Performance Evaluation of a Probabilistic Packet Filter Optimization Algorithm for High-speed Network Monitoring

Jan Coppens, Stijn De Smet, Steven Van den Berghe,
Filip De Turck and Piet Demeester

Department of Information Technology (INTEC)
Ghent University - IMEC
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium.
Tel.: +32 9 264 99 58, Fax: +32 9 264 99 60
E-mail: Jan.Coppens@intec.ugent.be

Abstract. Because of the ever-increasing popularity of the Internet, network monitoring becomes very mission critical to guarantee the operation of IP networks, e.g. to detect network failures and stop intrusion attempts. A majority of these monitoring tasks require only a small subset of all passing packets, which share some common properties such as identical header fields or similar patterns in their data. Nowadays, next to the increasing network speed, much of these tasks become very complex. In order to capture only the useful packets, these applications need to evaluate a large set of expressions. In this paper, we present a platform independent filter and pattern matcher optimization algorithm, which reduces the required number of evaluated expressions. The performance of the algorithm will be validated both analytically and by means of a high-speed monitoring system.

1 Introduction

In today's society, computer networks are mission critical for businesses, research institutes and other organizations. The operation of the network has to be ensured at all expense. Network monitoring can be used to track down anomalies in normal network behaviour such as the failure of certain network components, detect intrusion attempts and security breaches and gather flow and network usage statistics in order to re-dimension the network over time. Network monitoring is very common at low network speeds. Nowadays, most corporate and ISP networks are widely deployed. Keeping these national and international networks up and running becomes very hard due to scope of the network. As time goes by, not only the size but also the speed of the networks increases. Network backbones with a capacity of 10Gbit and beyond are not a curiosity anymore. Monitoring techniques that used to be effective at low speeds are becoming less and less useful when applied in current high-speed backbones.

Next to the ever-increasing network speed, network monitoring applications tend to become more complex and demanding (e.g. Snort[8]). Where early monitoring applications commonly require little information from the network (e.g.

aggregated traffic statistics based on simple filter rules), more recent tools may need a much more significant amount of data, possibly including both header and entire payload. To make matters even worse, the amount of processing required on this data tends to increase. Because attackers get cleverer and find new and more complicated ways to launch network attacks, network security applications have to keep up with these new threats. When a security application needs to detect for instance an Internet worm or various other forms of cyberattacks, very computational intensive processing, such as pattern matching, is required.

In the remainder of this paper we will describe a platform independent optimization algorithm and evaluate its performance. Section 2 addresses common techniques used in network monitoring such as packet filtering and pattern matching. In section 3 we propose an optimization algorithm that reduces the number of expressions that need to be evaluated on each packet. After these initial optimization techniques, a more advance probabilistic optimization algorithm is discussed in section 4. Because we deal with a lot of different unknown parameters, a mathematical performance study in section 5 will validate the effectiveness of the algorithm. To verify the performance of the algorithm in an implemented monitoring system, section 6 compares the number of clockcycles used to evaluate multiple expressions with and without the optimization. Finally, future work and a conclusion are presented in section 7 and 8.

2 Packet Filtering and Pattern Matching

A well-known technique to eliminate unnecessary packets in a captured flow is "filtering". Packet filtering will select only those packets that share some common properties, such as identical header fields (e.g. IP protocol = 6). Unlike packet sampling, filtering is a deterministic process, i.e. the set of filtered packets does not depend on the place (in time and space) of the packets in the captured flow. Because of this deterministic nature, every packet has to be carefully analyzed in order to determine whether it belongs to the set of filtered packets. Depending on the complexity of the packet filter, this evaluation can be very computational intensive. If the filter expression consists of a logical combination[1] of multiple headerfields, the application needs to parse and verify numerous expressions before it can make a decision.

