twenty packets. Sampling period cannot be less than 1. Periodic sampling is applied as:

```c
mapi_apply_function(fd,"SAMPLE_PACKETS",20,PERIODIC)
```

In case of periodic sampling, a simple counter is incremented in each packet arriving to the function. If the counter reaches the sampling period, the packet is given for processing to the next function and the counter is reset to zero, elsewhere zero is returned indicating that packet will not be processed by proceeding functions of the flow.

**Hashing based sampling**

Hashing based sampling can be used to make sure that all packets of a random flow are captured. The hash is only calculated for TCP and UDP packets, all other packets are discarded. Based on the source IP, the destination IP, the source port and the destination port a hash value between 0 and 1024 is calculated. The user can specify which part of this hashing range must be sampled. The function can be used as a simple filter, with a packet capture function after the filter, or as a function returning packet data. The packet data returned by the function contains IP and part of the TCP or UDP header. Function prototype:

```c
mapi_apply_function(flow_descr,"SAMPLE_PACKETS",range,keep_results,HASH);
```

The range parameter is an integer between 0 and 1024, identifying the 0-range part of the hash result that must be sampled. Keep_results is a boolean(int) to enable the storage of packet data in a ringbuffer. Packet Data:

- source IP
- destination IP
- source port
- destination port
- protocol
- timestamp
- TCP Data
  - sequence
  - tcp flags

The application can read the results from the hashing based sampling with the mapi_read_results function. If keep_results was 0, no results are stored and this function will return NULL pointers.

### 4.4.8 STR_SEARCH - Pattern matcher

The pattern matching function is based on the Boyer-Moore algorithm. This algorithm tries to skip parts of the text that do not contain the pattern we search for. As a result, the number of comparisons needed is significantly reduced in contrast with the naive algorithm. In order to perform the skips, it builds two tables: the bad character and the good suffixes table. The bad character table works as follows: if the mismatching character appears in the search string, the search string is shifted so that the mismatching character is aligned with the rightmost position at which the mismatching character appears in the search string. If the mismatching character does not appear in the search string, the search string is shifted so that the first character of the pattern is one position past the mismatching character in the input. The second table, called the good suffixes table, is also triggered on a mismatch. If the mismatch occurs in the middle of the search string, then there is a non-empty suffix that matches. The heuristic then shifts the search string up to the next occurrence of the suffix in the string. The construction of bad character and good suffixes table is performed once during the preprocessing phase of the pattern matching function.

Pattern matching inside MAPI also permits searching for patterns containing non-printable characters. The pattern matching function supports a special notation that uses the “pipe” character (‘|’). For every character that cannot be typed, user encloses its hexadecimal value inside pipes. At the initialization phase of pattern matching functions, the pattern is parsed and hexadecimal values are converted to ASCII characters. For example, pattern “abc|12” searches for “abc” followed by the character with hexadecimal value 12. Multiple hexadecimal values can be enclosed inside pipes, e.g. “[00 00 00 00]” is a pattern of four consecutive null characters (spaces between hexadecimal values can be omitted). If user wants to search for the pipe character itself, then pipe character has to be escaped. For example, “ab\|” searches for “ab” followed by the pipe character. Only the slash and pipe characters can be escaped.
4.4.9 TO_FILE - Storing packets in a file

Export to tcpdump trace format

This packet export function creates a libpcap/tcpdump file from all remaining packets. The file can be used for offline processing of the packets, or viewing them in Ethereal, tcpdump,... In order to export all remaining packets to a tcpdump trace-file, the user can apply the following function to his configured flow.

```c
int mapi_apply_function(fd, "TO_FILE", MFF_RAW, "filename.cap", count);
```

Where `fd` is the flow descriptor, “TO_FILE” identifies the function that writes packets to a trace file, MFF_RAW defines the file format, “filename.cap” is the output file and `count` configures the number of packets that should be exported. The counter is used to limit the number of packets in the file. A counter value of 0 is used to give an infinite limit. Capturing is stopped by resetting the shared variable or when the counter-value is reached. The capture file is created after reaching the counter-value or on interruption by the shared variable. This is done in a separate thread, so the normal MAPI-processing is not interrupted. As usual, “mapi_apply_function” returns the identification value of the applied function.

In order to determine if all packets are already written to the tracefile, the user can call the “mapi_read_result” function.

```c
char result;
char mapi_read_result(fd, functID, &result);
```

Where `fd` is the flow descriptor and `functID` a reference to the “TO_FILE” function. Result will be zero if `count` packets are already exported, non-zero otherwise.

The IPFIX protocol for flow export

IPFIX is a protocol for flow export which is being standardized by the IETF. The standard has not been finalized, but it is clear that the record format will be Cisco NetFlow v 9 with minor changes. Unlike Netflow v 9, IPFIX will support reliable transport.

IPFIX has a notion of flow which is different from the MAPI flow. A MAPI flow is a set of packets satisfying some condition. The only condition is that it can be expressed in MAPI, e.g. as a BPF filter. An IPFIX/NetFlow flow, on the other hand, contains all packets that have a common 5- or 7-tuple of protocol number, source IP address, destination IP address, source port, and destination port. If input and output interfaces are known, these are also included in the tuple.

MAPI flows will often consist of multiple IPFIX flows. For these situations, MAPI lets the application programmer request statistics about the IPFIX flows in the form of IPFIX flow records.
Clients may collect IPFIX flow records directly over the MAPI using shared memory. One particular client is the Flow Record Exporter, which can distribute flow records to recipients like files or standard output as well as provide NetFlow/IPFIX export via the network.

Furthermore, the MAPI daemon is able to export flow records directly over UDP. MAPI provides flow records in IPFIX format only. Clients which need a different format must go via the Flow Record Exporter. The IPFIX functionality in MAPI is based on the nProbe program by Luca Deri. Here is how an application requests IPFIX flow record generation with default parameters.

```c
fd = mapi_create_flow("/dev/dag0");
funct_id = mapi_apply_function(fd, "FLOW_REPORT",
NULL, NULL, NULL, NULL);
for (;;) {
    mapi_read_results(fd, funct_id, &buffer);
}
```

IPFIX records will be returned in the buffer, which will contain the length of the IPFIX record in the first four bytes, followed by the record data. The apply_function call for flow reporting has the following parameters:

```c
mapi_apply_function(fd, "FLOW_REPORT",
record_type,
transport,
key_template,
record_template)
```

- record_type - "NETFLOW_V5", "NETFLOW_V9" or "IPFIX". Default is "IPFIX".
- transport - "SHMEM" - shared memory or "UDP". Default is "SHMEM".
- key_template - flow key specification. See below.
- record_template - flow record specification. See below.

**Flow key specification** The following properties may be used as part of the key for distinguishing flows:

- Incoming interface
- Outgoing interface
- Source IP address
- Destination IP address
SCAMPI Deliverable D2.3
Enhanced SCAMPI Implementation and Applications

- IP Protocol type (TCP, UDP, ICMP,...)
- IP version number (IPv4, IPv6)
- Source port number
- Destination port number

For source address and destination address, separating by full match is supported as well as separation by prefix match. Port numbers are only relevant for TCP and UDP flows. Interfaces are not relevant where SCAMPI is deployed as a probe. All meaningful combinations of the above may be used to distinguish IPFIX flows. The key_template parameter to mapi_apply_function is used to choose what to use.

**Flow record specification**  MAPI is able to include any combination of the following information fields in flow records returned to the client:

- Source IP address
- Destination IP address
- IP Protocol type (TCP, UDP, ICMP,...)
- IP version number (IPv4, IPv6)
- Source port number
- Destination port number
- Number of packets
- Number of bytes
- Type of service (IPv4)
- Traffic class octet (IPv6)
- Flow label (IPv6)
- Timestamp of the first packet of the flow
- Timestamp of the last packet of the flow
- Unique identifier of the observation point (the SCAMPI probe)
- ICMP type and code
- input interface
- output interface
SCAMPI Deliverable D2.3
Enhanced SCAMPI Implementation and Applications

- multicast replication factor
- average time between packet arrivals
- standard deviation of time between packet arrivals
- average packet size
- standard deviation of packet size
- a serial number

Input interface, output interface and multicast replication factor are not applicable where SCAMPI is deployed as a probe.

The client may choose to have all meaningful combinations of the above included in the flow records returned. The record_template parameter to mapi_apply_function is used choose what to include. MAPI only maintains those statistics which some client has requested. Thus, administrators may avoid doing costly computations in a high data rate situation.

When the set of information fields being monitored is increased, because a new client asks for fields which were so far not monitored, all existing sub-flows are expired.

**Implementation** The generator runs as the following threads:

- flow thread - processes arriving packets.
- export thread - exports record for completed flows.
- walkHash thread - scans the flow table for expired flows.

Access to shared resources is protected by mutexes. The flow table is a hash table. The hash function for IPV4 is

\[
hash = srcHost + dstHost + sport + dport
\]

For IPV6, all 32 bit groups of the source and destination addresses are added together. The flow record generator implements the standard mapidflib methods. The `process` method receives the packets which constitute the flow. The `client_read_result` method passes records to the client in shared memory.

**4.4.10 TO_BUFFER - Exporting packets to a buffer**

The TO_BUFFER function does not take any parameters and stores all packets in a buffer. Depending on the driver being used, packets are processed in different ways.
In the original approach, that can be used for all devices, a new buffer is allocated for each TO_BUFFER function. All packets in the flow are copied to this buffer and the application can read them from this buffer using a semaphore.

To avoid the cost of copying a packet received by a regular NIC multiple times (once for each flow that monitors the interface) we have introduced a common buffer where packets are being stored and is shared between the applications monitoring the same interface. This circular buffer (aka ring) is created by the MAPI driver handling regular NICs. The buffer is located on shared memory and read access is given to the effective group id of MAPId. Only applications that belong to this group id can read the packets in the buffer, thus protecting the data from 3rd parties. Applications access captured packets by applying function “TO_BUFFER” on a flow. On the first invocation of “TO_BUFFER” for a packet received on a regular NIC, the packet is being copied in the ring created by the NIC driver and its index in the buffer is stored for the following calls to “TO_BUFFER”. This index is pushed to a FIFO queue that is created for each “TO_BUFFER” instance in shared memory. Consecutive calls to “TO_BUFFER” regarding the same packet, push the previously stored index to their queues. The stored indexes are pulled by the mapi_get_next_pkt() call to retrieve “interesting” packets in the shared circular buffer and return a pointer to the packet data to the application. A packet will be dropped by a single flow when the queue of its “TO_BUFFER” function is full, or will be dropped for all associated flows when the shared circular buffer is full.

4.5 The MAPI Drivers

MAPI drivers are user-space components, which implement lower-layer driver-dependent part of MAPI. The purpose of MAPI drivers is to provide a device-independent interface, which can be used by the upper-layer part of the MAPI to open streams, get packets and apply functions to them in an uniform way, while packets are being captured by different hardware monitoring adapters. The upper-layer part of MAPI uses an extensive set of data structures and functions to implement application of functions on opened flows and data packets. MAPI drivers use kernel device drivers and function libraries supplied with hardware monitoring adapters. There is one MAPI driver for each type of hardware monitoring adapter supported by MAPI.

4.5.1 Support for loadable drivers

It is the support for loadable drivers that makes it possible to run the same application on top of different types of hardware without having to make any changes to it. A MAPI driver works in a similar way as kernel drivers do in that each drivers allow access to specific type of hardware through a well defined interface.

The defined interface of MAPI drivers has many similarities to the MAPI in-
The following calls are defined by the MAPI driver interface:

**mapidrv_add_device**  adds a new device that should be handled by this driver.

**Arguments**

- **char devname**  name of device.
- **int devid**  unique ID of the device

**mapidrv_create_flow**  creates a new flow.

**Arguments**

- **int devid**  device identifier. Specifies which device should be used to create the flow.
- **int fd**  flow descriptor. Unique identifier for the flow.
- **char **devoid**  the driver should store the devoid of the device in this argument.

**mapidrv_create_offline_flow**  creates a new offline flow

**Arguments**

- **int devid**  device identifier. Specifies which device(a file is treated as a device) should be used to create the flow.
- **int format**  specifies which format the file is in. speed=0 means the file is processed as quickly as possible. 1 means that the file is processed in the same speed as it was recorded.
- **int fd**  flow descriptor.
- **char **devoid**  the driver should store the devoid of the device in this argument.

**mapidrv_close_flow**  closes an open flow

**Arguments:**

- **int fd**  flow descriptor

**mapidrv_apply_function**  applies a new function to an existing flow
Arguments

`int fd`     flow descriptor.

`char function` name of function being applied.

`mapiFunctArg *fargs` string containing the function arguments

`mapidrv_read_results` returns results or information about results.

Arguments

`int fd`     flow descriptor.

`int fid` function ID.

`mapid result_t result` structure where results are stored.

`mapidrv_connect` connects to a flow. This tells the driver to start processing packets for this flow.

Arguments

`int fd`     flow descriptor

`mapidrv_get_flow_functions` returns a linked list containing the descriptions of functions that have been applied to a flow.

Arguments

`int fd`     flow descriptor

`int mapidrv_load_library` loads a new library

Arguments

`char *lib` name of library file that should be loaded

`mapidrv_unload_library` unloads a library. If functions from the library is in use by a flow, the library can not be unloaded.

