INFORMATION SOCIETY TECHNOLOGIES (IST) PROGRAMME

A Scaleable Monitoring Platform for the Internet
Contract No. IST-2001-32404

D3.4v4 "Description of Experiment Results"

Abstract: This document provides description of evaluation tests performed on the SCAMPI architecture. This is the fourth release of D3.4 deliverable. It includes performance tests of the SCAMPI architecture software running on top of 1 GE SCAMPI with monitoring functions implemented in software and firmware. It also includes tests with 10 GE SCAMPI adapter and several application tests.

Contractual Date of Delivery	30 September 2004
Actual Date of Delivery	2 April 2005 (v4)
Delivarable Security Class	Public
Editor	Sven Ubik
Contributors	IMEC, FORTH, UNINETT, CESNET, FORTHnet

The SCAMPI Consortium consists of:

TERENA	Coordinator	The Netherlands
IMEC	Principal Contractor	Belgium
FORTH	Principal Contractor	Greece
LIACS	Principal Contractor	The Nedherlands
NETikos	Principal Contractor	Italy
UNINETT	Principal Contractor	Norway
CESNET	Principal Contractor	Czech Republic
FORTHnet	Principal Contractor	Greece
4Plus	Principal Contractor	Greece
Siemens	Principal Contractor	Germany

Contents

0 What is new in release D3.4v4, D3.4v3, D3.4v2

The following parts were added in D3.4v4:

- Packet capture test with 1 GE SCAMPI adapter was repeated with the latest version of firmware and driver, see table 2 and corresponding comments.
- Firmware header filtering test with 1 GE SCAMPI adapter was repeated with the latest version of firmware and with the more complex header filter, see table 6 and corresponding comments.
- Firmware sampling test with 1 GE SCAMPI adapter was repeated with the latest version of firmware and driver, see table 8. This time we also tested sampling with 64-byte packets (in the previous test we only used 1500-byte packets).
- Packet capture test with 10 GE SCAMPI adapter was repeated with the latest version of firmware and driver, which was specifically designed for this adapter (the previous test was done with firmware designed for 1 GE adapter which was only adopted for 10 GE adapter). See table 10 and corresponding comments.
- Firmware sampling test with 10 GE SCAMPI adapter was added, see table 11 and corresponding discussion.

The following parts were added in D3.4v3:

- Firmware header filtering and statistics with 1 GE SCAMPI adapter (new version)
- Firmware sampling with 1 GE SCAMPI adapter
- Firmware header filtering and statistics with 10 GE SCAMPI adapter (preliminary version)
- More information about test setup to allow test replication

The following parts were added in D3.4v2:

- Firmware header filtering with 1 GE SCAMPI adapter
- Packet capture with 1 GE SCAMPI adapter (new version with firmware supporting header filtering)
- Intrusion detection application test - extended with experiments with Piranha algorithm
- QoS monitoring application test - extended with a new case study

1 Introduction

The purpose of the workpackage WP3 "Experimental evaluation" is to do functionality and performance tests of the SCAMPI architecture in order to validate its applicability for passive network measurements.

This is the fourth release of D3.4 deliverable, which includes description of the following evaluation tests:

- Packet capture with 1 GE SCAMPI adapter
- Software header filtering with 1 GE SCAMPI adapter
- Firmware header filtering and statistics with 1 GE SCAMPI adapter
- Firmware sampling with 1 GE SCAMPI adapter
- Firmware header filtering and statistics with 10 GE SCAMPI adapter
- Packet overhead test using PAPI
- Intrusion detection application test
- QoS monitoring application test

2 Configuration

We used Spirent AX/4000 packet generator to produce a stream of packets with the specified packet size and the number of packets per second. The configuration is shown in Fig. 1. The SCAMPI architecture was installed on the PC on the left. It included 1 GE or 10 GE SCAMPI adapter, SCAMPI adapter driver, middleware libraries (libcombo, libscampi, libscampid), MAPI and applications. The PC itself was Celeron 1.7 GHz with 256 MB RAM. The operating system was Linux with kernel 2.4.25. The way that packets passed through the SCAMPI architecture is illustrated in Fig. 2. We used 1500-byte and 64-byte packets and observed packet loss rate in various scenarios and various packet rates.

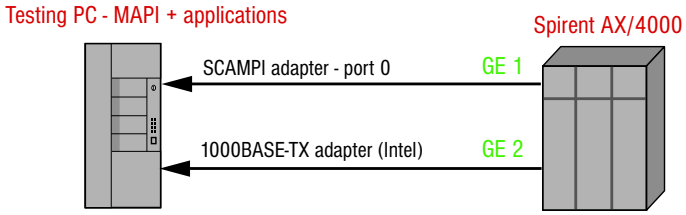

Figure 1: Configuration for the maximum load test

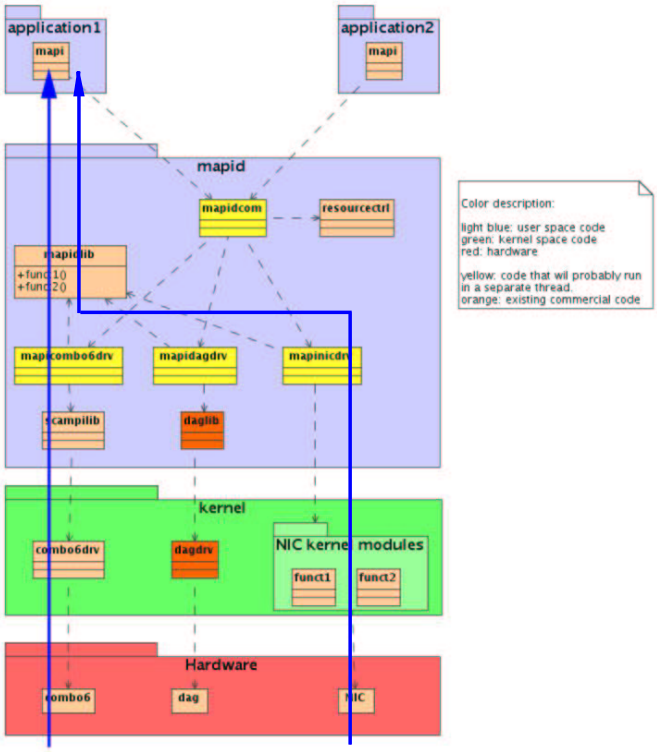

Figure 2: Packet passing through the SCAMPI architecture

3 Packet capture with 1 GE SCAMPI adapter

We tested the maximum number of packets per second that can be obtained from 1 GE SCAMPI adapter. We performed the test twice.

The first test was done with the initial version of firmware and driver for 1 GE SCAMPI adapter. In this test we counted packets that could be received from MAPI and for comparison purposes we also used Intel NIC as an alternative to 1 GE SCAMPI adapter.

The second test was done with the latest version of firmware (version 1_current, which was slightly modified version 1_00_07) and driver for 1 GE SCAMPI adapter. In this test we counted packets read directly from thr driver (without MAPI), because we were interested primarily in performance of the SCAMPI adapter itself with just the necessary driver.

Initial firmware and driver version

For the first test we used the following simple program written on top of MAPI, which applies `PKT_COUNTER` MAPI function to the flow to count packets:

```
#include <stdio.h>
#include <unistd.h>    /* sleep */
#include "mapi.h"

int main(void)
{
  int fd;
  int counter_all;
  int total_packets=0;
  unsigned long long *cr;

  fd=mapi_create_flow("/dev/scampi/0");
  // Or use "eth1" for Intel NIC
  // fd=mapi_create_flow("eth1");
  counter_all=mapi_apply_function(fd, "PKT_COUNTER");
  mapi_connect(fd);

  while(1) {
    usleep(100000);
    cr=mapi_read_results(fd, counter_all, 1);
    total_packets+=*cr;
    printf("pkts all: %lld\n", total_packets);
    // With Intel NIC result of PKT_COUNTER is cummulative
    // printf("pkts all: %lld\n", *cr);
  }

  return 0;
}
```

In this test we used the initial version of adapter firmware and driver, which functioned as a regular NIC card. No monitoring capability was yet implemented on the adapter. The results

are summarized in Table 1.

In this test and in all following tests, NA stands for "not applicable" when the corresponding number of packets per second at given packet size exceeds link rate and NT stands for "not tested" when we skipped this measurement, because there was already significant packet loss for a lower packet rate.

	SCAMPI adapter		Intel adapter	
	1500-byte	64-byte	1500-byte	64-byte
4000 p/s	0%	0%	0%	0%
5000 p/s	0%	0%	0.004%	0.003%
10000 p/s	0%	0%	0.016%	0.017%
17500 p/s	0%	0%	0.019%	0.021%
25000 p/s	0%	0.004%	0.031%	0.030%
37500 p/s	0%	0.005%	0.055%	0.050%
50000 p/s	10.3%	0.017%	0.039%	0%
75000 p/s	40.1%	0.031%	0.230%	0.002%
100000 p/s	NA	9.6%	NA	0.057%
150000 p/s	NA	39.4%	NA	59.1%

Table 1: Packet loss rate for packet capture with 1 GE SCAMPI adapter and 1 GE Intel NIC

We can observe the following points:

- The maximum rate that could be processed without any loss by the Intel NIC was 4000 packets per second for both 1500-byte and 64-byte packets.
- The maximum rate that could be processed without any loss by 1 GE SCAMPI adapter was 37500 packets per second for 1500-byte packets and 17500 packets per second for 64-byte packets.
- When using the Intel NIC, packet losses started earlier than when using the 1 GE SCAMPI adapter, which was able to process much more packets without losses.
- The maximum IP layer bit rate that could be processed when using the Intel NIC was 898 Mb/s for 1500-byte packets.
- The maximum IP layer bit rate that could be processed when using 1 GE SCAMPI adapter was 538 Mb/s for 1500-byte packets.
- Lower maximum throughput of 1 GE SCAMPI adapter corresponds to the real achievable throughput of PCI 32-bit/33 MHz bus.

Latest firmware and driver version

Later in the project we repeated the packet capture test with the latest version of adapter firmware and driver, which already supported monitoring capabilities (but in this particular test we did not use any monitoring capability, we just read packets). The results are summarized in Table 2.

We can observe the following points when comparing the latest version of firmware and driver for the 1 GE SCAMPI adapter with the initial version tested in the previous experiment:

	SCAMPI adapter	
	1500-byte	64-byte
4000 p/s	0%	0%
5000 p/s	0%	0%
10000 p/s	0%	0%
17500 p/s	0%	0%
25000 p/s	0%	0%
37500 p/s	0%	0%
50000 p/s	0%	0%
75000 p/s	18%	0%
100000 p/s	NA	0%
150000 p/s	NA	0%
250000 p/s	NA	0%
500000 p/s	NA	0%
550000 p/s	NA	6%
600000 p/s	NA	15.5%

Table 2: Packet loss rate for packet capture with 1 GE SCAMPI adapter with latest version of firmware and driver

- When capturing 1500-byte packets, we could capture up to 50000 packets per second without any packet loss (it was 37500 packets per second with the initial version). Packet loss rate at 75000 packets per second was 18% (it was 40.1% with the initial version). The maximum achieved data rate was aproximately 738 Mb/s. A theoretical throughput of PCI bus running at 32-bit / 33 MHz is approximately 1 Gb/s. However, in practice the achievable throughput is always lower. It is likely that we reached the limit of PCI bus throughput.

- When capturing 64-byte packets, we could capture up to 500000 packets per second without any packet loss (it was 17500 packets per second with the initial version). A significant improvement over the initial version was achieved primarily because more effective transfer of packets from the card to the host computer with the new driver.

