Basic Research in Computer Science

BRICS RS-02-43 Christiansen & Fleury: Using IDDs for Packet Filtering

Using IDDs for Packet Filtering

Mikkel Christiansen
Emmanuel Fleury

BRICS Report Series **RS-02-43**

ISSN 0909-0878 **October 2002**

**Copyright © 2002, Mikkel Christiansen & Emmanuel Fleury.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

**See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/02/43/`

Using IDDs for Packet Filtering

Mikkel Christiansen Emmanuel Fleury

BRICS*
Department of Computer Science
Aalborg University

Email: {mixxel,fleury}@cs.auc.dk

October 29, 2002

Abstract

Firewalls are one of the key technologies used to control the traffic going in and out of a network. A central feature of the firewall is the *packet filter*. In this paper, we propose a complete framework for packet classification. Through two applications we demonstrate that both performance and security can be improved.

We show that a traditional ordered rule set can always be expressed as a first-order logic formula on integer variables. Moreover, we emphasize that, with such specification, the packet filtering problem is known to be constant time ($O(1)$). We propose to represent the first-order logic formula as *Interval Decision Diagrams* [ST98]. This structure has several advantages. First, the algorithm for removing redundancy and unnecessary tests is very simple. Secondly, it allows us to handle integer variables which makes it efficient on a generic CPUs. And, finally, we introduce an extension of IDDs called *Multi-Terminal Interval Decision Diagrams* in order to deal with any number of policies.

In matter of efficiency, we evaluate the performance our framework through a prototype toolkit composed by a *compiler* and a *packet filter*. The results of the experiments shows that this method is efficient in terms of CPU usage and has a low storage requirements.

Finally, we outline a tool, called *Network Access Verifier*. This tool demonstrates how the IDD representation can be used for verifying access properties of a network. In total, potentially improving the security of a network.

*Basic Research in Computer Science (`www.brics.dk`)

1 Introduction

The Internet firewall is one of the key technologies used by network administrators for controlling access to an organizations network. The main reason for the success is that the firewall allows centralized filtering of traffic entering and exiting the protected network. The central filtering mechanism of the firewall is the packet filter. It operates by identifying a policy by comparing the protocol header fields of a packet with a filter specification. In this paper we focus on the packet filtering mechanism, and in particular on how packet filters can be improved both in terms of security and performance.

The primary aspect of packet filtering is the issue of packet classification. Packet classification has been subject of much study in recent time, for example see [LS98, GM99, FM00]. The reason being that the ability to classify packets plays a central role in routing and in the Differentiated Services Architecture. However, the requirements to the packet classification scheme may be quite different from one application to the other. One example is routing on the Internet, where the classifier is used for choosing an interface based on a routing table. Here the classification only uses one or two of the address fields in the packet header determine route, where a firewall may classify packets based on any number of packet header fields TCP and/or IP. An related example is whether the classification algorithms should support dynamic updates of the specification or not. This is, for instance, the case with dynamic routing. Firewalls, on the other hand, uses more static specifications. An final difference may be the option to use dedicated hardware or not.

Given these differences, common performance measures of packet classification algorithms still remain. This includes classification time, space complexity, and performance of the optimization phase. Often worst case complexities are given in along with empirical measurements.

An other aspect of packet filtering is ability to analyze and check the filter specification before taking it into use. Current security audits rely on performing tests on the actual network by using port scanning or more advanced tools such as Nessus [nes]. Performing *off-line* security audits allow administrators to perform complete tests of their networks and minimize the requirement to perform test on the actual networks. However, a central issue for tool design is that the tool is based on a strong foundation, which in the case of packet filtering means a sound and complete representation of filter specifications.

In this paper we present a packet classification scheme that is well suited for packet filtering and can be summarized as follows:

- Sound representation of packet filters that is compatible with the traditional representation, e.g. ordered rule based filter description.
- Scalable in terms of the number of header fields, policies used in the

specification.

- Efficiently classification complexity ($O(1)$), assuming that the number of bounded fields.
- Compact and static representation of filter specification using decision diagrams.
- Access to techniques for verifying properties of filter specifications.

The key idea in the packet classification scheme it to transform a traditional rule based representation of a packet filter into a boolean expression represented as a decision diagram, similar to the approach presented in [Haz99]. However rather then using the widely known Boolean Decision Diagrams (BDDs) [Bry86] as in [Haz99] we use the less explored Interval Decision Diagrams (IDDs) [ST98]. IDDs operate on integer ranges rather then booleans thus providing the access to efficient classification of packets on generic CPUs.