Arguments

`char *lib` name of library

`mapidrv_get_errno` returns the last error code generated by a flow
Arguments

**int fd**  
flow descriptor

The only restrictions placed on MAPI drivers are through the defined interfaces. Drivers are therefore free to implement them as they see fit. Most drivers will however use the mapidlib to do most of the work. The mapidlib is a generic library that can be used by all drivers to keep track of open flows, loaded libraries and functions applied to flows. Many of the interfaces defined for a driver will therefore simply forward the call to mapidlib which will do all or most of the processing.

The main job of a driver is therefore to initialize a device and retrieve new packets from it. A driver is also responsible for retrieving packets from files. Each file format supported by MAPI is assigned a MAPI driver and that driver is then responsible for decoding this specific format.

To be able to handle both the capturing of packets as well as replying to commands from user applications, most drivers will use multi threading where a separate thread is created for reading packets from the device and send them to mapidlib for processing.

The configuration file is used to specify which devices and file formats that should be handled by which MAPI drivers. Only one instance of a driver is loaded so this driver must be able to handle multiple devices and file formats simultaneously.

### 4.5.2 The MAPI NIC driver

The NIC driver allows MAPI to be used on top of a generic NIC (Network Interface Card). In order not to tight the drivers on a specific NIC, MAPI has been developed on top of the libpcap (pcap stands for packet capture) library [17]. The Packet Capture library provides a high level interface to packet capture systems that is independent of the operating system. The advantage of using pcap is manifold and include:

- **Portability:** ability to run on both Unix (including the various flavors like MacOSX, Linux and FreeBSD) and Windows with a single code source tree.

- **API Transparency:** ability to transparently exploit platform-specific features (e.g. on Linux with some custom kernels it is possible to enable a memory-mapped facility for moving captured packets to user space at high speeds) when available.

- **Popularity:** the API is very popular in the network community, and there are several pcap flavors available that allow specific NICs to be used from pcap (e.g. the DAG [22] card manufactured by Endace) hence available to the driver.
The main drawback of pcap is that it is yet another layer between the NIC and the monitoring application. This means that there is a little performance loss due to pcap that is vastly compensated by the flexibility of its usage.

**Driver on top of libpcap**

The MAPI driver based on pcap is implemented into the mapinicdrv.c source file. This driver uses pcap for:

- opening an adapter and reading packets from a generic NIC
- reading packets from a dump file saved in tcpdump format

Once the driver receives the packet, the MAPI core as described earlier in this document handles it. This means that the interaction with the pcap library is limited to a few `pcap_xxx` (e.g. `pcap_open_live` that opens an adapter and `pcap_next` that fetches the next available packet from the NIC) calls that allow packets to be moved off the NIC to the MAPI core.

**Kernel-level ring buffer**

Passive packet capture is a costly activity on a generic PC, as it requires a tight interaction between the NIC/kernel and the monitoring applications running in user space. This means that unless an accelerated network card is used, the PC cannot capture traffic at wire speed on gigabit links. The result is a severe packet loss at high speeds or even at 100 Mbit when some traffic conditions are verified (e.g. under attack due to a worm/virus).

In order to improve packet capture at high speeds, the Linux kernel has been strongly modified. The result is a new socket type named PF_RING that works as follows.

The main network code of the Linux kernel has been modified so that packets received from an adapter bound to a PF_RING sockets are diverted from the normal journey and send to the ring Linux kernel module (figure 4.5). This means that incoming packets from these adapters are not received by applications unless they use a PF_RING socket. Once the ring module receives the packet, it is copied into each ring that has both an empty slot and that is bound to the same NIC where the incoming packet is coming from. User space applications read packets from the ring socket via a memory mapped area obtained via the `mmap()` system call.

In order to enable legacy applications based on the popular pcap library to take advantage of PF_RING, the libpcap library has been extended. Namely it has been enhanced so that if the ring module is available the pcap library uses it, if not the usual PF_PACKET Linux socket is used. The result is that passive monitoring applications or drivers (e.g. MAPI) can be compiled with the enhanced libpcap and take advantage of the PF_RING facility if available.
The speedup obtained with PF_RING is very large as it allows on a Pentium IV 1.7 GHz with a 32-bit Gigabit Ethernet card, to capture packets at over 550'000 pkt/sec that’s basically close to the theoretical hardware limit. Another advantage is that the PF_RING is very light, hence contrary to what happens with a vanilla Linux kernel, packet capture at high speeds does not require many CPU cycles. The outcome is that monitoring applications have plenty of CPU cycles available for traffic analysis. In the previous setup, a complex NetFlow probe can handle over 395’000 pkt/sec that is much more that any commercially available hardware NetFlow probe that will not go over 250’000 pkt/sec.

4.5.3 The MAPI driver for DAG cards

The MAPI DAG driver is a relatively simple driver. It currently do not support any of the new DAG cards with a co-processor, and all processing of packets is therefore done in software inside MAPId. The first time a flow is created, the MAPI DAG driver initializes the DAG card and starts a separate thread for reading packets from the device. The DAG driver has also implemented support for reading files in the DAG ERF format.

As part of the support for DAG cards, there has also been implemented a special version of the TO_FILE function which supports storing traces to disk using the DAG ERF format. This function is included in a separate DAG function library.

4.5.4 The MAPI driver for the SCAMPI adapter

MAPI driver for SCAMPI adapter (based on Combo6 mainboard) is similar to the MAPI driver for DAG adapter. While the MAPI driver for the DAG adapter
uses dagdrv kernel device driver and daglib library supplied with the DAG adapter, the MAPI driver for the SCAMPI adapter uses combo6drv kernel device driver and libscampi and libscampid libraries developed within the SCAMPI project. The former library opens and closes the device and retrieves packets. The later library merges requests on header filters, samplers and payload searches and submits the result to the SCAMPI adapter. Both libraries use the combo6drv kernel device driver. The libraries provide the following functions available to the programmer. They are used by MAPI implementation, but they can also be used by a standalone application, running directly on top of the SCAMPI adapter, its driver and libraries, when we do not require MAPI functionality.

`int scampi_open(char *name)` : Opens SCAMPI device and creates a new stream
  - name : device filename
  - return value : >=0, file descriptor; -1, error occurred

`int scampi_close(int fd)` : Closes a stream and SCAMPI device
  - return value : 0, ok; -1, error occurred

`int scampi_set_option(int fd, int option, void *value)` : Sets option of a stream
  - fd : file descriptor
  - option : name of option to be set (currently supported option is SCAMPI_LIB_FILTER, which sets packet header filter, programmers should preferably use scampi_set_header_filter() and related functions)
  - value : value of option to be set (e.g., BPF header filter)
  - return value : 0, ok; -1, error occurred

`int scampi_get_option(int fd, int option, void *value)` : Gets option of a stream
  - fd : file descriptor
  - option : name of option to be get
  - value : value of requested option
  - return value : 0, ok; -1, error occurred

`unsigned char *scampi_get_next_packet(int fd, int *size)` : Returns pointer to one new packet and locks this packet in the ring buffer
  - fd : file descriptor
  - size : size of new packet in bytes including Ethernet header
  - return value : !=NULL, pointer at the first byte of Ethernet header of new packet; NULL, error occurred

`void scampi_unlock_packets(int fd)` : Unlocks any packets locked on behalf of this application
  - fd : file descriptor

`int scampi_set_header_filter(int fd, char *header_filter)` : Sets header filter
  - fd : file descriptor
  - header_filter : BPF syntax of header filter
  - return value : >=0, header filter ID; -1, error occurred
int scampi_delete_header_filter(int fd, int filter_id) : Removes header filter
- *fd : file descriptor
- filter_id : header filter ID
- return value : 0, ok; -1, error occurred

int scampi_set_sampler(int fd, int sampler_type, int sampler_rate) : Sets sampler
- *fd : file descriptor
- sampler_type : NONE, DETERMINISTIC or PROBABILISTIC
- sampler_rate : one in this number of packets will be dropped (deterministically or probabilistically)
- return value : >=0, sampler ID; -1, error occurred

int scampi_delete_sampler(int fd, int sampler_id) : Removes sampler
- *fd : file descriptor
- sampler_id : sampler ID
- return value : 0, ok; -1, error occurred

int scampi_set_payload_filter(int fd, char *payload_filter[]) : Sets payload search
- *fd : file descriptor
- payload_filter : an array of strings to be searched on payload (packet matched when at least one of the strings was found)
- return value : >=0, payload search ID; -1, error occurred

int scampi_delete_payload_filter(int fd, int filter_id) : Removes payload search
- *fd : file descriptor
- filter_id : payload search ID
- return value : 0, ok; -1, error occurred

4.6 MAPI on network processors

The original aim of SCAMPI was to build a monitor capable of handling truly high speeds, by pushing as much of the processing to the lower levels of the processing hierarchy. Beware that in general the ‘lowest possible level’ only means the level that has the fewest intermediate layers beneath it in packet processing. Accordingly, if specialised hardware is able to deliver packets directly to the address space of an application (without intervention of the kernel), kernel and userspace are at the same level. This is the case, for instance, for DAG cards [8].

However, unlike the COMBO6 interface, the functionality of the DAG card is fixed. While it performs a fair amount of useful processing, we are currently not able to push more of the processing down to the hardware. Network processor interfaces, like the IXP1200-based cards, on the other hand, resemble the COMBO6 in that they are ‘programmable’. This section discusses how we exploited the programmability of the IXP1200 to provide a MAPI implementation capable of push-
ing packet processing functions to the lowest levels of the processing hierarchy. The implementation will be referred to as MAPI-X throughout this document.

4.6.1 IXP1200 overview

The IXP1200 contains a single StrongARM processor (running Linux) and 6 microengines (MEs) running no operating system whatsoever. Each ME supports 4 hardware contexts that have their own PCs and register sets (allowing it to switch between contexts at zero cycle overhead). Each ME runs its own code from a small (1K) instruction store. Figure 4.6 shows the main components of an architecture where a number of IXP1200 boards are plugged into the PCI backplane of a host.

Figure 4.6: Main components of the IXP1200

The IXP1200 evaluation boards are plugged into the PC’s PCI slots and each contain a single IXP1200, on-board DRAM and SRAM (256MB and 8MB respectively) and two Gigabit Ethernet ports. The boards drawn in Figure 4.6 are based on the Radisys ENP-2506 IXP1200 board.

MEs control the transfer of network packets to SDRAM in the following way. Ethernet frames are received at the MACs and transferred over the proprietary IX bus to an IXP buffer in 64 byte chunks, known as mpackets. From these buffers the mpackets can be transferred to SDRAM. MEs can subsequently read the packet data from SDRAM in their registers in order to process it.
4.6.2 FFPF overview

As outlined in D1.3, the design for MAPI-X is based on the design of FFPF (the fairly fast packet filter) that is developed at Leiden Universiteit and Vrije Universiteit Amsterdam\(^1\). As the interface of FFPF is more than equivalent in expressive power to the MAPI, it was not difficult to implement the MAPI on top of it. While FFPF and the ‘official’ implementation of the MAPI differ in many aspects (e.g., FFPF is currently a kernel-based approach, while MAPI is a user-space application), the main distinction is that the ‘official’ MAPI implementation is much more complex. None of this is important for the implementation of MAPI-X, however, as the only criterion here is whether the MAPI can be expressed in terms of the FFPF API.

The original FFPF design consists of a Linux kernel module that is responsible for receiving packets and passing them to userspace without copying. To this end, FFPF employs a single, shared packet buffer known as \(\text{PBuf}\) in which all packets are stored that are of interest to at least one of the applications in a group. A group is a collection of applications with the same access right to packets. As there is no notion of groups in SCAMPI, we will ignore this aspect and assume all applications are in the same group. FFPF maps the packet buffer to the address space of the applications in the group, so that (in principle) each application can access all the packets in this buffer. In addition, for every MAPI-like flow, FFPF maps another circular buffer to the address space of each application which contains pointers into the shared packet buffer. This buffer, known as ‘index buffer’, or \(\text{IBuf}\), is not shared. If the packet matches a flow-specific expression (e.g., a filter), FFPF pushes the packet in \(\text{PBuf}\) and enters the index in the flow’s \(\text{IBuf}\). If another flow is also interested in this packet, instead of copying the packet, FFPF just adds the packet index to the other flow’s \(\text{IBuf}\). There exists a third flow-specific buffer, a byte array known as \(\text{ABuf}\), that is shared between kernel and application. It is used as a generic information buffer which can be used by the application and the expression (really a simple program) operating in the kernel on its behalf, as they see fit (e.g. to report results).

FFPF supports several different ways of managing its circular buffers \(\text{PBuf}\) and \(\text{IBuf}\). In this section, we only mention the one that corresponds to the buffer management in Scampi. In this case, the buffers are controlled by read and write (R and W) pointers, where the W pointer is handled by the system and the R pointer is explicitly advanced by the reader (see D1.3 for details). In practice, this means that the rate at which packets can be processed is determined by the slowest reader.

4.6.3 An implementation of MAPI on IXP1200s

Given an implementation of MAPI on top of FFPF, we now discuss how MAPI-X support was added for the IXP1200. The FFPF software structure is modular and allows for different ‘hooks’ that provide it with packets. For instance, the default

\(^1\)See http://ffpf.sourceforge.net/
Linux implementation provides a netfilter hook as well as a net_if_rx hook (where the latter is slightly faster as it grabs packets at an earlier stage). Support for the IXP1200 was provided by adding a third hook known as the i xp hook. It is responsible for handling the packets that have been received by the IXP1200-based network card. Compared to a regular NIC, or a DAG card, the IXP1200 configuration is different in that MAPI functions (e.g., filters) can be installed on the network card itself.