4 Software header filtering with 1 GE SCAMPI adapter

Next we tested the maximum number of packets per second that can be processed with packet header filtering in MAPI. We generated testing packets in three different scenarios where the number of packets that passed through the filter and which were then sent to the application was 10%, 50% and 90% of all packets coming to the SCAMPI adapter.

In this test we used the second release of adapter firmware and driver, which had improved performance and already included support for header filtering on the adapter, which was however not utilized in this test.

The results are summarized in Table 3. The number at the top of each column indicates the percentage of packets passed through the filter.

	1500-byte			64-byte		
	10%	50%	90%	10%	50%	90%
5000 p/s	0%	0%	0%	0%	0%	0%
10000 p/s	0%	0%	0%	0%	0%	0%
17500 p/s	0%	0%	0%	0%	0%	0%
25000 p/s	0%	0.03%	23%	0%	0%	0%
37500 p/s	0%	0.09%	NT	0%	0%	62%
50000 p/s	0%	26%	NT	0%	10.6%	NT
75000 p/s	23%	NT	NT	0%	NT	NT
100000 p/s	NA	NA	NA	0%	NT	NT
250000 p/s	NA	NA	NA	12%	NT	NT
500000 p/s	NA	NA	NA	NT	NT	NT
750000 p/s	NA	NA	NA	NT	NT	NT

Table 3: Packet loss rate for software header filtering with 1 GE SCAMPI adapter

We can observe the following points:

- It was possible to process more shorter packets than longer packets and the packet rate when significant packet loss starts is almost the same as in the previous test without header filtering and was limited mostly by system ability to pass corresponding data volume from the adapter to the user space.
- When header filtering passes approximately less than 50% of packets to further processing, we can deal with higher packets rates on the monitored line than without header filtering.
- When header filtering passes more than 50% of packets then the additional overhead caused by header filtering actually causes higher packets loss than without header filtering. However, this is not the usual case in network monitoring.

5 Firmware header filtering and statistics with 1 GE SCAMPI adapter

In this test used scampi-ph1 firmware, which implements LUP (Lookup Processor) for header filtering, SAU (Sampling Unit) for sampling and STU (Statistics Unit) for statistics. This firmware requires corresponding scampi-ph1 version of device driver.

We loaded the device driver and firmware using the following sequence of commands:

```
#!/bin/sh

export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
export LD_LIBRARY_PATH=/usr/local/src/scampi/liberouter/sys_sw/libcombo:/usr/local/src/scamp

FIRMWAREDIR=/usr/local/mcs/scampi/1_02_05

# Detach child drivers to be able to unload old driver
csboot -D -v

# Unload old driver
rmmod -v scampi-c6ph1
rmmod -v scampi
rmmod -v combo6

# Load new driver
modprobe -v scampi-c6ph1

# Load firmware
csboot -v -f 8 $FIRMWAREDIR/mtx-1-2v2000.mcs \
          -f 9 $FIRMWAREDIR/mtx-1-2v2000.mcs \
  -f 0 $FIRMWAREDIR/combo6-1-2v3000.mcs

# Attach child driver
csboot -a -v
```

In order to minimize software overhead, in addition to using MAPI we also read packets and statistics directly from `libscampi` and `libcombo` libraries.

We prepared `scampi_init` utility to initialize HFE, LUP, SAU and STU blocks. The utility parses specified header filter string, prepares corresponding nanoprogram for LUP, loads content of CAM and SRAM memories and enables specified SAU and STU blocks. The utility is used as follows:

```
scampi_init: Usage: scampi_init [options] header_filter
-d device Combo6 device (default /dev/combosix/0)
-m {d | b | p} sampling mode (default deterministic)
d - deterministic
b - byte deterministic
p - probabilistic
```

```
-r threshold pass packet when threshold is reached
number of packets for d mode (default 1)
number of bytes for b mode (default 3000)
pass probability for p mode (default 0.5)
-a <0-15>SAU ID (default 0)
-t <0-255>STU ID (default 1)
-h this help message
```

Packet filtering switched off

We first tested the maximum number of packets per second that can be read from the adapter without packet header filtering. In contrast to the test in section 3 we used a newer firmware and driver version. We read packets from MAPI and directly from `libscampi`. *Important notice: this test was outdated by the test with the latest version of firmware and driver, refer to Table 2. We leave this test here just for completness. You can proceed to the test of header filtering performance below.*

The results are summarized in Table 4.

	SCAMPI adapter + libscampi		SCAMPI adapter + libscampi + MAPI	
	1500-byte	64-byte	1500-byte	64-byte
5000 p/s	0%	0%	0%	0%
10000 p/s	0%	0%	0%	0%
17500 p/s	0%	0%	0%	0%
25000 p/s	0%	0%	0%	0%
37500 p/s	0%	0%	0.47%	0.008%
50000 p/s	0.18%	0.04%	10.2%	11.8%
75000 p/s	30.1%	20.1%	37.7%	57.8%
100000 p/s	NA	39.8%	NA	62.1%
150000 p/s	NA	63.3%	NA	75.4%

Table 4: Packet loss rate for packet capture with 1 GE SCAMPI adapter with firmware and driver supporting header filtering (but with all packets passing through)

We can observe the following points:

- With the second firmware and driver release the SCAMPI adapter can capture all packets without losses up to a slightly higher rate than with the first firmware and driver release. It should be also noted that in this test we actually passed whole packets to the application, whereas in the test with the first release we only counted packets.
- MAPI adds some overhead and slightly reduces the maximum number of captured packets when compared to reading directly from libscampi.

Next we tested the maximum number of packets per second that can be processed with packet header filtering in firmware enabled. The current version of libscampid library that prepares the content of CAM and SRAM memories on the adapter supports the following subset of BPF (tcpdump style) syntax and semantics:

```
tcp
udp
[ src | dst ] host A.B.C.D
[ src | dst ] net A.B.C.D mask E.F.G.H
[ src | dst ] net A.B.C.D/E
[ src | dst ] port A
[ src | dst ] port \[A - B \]
```

The last term is actually an extension to BPF mnemonics, which allows to match a range of ports.

We generated testing packets in three different scenarios where the number of packets that passed through the filter and which were then sent to the application was 10%, 50% and 90% of all packets coming to the SCAMPI adapter.

We performed the test three times - with the first version of firmware that supported header filtering, with the latest version of firmware (version 1_current, which was slightly modified version 1_00_07) and with the latest version of firmware and more complex header filter.

First version of firmware supporting header filtering

We used `scampi_init` utility to configure the card with the following header filter `src port 2000`. This filter requires one comparison of header fields in CAM (it is not specified explicitely in the filter, but comparison in CAM is always performed, in this case it was "match all" comparison) and one arithmetic comparison of source port number using LUP nanoprogram. We used the following command:

```
scampi_init src port 2000
```

As a result the following LUP nanoprogram was created:

```
#include "instructions.def"
#define ko 0              ;deny interface
#define ok 65536          ; accept interface

EXE ok
#step
EXE ko
#step
```

We then configured Spirent generator to send some packets from source port 2000 and some packets from source port 2001.

The results of the test with the first version of firmware that supported header filtering are summarized in Table 5. The number in the left column indicates the total number of packets per second sent to the adapter (packets that passed header filter plus packets that did not pass header filter). The number at the top of each column indicates the percentage of packets that passed header.

We can observe the following points:

	1500-byte			64-byte		
	10%	50%	90%	10%	50%	90%
5000 p/s	0%	0%	0%	0%	0%	0%
10000 p/s	0%	0%	0%	0%	0%	0%
17500 p/s	0%	0%	0%	0%	0%	0%
25000 p/s	0%	0.05%	0.03%	0%	0%	0%
50000 p/s	0%	0.04%	0.76%	0%	0.03%	0.05%
75000 p/s	0.02%	11.4%	28.2%	0%	0.07%	0.06%
100000 p/s	NA	NA	NA	0%	0.03%	0.07%
250000 p/s	NA	NA	NA	0.06%	8.5%	50.1%
500000 p/s	NA	NA	NA	0.08%	54.3%	73.9%
750000 p/s	NA	NA	NA	42.1%	69.2%	83%

Table 5: Packet loss rate for firmware header filtering with 1 GE SCAMPI adapter

- With unwanted packets filtered out on the adapter and not passed to the host computer we can process much higher number of packets per second than without firmware filtering. For instance, with 64-byte packets when 10% of packets pass through the filter, we can process up to 100000 packets per second with zero packet loss and 500000 packets per second with packet loss of 0.02%.
- A good point is that we can pass data rate up to the full real achievable throughput of PCI bus. With 1500-byte packets we achieved incoming data rate (at the Ethernet frame level) of 587 Mb/s. Packet loss in high packet rates is now lower than with previous firmware and design releases.
- An issue that needs to be improved is occassional packet loss at lower packet rates.
- The number of packets counted by STU was only very slightly higher (just a few packets) than the number of packets retrieved over the PCI bus. This difference was most likely caused by a few packets that arrived between reading a packet from the driver and reading STU from libcombo library a little bit later in the application code. It is important to note that when we want to just read STU without reading the packets themselves from the driver, the firmware must be configured such that packets are not send from LUP (Lookup Processor for header filtering) to any SAU (Sampling Unit), that is they are send just to some STU (Statistical Unit). When packets are send to some SAU then those packets that are passed through some SAU must be read from the driver. When we do not catch up in reading packets from the driver, the card eventually hangs up. To make the solution more robust, we should remove this problem.

Latest version of firmware

In this test we first used the same filter configuration and we generated the same packets as in the previous test. We then repeated the test again with more complex header filter `udp and src net 10.0.0.1/32 and dst net 10.0.0.2/32 and src port [1999-2000] and dst port [3357-3358]`. This filter required one comparison of header fields in CAM (even though the filter specified comparison of more fields - protocol, source IP subnet and destination IP subnet - all these fields are

compared at once in CAM) and four arithmetic comparisons to check that source and destination port numbers are within specified ranges. We used the following command:

```
scampi_init udp and src net 10.0.0.1/32 and dst net 10.0.0.2/32 \
  and src port [1999-2000] and dst port [3357-3358]
```

As a result the following LUP nanoprogram was created:

```
#include "instruction.def"
#define ko 0 ;deny interface
#define jump 0 ;length of the condition jump
#define ok 65536 ;accept interface

GTE jump, 26, 1999
EXE ko
LTE jump, 26, 2000
EXE ko
GTE jump, 27, 3357
EXE ko
LTE jump, 27, 3358
EXE ko
EXE ok
#step

EXE ko
#step
```

We then configured Spirent generator to send some packets from source port 2000 and some packets from source port 2001, all from IP adress 10.0.0.1 to IP address 10.0.0.2.

The results of the test with the latest version of firmware are summarized in Table 6. The number in the left column indicates the total number of packets per second sent to the adapter (packets that passed header filter plus packets that did not pass header filter). The number at the top of each column indicates the percentage of packets that passed header.

We can observe the following points when compared with the first version of firmware that supported header filtering:

- When filtering 1500-byte packets we could process up to 500000 packets per second without any packet loss regardless of what percentage of packets passed through the filter (it was only 17500 to 50000 packets per second with the previous version, depending on the percentage of packets that passed through the filter). With 75000 packets per second and 10% of packet passing the filter the measured packet loss rate was higher than with the previous version.

- When filtering 64-byte packets we could process up to 500000 packets per second without any packet loss regardless of what percentage of packets passed through the filter (it was only 25000 to 100000 packets per second with the previous version, depending on percentage of packets that passed through the filter). That is a significant improvement has been achieved.