However IDDs can only be used for classifying between two policies. To alleviate this problem we introduce the concept of Multi Terminal Interval Decision Diagrams (MTIDDs), that provide access to using any number of policies. This extension is similar to the MTBDD extension of BDDs described in [Bry86] which is suggested for packet classification in [AH02].

To demonstrate the potential of using IDDs for representing filter specifications, we outline a tool called Network Access Verifier (NAV). The key concept of the verifier is the ability perform a reachability of analysis of an entire network, for instance proving whether the network is vulnerable to IP spoofing.

In the following sections we first describe background and related work. Then in Section 3 we describe our model of packet filtering. Section 4 continues by introducing IDDs and show how we represent filter specifications using IDDs. In section 5 we describe the first of two applications which takes advantage of the packet classification scheme. This first application is a high performance packet classifier that provides empirical evidence showing that the performance of the scheme corresponds to expectations. In Section 6 we outline the second application which is NAV, through which we demonstrate the strength of using the IDD representation of packet filters. Finally in Section 7 we state conclusions and describe future work.

2 Related Work

In [Haz99] Hazelhurst presents the idea of transforming firewall packet filters into boolean expressions that are represented as BDDs. The paper describes an algorithm for transforming a Cisco firewall filter into a BDD, including the

handling of issues with overlapping rules. The main use of BDDs in this paper is for a tool that can be used analyzing and test filters. A later paper by Hazelhurst *et. al* [HAS00] focus on using using the BDD structures for performing packet classification. The conclusion is that BDDs can improve the lookup latency on systems using dedicated hardware such as FPGAs, while they do not perform well on generic CPUs. In [AH02] Attar and Hazelhurst use N-ary decision diagrams for improving the lookup performance. The experimental results show that the lookup time can be significantly improved by using this method, however at the price of increased memory usage. Furthermore the idea of using MTBDDs to handle the more general packet classification is suggested.

Several papers propose algorithms for packets classification on multiple fields for generic CPUs [BMG99, FM00, Sri01, BV01].

Begel *et. al* [BMG99] proposes a fully general packet filter framework. Filters are specified in a declarative predicate language, that are compiled into a flow graph, and then optimized before being executed on a virtual machine model. Optimization is performed on the flow-graph by using redundant predicate elimination for removing redundancies and rearranging non-optimal code sequences. An interesting point is the introduction of a safety verifier that checks the validity of the programs before they are executed on the virtual machine. This prevents the user from running programs with infinite loops or memory faults. The evaluation of the tool shows good performance. However only with small test cases are applied.

In [BV01] Baboescu and Varghese describe a scheme called Aggregate Bit Vector (ABV). The aim of the scheme is to provide scalable packet classification (100,000 rules) to handle large filters while also providing efficient classification times on generic CPUs. The scheme is an extension of the bit vector search algorithm (BV) described in [LS98]. The first optimization of the BV scheme consists of minimizing the number of unused bits in the bit vectors, by taking advantage of the observation that the number of rules overlapping in a filter is likely to be small. This is technique referred to as aggregation. Secondly, to take full advantage of using aggregation the order of the rules is rearranged. However, again due to the issues of overlapping rules, it is not possible. But by modifying the BV scheme to first find all matches and then computing the lowest cost match this is made possible.

In comparison with the approach presented in this paper, both the BV scheme and the ABV scheme solve a more general packet classification problem the we do. The reason being that in BV and ABV issues of overlapping rules are handled in the classification algorithm while we remove the overlap between rules when building the decision diagram structure.

An other active area for research is on tools for managing and analyzing filters. An example is the tool presented in [HSP00] which can be used for detecting an resolving packet conflicts in packet filters. Here a scheme is in-

troduced to resolve packet conflicts by adding resolve filters. An other tool, presented in a paper by Eronen and Zitting [EZ01], presents a tool that uses constraint logic programming for analyzing packet filters. Similar to the work presented in [Haz99] this tool transforms packet filters to boolean expressions before performing the analysis.

3 Packet Filtering

The problem of packet filtering is to match a packet header with a policy. This decision is based only on the header of the current examined packet and a set of rules, also called '*filter*'.