In current monitoring applications, there exist several different filtering techniques, which are mainly platform dependent. Based on the depth of the protocol stack, one of the available protocol filters is preferred. If we only consider the UNIX platform, the main protocol filter is BPF (Berkley Packet Filter). BPF (used in tcpdump[4]) allows an application programmer to write filter expressions using a straightforward syntax. A. Begel et al.[1] have implemented a basic platform dependent optimization of BPF, called BPF+. The drawback of both BPF and BPF+ is that only some "lower" level protocols are supported. These

[1] A logical combination of elements is a set of elements combined with the logical operators AND, OR, XOR, NOT and parentheses

protocols include Ethernet, IP, IPv6, ICMP, TCP, UDP... Berkley Packet Filters only support a select set of Layer 2 (datalink layer), Layer 3 (network layer) and Layer 4 (transport layer) protocols. If an application needs to go higher in the protocol stack, other protocol filters, such as "Ethereal Display Filters"[6], should be used. When going one step higher in the protocol stack, the application is confronted with a plethora of new and more complex protocols. This results in a far more time intensive parsing process.

In addition to packet filtering, pattern matching can be used to select a set of packets that contain a common pattern or string in their data. Contrary to the fixed position[2] of the header fields in packet filtering, the position of the searched pattern is unknown, if at all present in the packet, to the applications. This means that the pattern matching process is far more computational demanding than packet filtering. However, next to the simple but slow greedy search algorithms, faster and more clever algorithms are available (e.g. Boyer-Moore[2], Boyer-Moore-Horspool[3], Set-wise-Boyer-Moore-Hospool and ExB[5]). In the following section we will describe an algorithm that can be used to improve the performance of both filtering and pattern matching even further.

3 Basic Optimization of an Expression Evaluation Algorithm

Regardless whether we use filtering or pattern matching, when analyzing a captured packet, multiple expressions need to be evaluated. A concatenation of expressions or a "rule" can be for example:

```
src 10.10.10.173 and port 80 or not port 25
ip.dst eq 192.168.4.1 and http.request
find "/usr/bin/perl" or find "/bin/sh"
```

The first expression is a BPF expression, the second an Ethereal Display Filter and the last is a general pattern matching expression. Of course, a combination of the previous expression formats is possible. In this section we propose a platform independent optimization algorithm that will reduce the number of expressions that need to be evaluated. To illustrate the optimizations, we will use the BPF syntax in our algorithms and examples. Similar algorithms will be used when dealing with other languages.

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 1 to the right-most expression.

[2] In case of variable length headerfields, the position is not fixed, but can be determined by analyzing the header

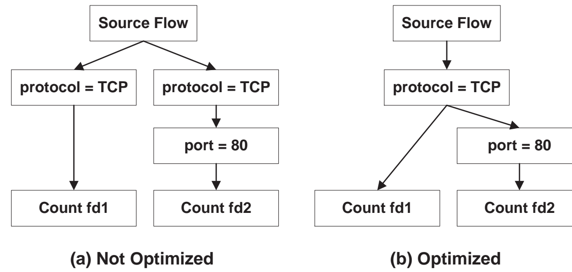


Fig. 1. Optimization of filter expressions

3.1 Basic Optimization Algorithm

In this section, we will introduce the term *atomic expression*. An atomic expression or atom is a filter expression that can not 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 Operator e | not e | (e) | Atom
Operator ::= and | or | 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.

```
(host X or host Y) and (port 1 or port 2)
host Z and (port 1 or port 2)
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
  host X refcount:1)
AND
 (OR
   port 1 refcount:2
   port 2 refcount:2)
 host Z refcount:1
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.