	1500-byte			64-byte		
	10%	50%	90%	10%	50%	90%
5000 p/s	0%	0%	0%	0%	0%	0%
10000 p/s	0%	0%	0%	0%	0%	0%
17500 p/s	0%	0%	0%	0%	0%	0%
25000 p/s	0%	0%	0%	0%	0%	0%
50000 p/s	0%	0%	0%	0%	0%	0%
75000 p/s	22.8%	11.4%	17.7%	0%	0%	0%
100000 p/s	NA	NA	NA	0%	0%	0%
250000 p/s	NA	NA	NA	0%	0%	0%
500000 p/s	NA	NA	NA	0%	0%	0%
750000 p/s	NA	NA	NA	48.4%	11.7%	31.3%

Table 6: Packet loss rate for firmware header filtering with 1 GE SCAMPI adapter with latest version of firmware and driver

- An issue to be improved is that even though we just counted packets on the card using STU (that is we did not read packets themselves from the driver), the card could not process packets at full line rate of 1 Gb/s at any packet length.
- We found that results did not change when we used a more complex header filter instead of a simple header filter. That is the performance was not affected by higher number of arithmetic comparisons and LUP nanoprocessor instructions performed.

6 Firmware sampling with 1 GE SCAMPI adapter

In this test we sent different rates of packets to the adapter and set SAU to pass through 10%, 1% or 0.1% of packets. For instance, we used the following command to set SAU to pass each 10th packet:

```
scampi_init -r 10 src port 2000
```

We performed the test twice - with the first version of firmare and driver that supported sampling and then again with the latest version of firmware (version 1_current, which was slightly modified version 1_00_07).

First version of firmware supporting sampling

The results are summarized in Table 7. The number in the left column indicates the number of packets per second sent to the adapter (before sampling). The number at the top of each column indicates the percentage of packets that should pass sampler. The first number before a slash is packet loss rate counted from read packets.

	1500-byte		
	10%	1%	0.1%
1000 p/s	1.5%	2.2%	2.0%
5000 p/s	1.9%	2.0%	2.0%
10000 p/s	0%	1.7%	1.0%
25000 p/s	0%	1.7%	0%
50000 p/s	0%	2.0%	0%
75000 p/s	17.6%	17.6%	17.5%
82347 p/s	23.8%	27.3%	23.7
250000 p/s	NA	NA	NA

Table 7: Packet loss rate for firmware sampling with 1 GE SCAMPI adapter

We can observe the following points:

- The indicated packet loss is a difference between expected number of sampled packets and the real number of sampled packets. Packets always go through the SAU block on the adapter. When we did not use sampling in the previous test, we just configured one of SAU blocks to deterministically pass 1 out of 1 packets. Here we configured the same SAU block to pass 1 out of 10 packets and so on. We will investigate why packet loss was higher in this test than in simle packet reading. Due to limited time we tested only deterministic sampling for 1500-byte packets. We will also test probabilistic sampling and also for smaller packets.

Latest version of firmware and driver

The results are summarized in Table 8. The number in the left column indicates the number of packets per second sent to the adapter (before sampling). The number at the top of each

	1500-byte			64-byte		
	10%	1%	0.1%	10%	1%	0.1%
1000 p/s	0%	0%	0%	0%	0%	0%
5000 p/s	0%	0%	0%	0%	0%	0%
10000 p/s	0%	0%	0%	0%	0%	0%
25000 p/s	0%	0%	0%	0%	0%	0%
50000 p/s	0%	0%	0%	0%	0%	0%
75000 p/s	1.83%	0.18%	0%	3.3%	0.33%	0%
82347 p/s	2.45%	0.25%	0%	5.82%	0.58%	0%
250000 p/s	NA	NA	NA	5.82%	0.58%	0%

Table 8: Packet loss rate for firmware sampling with 1 GE SCAMPI adapter with latest version of firmware and driver

column indicates the percentage of packets that should pass sampler. The first number before a slash is packet loss rate counted from read packets.

We can observe the following points:

- The problem of packet loss at lower packet rates that was present in the first version of firmware and driver supporting sampling was corrected and we can now sample packets with zero packet loss up to incoming rate of 50000 packets per second when deterministically sampling one from 10 packets.
- There was a packet loss at high packet rates for 1500-byte packets. This packet loss was probably caused by approaching throughput of PCI bus, as discussed in packet capture test (we have to read packets from the driver when sampling)
- There was also a packet loss for 64-byte packets at rates starting from 75000 packets per second. This was surprising because at these rates the packet capture test was able to read all packets. And in the packet capture test we were reading all packets, whereas in sampling test we are reading only a portion of packets. We can note that the packet loss rate when sampling 1% of packets is exactly 1/10th of the packet loss rate when sampling 10% of packets. Therefore it is likely that the limitation was inside or after SAU unit and not before SAU unit.

7 Firmware header filtering and statistics with 10 GE SCAMPI adapter

We performed the test twice - with the first version of firmware which already supported also 10 GE SCAMPI adapter (version 1_00_05) and with the latest version of firmware which was specifically designed to support 10 GE SCAMPI adapter (version 2_00_02).

We generated testing packets in three different scenarios where the number of packets that passed through the filter and which were then sent to the application was 10%, 50% and 90% of all packets coming to the SCAMPI adapter.

First version of firmware supporting 10 GE SCAMPI adapter

We did a preliminary test of header filtering and statistics with 10 GE SCAMPI adapter and scampi-ph1 firmware. However, this firmware was not optimized for the 10 GE SCAMPI adapter.

The results are summarized in Table 9. The number in the left column indicates the total number of packets per second sent to the adapter (packets that passed header filter plus packets that did not pass header filter). The number at the top of each column indicates the percentage of packets that passed header.

	1500-byte			64-byte		
	10%	50%	90%	10%	50%	90%
10000 p/s	8.9%	0%	0.9%	0.31%	0%	0.05%
25000 p/s	21.7%	0.05%	2.6%	1.4%	0%	0.2%
50000 p/s	48.6%	0%	4.7%	2.8%	0%	11.6%
75000 p/s	34.1%	14.0%	27.9%	0%	0%	0.8%
100000 p/s	91.1%	3.9%	46.3%	5.7%	15.3%	0.84%
250000 p/s	NT	NT	NT	15.7%	23.6%	47.8%

Table 9: Packet loss rate for firmware header filtering with 10 GE SCAMPI adapter

We can observe the following points:

- Packet loss rate was higher with 10 GE SCAMPI adapter than with 1 GE SCAMPI adapter at the same packet rate. Packet loss rate also showed strange patterns. Sometimes there was lower packet loss with higher percentage of packets passing through header filter than with lower percentage of packets. In some cases no packets passed through at a first try and we needed to repeat the same test and then packets started to pass.

Latest version of firmware

The results are summarized in Table 10. The number in the left column indicates the total number of packets per second sent to the adapter (packets that passed header filter plus packets that did not pass header filter). The number at the top of each column indicates the percentage of packets that passed header.

We can observe the following points:

- Packet loss rate was lower than with the first version of firmware that supported 10 GE SCAMPI adapter. However, packet loss rate was still high and will need to be eliminated

	1500-byte			64-byte		
	10%	50%	90%	10%	50%	90%
10000 p/s	1.2%	0%	0.14%	0.09%	0%	0.01%
25000 p/s	2.7%	0%	0.28%	0.21%	0%	0.03%
50000 p/s	5.5%	0%	0.68%	0.46%	0%	0.05%
75000 p/s	0%	0%	5.8%	0%	0%	0%
100000 p/s	11.2%	0%	1.3%	1.0%	0%	0.11%
250000 p/s	27.5%	6.2%	NT	2.3%	0%	0.23%

Table 10: Packet loss rate for firmware header filtering with 10 GE SCAMPI adapter with latest version of firmware

for the practical use of 10 GE SCAMPI adapter. There were strange patterns - packet loss was zero when 50% of packets passed the filter, but it was higher when only 10% of packets passed the filter. This effect was present for both 1500-byte and 64-byte packets There was also one occasion when packet loss was zero for 64-byte packets when 90% of packets passed the filter, but it was higher for lower packet rates.

8 Firmware sampling with 10 GE SCAMPI adapter

In this test we sent different rates of packets to the adapter and set SAU to pass through 10%, 1% or 0.1% of packets.

We performed the test with the latest version of firmware (version 2_00_02).

The results are summarized in Table 11. The number in the left column indicates the number of packets per second sent to the adapter (before sampling). The number at the top of each column indicates the percentage of packets that should pass sampler. The first number before a slash is packet loss rate counted from read packets.

	1500-byte			64-byte		
	10%	1%	0.1%	10%	1%	0.1%
1000 p/s	0%	0%	0%	0%	0%	0%
5000 p/s	0%	0%	0%	0%	0%	0%
10000 p/s	0%	0%	0%	0%	0%	0%
25000 p/s	17.4%	0%	0%	0%	0%	0%
50000 p/s	NT	0%	0%	0%	0%	0%
75000 p/s	NT	0%	0%	0%	0%	0%
82347 p/s	NT	0%	0%	0%	0%	0%
250000 p/s	NT	18.1%	4.9%	0%	0%	0%
500000 p/s	NT	NT	NT	48.7%	0%	0%
1000000 p/s	NA	NA	NA	NT	0%	0%
2500000 p/s	NA	NA	NA	NT	28.5%	28.6%

Table 11: Packet loss rate for firmware sampling with 10 GE SCAMPI adapter

We can observe the following points:

- Packet loss rate showed a good characteristics in that it was reliably zero until certain packet rate. However, for 1500-byte packets losses started rather early, when the portion of packets that passed the sampler was still significantly lower than PCI bus throughput. Packet loss rate for 64-byte packets was much better, even though it started significantly earlier than at full link rate.

9 Packet overhead test

We used the PAPI (Performance API) [1] version 3.0 beta 2 to find the number of instructions and cycles required to process one packet in various scenarios. The configuration is shown in Fig. 3. The SCAMPI architecture was installed on the PC on the left. The PC on the right was used to generate packets satisfying required criteria, such as including a specified text string.

A sample program written on top of MAPI to check various scenarios of packet processing is shown in Appendix A. First, a new stream of packets was created. Then, various functions were applied to this stream one at a time or several of them together. Finally, part of the code in the sample program and inside the MAPI were wrapped around by calls to PAPI to find the overhead of the particular piece of code. We will describe these PAPI calls in more detail below.

We used a set of calls to PAPI shown in Appendix B to initialize PAPI library and start counting of virtual instruction and cycle counters in the client application written on top of

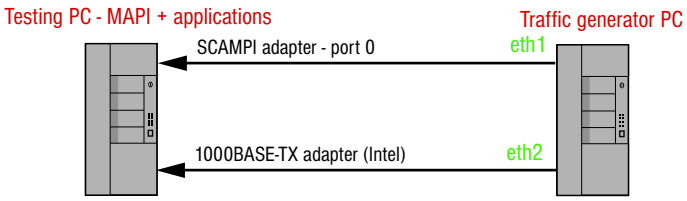

Figure 3: Configuration for the packet overhead test

MAPI, as well as inside the mapid daemon. We then used a set of calls to PAPI shown in Appendix C to read virtual instruction and cycle counters in various places inside the code.

The number of instructions and cycles measured as required for processing of one packet when various functions were applied to a flow is given in Table 12. The packet processing overhead includes resources spent in MAPI internal function `mapid_process_pkt()` and all functions applied to a packet from this function.

The number of instructions and cycles measured as required for certain MAPI calls, which are the most important calls from the performance viewpoint, are given in Table 13.

The number of additional instructions measured as required for processing of one packet when more flows were opened and the pacticular function were applied to each of these flows are indicated in Table 14. For example, when three flows are opened, the particular MAPI function is applied to all three flows and an arriving packet also belongs to all three flows, then the processing overhead for this function per packet will be the sum of the number indicated in Table 12 and twice the number indicated in Table 14.