The filters are defined as an ordered list of independent rules. Each rule specify both a set of headers and what policy to apply to the packet. For example, in Cisco-like syntax, one can define the rule set represented on Figure 1.

```
access-list 108 permit tcp any any eq www
access-list 108 deny tcp any any
access-list 108 deny ip any any
```

Figure 1: *Example of a filter in a Cisco-like syntax.*

The first rule applies the policy "`permit`" to any TCP packet when the destination port is equal to "`www`". if the incoming packet is not matching the first rule, it is compared to the second one, which states that the filter apply the policy "`deny`" to any TCP packet. If, again, the incoming packet is not matched with this rule, it is compared to the last one which apply the policy "`deny`" to all IP packets.

A naive approach would be to use this filter specification strait forward. But, this way of specifying a filter is strongly dependent of the order of the rules in the list. Keeping this order prevent a lot of possible optimizations both in space storage for the rules set and in speed to perform the classification of each packet.

The worst case complexity of such naive algorithm is $O(n \cdot m)$, with n the number of rules, m the number of fields to check in the header. If we assume the number of fields as constant (as we are dealing only with known protocols with a known number of fields), we have a linear complexity in the number of rules ($O(n)$). This complexity analysis show that the number of rules has great impact on the performance of the packet filter.

In this section we propose to consider a filter as a first-order logic formula on integers. We show that not only we have the same expressive power than the ordered rule-set representation, but also that this way of specifying a filter

allow us to deal with a constant time complexity $O(1)$ concerning the packet classification problem.

3.1 Specifying Filters as First-Order Logic Formula

Specifying filters as first-order logic formula on integer variables is immediate. In order to do it right we introduce a formal framework of the problem in order to be able to prove formally the properties we are interested in.

Let H be the finite set of all the possible headers, and $\Pi = (\pi_1, \pi_2, \ldots, \pi_p)$ the set of all the policies. A *rule* is given by a set of headers ($\eta \in \mathcal{P}(H)$[1]) and a policy ($\pi \in \Pi$):

$$r = (\eta, \pi), \text{ with } \eta \in \mathcal{P}(H) \text{ and } \pi \in \Pi. \tag{1}$$

For example, a rule which drops the packets that have the field 'source IP' set to 192.134.*.* and use the protocol TCP would be written:

$$r = ((sip = 192.134.*.*) \wedge (proto = TCP), \texttt{DROP}) \tag{2}$$

We define a *filter* as a set of rules over $\mathcal{P}(H) \times \Pi$:

$$\varphi = ((\eta_1, \pi_{k_1}), (\eta_2, \pi_{k_2}), \ldots, (\eta_n, \pi_{k_n})), \text{ with } \pi_{k_i} \in \Pi, \ \forall i \leq n. \tag{3}$$

By extension, we define a filter $\varphi = (\eta_i, \pi_{k_i})_{i \leq n}$ as a function that maps one header to a set of policies. Formally, the function $\varphi : H \to \mathcal{P}(\Pi)$ is defined such that:

$$\varphi(h) = \{\pi_{k_i} \in \Pi / h \in \eta_i\} \tag{4}$$

We say that two filters φ and φ' are *equivalent* iff for all $h \in H$ we have $\varphi(h) = \varphi'(h)$. And we note $\varphi \equiv \varphi'$

We define a *normal form filter* as a filter with no duplicate policy in the rule set. And, finally, we call a *valid filter*, a filter in which the set of headers $(\eta_i)_{i \leq n}$ are a partition of H. Formally a *partition* is defined as:

Definition 1 *Let H be a set and $(\eta_i)_{i \leq n}$ such that, for all $i \leq n$, $\eta_i \in \mathcal{P}(H)$. Then, $(\eta_i)_{i \leq n}$ is a* partition *of H iff:*

1. $\bigcup_{i \leq n} \eta_i = H$,
2. $\eta_i \cap \eta_j = \emptyset$, $\forall i, j \leq n$ *with* $i \neq j$.

[1] Where $\mathcal{P}(A)$ is the powerset of A.

3.2 Ordered Filters vs First-Order Logic Filters

A filter has to be valid in order to avoid any ambiguity while the classification of a given header. The ambiguity was previously avoided by ordering rules in the list. This order was intended to prioritize a rule over the others, as we had illustrated it in our first example.

In order to prove the equivalence between an ordered filter and a first-order logic only filter, we have first to define formally what is an *ordered filter*.