In order to gain high speed processing, we implemented true zero-copy handling of network traffic up to user space. The programming language used for the MEs was Intel’s microengine C on the IXP network processor, which provides a high-level structured programming method, and yet is efficient enough to process data at line speed. Exploiting as much as possible the hardware support on IXP1200 platforms for fast path processing while minimising communication between micro-engines and the core (StrongARM processor) as well with the host processor, proved to be a key factor for the successful implementation. The following sections will cover the system implementation in detail.

A single microengine is responsible for receiving packets from the network ports and storing them in PBuf. Should a single microengine be insufficient, more than one can be dedicated to this task. All remaining microengines execute application code. In the case of MAPI-X FFPF, these microengines correspond to high-speed MAPI flows at a higher level.

On each of the microengines a main loop is provided by the application framework. Given the appropriate privileges (determined by admission control, described elsewhere in this document), programmers may ‘plug in’ application code such as compiled FFPF expressions in this loop and load the complete program on the microengine.

Packet reception

In the MAPI-X architecture, one ME with four threads is assigned per Gigabit port for receiving packets and storing them in the shared packet buffer. In the absence of additional processing (i.e., with a ‘NULL-filter’), MAPI-X is capable of receiving packets at line rate. When a packet arrives at the MAC, an infinite loop in the receive threads will transfer packets from MAC buffer to the shared packet buffer. Processing takes place as follows.

First, threads have to allocate a free slot in the packet descriptor queue by issuing a pop instruction from the SRAM FIFO described in the next section. If there is no available slot, the packet will be dropped and a counter of dropped packets will be increased. Since the IXP1200 external data bus, the IX bus, segments all incoming packets into 64 byte mpackets, each thread will process one mpacket at a time until the entire packet is placed into a packet buffer slot. The IXP hardware defines Gigabit ports as fast ports by removing serialization restrictions in the receiving process. That means the receive threads have to issue speculative receive requests to the IX bus to check the availability of incoming packets. Unlike the case with
slow ports (fast Ethernet), these requests are not blocked until a new mpacket is
detected in the MAC. Instead, if there is no packet waiting to transfer in the MAC,
a special message will be returned to the thread that sent the request. In this case,
the thread simply ignores the request and continues with the loop.

The buffers

in MAPI-X, PBuf is a large shared memory area residing in the IXP board’s
SDRAM in which all packets are stored that are of interest to at least one of the
applications. The buffer is divided into 2KB slots so that the largest Ethernet data-
gram of 1500 byte can fit in one slot\(^2\). A shared buffer located in SRAM, known as
packet descriptor queue, contains one packet descriptor structure for a correspond-
ing slot in the packet buffer:

```c
typedef struct {
    unsigned int next_buff:16; // pointer to the next de-
scriptor in the queue
    unsigned int buff_size:16; // actual size of the packet in the slot
    unsigned int buff_flag; // additional information for this slot
} buffer_queue_entry_t;
```

The bitmap buff_flag field indicates which flows are interested in this
packet. The reason to implement a packet queue descriptor instead of packet buffer
alone is to exploit the hardware-supported SRAM FIFO operations and avoid com-
plex buffer allocation (and freeing) tasks at runtime. Since both the packet buffer
and the packet descriptor queues are contiguous memory areas, mapping between
a descriptor in SRAM and the corresponding packet slot in SDRAM is straightfor-
ward.

As mentioned earlier, each flow also has an IBuf, a circular buffer containing
pointers to the packet slots in which the flow is interested. Each flow is bound to
a filtering ME which scans the packet buffer and fills up this circular buffer. Dur-
ing this process, the W pointer is updated, pointing to the last available packet in
the flow. On the other hand, the user-space application will update the R pointer
whenever it finished processing packets in the flow. Note that tying a flow to an
ME, means that a maximum of five flows can be directly supported by the hard-
ware, as at least one ME is dedicated to packet reception.

For clarity, we ow show the structure that is used for indexing the buffers. The
R pointer is denoted by producer, while an R pointer is called consumer:

```c
typedef unsigned long long ULL;
typedef struct {
    ULL consumer;        // R pointer
    ULL producer;        // W pointer
    ULL handles[SRAM_QUEUE_SIZE]; // packet indices
} flow_t;
```

\(^2\)Currently, there is no support fo Jumbo frames
The flow-specific ABuf buffers are simple chunks of SDRAM shared among filter MEs, StrongARM core modules and/or user applications to store flow specific information.

Filter/function implementation

In the current MAPI-X design a single ME is dedicated to a flow, using all four threads to process received packets one by one. If the packet matches the filter criteria, the ME places a packet descriptor in the IBuf of this flow. For the IXP1200 this means that a maximum of five flows can be supported at the same time. If a packet is not classified as interesting to the flow, a special function known as pkt_drop() is called. This function will mark this packet as finished by setting the appropriate bit in the flag of the packet descriptor and checks if all other MEs also finished processing this packet. If this is the case, the packet descriptor will be pushed back to SRAM FIFO, saving space for new incoming packets. The actual filter expression is placed in a function known as filter_impl() which is stored in a separate file filter.c. An example of a filter function in pseudo-code is given below. In this example, first some information is extracted from a packet. Here we have just used a few fields for illustration purposes. Eventually, the sample code accepts all TCP packets that are sent to TEST_PORT.

```c
// example of filter_impl: user code may be plugged in the marked area
int filter_impl(flow_t *flow, buffer_handle_t curbuf) {
    UINT pkt_buf_addr;
    UINT ip_verslen, tcp_src_port, tcp_dest_port;
    UINT ip_saddr, ip_daddr;
    UINT temp_ether_header[4]; //temporary buffer
    UINT temp_ipv4_header[8]; //temporary buffer
    struct ixp_ethernet_header ether_header;
    struct ixp_ip_header ipv4_header;
    buffer_queue_entry_t *qentry = (buffer_queue_entry_t *)curbuf;

    // find the appropriate buffer
    pkt_buf_addr = (UINT)buffer_get_data_ptr(curbuf);

    // read link layer, 1st 2 quadwords and context switch until we have them
    sram_read(temp_ether_header, (void *)pkt_buf_addr, QWCOUNT2, 1, ctx_swap);

    // extract the ethernet header
    extract_ether_param(&temp_ether_header[0], &ether_header);

    // if NOT 802.3(see RFC 1122, 2.3.3) && NOT Arp Packets send to IP Stack (rfc1812 3.3.2)
    if(ether_header.protocol_length > ETHER_MTU && ether_header.protocol_length != ETHER_ARP) {
        // get version and header length
        ip_verslen = ixp_bytefield_extract(temp_ether_header[3], 2, 2);

        if (ip_verslen == IPV4_NO_OPTIONS) // if ipv4 no options (see RFC 791)
            { sram_read(temp_ipv4_header, (void*)pkt_buf_addr+QWOFFSET1), QWCOUNT4, 1, ctx_swap); // extract IPv4 header
    }

    // extract some more information (not used - for illustration purposes only)
}
```

scampi@ist-scampi.org 19th April 2004
ip_saddr = get_ip_source_addr (...);
ip_daddr = get_ip_dest_addr (...);

// treat TCP and UDP separately
if (is_tcp ( ...) )
{
  // get TCP ports
tcp_src_port = ixp_bytefield_extract(temp_ipv4_header[7], 0, 1);
tcp_dest_port = ixp_bytefield_extract(temp_ipv4_header[7], 0, 1);

  // we have extracted some relevant info now decide whether we
  // want the packet (i.e. execute the ‘real filter’)
  // Check is_pkt_interesting (i.e. is it sent to the TEST_PORT)
  if (tcp_dest_port == TEST_PORT) flow_enqueue(flow, curbuf);
  else pkt_drop(qentry, UENGINE_ID);
  
}
else if (is_udp(...))
{
  ....
}
else // IP with options or frag: not supported
{
  pkt_drop(qentry, UENGINE_ID);
  // IP with options or frag
}
else // NOT 802.3 and Arp packets
{
  pkt_drop(qentry, UENGINE_ID);
}
return 0;

In principle, applications may define the body of filter_impl themselves, provided they have the appropriate privileges. In practice, however, this process is too complex for all but a handful of power users. Instead, they will opt to use a template and write their filters in a high-level filtering language. A simple, yet powerful language that was recently defined at Leiden Universiteit and Vrije Universiteit Amsterdam for this purpose, is known as FPL-2 (FFPF Programming Language 2). A compiler which translates FPL-2 to C is described in [10].

Using the above template-based approach, the code produced by the FFPF compiler can be easily linked into the main loop of filter threads. The compiler needs to take the peculiarities of the IXP1200 platform into account in order to hide them from the user. For instance, since the packets are stored in SDRAM, to apply filters, packet headers must be read into SDRAM read registers before processing (which is the responsibility of the MEs themselves). Mutual exclusion among threads is handled by using a shared de-queue structure:

/* Dequeue state */
typedef struct _deq_state_t {
  unsigned int valid : 1; /* The state is valid (or not in use) */
  unsigned int discard : 1; /* Entire packet is bad */
}
buffer_handle_t buf;
} deq_state_t;

//shared state among 4 threads
volatile __declspec(shared) deq_state_t deq_state = {0};

The main loop on each thread waits for the valid bit to become true and only then starts processing it.

**StrongARM core components**

The StrongARM (SA) components are responsible for control tasks, including initialization and control of MEs, programming the FBI (the unit responsible for controlling the internal bus of the IXP connected to the MAC ports), memory mapping of SDRAM, SRAM and ScratchPad to StrongARM Linux and the host, initialization of different memory buffers, etc. Its main functionality is contained in a kernel module known as **ffpf.o**.

**Memory mapping.**

Different types of memory and registers on the IXP1200 are mapped directly to and can be accessed from the SA core. This is supported by the Linux kernel patched for SA cores and customized for IXP1200 chips. For safety, FFPF makes only specific areas of the board’s SDRAM aredirectly accessible to the user applications on the host.

**Programming the FBI**

In the FBI, there is a programmable unit called the ready-bus sequencer, with the following features: a 12-entry instruction store, with nine instruction categories, and a configurable maximum execution rate. The ready-bus sequencer probes, and sometimes controls, external devices on the IX bus. For example, it is the ready-bus sequencer that polls the MACs for information on which ports have data available for reception and transmission. The ready-bus sequencer updates the receive-ready and transmit-ready bits based on this information. Programming the ready-bus sequencer is a one-time task during the course of a project. Once enabled, the ready-bus sequencer executes all 12 of its instructions, in order, then potentially delays for a configurable amount of time, and finally repeats the entire process. In MAPI-X, to reduce the possibility of reading stale receive-ready bits, and to speed up the transmit process, we set the ready-bus sequencer to update the receive-ready and transmit-ready bits as fast as possible. This is done by setting the ready-bus sequencer to poll the receive-ready and transmit-ready bits, and then executes ten invalid so-called **FLWCTL** instructions. The result is that the receive-ready and transmit-ready bits are updated as fast as possible.

**Loading images and starting MEs**

The code for uEs in this project was written in the microengine C lan-
guage for IXP1200 processors with support of Intel Developer Workbench SDK 2.1. Normally, the programmer decides at compile time which portion of the code will run on which ME. The entire code will then be compiled and built into a single image file (known as uof file), to be used for all six MEs. However, by excluding certain MEs as target, they are unaffected by any subsequent ‘load’ operation. In other words, it is possible to generate code for a single ME, which is loaded without affecting existing code on other MEs.

MAPI-X support code in the SA core has to load this image into MEs and then instruct the MEs to start running the code. As described in previous sections, packet receiving from MACs is done separately by one ME per gigabit port. This code has to be loaded and started at start-up time. Whenever a user wants to create a flow and enter a filter, (s)he must supply MAPI-X with an appropriate uof image file. The process of generating a uof image can be described briefly as follows. A user writes his filter expressions in an appropriate language (e.g. BPF, or FFPF’s FPL1 or FPL2), i.e. a language supported by a MAPI-X FFPF compiler for IXP. The compiler generates microengine C code from filter expressions to a filter.c file. This code is then compiled and built into a user.uof image from filter.c and flow.c (containing MAPI-X support code) using the IXP SDK.

On the SA or the host, the user then issues an ioctl call to the ffpf.o kernel module with user.uof as a parameter. The ffpf.o module will load user.uof, restart the relevant ME(s) and return to the user a flow description structure of the circular buffer that stores packets that matched the filter.

User applications

User applications in MAPI-X FFPF can run both on the SA core and on the host computer. In the former case, because there is no communication via the PCI bus, the application can keep up with very high network rates. If the application is run on the host computer, the speed of packet processing will depend strongly on the PCI bus communication. As a result, a high-end user must be very careful in designing the data flow via the PCI bus. However, in both cases, the structure of the user application will be identical regarding buffer management and packet handling.

Note that in the case of applications on the host, the zero-copy MAPI-X implementation is only efficient if the packet is not touched often by the userspace application. Otherwise, the read operations over the PCI bus will start to dominate performance. For this reason, we have also implemented a version in which ‘interesting’ packets are explicitly sent to the host processor across the PCI bus as pseudo-Ethernet packets.