The numbers in these tables include the overhead of PAPI itself. In order to establish the overhead of PAPI itself we used two consecutive calls to individual PAPI functions, with no intervening program code between them, and summarized overhead of those functions used in the PAPI code fragment described in Appendix B. One set of calls to PAPI required 265 instructions or 700 cycles. The PAPI overhead should be deducted from the packet and MAPI function overhead tables.

Comments and observations:

- Unlike numbers of instructions, numbers of cycles differed in different test runs. We investigated this behaviour and thanks to advice from PAPI developers in PAPI mailing list [2] we managed to reduce this effect by proper sequence of PAPI calls. Numbers of cycles in tables are averages from 5 test runs rounded to 100 cycles.

- Packet processing overhead when just the function `COUNTER` was applied to a flow and the SCAMPI adapter was used was the same as when no function was applied to a flow. The reason is that packets and bytes were counted inside the SCAMPI adapter driver and this counting is performed all the time.

- We can see that the packet processing overhead when using the SCAMPI adapter is comparable to overhead when using the Intel Gigabit Ethernet adapter. The reason is that all packet processing is now done in software, rather than in firmware on the SCAMPI adapter. We will make a new round of tests when development of processing in firmware is completed.

Functions applied	Packet size	Packet processing overhead	
		SCAMPI adapter	Intel adapter
No function	any	321 ins. 1500 cyc.	318 ins. 900 cyc.
COUNTER	any	321 ins. 1500 cyc.	371 ins. 700 cyc.
BPF_FILTER	any	601 ins. 2000 cyc.	490 ins. 2800 cyc.
SAMPLE_PACKETS	any	379 ins. 800 cyc.	376 ins. 3700 cyc.
TO_BUFFER	any	537 ins. 8800 cyc.	8316 ins. 14700 cyc.
BPF_FILTER + TO_BUFFER	any	817 ins. 11900 cyc.	8491 ins. 15500 cyc.
COUNTER + BPF_FILTER + SAMPLE_PACKETS + STR_SEARCH + TO_BUFFER	1500 bytes	18850 ins. 26500 cycles	8541 ins. 3500 cyc.
STR_SEARCH	64 bytes	1393 ins. 3900 cycles	1388 ins. 2900 cyc.
STR_SEARCH	128 bytes	2149 ins. 3400 cycles	2144 ins. 2800 cyc.
STR_SEARCH	256 bytes	3697 ins. 4400 cycles	3692 ins. 3800 cyc.
STR_SEARCH	512 bytes	6757 ins. 7500 cycles	6752 ins. 6200 cyc.
STR_SEARCH	1024 bytes	12913 ins. 14500 cycles	12908 ins. 11300 cyc.
STR_SEARCH	1500 bytes	18459 ins. 20300 cycles	18452 ins. 15600 cyc.

Table 12: Packet processing overhead in instructions and cycles

Functions applied	mapi_get_next_packet()		mapi_get_result()	
	SCAMPI adapter	Intel adapter	SCAMPI adapter	Intel adapter
COUNTER	NA	NA	7152 ins.	553 ins.
TO_BUFFER	513 ins.	1246 ins.	NA	NA
BPF_FILTER + TO_BUFFER	522 ins.	1246 ins.	NA	NA

Table 13: MAPI function overhead in instructions and cycles

- The only case when the packet processing overhead depends on the packet size is then the function STR_SEARCH is applied to a flow. Obviously, the longer the packet, the more resources are required to search its payload.
- The overhead of mapi_get_result() function for PKT_COUNTER is bigger when the SCAMPI adapter is used, because the counter

Functions applied	Additional packet overhead	
	SCAMPI adapter	Intel adapter
`COUNTER`	21 ins.	74 ins.
`TO_BUFFER`	237 ins.	not supported
`BPF_FILTER` (the same filter)	86 ins.	82 ins.
`BPF_FILTER` (different filters)	274 ins.	191 ins.
`STR_SEARCH` (the same filter)	95 ins.	96 ins.
`STR_SEARCH` (different filters)	18156 ins.	4295 ins.

Table 14: Additional packet overhead for each additional flow opened

10 Application test - Intrusion detection

10.1 Description

MAPI offers all the functionality needed to build a fast intrusion detection system. We have implemented such an intrusion detection system, called *sids*. *Sids* takes as an input simple rules describing potential attacks and translates them into the appropriate MAPI functions. Each rule is mapped into a flow and can perform header analysis as well as packet payload inspection. The format of rules given as input to *sids* is fixed. As an example

```
count tcp and dst port 80; content: /bin/perl.exe; id:100;
```

describes a rule that searches for "`/bin/perl.exe`" pattern inside TCP packets that are destined to port 80. The first keyword describes what action should be done if a packet matches the rule. So far, *sids* supports counting the packets matching the rule (keyword "`count`" in the previous example) and copying them into files (keyword "`copy`"). The second argument is a BPF filter, describing the header we search for. Futhermore, using keyword "`content`" *sids* can perform pattern matching. One rule can contain more than one patterns to be searched.

As MAPI supports flow cooking, *sids* packet inspection can become stateful so as not to miss attacks on packet borders. Furthermore, packets can be read from trace files either on *tcpdump* or *dag* format. For practical purposes, we have developed a tool, *snort2sids* that converts precisely Snort rulesets into *sids* rules. This tool saves users from writing rules with hand as Snort[1] rulesets are large, frequently updated and publicly available.

Sids also comes with the ability of logging events into a database. The database supported is MySQL and Snort's Relation Schema for logging events into the database is followed. This way application that have been developed for displaying Snort log events can also be used for sids. Such an application is ACID, a web-based tool written in PHP, that presents Snort results in a graphical example. An example screenshot can be viewed in Fig.4, where the network protocol breakdown of attacks detected is displayed.

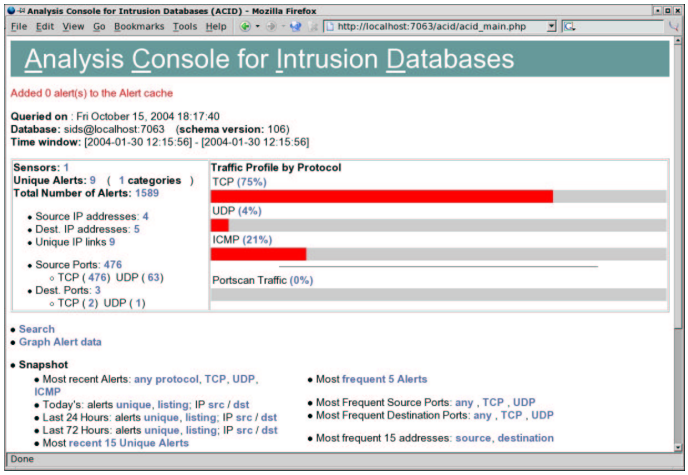

Figure 4: ACID on top of sids

[1] http://www.snort.org

10.2 Performance

We have measured the performance of *sids* under various conditions. The results presented here are preliminary as both MAPI and *sids* are in development stage but are representative of MAPI's processing capability. In our experiments, we used live Ethernet traffic at 100Mbps rate. Two machines connected back to back were involved in our measurements, a sender and a receiver. The intrusion detection system and MAPI daemon were running on the receiver, a Pentium 4 3GHz with 1GB main memory and 256KB L1 cache. All packets were uniform (tcp and destination port 5001) and were generated with *ttcp*[2] tool.

First, we examined how the processor load scales with the number of rules. Each rule was describing an attack on a specific port and more precisely attacks on a Web server. We tested two implementations of MAPI: an unoptimized one, where all flows are traversed linearly and an optimized one, where only the flows needed are examined (optimization is based on destination port). In Figure 5, we observe that *sids* can examine up to 500 rules without packet loss for the unoptimized version but the load remains stable for the optimized one. In the case of optimized version, no rules matched the traffic examined as we had no web traffic and consequently redundant flow traversal was avoided.

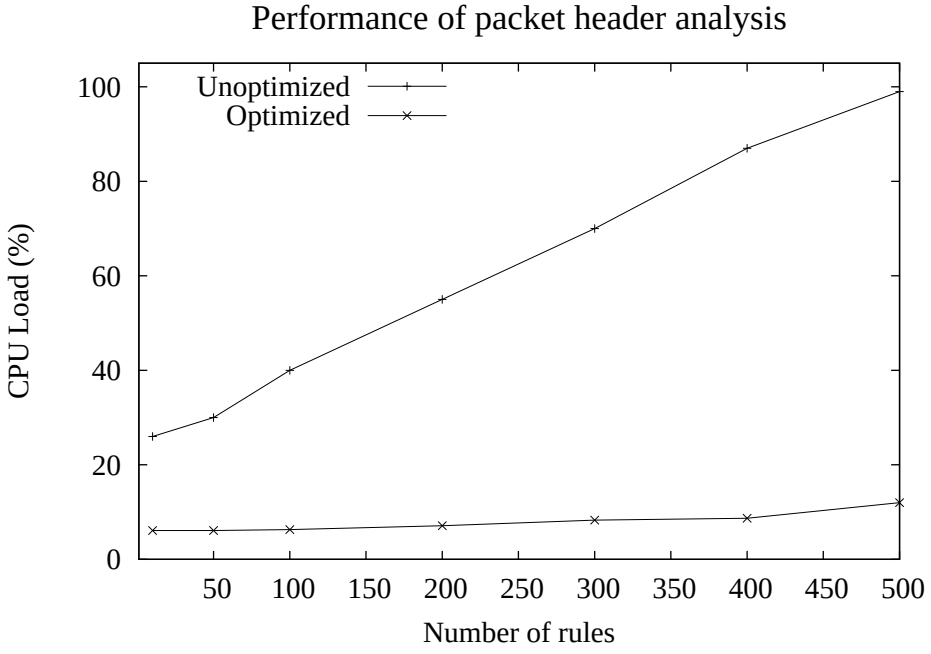

Figure 5: Performance analysis of header-only rules

[2]http://www.pcausa.com/Utilities/pcattcp.htm

Our second experiment was focused on measuring the performance of *sids* in case where rules search for patterns inside the packet. We used the unoptimized version of MAPI as no header description was provided by the rules, thus no optimization could be performed. In Figure 6, we see that *sids* can examine up to 150 rules containing patterns. The payload of each packet was constant, as generated by the *ttcp* tool (–abcde...–). Optimization on pattern matching is in progress and major performance gains are expected.