Lets call ψ an ordered filter iff $\psi = (\eta_i, \pi_{k_i})_{i \leq n}$ with $\eta_i \in \mathcal{P}(H)$, $\pi_{k_i} \in \Pi$ for all $i \leq n$ and we define an implicit order $\succ$ on the rules such that:

$$(\eta_i, \pi_i) \succ (\eta_j, \pi_j) \Leftrightarrow i > j \tag{5}$$

By extension, we call an *ordered filter* $\psi = (\eta_i, \pi_{k_i})_{i \leq n}$ a function that maps one header to one policy. Formally, the function $\psi : H \rightarrow \Pi$ is defined such that:

$$\psi(h) = \{\pi_{k_i} \in \Pi / h \in \eta_i \text{ and } h \notin \eta_j,\ \forall j < i\} \tag{6}$$

We will now state that for any ordered filter ψ we can build an equivalent valid filter φ'.

Proposition 1 *For any ordered filter $\psi = (\eta_i, \pi_{k_i})_{i \leq n}$, we can build a filter $\varphi = (\eta'_i, \pi'_{k_i})_{i \leq n}$ such that ψ and φ are equivalent.*

Proof 1 *The proof is strait forward from the definitions and the following construction of φ:*

- $\pi'_{k_i} = \pi_{k_i}$, $\forall i \leq n$,
- $\eta'_i = \eta_i \setminus \bigcup_{j<i} \eta_j$, $\forall i \leq n$.

So, φ' is given by:

$$\begin{aligned} \varphi = ((&\eta_1, \pi_{i_1}), \\ &(\eta_2 \setminus \{\eta_1\}, \pi_{i_2}), \\ &(\eta_3 \setminus \{\eta_1 \cup \eta_2\}, \pi_{i_3}), \\ &\ldots, \\ &(\eta_k \setminus \{\eta_1 \cup \cdots \cup \eta_{k-1}\}, \pi_{i_k})) \end{aligned}$$

By construction of φ, this filter is valid *and equivalent to ψ.*

Therefore, from the proposition 1 we can deduce that our formalism is, at least, as expressive than the current method.

3.3 Complexity of Packet Classification

Actually, removing the need of the order in the definition of a filter has some important consequences on the complexity of the packet classification problem. Indeed, if we consider a normal valid filter, classifying a packet is equivalent to evaluate a first-order logic formula on integer variables. This operation is known to be linear in the number of variables, or in other words in the number of fields (m) and logarithmic in the domain of the greatest field[2] ($\log(w)$, with w the wider ranger of the fields). Therefore, the complexity of such operation would be $O(m \cdot \log(w))$. Finally, if we consider that the number of fields in the header and the domain of each field are bounded, then we have a constant time complexity ($O(1)$).

Proposition 2 *Given a* normal valid filter*, and a bounded number of bounded fields, the problem of packet classification is* $O(1)$.

In conclusion, we proved that specifying a rule-set as an ordered-list or a first-order formula is equivalent, we even exhibit an algorithm to derive a first-order logic specification from any ordered list. We also shown that the complexity of classifying a packet with a normal and valid first-order logic specification is constant time ($O(1)$). In the next section we will describe an efficient data-structure for handling first-order logic formula.

4 Decision Diagrams

As we pointed out in the previous section, the packet filtering problem is equivalent to evaluate a first-order logic formula. Indeed, one of the most efficient data-structure, both in space storage and computational time, are the *decision diagrams*. The most famous of those are *binary decision diagrams* (BDD, [And97]). Using such data-structure to represent filters have been already investigated by S. Hazelhurst in [AH02, Haz99]. But, one main problem in such approach is that BDD are based on boolean variables only. Therefore, it is mandatory to consider one bit after one. As a generic CPU is used to consider one *word* of several bits in one operation, there is an overhead on extracting bits from words. In order to avoid this drawback, we chose to focus on another decision diagram structure called *interval decision diagram* (IDD, [ST98]). This structure allows us to perform classification on integer numbers within a domain (finite of infinite).

[2]Worst case of number of tests to perform in order to find the position of an integer variable on a partition

4.1 Interval Decision Diagrams

An IDD is a DAG structure in which each node correspond to a test on an integer variable. Each out going edge from a node is associated to an interval within the domain of the variable attached to the node. Finally, the edge is linked either to another node either to a boolean *terminal* ($True$ or $False$). More formally, the definition of an *IDD node* is given by:

Definition 2 *Let x be an integer variable defined on the domain $\mathbb{D}_x \subseteq \mathbb{N}$ and t a first-order logic formula on integer variables. We call t an* IDD node *iff one of the following hold:*

- $t \in \{True, False\}$,
- $t = (x \in I_0 \wedge t_0) \vee (x \in I_1 \wedge t_1) \vee \ldots (x \in I_k \wedge t_k)$.