```
AND
   ''port 1 or port 2'' refcount:2 node:1
   OR
      host Y refcount:2 node:2
      host X refcount:1 node:3
AND
   ''port 1 or port 2'' refcount:2 node:1
   host Z refcount:1 node:4
host Y refcount:2 node:2
```

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

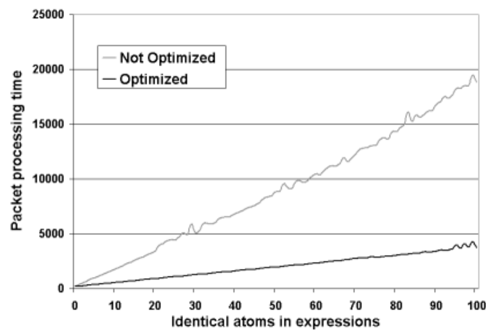


Fig. 2. Processing time of expressions containing identical atoms

3.2 Evaluating the Aggregated Expression

The evaluation step 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 2 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.

4 Probabilistic Expression Optimization Algorithm

Up until now, all atoms in an expression are evaluated in the same order as they are listed. To make some useful decision on the evaluation order of the atoms in an expression, we assume that we know the probability of the occurrence of an atom in a packet. This probability can be configured statically (e.g. when a network operator knows what traffic flows over his network) or through measurements. However, we can not evaluate each packet to obtain the required statistics, because this is exactly what we are trying to avoid. On the contrary, we can use *sampling* (probabilistic or deterministic) to select some packets in order to deduce these flow characteristics. We also can use the available evaluation information from the monitor and try to create a probability matrix. Note that this information is not that accurate because some atoms are not evaluated in case an expression short-circuits.

Let P_x be the probability that atom x is true for a packet. $1 - P_x$ then equals the probability atom x is false for an arbitrary packet. The algorithm reads:

1. Evaluate all atoms belonging to expressions that do not contain unevaluated AND | OR sub-expressions.
2. Evaluate expressions where all operands are evaluated.
3. For all remaining AND | OR expressions, compute recursively the evaluation order EO of all atoms in the (sub-)expression(s).
   - *"X and Y":* if $((1 - P_x) > (1 - P_y))$ then $EO(x) > EO(y)$
   - *"X or Y":* if $(P_x > P_y)$ then $EO(x) > EO(y)$

   i.e. For AND, compute all probabilities so both operators evaluate to false. For OR, compute all probabilities so both operators evaluate to true.

4. For each expression, order all atoms according to their evaluation order (i.e. maximize the probability an AND | OR expression short-circuits).
5. For all atoms with the highest EO in each expression:
   - Order these atoms according to the number of expressions they occur in.
   - Of all top ranked atoms, evaluate the one with the highest probability it evaluates to the required value.
6. Evaluate expressions where all operands are evaluated.
7. If there remain unevaluated expressions, return to 1

Note that the ordering process in steps 3, 4 and 5 can be done off-line or after each update of the probability matrix. To illustrate the algorithm, consider the following example. Let us assume an application configures a simple filtering rule "(SRC IP 10.0.0.1) AND (SRC PORT 80) AND (DST IP 10.0.0.2) AND (DST PORT 2045)". Network measurements have shown that the present atoms have the following probability of evaluating to false:

Atoms	$1-P_x$
(SRC IP 10.0.0.1)	50%
(SRC PORT 80)	60%
(DST IP 10.0.0.2)	20%
(DST PORT 2045)	30%

Based on the evaluation order of the individual atoms, more/less atoms will need to be evaluated in order to evaluate the expression.

Evaluation order	P
1. (DST IP 10.0.0.2)	20%
2. (DST PORT 2045)	44%
3. (SRC IP 10.0.0.1)	72%
4. (SRC PORT 80)	100%

(a) Worst case

Evaluation order	P
1. (DST PORT 2045)	30%
2. (SRC PORT 80)	72%
3. (DST IP 10.0.0.2)	86%
4. (SRC IP 10.0.0.1)	100%

(b) Random case

Evaluation order	P
1. (SRC PORT 80)	60%