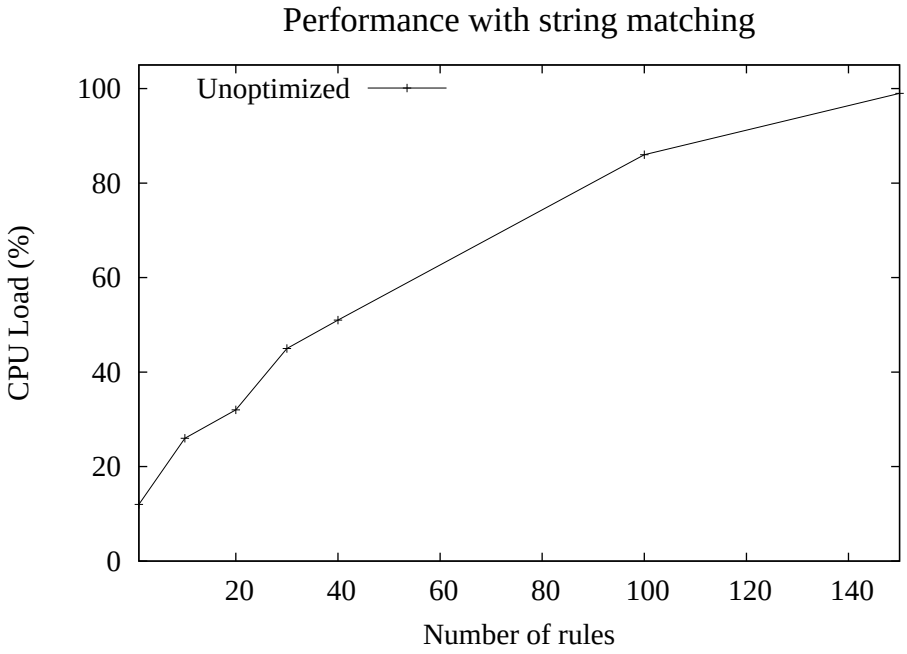

Figure 6: Performance analysis when rules contain patterns

10.3 Piranha: A Fast Lookup Pattern Matching Algorithm for Intrusion Detection

Pattern matching inflicts a significant cost to the performance of a nIDS. Previous research results suggest that 30% of total processing time is spent on pattern matching[5], while in some cases, like Web-intensive traffic, this percentage raises up to 80% [6]. Apart from processing time, memory demands of a NIDS may reach at high levels due to ruleset growth. Although algorithms with low memory demands have been developed, their performance in comparison with algorithms that consume more memory is still poor. Given the fact that link speed increases every year, pattern matching evolves to a highly demanding process that needs special consideration. Minimizing demands of pattern matching leaves headroom for further heuristics to be applied for intrusion detection, like anomaly detection or sophisticated preprocessors.

In our effort to decrease the overhead implied by pattern matching, we have designed and applied a pattern-matching algorithm called Piranha. The Piranha algorithm is based on the idea that if we find the rarest 4-byte substring of a pattern inside the packet payload then we assume that this pattern matches. Each pattern is now represented by its least popular 4-byte sequence, where popular reflects the number this substring was observed on all patterns. For

all instances of the rare substring, `Snort` is instructed to check the corresponding rule. Piranha itself can only handle patterns with length greater or equal to 4. For completness reasons, patterns with length less than 4 are handled separately.

Preprocessing

Piranha treats every byte-aligned pattern as a set of 32-bit subpatterns. For example, the pattern "/admin.exe" (R1) is considered as the set of all its 32-bit byte-aligned subpatterns, that is "/adm","admi","dmin", 'min.","in.e", "n.ex" and ".exe". The 32-bit partitioning was chosen as integers can perform fast operations. Pattern matching can then be formulated in terms of an AND operation. Every pattern is represented by a gate. The gate has as many inputs as the number of its 32-bit subpatterns. Each input represents whether the 32-bit subpattern has appeared in the payload or not. The gate for R1 can be seen on the right side of Figure7 with all its subpatterns consisting the inputs of the gate. Initially, all inputs are set to zero and whenever the sequence is seen on the packet then the input is switched on. Just like the behavior of an AND gate, if all inputs of the gate are switched on then output is set to true, that is we have a possible match as all sequences that form the pattern have appeared. However, the output must not be regarded as an exact match. For example, if the packet payload is "/admAAAdmin.exe", then, despite the fact that all 4-byte sequences for R1 have appeared, the pattern itself does not match. Each time the output of the gate is switched on, we consider it as false positive and `Snort` is instructed for further inspection. In order to find fast which inputs to switch on, an index table is maintained. The index table keeps for all 4-byte sequences a list of all patterns that contain them. For example, if we assume that we only have the patterns "/admin.exe" (R1) and "/admin.sh" (R2), a view of the table is displayed in Figure 7. Sequences "/adm","admi","dmin" and "min." appear in both patterns while ".exe","in.e" and "n.ex" exist only in R1 and "in.s" and "n.sh" only in R2. Each time a node of index table is reached then the appropriate input is switched on. As an example, if payload is "min.exe", we firstly access the "min." entry of index table and we switch on the `min.` inputs for R1 and R2, then we access the "n.exe" entry and switch on the input for R1 and finally the ".exe" entry is traversed. The performance of Piranha for a subset of our packet traces, in terms of running time and false positives per packet, is displayed in Table 15.

Although gates present a low rate of false positives, their performance is poor as a lot of steps and transitions are needed in order to take a decision whether a pattern matches or not. In a typical case, the index table is firstly accessed , then the appropriate input is switched on and then the whole gate is checked if all inputs are switched on. In our effort to reduce the number of steps – and consequently memory accesses – an optimization phase takes place. The optimization phase involves the procedure of selecting one input for each gate, a representative sequence. The rarest sequence is chosen as representative, that is the sequence found in the least number of rules. In order to find which is the sequence that is appeared the least times in rules the index table is consulted. All other inputs are removed from the gate as well as the corresponding nodes from the index table. For example, trying to optimize our previous example we keep sequence "n.ex" as representative for pattern R1 and "n.sh" for R2. The optimized view of index table can be seen at Figure 8. After the optimization phase, every gate has only one input so it can be totally removed (output is equal to input) as we can use the index table for the searching phase – if a node of the index table is reached then a possible match is triggered –.

The effect of optimization is shown in Table 15, in terms of running time and false positives.

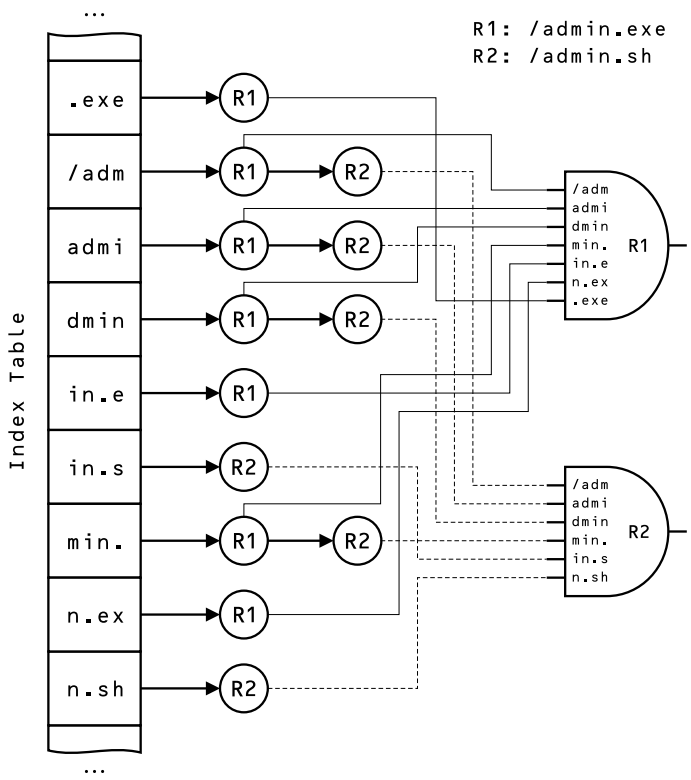

Figure 7: An example of index table and gates for two patterns. When all inputs of gate are switched on then pattern is possibly matched

The "full gates" column represents the unoptimized case of Piranha, whether the "representative sequence" refers to the optimized case. Although false positives per packet increase as now only one input triggers possible match, the performance increases due to decrease of steps and compactness of memory footprint. Performance is increased by up to 36% even if false positives are two to three times more. With further optimization during the searching phase as it is described in Section 10.3, false positives and running time drop significantly.

Searching

The searching phase of Piranha is straightforward. For each 4-byte sequence of the packet payload, the index table is consulted in order to find the patterns that contain this sequence. All these patterns are then sent to `Snort` for further inspection. Following our previous example, if payload is "/login.sh" we have to check sequences "/log","logi","ogin", "gin.",in.s" and "n.sh". According to the index table "n.sh" is found to pattern R2 so we assume that R2 is matched. The rest of the sequences are not contained in any pattern so no checks are necessary. Such an approach would trigger further inspection multiple times for each packet, as shown in Table 16 (`No check` case). We observe that, in average case, in an unoptimized search we trigger one rule per packet which is prohibitive in terms of performance. In our effort to reduce false positives, we perform a trivial check before the decision that a pattern is matched. The last two characters of the pattern are checked against the corresponding two characters in payload

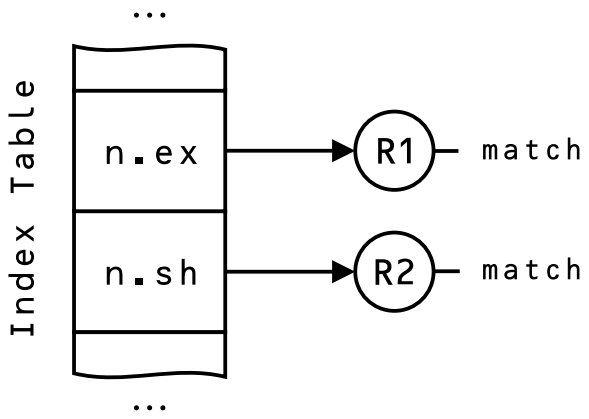

Figure 8: Optimized view of index table

	Full gates		Representative sequence	
	Running time	False positives	Running time	False positives
forth.web	37.71	0.61	24.06	1.65
forth.tr	36.14	0.29	27.60	0.67
forth.tr2	34.58	0.29	27.04	0.63
ideval2	12.07	0.33	9.58	1.06
ideval3	13.16	0.25	10.68	0.93

Table 15: Effect of optimizing gate inputs. False positives increase but running time decreases as less steps and memory are required

and if the check succeeds then further inspection is triggered. The effect of this optimization is summarized in Table16. In some cases, up to 75% of triggers are eliminated while the minimum reduction reaches 50%.

	No check	Check last 2 bytes
forth.web	1.65	0.62
forth.tr	0.67	0.24
ideval2	1.06	0.32

Table 16: False positives per packet without and with checking last 2 bytes of pattern against payload

10.4 Experiments

We evaluated the performance of Piranha against E^2xB and MWM algorithm in *Snort 2.2* using a set of packet traces. All Snort preprocessors were disabled.

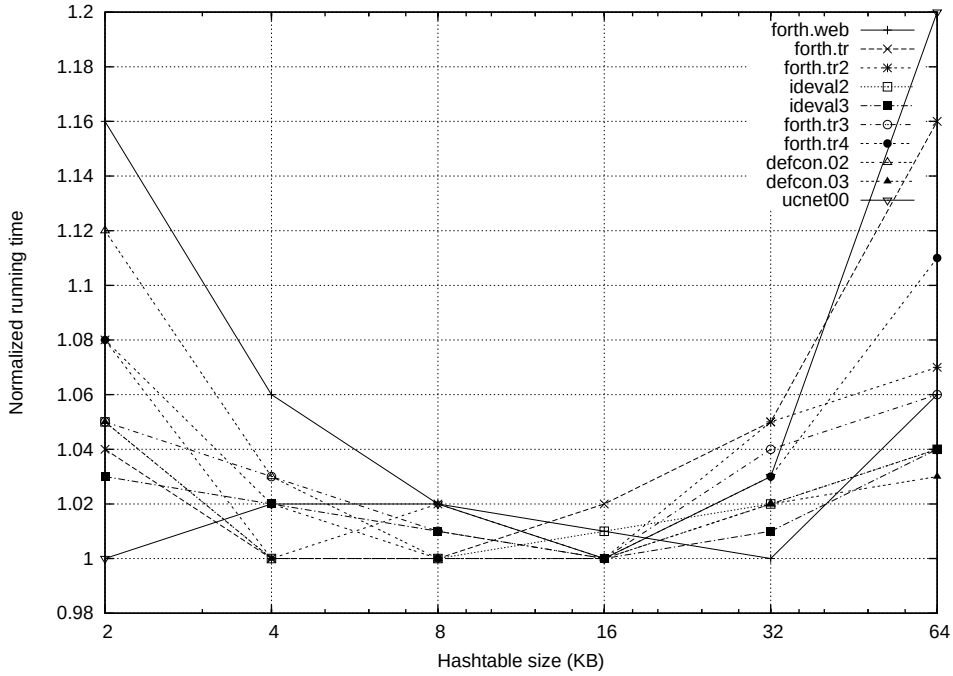

Figure 9: Effect of hashtable size on running time

Environment

All the experiments were run on a Pentium 4 processor running at 2.80GHz, with 8KB of L1 cache, 512KB of L2 cache and 1GB of main memory. The host operating system is Linux (kernel version 2.4.0, Redhat 9.0). We use five sets of packet traces from diverse environments. The first set consisted of a full packet trace containing Web traffic (`forth.web`), generated by concurrently running a number of recursive `wget` requests on popular portal sites from a host within the FORTH network. The second set contains two full packet (`forth.tr` and `forth.tr2`) traces collected in a local area network at Institute of Computer Science inside FORTH. The third set includes a full-packet trace from the DEFCON "capture the flag" data-set (`defcon.02`). This trace contains numerous intrusion attempts. The fourth set consisted of two full-packet traces (`ideval2` and `ideval3`) which were collected during the DARPA evaluation tests at MIT Lincoln Laboratory. Finally, a header-only trace with uniformly random payload (`ucnet00`) collected on the OC3 link connecting the University of Crete campus network (UCNET) to the Greek academic network (GRNET)[7] is used.