With $(I_i)_{i \leq k}$ a partition of $\mathbb{D}_x$ and $(t_i)_{i \leq k}$ a set of IDD nodes. We note: $t = x \rightarrow (I_0, t_0)(I_1, t_1) \ldots (I_n, t_k)$.

We call an *IDD root*, an IDD node without predecessor. We say that a set of IDD nodes $(t_i)_{i \leq n}$ is *consistent* if there is only one root. Moreover, if t is an IDD node, let $var(t)$ be the function which give the integer variable tested on this node. More formally:

$$var(t) = \begin{cases} x, & \text{if } t = x \rightarrow (I_0, t_0)(I_1, t_1) \ldots (I_k, t_k) \\ t, & \text{if } t \in \{True, False\} \end{cases}$$

Finally, we call $I = ((t_i)_{i \leq n}, \succ)$ an IDD iff $(t_i)_{i \leq n}$ is a consistent set of IDD nodes and $\succ$ is an order on the integer variables such that for all $t \in (t_i)_{i \leq n}$ with $t = x \rightarrow (I_0, t'_0)(I_1, t'_1) \ldots (I_k, t'_k)$, we have $x \succ var(t'_i)$ for each $i \leq k$.
For example, if we consider the logic formula:

$$(x = 0 \wedge y \leq 3) \vee (1 \leq x \leq 6 \wedge z \leq 6) \vee (x = 7 \wedge y = 1)$$

The corresponding IDD would be (see Figure 2):

$$\begin{aligned} t_0 &= x \rightarrow (\{0\}, t_{00})\ ([1,6], t_{000})\ (\{7\}, t_{01}) \\ t_{00} &= y \rightarrow ([0,3], T)\ ([4,7], F) \\ t_{01} &= y \rightarrow (\{0\}, F)\ (\{1\}, T)\ ([2,7], F) \\ t_{000} &= z \rightarrow ([0,6], T)\ (\{7\}, F) \end{aligned}$$

IDD structures can easily be used for describing a filter. On Figure 3, we represent a very simple filter as an IDD. This example is testing the 'source IP' variable that we splitted into four sub-variables (sip_i) which are easier to test. It can be noticed that all non-relevant tests have been removed from the IDD structure.

On the Figure 3 the terminal $DROP$ is assumed to be $\neg ACCEPT$, as we handle only boolean terminals. We did not represent it, because it is assumed that an edge which is not represented just leads by default to $DROP$.

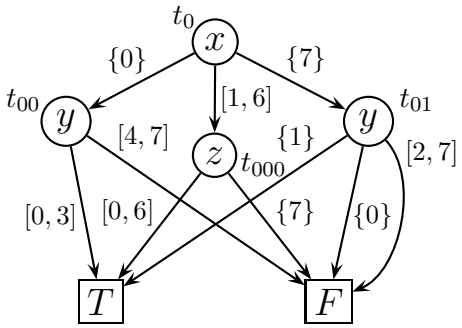

Figure 2: *Example of an Interval Decision Diagram (IDD).*

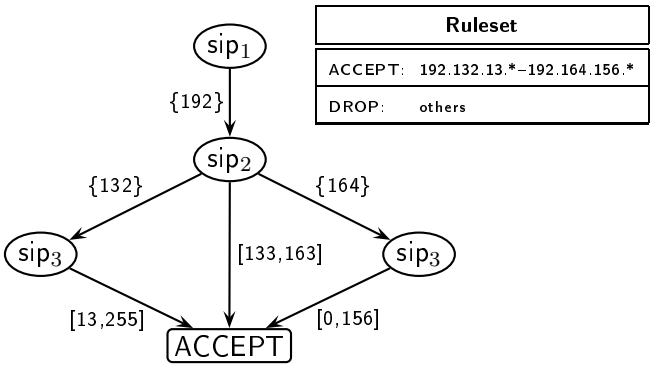

Figure 3: *IDD representing a filtering rule.*

4.2 Boolean Operations on Interval Decision Diagrams

As IDD are representing first-order logic formulas on integer variables, we can perform all the usual logical operations as negation ($\neg$), and ($\wedge$), or ($\vee$), and so on. Some examples are given on Figures 4 and 5. Figure 4 represent two formulas φ_1 and φ_2. Figure 5 represent the result of $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$. The edges labeled by $*$ are denoting the complement of all the other edges. For example, if a node has four edges labeled by $[0, 2]$, $\{9\}$,$[12, 15]$ and $*$ and has a range of $[0, 15]$, then $*$ stand for $[3, 8]$ and $[10, 11]$.