After processing a packet as required, the application has to free up its buffers in the Shared Packet Buffer to create room for new packets. For
this purpose, the application uses support code that sets its DONE bit in the flag field of a packet descriptor and then checks if all other flows finished using this packet. If this is the case, it will issue an `SRAM_PUSH` instruction to push the packet descriptor back to SRAM FIFO. This instruction is supported by hardware for both MEs and SA core, making the implementation fairly straight-forward.

**Communication to the host via the PCI bus**

In MAPI-X, communication via the PCI bus between the IXP1200 board and the host computer is done by memory mapping the SDRAM memory of the board to the host. This is a rather efficient method since IXP1200 supports sophisticated DMA channels which can move data from SDRAM to the PCI and vice versa. These channels can be accessed directly from MEs and SA core using DMA registers. With minimal communication across the PCI bus, memory mapping is sufficiently fast to handle high speed packet processing. To support memory mapping, we have patched the original PCI driver for the IXP as provided by Intel. From the host computer, in order to access the SDRAM memory area of the IXP, a user application has to open the device first and then issue a `mmap` command.

With the support of SDRAM mapping to the host, a polling method is used to communicate between applications in the host and applications or kernel modules in the SA core. The application in the host will poll the R and W pointers of its flow to check if there are packets in the flow. After processing the packets, the application will instruct modules in the SA core to free up those buffers by updating the R pointer of the flow. Meanwhile, an infinite loop in the MAPI-X FFPF module in the SA is polling the R pointer to free up used buffers and updating the W pointer of the corresponding flow.

### 4.6.4 Preliminary results

Preliminary results show that MAPI-X is able to keep up with speeds higher than 40 kpps, for packet sizes from 64 byte onwards. We are not sure what the highest rate is that can be sustained by this configuration as we currently do not have a sufficiently accurate traffic generator.
Chapter 5

Admission and Resource Control

Admission control is implemented as a daemon process known as Authd. The resource and access control daemon itself has not changed in a significant manner, since D2.2. The only major modifications include the front-end to Authd, as discussed in Section 5.3. In addition, we have now fully implemented accounting of resources to avoid exhausting the resource capacity of the platform (Section 5.4). For example, if a function takes up a certain amount of buffer space and CPU time, the total number of times this function can be applied needs to be less than the total resource capacity of the system. All of the changes to Authd are implemented and integrated with MAPId.

5.1 Admission control overview

Whenever a user wants to create a SCAMPI ‘flow’ and apply various functions to it (e.g., samplers, string searches, etc.), the SCAMPI admission control checks whether the user has the appropriate privileges to create this flow with this specific combination of functions and options applied to it. For example, while certain users may not have the right to create a flow that receives all traffic with a string search algorithm applied to every packet, he/she may be granted the right to create such a flow provided the string search algorithm is preceded by a sampler that samples just 10% of the packets. The example shows that admission control, when taking a decision, should take the entire flow specification into account. This way, it is possible to specify that a certain combination of functions is allowed, while each of the constituent functions in isolation is not.

For clarity’s sake, the major steps of the entire admission/resource control processed are again shown in Figure 5.1. The information required to authorize a flow that a user wants to have connected include:

- the user’s public key;
- the user’s credentials (using the Keynote format [3]);
The device name and the list of the applied functions are used to generate assertions about the flow. Such assertions cover for example the type of the functions applied, the number of instances of a specific function type, the position of the instances (in the linear list of applied functions) and the arguments passed to them. Credentials that are provided by an authority specify to what rules such assertion should conform. Example of the types of assertion that may be expressed in credentials can be found in D1.3 and are explained further in Appendix B. We now explain the various steps of Figure 5.1.

The first thing a user needs to do is tell the system what his/her credentials are, and what public key can be used to identify this user (indicated by '1' in Figure 5.1). This only needs to be done once per application. Whenever the user wants to instantiate a flow, he/she also has to provide authentication to show that he/she really is the user that corresponds to the public key. The way this is done is by supplying a nonce value (an integer number), together with an encryption of this integer with the user’s private key. If the encrypted nonce decrypted with the public key equals the original value of the nonce, the user’s request is authenticated. By using the flow descriptor as the nonce value, the authentication is also tied to this particular request.
The user creates a flow (indicated by '2') and if successful a flow descriptor corresponding to the specified device is returned. The flow descriptor is used in all subsequent calls, e.g. to apply functions to the flow (indicated by '3'). The functions to be applied to the flow form a linked list of functions (each possibly having a linked list of parameters). The order in the list indicates the order in which the functions are to be applied.

Only when all the functions that a user wants to specify for this flow are applied, does the user call connect_flow(). This results in the following actions (number '4' in the figure):

1. An admission control request structure is created for the Authorisation of the flow. The list of functions of the flow are serialized and stored in the request structure along with the flow’s authorisation information and some extra parameters (e.g. the application domain and device name).

2. The serialised request structure is sent by MAPId to the authorisation component (Authd).

3. Authd authenticates the flow (through the encrypted nonce challenge). If authentication failed jumps to 6.

4. Authd processes the request structure and extracts information regarding:
   - the occurrence and order in which functions are applied;
   - the value of parameters that are passed to the functions;
   - the resource consumption.

5. The information extracted in (4), together with the credentials, is then evaluated for compliance with Authd’s security policy.

6. An admission control result structure is returned to MAPId containing information about: (a) whether the flow was accepted or rejected, (b) reason for rejection, and (c) resource requirements

7. If the request was accepted by Authd MAPId will instantiate the flow and reserve the resources (number ‘5’ in Figure 5.1.

The admission control result structure is as follows:

```c
struct adm_ctrl_result
{
    int PCV;          //!< The compliance value of a flow, as returned by keynote
    int error;        //!< Error feedback
    size_t resources_num; //!< Number of required resources
    resource_required_t required[RESOURCE_CTRL_MAX_RESOURCES]; //!< Required resources array
};
```

scampi@ist-scampi.org 19th April 2004
The structure that is used to contain all information necessary for admission is shown below:

```c
//! Admission control request structure
/** Contains all the required data for a client’s request to be authenticated and authorized. */
struct adm_ctrl_request
{
   //! Public key
   unsigned char pubkey[MAX_PUBKEY_SIZE];
   //! Credentials
   unsigned char credentials[MAX_CREDENTIALS_SIZE];
   //! Random number provided by mapi
   unsigned int nonce;
   //! Encrypted nonce with private key
   unsigned char encrypted_nonce[MAX_ENC_NONCE_SIZE];
   //! The length of the encrypted nonce
   size_t encrypted_nonce_len;
   //! The number of name-value pair assertions
   unsigned int pairs_num;
   //! The name-value pair assertions
   adm_ctrl_pair_t pair_assertions[MAX_PAIR_ASSERTIONS];
   //! The number of the serialized functions in the buffer
   unsigned int functions_num;
   //! The buffer with the serialized function list
   unsigned char function_list[MAX_FUNCTION_LIST_SIZE];
};
```

## 5.2 Configuration

The configuration of Authd is currently stored in a configuration file. All the variables are located in `config.h`. The sizes defined for keys and strings in `config.h` should be taken in mind when testing, and set appropriately. Also the location of the MAPld keys and policy file is defined in `config.h` and is by default set to `/etc/.scampi` The filenames are respectively "authd_key.pub", "authd_key.priv" and "policy". The location of resource control DB files is defined to be in `/etc/scampi/resourcectrl` and the main database file named "resource.db".

## 5.3 The front-end

Originally, the communication between MAPld and Authd could only be performed using shared memory. While the shared memory option is still supported, it is also possible to connect to Authd using other communication primitives. For this purpose, the back-end of the daemon (where authorisation checking and resource calculation are performed) is fully decoupled from the front-end, which takes care
of communication. The design is modular and replacing a communication package with another one is trivial, and we have implemented several such modules in the current distribution.

In particular we have implemented a solution that allows one to connect to Authd using sockets and OpenSSL \(^1\). The implementation provides the following features:

- Connect to Authd from anywhere;
- All communication across the socket is encrypted;
- Client authentication;
- Server authentication.

The above features allow Authd to be used in a wider context than just on a single SCAMPI monitoring node. For instance, it is possible that an organisation installs a single authorisation daemon for an entire network. Moreover, the daemon is fully stand-alone and can be used without modification by projects other than SCAMPI as well. In fact, Authd provides a generic, application-neutral authorisation and admission control solution that can be applied to any project in need of such a facility.

### 5.4 Resource control

The meaning of ‘resource control’ has changed during the lifetime of the SCAMPI project. Originally, it was intended to be strict resource control, enforced by the Open Kernel Environment. Since moving our code out of the kernel, however, we have taken a more admission-control based view on resource. Whenever a user tries to instantiate a flow, Authd checks the resources that will be consumed by all the functions (filters, samplers, counters, etc.) that constitute the flow. Upon completion, it will return a resource-consumption report to MAPId. It is the responsibility of MAPId to decide whether or not the flow should be admitted. The reason for placing the responsibility at the side of MAPId is that it facilitates variation in enforcement policies. For instance, one site may decide to enforce strict, conservative resource control, while another does not enforce resource control at all and simply registers the resource consumption.

In a database, implemented using Berkeley DB, we store formulas about the resource consumption\(^2\). The formula language is able to distinguish between different resources (e.g. CPU, host memory, memory on the network card, etc.) and

\(^1\)OpenSSL is an open source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library, see http://www.openssl.org/

\(^2\)Berkeley DB is a free, simple and fast database approach (see http://www.sleepycat.com/).
may use a function’s parameters to calculate variable resource consumption values. For instance, a checksum function may take CPU time proportional to the area it needs to checksum.

In Authd the resource consumption of a function for a specific is handled as 3-tuple, consisting of the a key into the resource database (identifying the resource) a value for the fixed cost of this function for this resource, and a variable_cost_formula that is able to calculate the variable cost, depending for instance on the function parameters. The struct used for this purpose is shown below.

```c
/// Resource consumption of a function in a specific library
struct resource_consumption
{
    u_int32_t rkey; //!< Referred resource key
    u_int32_t fixed_cost; //!< Fixed cost of function
    // Formula string that calculates the resource consumption depending
    // on function parameters
    // char variable_cost_formula[RESOURCE_CTRL_MAX_VAR_FORM_LEN];
};
```

The variable cost formula is a string following printf’s format for inserting function parameters values into an arithmetic expression. E.g. in the formula "10 + %1$d * 5 + %3$llu / 2". "%1$d" represents the first parameter of the function which is treated as an integer and "%3$llu" the third argument of the function which is treated as an unsigned long long. The arithmetic expression that results after the replacement of the function’s parameters in the formula is evaluated to the variable cost of the function instance. Special care needs to be taken when writing these formulas, since invalid parameter type specification could unpredictable application behaviour.

We also store the available capacity of the resource: It is realised that it is not always possible to arrive at an accurate estimate for the resource consumption, but we do believe that we have catered to approximately all common cases. A failure by the admission control daemon, may lead to overcommitted resources. On the other hand, we believe this is an acceptable price that is paid for the new view within the project on resource control.
Chapter 6

SCAMPI Error handling

Because all components in the SCAMPI system communicate mainly throughout APIs (or IPCs), error codes can be returned as the return value of a function. All error messages in SCAMPI will be handled and processed in a unified way. When some operation (either a function, library, module or driver) in the SCAMPI system fails and generates an error, this error will be returned to the component that executed the operation. Based on the error code, the component knows why the operation failed. In fact, the error code can be interpreted by every other component in the system.

Every component in the system that can fail and generate errors has a unique set of error codes assigned to it. Upon failure, one of these error codes is returned to the previous component, i.e. the component that called the routine. In the worst case scenario, the error code is propagated towards the application layer (MAPI). The MAPI interprets these error codes and assigns a human-readable error message to the code, by calling the error library. This error library is centralized and shared between all components. (Depending on the different languages supported, the appropriate localization library is used.)