Effect of hashtable size

A complete index table of 32-bit-long patterns would normally contain 2^{32} entries, an outrageous number in terms of memory usage. In order to keep the memory footprint as small as possible, the index table was implemented as a hashtable. As the memory footprint and locality of accesses is critical to the performance of the algorithm, we determined the optimal size of the hashtable by obtaining the running time for different sizes and for all available traces. Results are summarized in Fig.9. Running times for each set are presented normalized to the lowest

value. The time was measured using the *time* facility of operating system.

Small hashtables suffer from conflicts and consequently longer chains have to be traversed in order to find the correct index. A large hashtable, on the contrary, has less conflicts but for every access a performance penalty is paid due to poor cache behavior. We observe that optimal size of the hash table for most of the traces is around 16KB and this is the size we used for all our experiments presented in the paper.

Comparison against other algorithms

	Piranha		MWM		E^2xB			
	pattern length		pattern length		pattern length			
	>=4	all	>=4	all	>=4	all	Piranha vs. MWM	Piranha vs. E^2xB
forth.web	21.05	30.17	25.32	33.59	28.86	34.12	10.18	11.57
forth.tr	23.78	30.78	30.80	35.65	29.80	31.18	13.66	1.28
forth.tr2	26.55	30.37	30.23	36.12	29.91	30.46	15.91	0.29
ideval2	8.49	11.36	9.68	12.70	10.84	13.25	10.55	14.26
ideval3	9.88	12.89	11.26	14.58	12.69	15.25	11.59	15.47
defcon.02	7.06	9.91	8.99	9.97	9.42	9.96	0.60	0.50
defcon.03	7.20	8.74	8.59	9.20	8.18	8.99	5.00	2.78
ucnet00	3.11	3.59	3.48	4.21	3.59	3.81	14.72	5.77

Table 17: Effect of small patterns on running time

We compared Piranha against MWM[8] and E^2xB[9, 10] on all available traces. In our experiments, we measured running time in user space (kernel time was negligible). Results are presented in Fig.10. Times are presented normalized against the running time of Piranha algorithm.

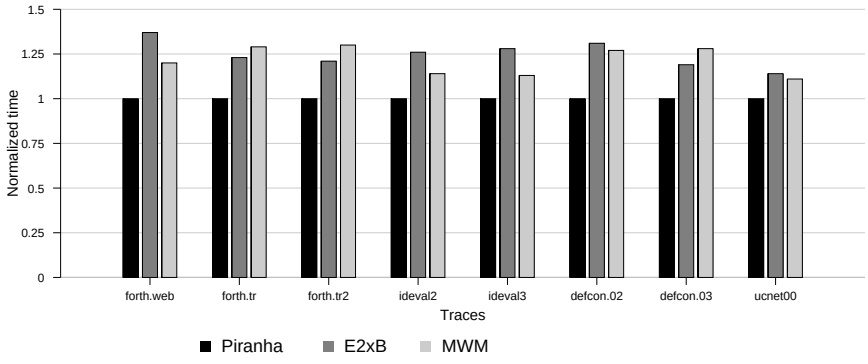

Figure 10: Running time for E2xB,Piranha and Mwm for patterns with length greater or equal to 4

The performance of Piranha is consistently better compared to other algorithms. Improvement ranges between 10 and 23.50%, with the results remaining the same for the rest of the

traces that are not displayed in Fig.10. We also compared our algorithm with AC-Banded[11], an optimized implementation of Aho-Corasick[12], but running time of AC-Banded was two to four times the time of our algorithm. Results in Fig.10 are for patterns with length greater or equal to four, as four is the length that can be natively handled by Piranha. For completness reasons, the case of small pattern was also implemented. Small patterns impose a performance bottleneck for both Piranha and MWM as well as E^2xB. MWM can natively handle patterns with length greater or equal to two while patterns with length one are examined separately. The overhead that small patterns impose in terms of running time can be seen on Table 17. In average case, running time was decreased by 25% for Piranha and 20% for MWM. The effect on E^2xB is smaller as it is not dependent to pattern legth but proportional to the number of patterns. In the last column of the table we can observe the performance benefit of Piranha against MWM for all pattern lengths. Despite the performance bottleneck, our algorithm still performs better for all available traces, except the case of `defcon.02` trace where improvement is marginal. However, our main contribution is focused on patterns with a fair enough large size as only 3% of patterns have length less than four.

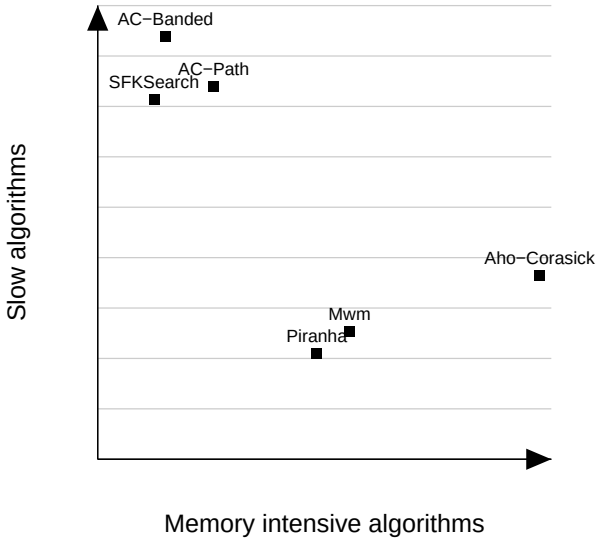

Figure 11: Memory usage against processing time

Piranha does not only perform better in terms of processing time but also in terms of memory usage. While MWM requires 45MB of memory to process the full ruleset, AC-Banded 96MB and Aho-Corasick 140MB, Piranha consumes only 37MB. Efforts have been made recently in order to develop algorithms with low memory consumption. Tuck et al. in [13], have developed two modified versions of Aho-Corasick, AC-Bitmap and AC-Path, that reduce memory usage. AC-Bitmap needs 20MB memory while AC-Path only 15MB. However, such algorithms present very high processing time. Comparing Piranha with AC-Bitmap and AC-Path, we observed that they need, in average, three to four times more processing time. `Snort` also comes with `SFKSearch`, an algorithm that requires only 14MB of memory, but its performance compared to others is poor – three to four times more processing time against Piranha –. The tradeoff between memory usage

and processing time can be seen in Fig.11. Algorithms with low memory usage need three to four times more processing time, while algorithms with high memory usage present high processing capacity. Although the assumption that low memory means high processing time cannot be generalized, there are strong indications that this tradeoff might hold for other algorithms that are not discussed here.

Evaluation on different architectures

We evaluated the performance of Piranha on different hardware architectures. Our testing environment, besides the machine described in Section10.4, was consisted of a Pentium Xeon 2.66 GHz with 8KB L1 cache, 512KB L2 cache and 1GB main memory and a Pentium Xeon 2.4 GHz with 8KB L1 cache, 512KB L2 cache and 512MB main memory. Results are presented in Fig.12. Running time is normalized against the time of Piranha running on P4 at 2.8GHz.

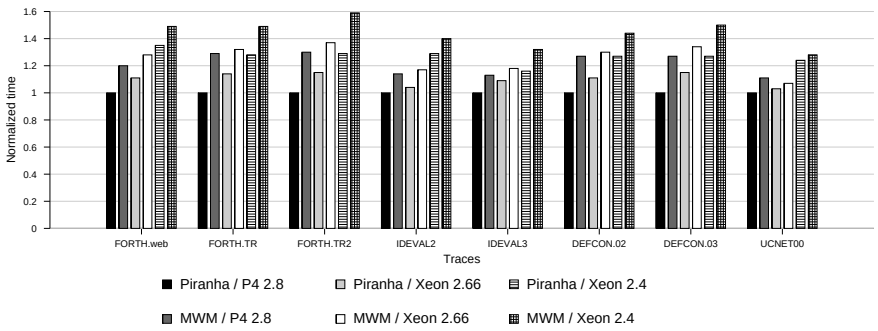

Figure 12: Performance of Piranha and MWM on different architectures

Independently of the underlying hardware platform, Piranha performs better for all traces. As processor clock speed decreases, performance of both algorithms is decreased as expected. However, the performance gap seems to decrease with the clock speed for specific traces while for others it remains constant. On Pentium Xeon 2.66GHz, improvement waves between 11.1% and 16% while on Xeon 2.4GHz between 7.8% and 18.8%.

Performance for extended rulesets

Over the last few years we have been observing an increasingly larger number of Internet-based attacks. This increase can be seen in the number of Snort rules. Indeed, as Figure 13 indicates, while in the first months of 1999 there were only one hundred rules, current rulesets consist of 2500 rules. Pattern matching algorithms should not only focus on current rulesets but should also perform well for increasingly larger number of rules. We evaluated Piranha against MWM to largest sets of rules. The extended sets were constructed by multiplying the default ruleset, with patterns permutated at random positions. According to our results, as shown in Fig.14, performance benefit of Piranha is invariant to the number of rules. The x-axis represents the number we multiplied the default ruleset, while the y-axis is the fraction of running time of MWM to the time of Piranha.