2. (SRC IP 10.0.0.1)	80%
3. (DST PORT 2045)	90%
4. (DST IP 10.0.0.2)	100%

(c) Optimal case

Using the optimal evaluation order, the expression has 60% chance to short-circuit after the first evaluated atom (i.e. there is a 60% chance only one atom needs to be evaluated), 80% after the second atom and 90% after the third.

5 Mathematical Performance Model

Because the algorithm is highly dependent on application and network characteristics, it is quite difficult to validate the effectiveness of the proposed algorithm in a generic fashion. The algorithm performance depends on:

- Characteristics of the captured flow
- Number of expressions that are configured by one or multiple applications
- Structure of the different expressions (type and complexity of operators)
- Number of atoms in an expression
- Diversity of the atoms in an expression
- Popularity of atoms in both the expressions and captured flow

However, despite the previous uncertainty, this section mathematically proves the algorithm performs at least as good as the non-optimized algorithm. On top of this, we will backup the statement that our algorithm performs, with a high probability, much better than the latter one. We define:

a_i = Atomic expression ($i = 1 \rightarrow m$)

e_j = Expression ($j = 1 \rightarrow n$)
c = Cost of evaluation of an atom
$c' \ll c$ = Cost of retrieving the evaluated value
$x_{ij} = 1$ if a_i in e_j, else 0
$\forall_j \sum_{i=1}^{m} x_{ij} \geq 1$
$A = \bigcup_{i=1}^{m} a_i$
$a_i^j \in A'_j \subseteq A$ = Ordered subset of A containing the atoms in e_j
$Pr_j^{\{a_1^j,\cdots,a_k^j\}}$ = Probability that e_j is not yet solved if first $(k-1)$ a_i^j in A'_j are evaluated, probability that a_k^j needs to be evaluated in e_j
$\forall_j \ Pr_j^{\{a_1^j\}} = 1$ (i.e. at least one atom needs to be evaluated)

Practical experiments show that the evaluation time c of a single BPF atom is about 174 clock cycles. Even though this cost depends heavily on the type of BPF atom, the cost of retrieving an already evaluated atom c' is significantly lower, i.e. 42 clock cycles.

Cost 1 Without optimization we get the following cost estimation (i.e. all atoms of all expressions are evaluated):

$$E[cost_1] = \sum_{j=1}^{n} \sum_{i=1}^{m} (c \times x_{ij}) \tag{1}$$

Cost 2 With short-circuit optimization we can reduce the cost to:

$$E[cost_2] = \sum_{j=1}^{n} \sum_{i=1}^{m} (c \times x_{ij} \times Pr_j^{\{a_1^j,\cdots,a_i^j\}}) \tag{2}$$

$$cost_2 \geq n \times c$$
$$cost_2 \leq \sum_{j=1}^{n} \sum_{i=1}^{m} (c \times x_{ij})$$

Cost 3 When we eliminate all duplicate evaluations from the short-circuit optimization function, the cost can be reduced even further. Note that the order of evaluation of the atoms in an expression depends on their position in A'_j. This can for instance be from left to right.

$eval_{i1} = (x_{i1} \times Pr_1^{\{a_1^1,\cdots,a_i^1\}})$
$eval_{ij} = eval_{i(j-1)} + (1 - eval_{i(j-1)})(x_{ij} \times Pr_j^{\{a_1^j,\cdots,a_i^j\}})$
$eval_{ij}$ = Probability a_i is already evaluated in e_j

$$E[cost_3] = \sum_{j=1}^{n} \sum_{i=1}^{m} (x_{ij} \times Pr_j^{\{a_1^j,\cdots,a_i^j\}} \times \tag{3}$$

$$((1 - eval_{ij}) \times c + eval_{ij} \times c'))$$

$$cost_3 \geq c + (n-1) \times c'$$

$$cost_3 \leq (m \times (c - c')) + \sum_{j=1}^{n}\sum_{i=1}^{m}(c' \times x_{ij})$$

Cost 4 Taking the probabilistic optimization algorithm into account gives us the following cost function:

$a_i \in A' \subseteq A =$ Ordered subset of A containing all atoms
$Pr^{\{a_1,\cdots,a_k\}} =$ Probability that a_k needs to be evaluated (depends on all $Pr_j^{\{a_1,\cdots,a_k\}}$)
Order $a_{i(1\rightarrow m)}$ so $min \sum_{i=1}^{m}(Pr^{\{a_1,\cdots,a_i\}})$

$$E[cost_4] = \sum_{i=1}^{m}(Pr^{\{a_1,\cdots,a_i\}} \times (c - c') + \tag{4}$$