The different components in the SCAMPI system that can generate errors are:

1. The hardware driver (COMBO6 driver, DAG driver and NIC kernel modules)
2. A hardware specific Library (scampilib, daglib)
3. A hardware specific MAPI (mapinidrv, mapcombo6drv, mapidagdrv)
4. The MAPId library (mapidlib)
5. The admission/resource control module (admcrtl)
6. The MAPId communication interface (mapiipc)
7. The MAPI interface

1Currently, only english is supported.
All these components have a unique set of error codes that represent a certain error and are equal for all "instances" of this component. The scampilib for example will generate errors with the same semantics as the daglib. This way, independent of the underlying hardware or driver, errors are interpreted in the same way. Because components at a same level (e.g. NIC, DAG and COMBO6 driver) have different functionality, they cannot share exactly the same error semantics. Some of the errors in their set will be shared, while others are unique for a certain component. If a component receives an error from another component, it can interpret this error and handle it, or it can propagate this error to the upper layers. The range of all errors that can be generated by a certain component is statically determined. If a component receives an error from the component it called, this error can either be “thrown” by the component itself (in the case the error is in the error scope of the component), or by another component (if the error is not in the error scope of the component). If the scampilib for example returns an error code to the mapicombo6drv, the mapicombo6drv can return the exact same error code to the mapidlib. Based on the scope of the error code, the mapidlib knows the error code was not generated by the mapicombo6drv, but by the scampilib. The scopes is defined as (depends heavily on the amount of errors needed by the different layers):

1. No error in operation (code 0)
2. Hardware driver (code 0001-1024)
3. Hardware specific Libraries (code 1025-2048)
   - mapinicdrv specific (code 1100-1199)
   - mapicombo6drv specific (code 1200-1299)
   - mapidagdrv specific (code 1300-1399)
4. Hardware specific MAPI (code 2048-3072)
5. mapidlib (code 3073-4096)
6. Resource control (code 4097-5120)
7. mapidcom (code 5121-6144)
8. MAPI (code 6145-7168)

If a component receives an error from another component, it will execute one of the following operations:

1. Neglect the error and continue processing (e.g. in case of non-critical errors)
2. Do some error-specific or global error processing and continue its operation
3. Quit its operation and propagate the same error to its caller
4. Quit its operation and return a new error in its range to its caller (e.g. an aggregated error)

5. Handle the error by cleaning up some initialization and propagate the error to its caller

6. Handle the error by cleaning up some initialization and return a new (aggregated) error to its caller

When a MAPI operation returns a valid state (a flow descriptor $\geq 0$, a pointer $\neq$ NULL,...), the function has succeeded. When the return value is invalid (-1 for integer-returning functions, NULL for pointer-returning functions), an error (or multiple errors) occurred. The interpretation of the different errors will be done statically. A library with the translation of each error to a human-readable error message is defined. Because error-processing IO is not that time critical, all translations will be listed in a flat file that needs to be parsed each time the library is called. This file has the following layout:

```plaintext
ERRORCODE1 = "Adapter out of memory"
ERRORCODE2 = "Unknown operation"
...
```

The error-library can be used by any component that needs to do this translation, for debugging purposes for example. If required these messages can be translated in any language, by including the appropriate localization file. If a component needs to know what kind of failure caused the error in order to do some processing and handle the error, the programmer of the component needs to check the list of all errors that can occur and manually code, if needed, the error handling in the component. In most cases, a component will notice that an error was thrown and propagate the error to the application without any further error-specific processing. Once the error reaches the application, the application can call the error library to print a human-readable error message.

The implementation uses different error-schemes. To the user of the MAPI, every function returns a valid state, or an error state. If a function returns an error, the application can request the errorcode and translated message from the MAPI, with a function call to:

```c
int mapi_read_error (int* err_no, char* errorstr);
```

This function has 2 parameters, and will fill `err_no` with the error-code, and `errorstr` with the error-message. Both pointers should be initialized and allocated large enough. In the MAPI Daemon, all error states are stored in a linked list, where each clientID that has an error, is assigned to its error. Every time a new error is reported, the old error-code is removed, and when an error is read, the error state is cleared. All MAPId functions return with the error-code if an error occurred, or 0 when no error occurred. The mapidlib functions report errors in two different ways to the MAPI Daemon.
1. Functions with no return value: These functions will return an integer, reporting their error state. This value will be stored by the MAPId in the errorfield assigned to the client.

2. Functions returning some value: These functions will return an invalid value for the return value, and the error will be stored inside mapidlib and is assigned to the flow the function was used for. The MAPId will read this value and store it again in the errorfield assigned to the client.

When implementing a new function or driver in the SCAMPI system, the following error-handling issues should be considered:

- Packet-processing functions: The init-part of these functions returns an integer, reporting their error. If this value is positive (i.e. non-zero), the mapidlib will stop adding this function and will release all resources assigned to it. The same will happen in the MAPId, and the error will be reported to the client. The processing function itself can’t report errors, this error-handling could interfere with the normal packet-processing, so all resources should be allocated before the function is used.

- Additional drivers: All driver functions return integer values reporting their errors to the MAPId. On error, they should release all allocated resources and report the error. In addition, the drivers need one error-reporting function. This function reads the error-value assigned to the flow from the MAPI Daemon.

```c
int mapidrv_get_errno(int fd)
{
    return mapid_get_errno(fd);
}
```

Errors occurring inside the MAPI at the application side can not be processed in the same way as errors occurring in the MAPI Daemon. When the client application translates an error reported to it by the MAPI, and the local MAPI interface generated an error, the error will be sent to the MAPI Daemon for translation and the current error-state in the MAPId for that client is cleared. The translated message is returned to the client.

The translation itself reads the flat errorfile and returns the error string. If the errorcode has no error string assigned, it will return “Error not found”. The error file is generated from another file, with this structure:

```c
#define ERROR_NAME 1000
1000 = "Error in human readable form"
...
```

scampi@ist-scampi.org 19th April 2004 Page 71
The Makefile generates 2 files from this file, one C headerfile with the \#define's, and the error-file read by the MAPI Daemon. That way it is possible to use symbolic error-names in the MAPI-code and assign different symbolic errors to the same error-code.
Chapter 7

The SCAMPI Adapters

The SCAMPI adapters are sets of the several COMBO cards [28]. The COMBO cards are based on programmable hardware (FPGA) and their main advantage is flexibility. The same hardware can be used in many different applications just with the change of firmware downloaded into FPGA’s.

All SCAMPI adapters are sets of three COMBO cards. The first is the PCI mother card, second is the add-on interface card and third is the time stamp card. We have three sets of SCAMPI adapters now:

- SCAMPI-4MTX: COMBO6, COMBO-4MTX and COMBO-PTM
- SCAMPI-4SFP: COMBO6, COMBO-4SFP and COMBO-PTM
- SCAMPI-2XFP: COMBO6, COMBO-2XFP and COMBO-PTM

7.1 Family of the COMBO cards

7.1.1 COMBO6

The COMBO6 card is the mother PCI (32/33) card with XILINX FPGA, memories and connectors for add-on and extension cards (figure 7.1). The main parts are:

- VIRTEX II - XC2V3000 (up XC2V6000 can be used)
- PCI interface chip PCI9054
- CAM - 2Mb ternary CYNSE70064A
- 3xSSRAM - 2MB 512K36
- Extension/test connector
- DRAM connector for PC DDR up 2GB

The card is in production now. It will be replaced with the COMBO6X card to fix some minor flaws with the throughput of CAM and DRAM and improve the speed of the PCI bus.
7.1.2 COMBO6X

COMBO6X is the mother PCI (64/66) card with XILINX FPGA, memories and connectors for add-on and extension cards. The main parts are:

- VIRTEX II PRO - XC2VP50 (XC2V70 can be used)
- VIRTEX II PRO - XC2VP4 with PCI core (PCI-X can be used)
- CAM - 2Mb ternary CYNSE70064A
- 3x SSRAM - 2MB 512K36
- Extension/test connector
- DRAM connector for PC DDR up 2GB

This card is under design now. The VIRTEX II PRO has Power PC processors inside the chip. A combination of the FPGA and the processor in one chip can bring new interesting ideas in network monitoring.

7.1.3 COMBO-4MTX

The COMBO-4MTX is an add-on card with four copper GbE interfaces (figure 7.2). This card can be used in many of the network applications. The main parts are:

- VIRTEX II - XC2V1000 (up XC2V3000 can be used)
- 2xSSRAM - 2MB 512K36
- 4xGbE copper interface

The card is in the production now.
7.1.4 COMBO-4SFP

COMBO-4SFP is an add-on card with four SFP cages for GbE interfaces (figure 7.3). The Serdes chips support also 2.5Gb/s (Infiniband) and with a change of the crystal 1.062GB/s (Fiber channel) or 2.12Gb/s (OC48) can be supported. The main parts are:

- VIRTEX II - XC2V1000 (up XC2V3000 can be used)
- 2xSSRAM - 2MB 512K36
- 4xSFP cages for multimod, monomod, CWDM and/or copper transceivers

This card is in production now.

7.1.5 COMBO-2XFP

The COMBO-2XFP is an add-on card with two XFP cages for 10GbE interfaces (figure 7.4). The main parts are:
SCAMPI Deliverable D2.3
Enhanced SCAMPI Implementation and Applications

- VIRTEX II - XC2VP20 (up XC2V3000 can be used)
- SSRAM - 2MB 512K36
- 2xXFP cages for multimod, monomod, CWDM and/or copper transceivers

Figure 7.4: The COMBO-2XFP add-on card

The samples of the cards are back from manufacturing and work on their activation is ongoing.

7.1.6 COMBO-PTM

COMBO-PTM is a PCI(32/33) with XILINX Spartan3 FPGA, embedded processor, precise crystal and several connectors (figure 7.5). The main parts are:

- Spartan3 - X3S200 (X3S400 can be used)
- TI processor - MSP430FI49IPM
- 2xRS232/485 interface for external GPS card
- Extension/test connector

The samples of the cards are back for manufacturing and work on their activation is ongoing.

7.2 Firmware of the SCAMPI adapter

The VHDL program defines an internal structure of the monitoring adapter. The design respects the configuration of the adapter for independent applications (typically 4 applications per card), however the implementation of the application concept (e.g. assignment of a packet to a particular application and possible duplication of packets) is done in the adapter driver.

The VHDL program, as shown in the Figure 7.6, has the following main blocks:
The basic concept of the VHDL design is a control word, which is assigned to each packet in the input filter. This word contains all information about further packet processing.

### 7.2.1 Timestamps

The timestamp is 64-bit fix-point integer. 32 bits represent the time in seconds (from 0:0:0 1.1.1970), 32 bits are fraction of second. We have to define two terms:

- **resolution** = the smallest increment of time counter (e.g., clock period).
- **precision** = the weight of the least significant bit of time representation structure.

In our case, the precision is $2^{-32}$ (about 230 picoseconds). The clock frequency is 100 MHz, therefore the timestamp resolution is 10 nanoseconds.

Timestamp of each packet is evaluated just after receiving the whole packet and since this time it is an attribute of the packet. The wire length of 64 bytes long packet is about 50 ns at 10 Gb/s, therefore the resolution of clock is high enough to assign a unique timestamp to each packet.
The TSU is designed like standalone card. It contains a small FPGA (time critical parts of unit), a universal CPU chip (digital PLL control circuit), a PCI bus and a special bus for timestamp request and response.

The TSU has PPS (Pulse Per Second) input and output. PPS is generally provided by the GPS receiver with absolute accuracy about 50 ns. The unit can be alternatively controlled by a NTP process in the host computer. It can operate even without any external control with free running clock. Accuracy of TSU:

- with PPS - about 1 us,
- with NTP - about 50 us (with ’close’ NTP server),
- without control - short time accuracy $10^{-7}$ (done by oscillator), long time accuracy is done by aging of the quartz.

### 7.2.2 HFE

The HFE is a configurable unit. Its role is to extract required fields from IP and TCP/UDP headers and to assemble the unified header structure.

### 7.2.3 LUP

The LUP unit represents the input classification unit of the adapter. It is an engine controlled by a set of pairs $(R,A)$, where $R$ is a Rule and $A$ is an Action. There can be stored up to 2000 such pairs in the CAM. A rule consists of a matching pattern and a set of conditions.
matching the width of the pattern is up to 272 bits and it is matched with the unified header. The pattern can contain 'do not care' bits. The result of matching is a pointer to the condition.

condition each condition is a comparison (arithmetic or logic) instruction with two operands: an addressed part of unified header and a 16-bit direct value. The instructions form a binary tree structure. The maximal depth of the tree is 8.

The action is a result of the classification and it is represented by a leaf of binary tree of comparison instructions. The action assigns to the packet a 32-bits word which controls further packet processing in the adapter structure.

7.2.4 Control word

The control word is an attribute of the packet. There is coded the required processing of the packet in the word. The structure of control word is as follows:

- SAMASK (16 bit) - sampling unit assignment. Each SAU has one bit in this field and '1' means that packed will be processed by this SAU.
- STATID (8 bit) - STU identification. The field addresses one of 256 STU units which will process the packet. Each packet is processed in some STU.
- PCMASK (8 bit) - PCK mask. Defines the set of checked patterns in the payload checker.

7.2.5 SAU - Sampling unit

As is stated before, the VHDL structure contains 16 sampling units. Each of them can be configured to do:

- probability sampling - the packet is passed through the unit with the probability 1/n,
- deterministic - each n-th packet is passed through.

The packet could be processed simultaneously in more then one SAU. If the packet is not passed through any of these units, it is discarded. The SAMASK field is masked by the result of processing in SAU. The '1' is remained only if the packet is passed through corresponding SAU.

Special configuration of SAU is the deterministic sampling with n=1. This case represents the situation, when all specified packets are required to pass to the application.
7.2.6 STU - Statistic unit

The design contains virtually 256 STU. However, there are implemented 256 sets of registers and only one processing unit. The field STATID selects a corresponding set of registers. The Statistic unit has two parts:

- packets lengths statistics. The input value is the length of the packet.
- statistics of intervals between two subsequent packets. The input value is the difference of two last timestamps.

There is a significant distinction between these two parts. It is no problem to join data of two ’length’ parts, while joining data of two ’time’ parts has no sense.