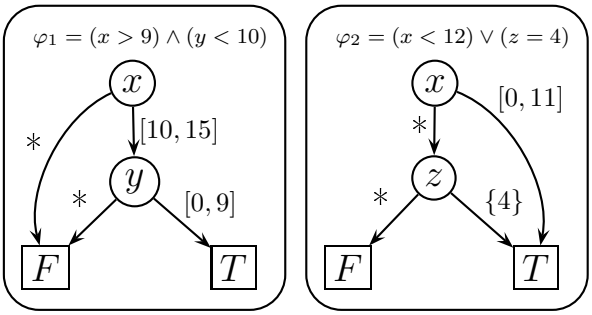

Figure 4: *Examples of Interval Decision Diagrams.*

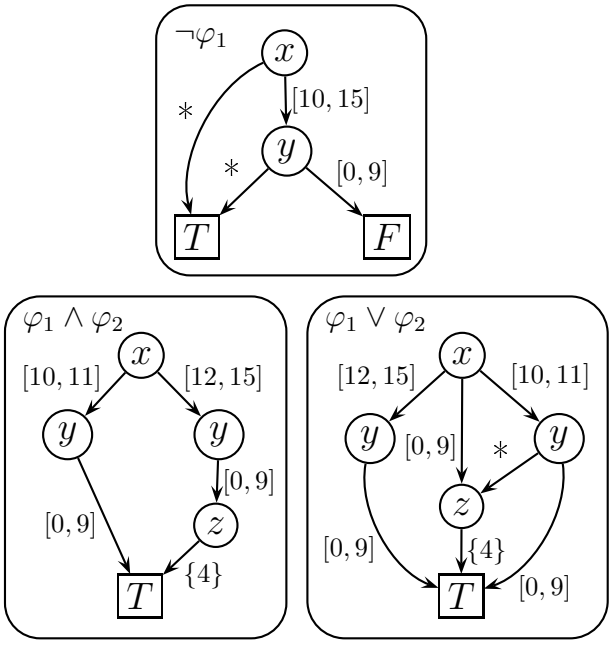

Figure 5: *Examples of boolean operations on IDDs.*

4.3 Optimization of Interval Decision Diagrams

As you can see on Figure 5, the result of $\wedge$ and $\vee$ operations is not a direct combination of φ_1 and φ_2. Indeed, some optimizations have been performed on the structure in order to prune redundant nodes and sub-trees.

Optimization process is very simple. It is performed by listing all the node of the IDD and applying the following optimization rules:

1. If a non-terminal node only has one outgoing edge, it must be pruned.
2. If two nodes have the same outgoing edges and represent the same variable, they must be merged into one.
3. If two edges of a node, with consecutive intervals, refer to the same child, they must be merged.

When all the nodes have been processed, the input IDD to the optimization function is compared to the resulting IDD. If they are equal a fix-point have been reached and the optimization terminates. If not, it takes the resulting IDD as the input and it performs the optimization function again.

This optimization algorithm is proved to always terminate (as all the rules are pruning nodes and none is adding one). It also guaranty, both, that the number of nodes will be minimal and that the depth of the IDD, for this given order[3], will be minimal [ST98].

4.4 Multi-Terminal Decision Diagrams

Unfortunately, in real life examples, you often have more than two policies. One good reason could be because the firewall allow the user to create his own policies. As IDDs are representing boolean formulas, they cannot provide more than two terminals and therefore they can't give an efficient way of dealing with more than two policies. The idea is now to extend the IDD structure with multiple terminals (MTIDD). This is directly derived from the multiple terminal binary decision diagrams (MTBDD, [And97]).

Figure 6 represent a filter which have more than two policies (*ALLOW*, *RESET*, *DROP*). As previously, one terminal is not represented. The *DROP* policy has been chosen as the default. The precise semantic is that all the edges which are not represented on the figure leads to the default policy.