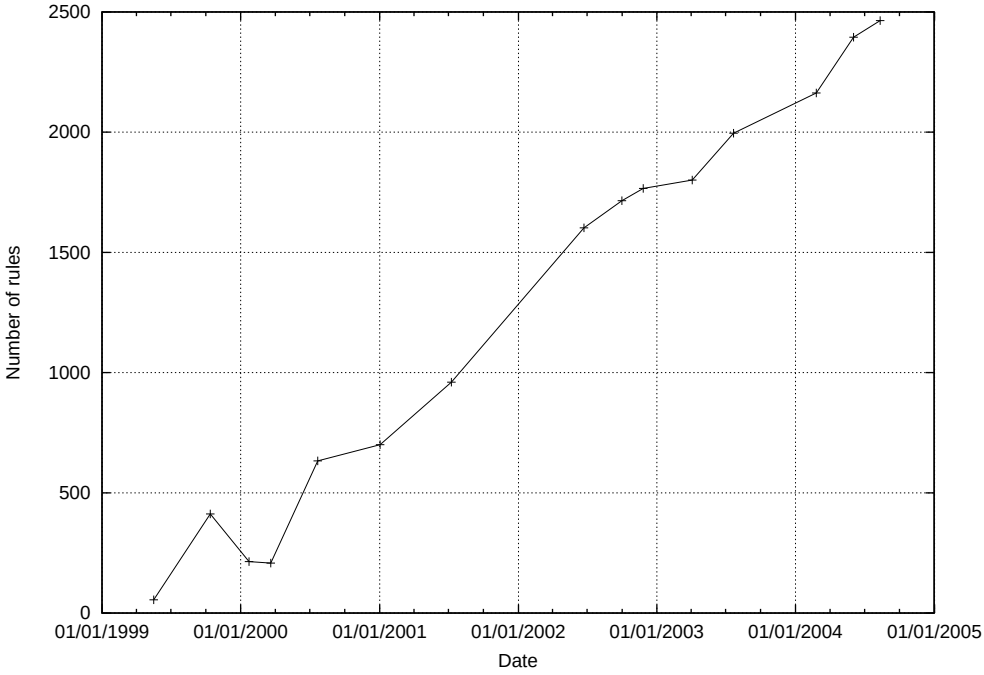

Figure 13: Evolution of Snort ruleset from 1999 until today

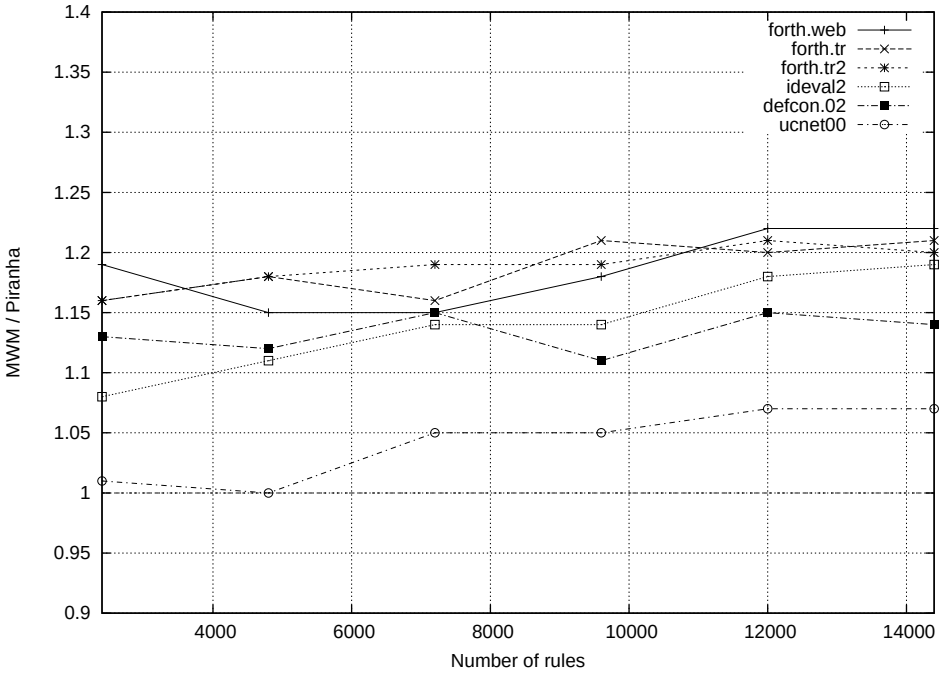

Figure 14: Relative performance (ratio of running times) of Piranha and *MWM*

11 Application test - QoS monitoring

In order to evaluate the functionality of the Quality of Service (QoS) monitoring application, some tests were conducted. These tests will validate the correctness of the measured QoS characteristics (one-way delay, jitter and packet loss) in a controlled environment.

11.1 Environment

To introduce delay and loss in the network, a Click-based[3] impairment node was configured in the network. To verify the measured QoS characteristics, they were compared to the results obtained by a Smartbits[4] network performance analysis system. This same Smartbits was used to generate the traffic. The used topology is illustrated in Fig. 15. The Smartbits system is connected to two access routers (Leucothea170 and Leucothea171). These routers contain a SCAMPI monitor running the QoS monitoring application. The application is configured to sample part of the captured packets and write useful data to the database. The core of the network only consists of one router (Leucothea173), which runs the impairment software.

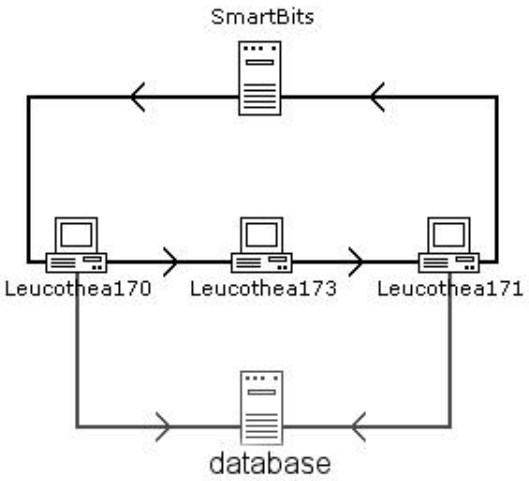

Figure 15: Experiment topology

The artificially introduced traffic generated by the Smartbits hardware contains random payload. This allows the SCAMPI QoS monitor to compute a unique hash over the individual packets. For each test a 10Mbit stream was generated during a period of 5 minutes. The frame size of each packet was 256 bytes.

11.2 One-way delay and jitter measurements

When comparing the delay and jitter obtained by the SCAMPI monitor and the Smartbits system, we varied the amount of introduced delay by the impairment node. The introduced delay was progressively increased from 0 to 0.5ms in steps of 0.05ms. Figure 16 shows the results obtained when using a sampling rate of 5% in the SCAMPI application.

The delay measured by the Smartbits software is generally 0.12ms higher than the delay measured with the SCAMPI application. This is due to the extra delay over the two links be-

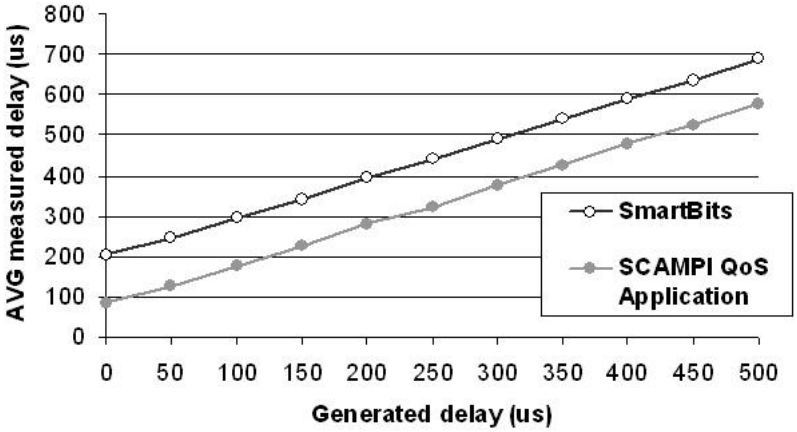

Figure 16: Delay measurements

tween the Smartbits and the access routers. Ignoring this additional constant delay experienced by the Smartbits, we see that the measured delay of the QoS application is exactly the same as the Smartbits results.

11.3 Packet loss measurements

Instead of delaying packets in the impairment node, in these tests random packets were dropped, generating packet loss. Using a probabilistic dropper, we progressively introduced more packet loss, ranging from 0% to 100% in steps of 10%. Figure 17 shows the comparison of the measured packet loss. In case of the QoS application, different sampling rates were used.

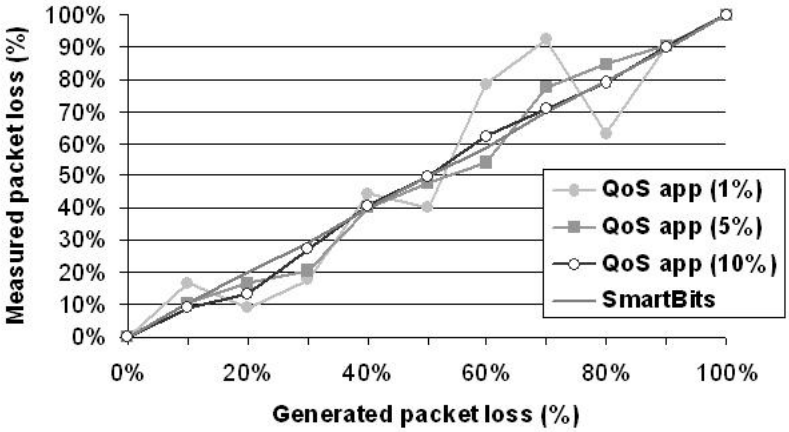

Figure 17: Packer loss measurements

The measurements with the Smartbits system show the correct packet loss. Using the QoS application, we try to approximate this correct result by choosing an appropriate packet sampling rate.

Comparing the results to the previously measured accuracy of the delay measurements, the packet loss deviates much more from the correct result as obtained by Smartbits. This is because, in case of the delay measurements, each sampled packet contributes to the average delay. Even though there is a high delay variation, the average will stay about the same. A subset of sampled packet will have the same average delay as the average of the entire set. This however is not the case with packet loss. A subset of sampled packets does not necessarily contain the same number of dropped packets as the entire flow. It greatly depends on which packets are sampled in both access points.

Figure 17 illustrates that the results improve when raising the number of sampled packets. For a sampling rate of 1%, we obtain very poor results. 5% is much better, while 10% already gives a good approximation of the real packet loss. Depending of the amount of precision needed by the application and the processing power of the SCAMPI monitor, an appropriate sampling percentage can be chosen.

11.4 Case study - measuring the QoS in an online gaming testbed

As a case study, the SCAMPI QoS monitoring application was used to measure the quality of service of playing online Xbox games on the Xbox Live network. Herefor, we used two different testbeds for the gaming tests. In the first testbed, the gaming consoles were distributed over different locations in Belgium. Two Xboxes were located in Antwerp, two in Hasselt and two in Gent (see Fig. 18). The Xboxes in Antwerp were connected to the ADSL network. In Hasselt, the Xboxes could access the Internet through Belnet, the Belgian national research network for education, research and public services. The Xboxes in Gent were able to use both access technologies, depending on the scenario. The second testbed was located in Gent at the INTEC laboratories. Here 3 Xboxes were used, each connecting to the Internet through different access technologies, i.e. Cable, ADSL and Belnet.

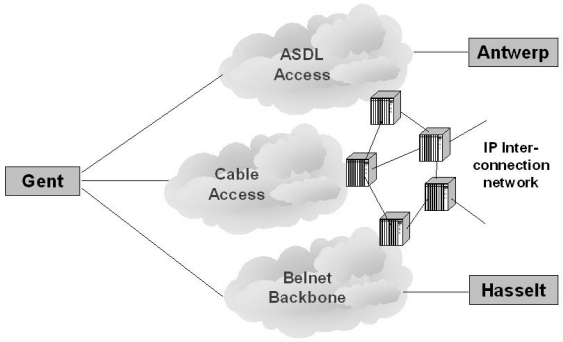

Figure 18: Setup of the online gaming testbed

To collect the measurement data, the SCAMPI QoS monitoring application was used. At each site, information about all sent and received packets is stored in a local Postgresql database. Later on, the databases of the 3 measurement sites are correlated to calculate the delay, packet loss and jitter. Both the ADSL and the cable Internet access technologies have limited upstream bandwidth (16000 Bps) and ADSL has a 3.3 Mbps downstream bandwidth limit.