Statistic of packet length

The statistic could be gathered for at least 1 second as full rate of 10Gb/s traffic represents less than $2^{32}$ bytes. The length part register set consists of following registers:

- number of packets
- accumulator of lengths of packets
- accumulator of squares of packets lengths
- min/max of packet length
- set of overflow flags

These data allow to calculate further parameters:

- average data rate
- average length of packet
- variation of packets length

The size of the accumulators is 32 bits (resp. 64 bits for accumulator of squares). In a standard situation, it is not necessary to clear registers, as the application can deal easily with register overflow. To achieve consistency of data, registers reading is done in two steps: registers value copying to latch and subsequent reading of latch registers.
Statistic of intervals

It seems reasonable to deal with inter-packet interval up to 1 minute. The size of the accumulator is 32 bits (resp. 64 bits for accumulator of squares). To implement such a time scale in 32 bits register, the interpretation is: 6 bits for seconds + 26 bits for fraction of second. Therefore the maximal duration of measurement is 64 seconds and the precision is about 15 ns (i.e. $2^{-20}$), e.g. still small enough to measure the interval between subsequent packets. The time part of STU set consists of following registers:

- number of packets
- accumulator of inter-packet intervals
- accumulator of square of inter-packet intervals
- min/max of inter-packet interval
- timestamp of last packet
- set of overflow flags (the same like in ’length’ part)

These data allow to calculate further parameters:

- total time of data flow (from first to last packet)
- average inter-packet interval
- variation of inter-packet interval

The interval statistic part implements the instruction ’clear and wait for the first packet’. It allows to measure statistic of specific data flow, as duration of the communication relation.

7.2.7 PCK - Payload checker

The PCK checks the payload of a packet for occurrence of defined patterns. Each pattern is up to 16 bytes long and the CAM can contain up to 120 patterns.

Method of checking: the CAM is configured for 272 bits word length. Each pattern is stored in 16 rows, shifted by 0,1, .. 15 bytes. All other bits in 32 byte wide window are ”don’t care”. Others 8 bits are matched with the field PCMASK of control word. This allows either

- to split all patterns into 256 groups and select any one of them

or

- to split all patterns into 8 groups and select any combination of them.
In each step, it is compared 32 bytes of payload with the CAM content. If any pattern matches, the PCK gives positive result. In the opposite case, the packet payload is shifted for 16 bytes and next portion of payload is compared again in following step. The process ends either when there are no more data or when a first match is found. Result of PCK should be interpreted this way:

- if there is no match, we are sure that none of the tested patterns are in the payload,
- if there is a match, we know that the founded pattern is in the payload and maybe the payload can contain some other pattern. The application has to repeat all tests. Usually, the real pattern is longer than 16 bytes and we find only packets suspected to contain the real pattern.

### 7.2.8 Adapter output

If the packet doesn’t pass any SAU nor PCK, it is discarded. In the opposite case, the packet and its attributes (timestamp, control word updated in SAU subsystem and eventually row id from PCK) are stored in the memory of the adapter until they are transfered to the software driver. In order to reduce the amount of data transfered through the PCI bus, the output circuit can be programmed to transfer only the first N bytes of each packet passing the SAU. Packets passing the PCK are not cut. The structure of the adapter output:

- **id** - 2 bytes - (e.g. interface number, reason of capturing - SAU/PCK),
- **control word** - 4 bytes
- **timestamp** - 8 bytes
- **rlen** - 2 bytes - record length (total length of this structure)
- **wlen** - 2 bytes - wire length (length of the packet including protocol overhead)
- **packet** - up to 16 kbytes.

### 7.2.9 Resource allocation

**LUP** - it is shared by all applications by parsing the set of rules and assembling new set of pairs (R,A). It is difficult to allocate rows in the table. In the worst case, adding one new rule may increase two times the number of current rows.

**SAU** - all 16 instances of SAU work in parallel and the SAU allocation is easy and straightforward.
SCAMPI Deliverable D2.3
Enhanced SCAMPI Implementation and Applications

**STU** - the 'length' part supports the parsing of rules as it is no problem to combine values from several STU. However, the 'time' part does not allow parsing and we have to resign from getting meaningful result in some cases (if there are two non-disjunctive rules of two applications, only one application of them can get 'time' statistic).

**PCK** - the control word allows to select an active subset of PCK patterns, so we can combine the rule and corresponding subset of patterns. The limitation is the CAM feature - only the first matching pattern is identified. The throughput of PCK is about 3 Gb/s.

### 7.2.10 Adapter limits

Despite the FPGA internal limits, this design of adapter VHDL structure has others limits:

**LUP functionality**  The LUP is able to classify packets as is described in chapter 3. The number of rows in the CAM is 2000. The LUP is a very complicated circuit, its design is ready and tested and it is not possible to change it in this phase of development.

**Control word**  We have currently only 32 bits available, and therefore we can address 16 SAU (any combination of them), 256 STU (just one of them) and 256 sets of pattern (just one of them) or 8 sets of patterns (any combination of these sets).

**STU**  We can address only one of 256 STU, what is not problem for the 'length' part of the STU as these data can be joined by simple postprocessing. However, it is a problem for the 'time' part of the measurements as these data can not be joined in any meaningful way.
Chapter 8

Applications

8.1 Packet Capture

The packet capture application is a simple network monitoring tool, which allows detailed per-packet network traffic analysis. The application which will be provided by the SCAMPI project has functionality comparable to the well-known tcpdump utility. It accepts a specification of packet header filter in the standard BPF (Berkeley Packet Filter) syntax and semantics, which is also used by tcpdump. Currently only the following subset of BPF set is supported. More header filtering capabilities should be provided by future releases.

- `[ src | dst ] host <ip_addr>` - source or destination ip address
- `[ src | dst ] net <net> mask <mask>` - source or destination subnet
- `[ src | dst ] net <net>/<len>` - dtto (alternative syntax)
- `[ src | dst ] port <port>` - source or destination port number
- `tcp` - TCP protocol
- `udp` - UDP protocol

These filter terms can be grouped together with `and` operator. They should allow to separate the required microflow or a set of microflows from the aggregate traffic.

The primary advantage of the SCAMPI packet capture monitoring application over tcpdump is the ability to use packet header filter implemented directly on the SCAMPI adapter (or another adapter, which provides such functionality) in its firmware. In this way, we can significantly reduce the volume of data transferred over the PCI bus to the host computer and thus reduce the host CPU load and capture more packets.
8.2 Flow Record Applications

8.2.1 Billing and Accounting

For Billing and Accounting two kinds of applications are under development:

**IP Range Traffic Count**

This application monitors a specific range of IPs and counts the total number of bytes that this range sends or receives. The traffic monitored is afterwards categorized as, e.g. www, e-mail, etc based on well known ports.

The application receives data concerning the customers, IP-pools from a specific database, and deploys flow-monitors for each customer, counting its traffic. The specific monitors make use of the flows that MAPI provides, thus making it possible to discriminating per-application traffic as sub-flows of the main (hierarchical) flow. In a configuration file that the application reads on start-up we hold the information about the services we want to measure. At first this is done by using well known ports, later on there will be the ability of applying application-specific modules concerning dynamic allocation of random ports; a method used mainly by peer-to-peer applications but also of other standard services (such as dynamic-ftp). This will provide a way of minimizing the amount of traffic that passes on uncategorized to "Other Services", resulting to a better, fairer pricing scheme.

The output of the application are the SDR (Service Detail Record Files) files, which have the following information:

1. customer_id - a unique ID used to correlate traffic data with a customer that has signed a contract
2. start_time - the beginning of the measurement period
3. end_time - the end of the measurement period
4. volume - the amount of traffic measured in Kbytes
5. direction - a binary field that represents the direction of the traffic (incoming, outgoing)
6. service - a service is discriminated by source/destination port numbers and one or more protocol numbers, changeable through the exporters configuration file.

**Virtual Streaming Server Traffic Count**

This application counts the total number of bytes that different streaming services (e.g. hosted radio stations) generate, when all of them are running on the same host and listen for requests to the same port.
As we can see above, the streaming server serves all connection requests through the same port e.g. port 80. Thus it is possible by applying a flow-monitor using a hierarchical flow on port 80 to catch each separate connection as a sub-flow. This gives us the ability to monitor each connection request separately. Based on the streaming protocol (e.g. HTTP, MMS) we can probe each connection request as to which radio station it is about and over which dynamic port it is going to be established. Afterwards we can deploy a new flow on the found port and measure the connection’s traffic. Aggregating the traffic data per customer, radio station, billing policies can be developed and applied.

### 8.2.2 Flow based reporting application

Flowrep is a flow based reporting application that collects raw Netflow data, generates reports which are stored in an SQL database and that users can view through a WEB interface. Figure 8.2 shows the various components in this application:

**SCAMPI IPFIX_exporter** collects flow data from MAPI and exports it using the IPFIX protocol.
NetFlow Collector collects raw NetFlow/IPFIX data and stores it to disk in flat files.

Report Generator analyzes the raw flow data and generates reports that are stored in an SQL database.

CLI Interface collection of small command line based applications for analyzing and printing information stored in raw flow files.

Maintenance scripts aggregates reports, deletes old records and updates information about available observation points, name of ports and AS numbers etc.

SQL Web interface allows users to browse the reports stored in the SQL database through a web interface.

SQL CLI interface allows users to retrieve information from the SQL database through a collection of command line based applications.

Database design All generated reports are stored in a PostgreSQL database. Figure 8.3 shows the basic principals of the database design. There are three tables that contains common information about devices, observation points and network links for all report types. Reports provide statistics for a certain time period and resolution. The first implementation provide statistics for an hour, a day, a week and a month. It will however be easy to add other time resolutions later. For each time resolution there are several report types and each report type has its own table.

Storing reports in separate tables provides a good isolation between the various reports so that it is easier to add and delete reports without influencing the performance of other reports.

An important design goal of the WEB interface was to make it as generic as possible so that when new report types or time resolutions are added, it is not
necessary to make any changes to the WEB interface code. This is achieved by having various SQL tables that describes time resolutions and how reports should be presented and plotted. This means that when a new report type is added, all that is necessary to do to get the WEB interface to display it correctly is to update a few SQL tables in the database.

**Report generator** The report generator that is used is based on flow-stat from the open source application Flowtools[24]. flow-stat reads raw Netflow records and generate a wide range of reports. The results from flow-stat is then simply inserted into the SQL database.

**WEB interface** The WEB interface allows users to select observation points and report types and see the results as tables or graphs. The user can then easily navigate between the various report types, observation points and time intervals.

The interface is implemented in PHP and uses Smarty templates so that the code for retrieving data from the SQL database is completely separated from the actual presentation of that data. This separation makes the code cleaner and easier to extend.

Figure 8.4 shows an example of what the WEB interface looks like. This example shows the IP protocol composition on one single observation point. The two most used protocols are shown, TCP and UDP, and the rest of the smaller protocols...
8.3 End-to-end Quality of Service Monitoring Application

QoS-monitoring analyses the behaviour of a specified (e.g. SLS monitoring) or random stream (e.g. CoS monitoring) throughout a system under observation (ranging from a single link to a concatenation of ISPs). We will develop and implement a two-layered architecture for QoS monitoring, i.e. a QoS monitoring layer and an application layer. The monitoring layer, belonging to a single Internet Service Provider (ISP), will provide end-to-end QoS statistics of the observed network to the application layer. These statistics include delay, jitter and packet loss. Any application in the upper layer can request these end-to-end QoS statistics from the monitoring layer.

8.3.1 Description and Environment

The architecture of the QoS monitoring application consists of 2 layers, a QoS monitoring layer and an application layer. The QoS monitoring layer provides network QoS statistics to the application layer above. This layer is part of a single ISP, which provides (on-demand) end-to-end QoS statistics to the application layer, without revealing any sensitive network topology information to the outside world. This way, the application can obtain end-to-end statistics between two access points of the network.

The integration of the monitoring layer with the SCAMPI architecture is illustrated in Figure 8.5. Based on hashing-based sampling (“trajectory sampling” [11]), the QoS parameters are measured. This information will be configured in the monitoring agents (i.e. the access points) throughout the MAPI. The results of the individual observation points are correlated in a centralized PostgreSQL database.
8.3.2 Components

Back-end (SCAMPI) The back-end of the application consists of several SCAMPI machines and a PostgreSQL correlation database. When monitoring an ISP network, all access nodes or peering points have to run the SCAMPI software (i.e. an instance of the mapid), an optional PostgreSQL database and a NTP client. The SCAMPI machines are configured so that all incoming traffic is filtered based on a hashing function. This hash is computed over a set of packet header fields and (optionally) part of the payload. Only invariable header fields with a high entropy are suitable. These include: Total length, Identification, src/dst address, src/dst port, sequence number,... If the hash function matches a packet, some information of the packet is written to a local PostgreSQL database. This information includes:

- source IP
- destination IP
- packet length
- packet identification
- packet protocol
- timestamp of arrival of packet
- site identification
- protocol dependent packet information

TCP
- source port
- destination port
- TCP flags (syn, ack, ...)
- TCP sequence number

UDP
- source port
- destination port
- hashes on the content

ICMP
- type
- echo identification
- echo sequence number

Figure 8.6 depicts the object oriented architecture of the database. In each table, a site-ID is added to allow the distinction between the different sites, when the trace-databases are joined. The network traffic for passive network monitoring consists of both TCP and UDP packets. TCP-packets can be distinguished from
each other, and correlating 2 different traces from 2 sites can be done based on the sequence-field. UDP-packets on the other hand, have no sequence field or something similar. We will generate 3 hash-values (simple rotating hash [14], Adler32 and CRC32) over the packet content to distinguish different UDP packets. Based on these hashvalues, together with source and destination IP/port and the packet length, we should be able to make the same correlation between the different sites. The hashes could be optimized by analysing a UDP trace and looking for the packet parts with the highest entropy in the trace. But considering the low bandwidth usage, there shouldn’t be a problem generating hashes over the full packet content.