More formally, the definition is very similar to the interval decision diagram's definition, except that we allow more than two terminals. In place of boolean as terminal we define a finite set $\mathbb{T}$ of terminals (T_1, T_2, ...). Lets first define a *MTIDD node*:

[3] Choosing a different order can sometimes leads to some gain

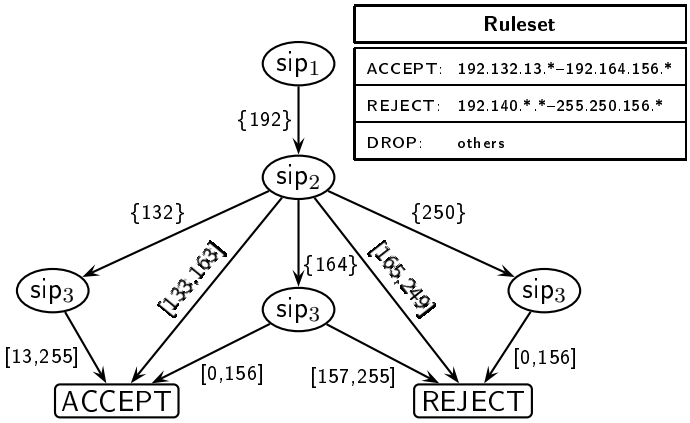

Figure 6: *MTIDD representing a filtering rule.*

Definition 3 *Let x be an integer variable defined on the domain $\mathbb{D}_x \subseteq \mathbb{N}$ and t a first-order logic formula on integer variables. We call t an* MTIDD node *iff one of the following hold:*

- $t \in \mathbb{T}$,
- $t = x \rightarrow (I_0, t_0)(I_1, t_1) \dots (I_k, t_k)$.

With $(I_i)_{i \leq k}$ a partition of $\mathbb{D}_x$ and $(t_i)_{i \leq k}$ a set of MTIDD nodes.

The notion of *root node* and *consistency* are the same, but we have to extend slightly the function *var*:

$$var(t) = \begin{cases} x, & \text{if } t = x \rightarrow (I_0, t_0)(I_1, t_1) \dots (I_k, t_k) \\ t, & \text{if } t \in \mathbb{T} \end{cases}$$

Finally, we call $I = ((t_i)_{i \leq n}, \succ)$ a MTIDD iff $(t_i)_{i \leq n}$ is a consistent set of MTIDD nodes and $\succ$ is an order on the integer variables such that for all $t \in (t_i)_{i \leq n}$ such that $t = x \rightarrow (I_0, t_0)(I_1, t_1) \dots (I_k, t_k)$, we have $x \succ var(t_i)$ for each $i \leq k$. For example (see Figure 7):

$$\begin{aligned} t_0 &= x \rightarrow ([0,4], t_{00})\ ([5,7], t_{000}) \\ t_{00} &= y \rightarrow ([0,3], T_1)\ ([4,15], T_2) \\ t_{000} &= z \rightarrow ([0,1], T_2)\ ([2,+\infty[, T_3) \end{aligned}$$

Performing packet classification on MTIDD in place of IDD does not imply any complexity overhead and can be see as a strait extension of a regular IDD. But, MTIDD are no more boolean formulas. In a matter of fact, we are computing MTIDD by combining non-overlapping IDDs (one by policy).

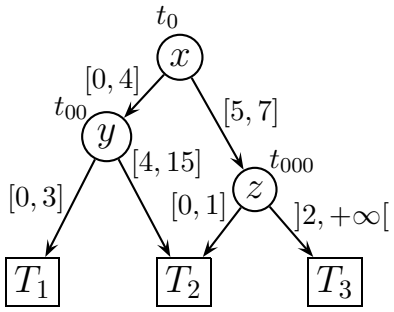

Figure 7: *Multiple-Terminal Interval Decision Diagram (MTIDD).*

In conclusion, we have presented an efficient data-structure to handle with first-order logic on integer variables (IDD), we described an algorithm to optimize in size and depth such data-structures. And, we proposed an extension of IDD in order to deal easily with several terminals (MTIDD). In the two next sections we will present the general architecture of a tool using such framework to classify packets and the basic algorithm of a network access verifier tool.

5 High Performance Packet Filtering

In the previous sections we described the IDD and MTIDD data-structures that we propose to use when performing packet filtering. This section focuses on evaluating the performance of the data-structure by describing a prototype tool that performs packet filtering using MTIDDs. In the following sections we first describe the architecture of the packet filtering toolkit and then evaluate the performance of the tool based an number of simple experiments.

5.1 Architecture

The architecture of the packet filtering toolkit is shown in Figure 8. The main components are the compiler, the packet classifier, and the NAV tool that we describe in Section 6. In the following we focus on describing the flow of data through the architecture and then the issues related to the design of the compiler and the packet classifier.

Figure 8 shows the overall architecture of the packet classification tool. The flow of data begins with a filter specification in a high level language. In our particular case we have simply chosen to use a Cisco-like access list language that supports overlapping rules and logging. Using a compiler the high-level specification is transformed into an MTIDD that has been optimized thus ensuring near optimum performance. After the compilation there are two directions for the data. Either the MTIDD can be used in a tool such as NAV, or it can be loaded into the packet classifier.