All Xboxes generate traffic by playing online games (CounterStrike and Project Gotham Racing 2) and by using voice chat during the games. Since we didn't know what kind of traffic

the games generated (Client-Server or Peer-to-Peer) we had to use a fast connection for the Xbox that hosts the game. We used the Xbox connected to Belnet at the INTEC laboratories in Gent to run a Counterstrike server or host a PGR2 race.

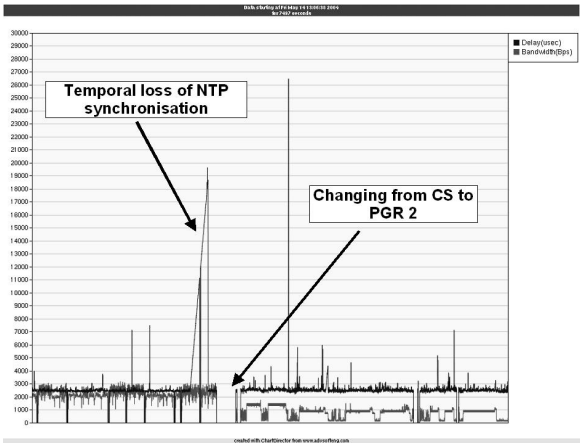

Figure 19: Delay and bandwidth from Hasselt to Gent (Screenshot of GUI)

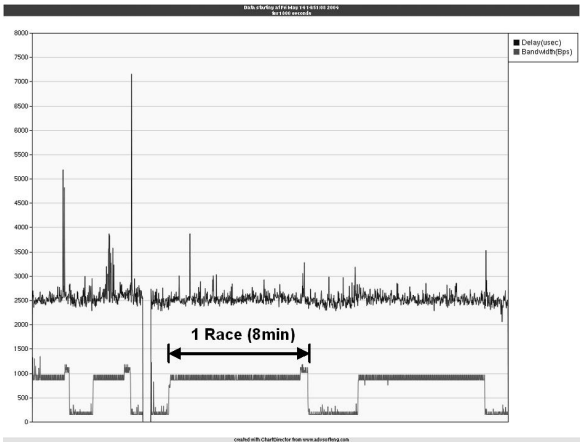

Figure 20: P2P traffic from Hasselt to Gent when playing PGR2 (Screenshot of GUI)

The screenshots in figure 19 and 20 were taken from the QoS monitoring application GUI when running the experiments. In Fig. 19 we see the importance of clock synchronisation of the measurement machines at the different sites. Because we do not have any GPS devices to do this synchronisation, we had to use the NTP protocol. In our setup, all monitoring machines are synchronised every second with the *ntp.belnet.be* time server. Because synchronisation was lost for about 10 minutes due to a network error, the measured delay was way off the correct value during this period.

Figure 20 shows the measured delay and bandwidth between Hasselt and Gent (both Belnet) when playing PGR2. This test illustrates the relatively steady delay between both sites. Between each race, the used bandwidth drops from about 800Bps to 200Bps.

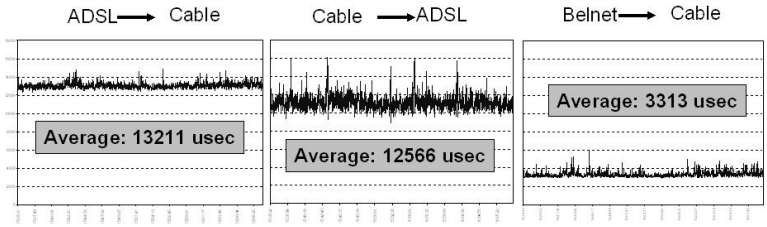

Figure 21: Delay measurement Counterstrike

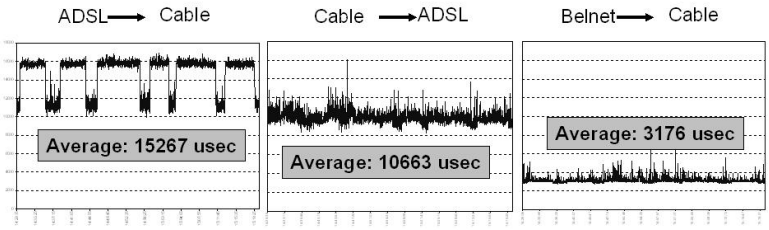

Figure 22: Delay measurement Project Gotham Racing 2

By using the SCAMPI QoS monitoring application, we could identify some differences between the traffic sent by the different Xbox Live games. The network protocol used by Counterstrike for example is based on server-client communication (i.e. there is only traffic between the server located in Gent and the clients in Antwerp and Hasselt), while PGR2 uses a peer-to-peer approach (i.e. we measured an equal amount of traffic between all participating sites). As shown in figure 21 and 22, we accurately measured the delay between the different sites with the SCAMPI software over a period of 40 minutes. In Fig. 22, we see a drop in bandwidth as well as delay between each consecutive gaming session when using the ADSL access technology.

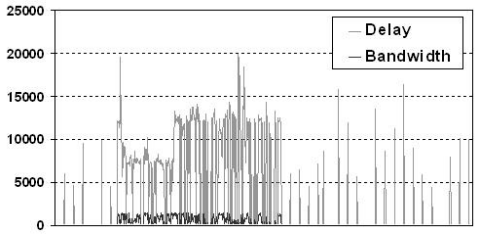

Figure 23: Influence of voice traffic on delay and used bandwidth

Because previous tests confirmed that Counterstrike uses a client-server protocol, we can clearly identify the P2P voice traffic between the different clients. Figure 23 shows the delay and used bandwidth of the voice chat. As expected, this traffic is very bursty and consumes only a small portion of the available bandwidth (max 1500 bps).

References

[1] J. Coppens, S. De Smet, S. Van den Berghe, F. De Turck, P. Demeester, "Performance Evaluation of a Probabilistic Packet Filter Optimization Algorithm for High-speed Network Monitoring", 7th IEEE International Conference on High Speed Networks and Multimedia Communications (HSNMC'04), Toulouse, France, June - July 2004.

A Sample program for the packet processing overhead test

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <sys/time.h>
#include "mapi.h"

int main(void)
{
  int fd;
  int counter_all, counter_bpf, counter_str;
  unsigned long long *cr, *cr2, *cr3;
  int bufid;
  struct mapipkt* pkt;
  int i, j;
  unsigned char *p;
  char *fce="testcombo6()";

  fd=mapi_create_flow("/dev/scampi/0");
  counter_all=mapi_apply_function(fd, "PKT_COUNTER");
  mapi_apply_function(fd, "BPF_FILTER", "src port 2005");
  mapi_apply_function(fd,"SAMPLE_PACKETS","5", "PROBABILISTIC");
  counter_bpf=mapi_apply_function(fd, "PKT_COUNTER");
  mapi_apply_function(fd,"STR_SEARCH", "www", 0, 1500);
  counter_str=mapi_apply_function(fd, "PKT_COUNTER");
  bufid=mapi_apply_function(fd, "TO_BUFFER");
  mapi_connect(fd);

  while(1) {
    sleep(1);

    cr=mapi_read_results(fd, counter_all, 1);

    pkt=mapi_get_next_pkt(fd, bufid);

    printf("size: %d\n", pkt->caplen);

    cr2=mapi_read_results(fd, counter_bpf, 1);
    cr3=mapi_read_results(fd, counter_str, 1);
    printf("pkts all:%lld bpf:%lld (%.2f\%%) str:%lld (%.2f\%%)\n",
     *cr, *cr2, ((double)(*cr2)/(double)(*cr))*100.0, *cr3, ((double)(*cr3)/(double)(*cr))*1
  }

  return 0;
}
```

B Initialization of PAPI performance analysis

```
#include <papi.h>

#define NUMBER_OF_EVENTS 3

int event_set = PAPI_NULL;
long_long values[NUMBER_OF_EVENTS], values2[NUMBER_OF_EVENTS];
long_long start_cycles, end_cycles, start_usec, end_usec;
long_long start_cycles2, end_cycles2, start_usec2, end_usec2;

int main(void)
{
  int retval;

  /* Initialize the PAPI library */
  retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT) {
    fprintf(stderr, "PAPI library init error!\n");
    exit(1);
  }

  /* Create event set */
  if (PAPI_create_eventset(&event_set) != PAPI_OK) {
    fprintf(stderr, "PAPI_create_eventset() failed\n");
    exit(-1);
  }

  /* Add events */
  if ((retval=PAPI_add_event(event_set, PAPI_TOT_INS)) != PAPI_OK) {
    fprintf(stderr, "PAPI_add_events(PAPI_TOT_INS) failed: %s\n", PAPI_strerror(
retval));
    exit(-1);
  }
  if ((retval=PAPI_add_event(event_set, PAPI_TOT_CYC)) != PAPI_OK) {
    fprintf(stderr, "PAPI_add_events(PAPI_TOT_CYC) failed: %s\n", PAPI_strerror(
retval));
    exit(-1);
  }

  /* Start counting */
  if (PAPI_start(event_set) != PAPI_OK) {
    fprintf(stderr, "PAPI_start() failed\n");
    exit(-1);
  }
```

```
  /* . . . */
}
```

C Using PAPI to read virtual counters

```
if (PAPI_read(event_set, values) != PAPI_OK) {
  fprintf(stderr, "PAPI_read()\n");
  exit(-1);
}
start_cycles=PAPI_get_virt_cyc();
start_usec=PAPI_get_virt_usec();
start_cycles2=PAPI_get_real_cyc();
start_usec2=PAPI_get_real_usec();
```

References

[1] *PAPI - Performance Application Programming Interface*, `http://icl.cs.utk.edu/papi`.

[2] Philip J. Mucci. Discussion in PAPI mailing list, `http://icl.cs.utk.edu/papi/custom/index.html?id=50`

[3] R. Morris, E. Kohler, J. Jannotti and M. F. Kaashoek, "The Click modular router", In Proc. 17-th Symposium on Operating Systems Principles, pages 217-231, 1999.

[4] SmartBits network performance analysis system, Spirent communications, "http://www.spirentcom.com/".

[5] Martin Roesch. *Snort: Lightweight Intrusion Detection for Networks*, November 1999, `http://www.snort.org`.

[6] M. Fisk and G. Varghese. *An Analysis of Fast String Matching Applied to Content-Based Forwarding and Intrusion Detection*, CS2001-0670 (updated version), University of California - San Diego, 2002.

[7] C. Courcoubetis and V. A. Siris. "Measurement and analysis of real network traffic", *Proceedings of the 7th Hellenic Conference on Informatics (HCI'99)*, August 1999, `http://www.ics.forth.gr/netgroup/publications/1999.HCI99.measurements.html`.

[8] *Snort 2.0 - Detection Revisited*, SnortFire, October 2002, `http://www.snort.org/docs/Snort_20_v4.pdf`.

[9] Kostas G. Anagnostakis and Evangelos P. Markatos and Spyros Antonatos and Michalis Polychronakis. "A Domain-Specific String Matching Algorithm for Intrusion Detection", *Proceedings of the 18th IFIP International Information Security Conference (SEC2003)*, May 2003.

[10] Evangelos P. Markatos and Spyros Antonatos and Michalis Polychronakis and Kostas G. Anagnostakis. "Exclusion-Based Signature Matching for Intrusion Detection", *Proceedings of CCN'02*, November 2002.

[11] Marc Norton. *Optimizing Pattern Matching for Intrusion Detection*, July 2004, `http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf`.

[12] A.V. Aho and M.J. Corasick. "Fast pattern matching: an aid to bibliographic search", *Communications of the ACM*, Vol. 18, No. 6, June 1975, pp. 333-340.

[13] Nathan Tuck and Timothy Sherwood and Brad Calder and George Varghese. "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection", *Proceedings of the IEEE Infocom Conference*, March 2004.