$$\sum_{j=1}^{n}(x_{ij} \times Pr_j^{\{a_1,\cdots,a_i\}} \times c'))$$

$$cost_4 \geq c + (n-1) \times c'$$

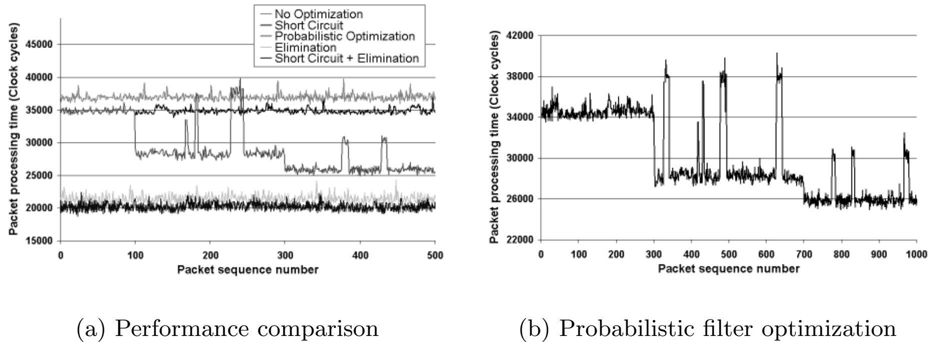
$$cost_4 \leq (m \times (c - c')) + \sum_{j=1}^{n}\sum_{i=1}^{m}(c' \times x_{ij})$$

Note that that both the $cost_3$ (3) and $cost_4$ (4) functions have the same the upper and lower boundaries. However, because we minimized the sum of the evaluation probabilities (this can be done because of the strict descending nature of the probability function), the overall curve of cost function (4) will be lower (or equal in the worst-case) than function (3).

6 Performance Measurements

To validate the proposed optimization algorithms, we have implemented them in the SCAMPI monitoring system [7]. We used a slimmed-down version of the DEFCON packet traces to generate traffic. To do the required measurements, we installed 20 applications, that each configure a BPF filter consisting of 5 atoms. These five atoms are uniformly selected out of a set of 15 random atoms that are present in the source trace.

Figure 3 (a) depicts the comparison of the performance of the various optimization techniques. Without any optimization the system needs about 37,100



(a) Performance comparison (b) Probabilistic filter optimization

Fig. 3. Packet processing time

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. Because of this perception, the probabilistic optimization algorithm will be of little use when used in combination with the elimination technique[3]. However, because elimination requires the monitoring system to keep state, it might be impossible for some systems to implement it. Therefor, Figure 3 (a) depicts the performance improvement of probabilistic optimizations based only on short-circuiting.

If we focus on the measurements of the probabilistic optimization, we can clearly identify the various implications of the different optimization steps. Figure 3 (b) illustrates this optimization in greater detail. In an initial situation all BPF filters are configured in the monitoring system. The different atoms are evaluated solely based on the number of times they occur in the expressions. This results in a packet processing time of about 34,700 cycles. Meanwhile, in order to deduce network characteristics, the monitoring system periodically samples packets for full analysis of all configured atoms[4]. When the (un)popularity of a certain atom raises, i.e. a high probability that an atom evaluates to true/false

[3] This is only true in our example. In the case where the cost of retrieving an already evaluated result is much higher or in applications with very long filter rules, elimination of duplicate evaluations combined with probabilistic optimizations will significantly improve the performance even further.

[4] For clarity, the processing time of these sampled packets is not shown in figure 3 (b)

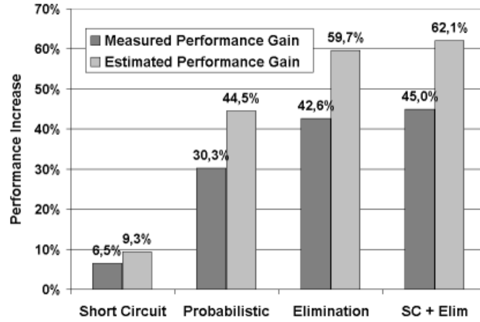


Fig. 4. Measured and estimated theoretical performance of average packet processing time

for a packet, the algorithm reorders the evaluation order of the atoms. In Figure 3 (b) we see that such a reordering takes place at packet 300. Here, the atom that has the highest probability to occur in a packet is evaluated first[5]. Because of this reordering, we see that the processing time drops to about 28,300 cycles. Between packet 300 and 700, we can spot some peaks in the processing time of the packets. This is due to the fact that now the most popular atom is evaluated first. All atoms that were evaluated prior to the most popular atom in the initial situation are delayed one evaluation. Apparently, there are some bursts of packets in the sourceflow that do not conform to the most popular atom. This situation is resolved at packet 700 when the peaks are detected and another reordering takes place, i.e. the second most popular atom is evaluated after the most popular atom.