All monitors write the gathered information of the captured packets in a row of a local (or distributed) table in the PostgreSQL database. After joining the databases, the database software will correlate all packets in the database. Because every packet sent from 2 measured hosts will be captured by both monitoring agents, we will find the same packet twice in the joined database, i.e. the packet is sampled in the incoming access point and the outgoing access point. When we correlate the timestamps of both measurements, we can deduce the delay and jitter between both access points. When a packet is listed only once in the database during a certain amount of time, we assume the packet is lost and the path between both access points experiences loss. The correlation software fetches the packets out of the joint monitoring table and processes the collected information in order to deduces the QoS characteristics. When calculating the delay for example, the software searches the information of a single packet at both the ingress and egress
SCAMPI Deliverable D2.3
Enhanced SCAMPI Implementation and Applications

of the ISP network. Subtracting their timestamps will result in the network delay.

After correlating the databases, we can calculate the following network characteristics between all sites:

1. one-way delay
2. one-way delay variation or jitter
3. packet loss

**Webbased front-end** The front-end illustrates the network characteristics in a GUI. To obtain the network QoS characteristics, the GUI queries the appropriate database tables, containing the requested information. A graph shows the delay, loss and jitter for each 2 access points. Other included features are:

- Show best connection between 2 sets of access points
- Reconfigure the hashing functions
- ...

### 8.4 Threshold Alerting for Traffic Engineering

#### 8.4.1 Description and Environment

Traffic engineering requires feedback from monitoring at different levels. A first level is a pure “diagnostic” monitoring (i.e. call holding time, average data volume, etc.). This is optionally complemented by an “operational” monitoring feedback (in which case we talk about “adaptive TE”). The application described below is such an example.

We assume a TE application is available that calculates paths through a network and the amount of traffic a path ideally should receive. Since flows cannot be split arbitrarily, a wrong level of multiplexing was forecasted or user behaviour deviates from the expected estimations, one cannot simply rely on a fixed division of flows amongst the given paths.

SCAMPI can be used to drive the division of traffic over the paths. In this case, the TE algorithm tells the network ingress node to balance the incoming traffic such that one path receives 70%, and the other 30% of the total traffic. In this application it is assumed that every flow is requested explicitly, after which a configuration for that flow is added to the node (an alternative mechanism for “implicit” invocations exists but its implementation wouldn’t add much to the setup).

When an “over-limit” event is received from one path, all traffic goes to the other paths. When an “under-limit” event is received, a path start attracting traffic again. When both paths are either under- or over used, traffic is divided according to the 70-30 division. Since flows have only a limited lifetime, this mechanism allows to balance the traffic close to the configured value.
8.4.2 Components

The application uses the following components:

1. Two MAPI applications (for each output interface of the node under test), receiving events for the over or under utilisation of the path.

2. A control layer, listening to these event and dividing traffic over the two paths (by adapting the Linux routing table in the node).

The algorithm that drives the control layer, divides paths into two states (‘green’ or ‘yellow’). A path becomes ‘yellow’ if it transports more traffic that was allocated to it, else it is said to be in the ‘green’ state. The next hop for new arriving flows is set:

- in a 70/30 division (using a weighted round robin method) if either both paths are in the green or both paths are in the yellow state.

- if only one of the paths is in the ‘green’ state, all new flows are mapped to it.

To demonstrate the ‘multi-application/single infrastructure’ nature of SCAMPI, additional applications can be ran on this node (e.g. for contributing diagnostic information important for traffic engineering such as flow interarrival time statistics).

A possible extension to this demonstrator is to use a similar setup for detecting congestion in the network. This doesn’t require additional components from the SCAMPI point of view, but only an extension to the TE control layer.

The mechanism for generating the traffic is still under investigation (either by driving the SmartBits or by incorporating a software-based traffic generator).

8.5 Security Applications

8.5.1 Intrusion Detection Application

Network-based Intrusion Detection is a research and development area that aims to improve the security of our cyberinfrastructure through the early detection of intrusion attempts. Network-based Intrusion Detection is usually deployed in the core (or the edge) of the Internet in order to identify possible intrusions as they are being launched. After a possible intrusion is identified, all the information regarding the intrusion is being logged, and the administrators of the system are (optionally) being alerted. The administrators, in turn, may take corrective measures to reduce the effects of this intrusion and possibly patch the security hole that led to the intrusion.

Network-based Intrusion Detection Systems are usually based on a set of rules (also called signatures). Each possible type of intrusion is described by one or more rules. For example, the following rule describes an attempt by an outsider to become super-user (i.e. root) in one of the local systems: $\text{OUTSIDE\_NETWORK} \rightarrow \text{LOCAL\_NETWORK TCP 23 content "su root"}$. The above rule states that if a
packet is sent from a computer located in the **OUTSIDE_NETWORK** (an alias for all computers outside the monitored organization) towards a computer in the **LOCAL_NETWORK** (an alias for all computers in the monitored organization) on **port 23** (the telnet port) using the protocol TCP, and the payload of the packet contains the substring “*su root*”, then this is a possible intrusion attempt. In a Network-based Intrusion Detection System each packet is checked against every rule. If the packet matches a rule, it is logged and the administrators may be notified.

Implementing a NIDS is rather a complicated task. Several basic operations like packet decoding, filtering, and classification, TCP/IP stream reconstruction, and string searching, must be crafted together to form a fully functional system. Each one of these operations alone requires deliberate decisions for its design, and considerable programming effort for its implementation. Furthermore, the resulting system is usually targeted to a specific hardware platform. For instance, the majority of current NIDSes are built on top of **libpcap** [17] packet capture library using commodity network interfaces set in promiscuous mode. As a result, given that **libpcap** provides only basic packet delivery and filtering capabilities, the programmer has to provide considerable amount of code to implement the large and diverse space of operations and algorithms required by a NIDS.

In contrast, MAPI inherently supports the majority of the above operations in the form of functions which can be applied to network flows, and thus, can be effectively used for the development of a complete NIDS. Consequently, a great burden is released from the programmer who has now a considerably easier task. As a matter of fact, we have developed a Network Intrusion Detection System based on MAPI. Based on the observation that a rule which describes a known intrusion threat can be represented by a corresponding network flow, overall implementation is straightforward. As an example, consider the following rule taken from the popular Snort [19] NIDS, which describes an IMAP buffer overflow attack:

```
alert tcp any any -> 139.91/16 143 (flags: PA;
    content:"|E8C0FFFFFF|/bin";
    msg: "IMAP buffer overflow";)
```

All packets that match this particular rule can also be returned by the following network flow, after the application of the appropriate MAPI functions:

```
fd = mapi_create_flow(dev);
mapi_apply_function(fd, BPF_FILTER,
    "tcp and dst host 139.91 and dst port 143");
mapi_apply_function(fd, TCP_FLAGS, "PA");
mapi_apply_function(fd, STR_SEARCH, "|E8C0FFFFFF|/bin");
```

Our MAPI-based NIDS operates as follows: During program start-up, the files that contain the set of rules are parsed, and for each rule, a corresponding network flow is created. Rules are written in the same description language used by Snort.

scampi@ist-scampi.org

19th April 2004
Enhanced SCAMPI Implementation and Applications

Figure 8.7: Comparison between Snort NIDS and MAPI-based NIDS.

Snort rules are converted by a separate module to the appropriate MAPI function elements, which are then applied to the related network flow. The rest of the functionality is left to MAPI, which will optimize the functional components of all the defined rules and deliver the packets that match any of them.

Implementing the above intrusion detection application using \texttt{libpcap} would have resulted in longer code and higher overheads. As shown in Figure 8.7, our implementation takes no more than 2000 lines of code, while the core functionality of other popular NIDSes, such as Snort, consists of roughly 30,000 lines of code\textsuperscript{1}. For example, \texttt{libpcap} does not provide any string searching facility, and thus the programmer would have to provide a significant chunk of code for the implementation of the chosen string searching algorithm. Instead of forcing the programmer to provide all this mundane code, MAPI already provides this frequently used functionality.

Note that a NIDS based on MAPI is not restricted to a specific hardware platform. MAPI operates on top of a diverse range of monitoring hardware, including more sophisticated lower level components like network processors, and thus, can further optimize overall system performance, considering that certain MAPI functions can be pushed to the hardware. Additionally, the functionality of the MAPI daemon can be shared by multiple concurrently running applications. For example, along with the intrusion detection application, one can develop a firewall application in the same fashion (i.e., in a few lines of code), adding this way extra capabilities to the overall system. Again, instead of providing code for the whole firewall operations, the programmer can use MAPI to reduce the development effort, and to effectively share resources by pushing the core firewall functionality into the MAPI daemon.

8.5.2 NetFlow IDS

IDSs (Intrusion Detection Systems) available on the market share the same working principle. A packet process loop captures packets from a network adapter and pass

\textsuperscript{1}Although the functionality of the two systems is not identical, it is clearly depicted a difference in code length of at least one order of magnitude.
them to the packet analyzer that looks for known signatures. The main limitation of this approach is that it is necessary to physically place the IDS on the network trunk where the packets are flowing and this can be an issue for many reasons (e.g. it is necessary to interrupt the normal network activities and add a component such as a network tap for duplicating packets).

Cisco NetFlow is a very popular protocol used mainly for network accounting and billing. As it is usually already present and active on many routers and switches of a generic network backbone, it would be interesting to also use the protocol for intrusion detection purposes. For this reason a popular network IDS named Snort [19] has been enhanced with a NetFlow support so it now behaves as a NetFlow collector.

Snort-NetFlow is able to decode incoming NetFlow v5 flows and pass them to the snort engine for signature checking. As the flows are not as many as the incoming packets that produce them, this allows snort to run at speeds much greater than those achieved on the original pcap-based snort. The main drawback of this approach is that NetFlow flows do not carry payload information so it is not currently possible to enable the snort signatures that explore the packet payload. Hence it is possible for instance to detect a portscan but it is not possible to detect trojans and worms that send emails with virus attachments. This is not a real limitation as IDSs placed on the backbone are not usually used for detecting payload problems but mostly other kind of attacks such as DoS that can be easily detected with Snort-NetFlow.

8.5.3 Denial of Service Attack Detection Application

Over the past few years many sites on the Internet have been the target of denial of service (DoS) attacks, among which TCP SYN flooding is the most prevalent [18]. Indeed, recent studies\(^2\) have shown an increase of such attacks, which can result in disruption of services that costs from several millions to billions of dollars.

The aim of denial of service attacks is to consume a large amount of resources, thus preventing legitimate users from receiving service with some minimum performance. TCP SYN flooding exploits the TCP’s three-way handshake mechanism and its limitation in maintaining half-open connections. Any system connected to the Internet and providing TCP-based network services, such as a Web server, FTP server, or mail server, is potentially subject to this attack. A TCP connection starts with the client sending a SYN message to the server, indicating the client’s intention to establish a TCP connection. The server replies with a SYN/ACK message to acknowledge that it has received the initial SYN message, and at the same time reserves an entry in its connection table and buffer space. After this exchange, the TCP connection is considered to be half open. To complete the TCP connection establishment, the client must reply to the server with an ACK message. In a TCP SYN flooding attack, an attacker, from a large number of compromised clients

\(^2\)2002 and 2003 CSI/FBI Cybercrime Survey Report. The 2003 report indicates that DoS attacks alone were responsible for a loss of $65 million.
in the case of distributed DoS attacks, sends many SYN messages, with fictitious (spoofed) IP addresses, to a single server (victim). Although the server replies with SYN/ACK messages, these messages are never acknowledged by the client. As a result, many half-open connections exist on the server, consuming its resources. This continues until the server has consumed all its resources, hence can no longer accept new TCP connection requests.

A common feature of DoS attacks is that they lead to changes in a measured statistic of a network traffic flow. Such statistics can include the type and size of packets, the number of half open connections, and the rate of packets associated with a particular application or port number; in the case of TCP SYN flooding the statistic is the number of TCP SYN packets. Based on the aforementioned property, DoS attack detection applications are commonly based on anomaly detection models, where the behavior of a measurable network characteristic is compared to its normal behavior, in order to detect deviations. An advantage of anomaly detection systems is that they do not require any a priori specification of attack signatures, hence they can detect new types of attacks. One approach for describing normal behavior is to use a static characterization; such an approach has the disadvantage of not adapting to trends and periodic behavior of normal traffic, e.g. the load of a networking system is much higher during peak hours compared to non-peak hours, which may eventually lead to an increased false alarm rate. Hence, anomaly detection systems should adaptively learn normal behavior, in order to track trends and periodic behavior.

**Anomaly detection algorithms**

The two anomaly detection algorithms that are implemented by the DoS attack detection application try to detect changes in some statistic of the traffic flow, based on measurements of the statistic in consecutive intervals of the same duration. In
both algorithms, the normal behavior is adaptively estimated from measurements of the mean rate.

The adaptive threshold algorithm signals an alarm when the measurements exceed some threshold \((\alpha + 1)\mu\), where \(\mu\) is the measured mean rate, for a number of consecutive intervals \(k\), Figure 8.8.