Figure 4 depicts the comparison of the measured and theoretical performance gain of the average packet processing time. The theoretical packet processing time is obtained by applying the cost functions of section 5 on the rule sets and packet traces of the measurements in this section. The measured performance increases of all optimization techniques have the same relative proportions as the expected theoretical estimation. However, because the software framework imposes some additional overhead, the measured performance is about 30% lower than the theoretical expected performance.

7 Future Work

Measurements have shown that the probabilistic expression optimization algorithm can significantly improve the performance of a packet filter. In order for the optimization algorithm to be efficient and adaptive, it needs continuous measurement information from the network. In our example we obtained this information

[5] Depends on the type of operators in the expressions.

through sampling. Although this technique provides accurate network measurement information, it consumes processing resources. Future work will investigate if we can reuse already measured information to obtain these statistics or reduce the number of atoms that need to be evaluated in the sampled set of packets.

8 Conclusion

In this paper we presented a platform independent algorithm for packet filter and pattern matcher optimizations. Because no assumptions were made regarding the used hardware or platform, this algorithm can be applied to different filter and pattern matcher implementations. Next to the mathematical approximation of the performance improvements of the algorithms, we implemented the algorithm in the SCAMPI framework[7], using the MAPI. Experiments show that we can achieve a significant performance boost when multiple applications each configure a set of rules. We combined the proposed algorithm with the knowledge or measurement of the probability a certain expression will evaluate to *"true"* or *"false"*. This way, using a self-reconfigurable set of rules, the algorithm is able to optimize the aggregated expression even further.

Acknowledgment

Part of this work has been supported by the European Commission through the IST-SCAMPI project (IST-2001-32404). The work of the fourth author is also supported by the Fund Of Scientific Research - Flanders (F.W.O.-V., Belgium)

References

1. A. Begel, S. McCanne, and S. L. Graham, "BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture", Proc. ACM SIGCOMM '99, August 1999.
2. R. Boyer and J. Moore, "A fast string searching algorithm", Commun. ACM, 20(10):762772, October 1977.
3. R. Horspool, "Practical fast searching in strings", Software Practice and Experience, 10(6):501506, 1980.
4. V. Jacobson, C. Leres and S. McCanne, "tcpdump manual page", Lawrence Berkeley National Laboratory, University of California, Berkeley, CA, 2001.
5. E. Markatos, S. Antonatos, M. Polychronakis and K. Anagnostakis, "Exclusion-based Signature Matching for Intrusion Detection", Proceedings of IASTED International Conference on Communications and Computer Networks (CCN 2002), October 2002.
6. Ethereal, "Sniffing the glue that holds the Internet together", http://www.ethereal.com/.
7. IST-SCAMPI, "A Scaleable Monitoring Platform for the Internet", http://www.ist-scampi.org/.
8. Snort, "The Open Source Network Intrusion Detection System", http://www.snort.org/.

This article was processed using the LaTeX macro package with LLNCS style