The CUSUM algorithm signals an alarm when the accumulated volume of measurements \(g_i\) that are above some traffic threshold, exceed some aggregate volume threshold \(h\), Figure 8.9.

**DoS attack detection application**

The DoS attack detection application works online, and takes as input load measurements in consecutive intervals of the same duration; this duration is typically of the order of tens of seconds. The measurements in consecutive time intervals are taken continuously using the MAPI interface, using its “PKT_COUNT” function. In its first version, the application detects SYN flooding attacks, hence the measurements consist of the aggregate number of SYN packets in consecutive time intervals; later versions will include other statistics, such as the number of UDP packets and the number of ICMP packets. The measurements are processed according to the two anomaly detection algorithms described in the previous subsections, based on which an alarm is signaled when a change of the normal behavior is detected.

Two different programs implement the two anomaly detection algorithms: the adaptive threshold algorithm is implemented by the program `adaptive`, whereas the CUSUM anomaly detection algorithm is implemented by the program `CUSUM`. The user is allowed to set specific values for the parameters of each algorithm. The command line for executing the program `adaptive` is

```bash
scampi@ist-scampi.org 19th April 2004 Page 98
```
SCAMPI Deliverable D2.3
Enhanced SCAMPI Implementation and Applications

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>percentage above which the measurement is potentially considered to be due to an attack, default value 0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>parameter of the moving average algorithm used to estimate the mean rate, default value 0.98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>number of consecutive intervals above which, if the measurements exceed the traffic threshold, an alarm is signaled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>duration (in seconds) of the intervals in which traffic measurements are taken, default value 10 seconds</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Parameters of the adaptive threshold algorithm

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>percentage above which the measurement is potentially considered to be due to an attack, default value 0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>parameter of the moving average algorithm used to estimate the mean rate, default value 0.98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>aggregate volume threshold above which an alarm is signaled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>duration (in seconds) of the intervals in which traffic measurements are taken, default value 10 seconds</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2: Parameters of the CUSUM algorithm

adaptive <a> <b> <k> <i>

where the different parameters are explained in Table 8.1. The command line for executing the program *cusum* is

cusum <a> <b> <h> <i>

where the different parameters are explained in Table 8.2.

The sensitivity of the two algorithms is determined mainly by the parameter $k$ for the adaptive threshold algorithm, and $h$ for the CUSUM algorithm. For this reason, the algorithms can be executed by giving only this parameter, in which case the other parameters obtain the default values shown in Tables 8.1 and 8.2.
Appendix A

Configuring, Compiling and Running the SCAMPI Software

The source of the SCAMPI monitoring software consists of several parts that need to be configured, compiled and executed in a Linux environment. The main directory contains the MAPI daemon source files as well as the test applications. The source of the standard SCAMPI library is located in subdirectory stdlib/. The directory adm_ctrl/ contains the admission and resource control source files, while the Ethereal patch is included in ethereal/. The source files of the combo6 and DAG library are listed in combo6lib/ respectively daglib/, the sources of the libpcap ring buffer are present in the ring/ subdirectory.

A.1 General configuration of the SCAMPI software

Before compiling the SCAMPI platform, the user needs to configure some parameters. The file Makefile.in contains the paths of the software libraries that are needed by SCAMPI platform as well as some compilation options. The compilation options are:

# Debug messages (not to be set if going in production)
DEBUG=1

# Debug messages for admission control (not to be set if going in production)
ADMCTRL_DEBUG=2

# Compile with Admission control (yes=1, no=0)
WITH_ADMISSION_CONTROL=1

# Enable logging with syslog (Supported only by admission control) (yes=1, no=0)
WITH_SYSLOG=1
A.2 Configuring and compiling the SCAMPI software components

A.2.1 The Ethereal Display Filters [optional]

The source files of the “Ethereal Display Filters” (EDF) are listed in the subdirectory ethereal of the main SCAMPI directory. Because of the large number of supported protocols in EDF, the EDF shared library is rather large (~13MB) and is not included in the SCAMPI distribution. Therefore, a first step is to download the sources of the latest Ethereal distribution (currently version 0.9.15) from http://www.ethereal.com/download.html. In order to create a SCAMPI share library, the sources of ethereal 0.9.15 should be patched with an included patch file listed in the subdirectory ethereal. Start in a directory where the Ethereal sources should be unpacked. Run:

tar xvjf ethereal-0.9.15.tar.bz2
patch -p0 < location_of_patch-ethereal-0.9.15
cd ethereal-0.9.15
aclocal && autoconf && automake
cd wiretap
aclocal && autoconf && automake
cd ..
cd epan
aclocal && autoconf && automake
cd ..
./configure –enable-shared –with-pic ...(other options)
mak e

After “make”, the shared library lib_etheral.so will be placed in the .libs subdir of the Ethereal sources. If the library should be installed on the system, type: “make install”. That way, the SCAMPI program will find the library by itself. Another option is to copy the library to the mapid executable dir. Currently, the only possible location is the mapid executable dir, and not the system
library paths, but “make install” is required in order for lib_ethereal to find its support libs.

A.2.2 The admission control daemon [optional]

The purpose of this daemon is to provide authorization services for the MAPI. Communication between the mapid and authd is performed using shared memory. To synchronize access to shared memory semaphores are used to ensure the consistency of the results. Routines are provided for easy access and manipulation of shared memory and semaphores. In order to compile the admission control daemon, install the following software:

- keynote 2.3 (Download from http://www.cis.upenn.edu/ keynote/)
- libsnprintfv v1.1 (Download from http://savannah.nongnu.org/projects/libsnprintfv) [resource control only]
- Berkeley DB 3.3.11 (Download from http://www.sleepycat.com/download/patchlogsdh.shtml) [resource control only]

Configuration

The configuration of authd is hardcoded at this time. All the variables all located in config.h. The sizes defined for keys and strings in config.h should be taken in mind when testing, and set appropriately. Also the location of the mapid keys, policy file and available mapi functions is defined in config.h and is currently set to /etc/scampi. The filenames are respectively authd_key.priv, authd_key.pub, policy and mapi_functions.

Compilation

To compile the admission control daemon type “make”. If keynote is not installed in one of the standard directories (e.g. /usr, /usr/local), place the keynote binary in your PATH and change the variables in ../Makefile.in to point to the keynote headers and library.

Setup

To actually start using admission control some files need to be install in “well-known” locations. These files are the admission control public and private keys, the authorization policy, the list of existing mapi functions and the file used to access shared memory and semaphores. To setup all these, run the setup.sh install script as root (desired user to own the files). Setup is setting the ownership of files to the username provided as argument. Setup can also perform the following actions:
SCAMPI Deliverable D2.3
Enhanced SCAMPI Implementation and Applications

Usage: setup.sh install|uninstall|policy_gen|creds_gen
- install (username): creates keys and a policy file in /etc/scampi, as well as /tmp/scampi_authd used for IPC
- uninstall: undo all the actions taken by install
- policy_gen (username): generates a policy file in /etc/scampi
- client: create keys for client usage. Keys are placed in files named “pub” and “priv”
- creds_gen (client’s public key) (conditions file) (output file): generates a credentials file for a user. The client’s public key is authorized using the conditions provided and the credentials are written to the output file (Run this as a privileged, so that it can read authd’s private key from /etc/scampi/authd_key.priv)

Testing
For testing purposes a test MAPI application has been created in ../. The keys have to be RSA at this point and were generated using keynote. The setup.sh script uses 1024bit keys. Step-by-Step guide:

1. generate client keys: ./setup.sh client
2. run setup.sh creds_gen pub conditions credentials as root
3. compose a function list or use of one already provided. The parser implemented by our test application is not very elaborate so keep it simple. Function arguments are seperated by any white space (this means no white spaces in strings) and strings should not be included in ‘”’.
4. run authd: authd [nodaemon]. If you run authd with nodaemon then it runs in the foreground. You still need to have DEBUG enabled on compile time to get DEBUG messages, but you will get the error messages when in the foreground. authd grabs the TERM signal so it can be closed gracefully. When not a daemon it also grabs QUIT & INT.
5. run ../mapid secure to enable admission control
6. run ../test_admctrl (user’s public key) (user’s private key) (credentials) (functions file). test_admctrl creates a flow on ’eth0’ and tries to apply the functions contained in the file and then connects the flow. Reports whether a connection has failed, because of insufficient privileges or communication error. Available function lists are:
   - funcs1: Succeeds with conditions
   - funcs2: Fails with conditions, because of STR_SEARCH argument depth
   - funcs3: Fails with conditions, because of a 2nd instance of PKT_COUNTER
A.2.3 Compiling the MAPI DAG driver

In order to compile the MAPI DAG driver, the object files dagapi.o and dagopts.o must already be available on the system. These object files are part of the DAG drivers provided by Endace.

A.2.4 Compiling the SCAMPI daemon

In order to use the ring buffer for high speed packet capture instead of the standard libpcap library, the user needs to patch the Linux kernel with the included patch and apply a patch to the libpcap library. Please check a Linux manual to find documentation concerning kernel patching, compilation and installation. To patch and install the libpcap library, first download libpcap version 0.8.1 from http://www.tcpdump.org/. Override the original libpcap files (pcap-int.h and pcap-linux.c) with the files enclosed in ring/libpcap-0.8.1/. Rebuild libpcap as usual and make sure that your libpcap-based application (e.g. mapid) is linked against this library. Next, build the ring module by typing “make” in the ring/module/ directory.

In the SCAMPI directory, type “make”. This will compile the MAPI daemon as well as the test applications.

A.3 Running the SCAMPI software

A.3.1 The MAPI daemon

When the ring buffer should be used instead of the standard libpcap library, first the user needs to load the compiled ring module into the modified kernel. Type the command “insmod ring.o” for 2.4.x Linux kernels or “insmod ring.ko” for 2.6.x kernels. The following options can be used:

- bucket_len : max packet capture len (default 128 bytes)
- num_slots : number of ring slots (default 4096)
- sample_rate : packet sampling rate (default 1)

Prior to running the MAPI daemon or a SCAMPI application, some configuration parameters can be defined in mapi.conf in the main SCAMPI directory. The layout of the file is as follows:

```
libpath=.
libs=mapidstdflib.so
drvpath=.

[driver]
device=eth0
```

scampi@ist-scampi.org 19th April 2004 Page 104
libpath defines the directory where the standard SCAMPI libraries are located, default the main directory. libs defines the SCAMPI function library, while drvpath the path to the drivers contains. For each hardware adapter in the SCAMPI monitoring box, a driver needs to be identified. mapi.conf contains a [driver] structure that lists the device and driver name of each present hardware card. After the configuration, the MAPI daemon can be started with the command:

```
./mapid
```

in the main SCAMPI directory. The “mapid” command has the following optional arguments:

Usage: ./mapid [OPTIONS]

- `-a, –admctrl`: Enable admission control
- `-s, –shmpath ( pathname )`: Use pathname for communication with admission control
- `-i, –shmid ( id character )`: Use id for communication with admission control
- `-r, –resctrl ( pathname )`: Set resource control home to pathname
- `-h, –help`: Display this message

## A.3.2 A SCAMPI application

To run one of the test applications, first start the MAPI daemon and then run the test application e.g. “./test_tcpdump”, preferably in a separate shell. For any other application, please consult the documentation of the application in particular.
Appendix B

Assertions

In this appendix, we briefly list some of the documentation that is provided with Authd about assertions and credentials. The rules below are meant both to convey the general idea and to show the expressiveness of the approach.

FUNCTION TYPE ASSERTIONS
- function is defined: (function_name) == "defined"
- function instances: (function_name).num == (number_of_instances)
- first position of an instance: (function_name).first == (first_position_of_function)
- last position of an instance: (function_name).last == (last_position_of_function)
- maximum value of a parameter: (function_name).param.(parameter_no).max == (maximum_value)
- minimum value of a parameter: (function_name).param.(parameter_no).min == (minimum_value)

FUNCTION INSTANCES ASSERTIONS
- instance position:
  (function_name).(instance_no).pos == (function_position)
  func.(function_position)_name == (function_name)
e.g. PKT_COUNTER.0.pos == 0 , func.0.name == PKT_COUNTER
- instance parameters:
  (function_name).(instance_no).param.(parameter_no) == (parameter_value)
  func.(function_position).param.(parameter_no) == (parameter_value)
e.g. TO_TCPDUMP.0.param.1 == 10000 , func.0.param.1 == 10000

WRITING CREDENTIAL CONDITIONS
To address an assertion in the credentials the type of the assertion must be specified, meaning whether the assertions should be checked as an integer, float or string.
E.g.:
- String condition_name == "string" or "($condition_name") == "string". Using the ~ operator we can use regular expressions
- Integer @condition_name == int or @$("condition_name") == int
- Float &condition_name == float or &@$("condition_name") == float
- unsigned long long $("condition_name") == "unsigned long long"

Example:
Conditions: app_domain == "MAPI" && device_name ~= "eth[0-9]" && @$("PKT_COUNTER.num") < 2 && @$("BYTE_COUNTER.num") < 2 && $("TO_TCPDUMP.param.1.max") < "847759389305035" -> "true";
Bibliography


Enhanced SCAMPI Implementation and Applications


[21] Cisco Systems, “Netflow services and applications”, 


[23] Ethereal - The world’s most popular network protocol analyzer, 
“http://www.ethereal.com”.