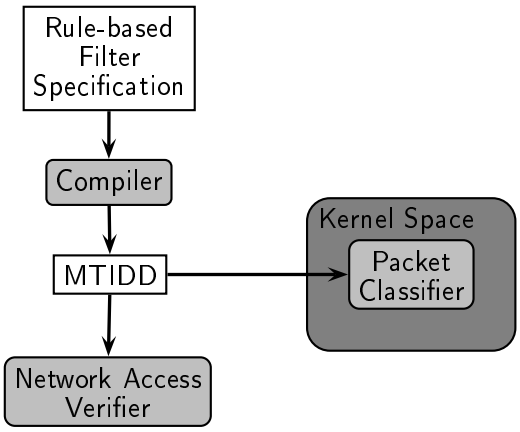

Figure 8: *Packet Filter Architecture.*

The compiler performs the transformation from the high-level filter specification into a MTIDD structure as we described it in Section 3.2. The overall approach consists of building an IDD for each of the policies used in the filter specification. These IDDs are then merged into an MTIDD representing the entire filter in a single decision diagram. An example is the result of compiling the filter specification in Figure 1. This results in an MTIDD built from two disjoint IDDs representing the policies: $PERMIT$ and $DENY$. At a more detailed level, the compiler operates by building an IDD for each of the rules in the order they are stated in the specification. Then, before adding an rule to the IDD with the corresponding policy, any overlap with previous rules is removed. This is done by removing any overlap between the current rule and the IDDs representing various policies used in the filter. This corresponds to the equivalence proof given in Section 3.2. For instance, from the example of Figure 1, when adding the second rule to the IDD representing $DENY$, we remove the part of the rule which overlaps with the IDD of the $PERMIT$ policy.

Having described the main idea of the compiler we move on and look closer at the design of the packet classifier. As shown on Figure 8 an actual implementation packet classifier will run in kernel space and serve as the core classification mechanism in a packet filter, however, in the prototype we chose only to do a user space implementation. The majority of code consists of initializing the MTIDD data-structure describing the filtering policy. To represent the MTIDD we used a structure fairly similar to the a adjacency-list representation of directed acyclic graphs described in [Sed02], with the exception that adjacency-lists are arrays, thus allowing fast search of the partitions. Figure 10 illustrates the organization of this structure based on the example

IDD shown in Figure 9. To limit the processing overhead, each node is initialized to have a pointer to a comparison function which is used to perform a binary search for the matching partition entry. This allows us to use different functions based on the size of the header field without any processing overhead. The worst case number of comparisons necessary to classify a packet is: $m \cdot \log(w))$ where m is the number of fields and w is the maximum number of intervals in the largest field. The actual search function simply consists of traversing the DAG, performing a binary search at each non-terminal until a leaf is reached.

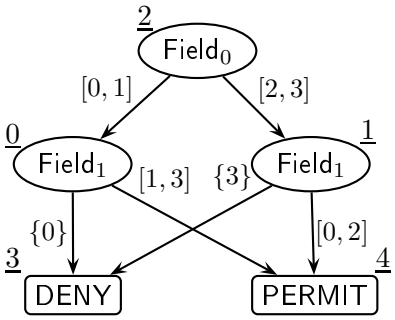

Figure 9: *Filter example.*

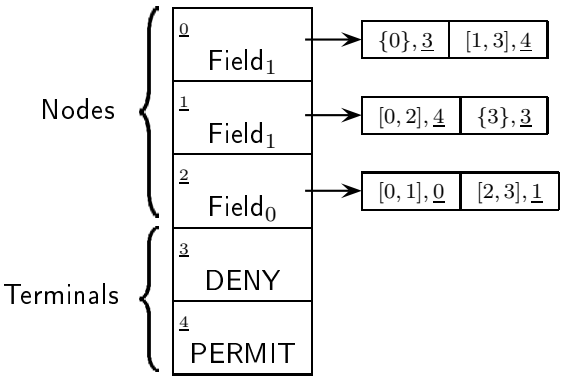

Figure 10: *Organization of an IDD structure in the packet classifier.*

The main strength of this architecture is that the whole complexity of packet filtering lies is in the compiler that runs in user-space while packet classifier, that run in kernel space, is very simple. A consequence of this design is that the filtering policy is more static since any change requires recompilation of the filter specification. On the other hand compiling that filter specification before actually loading it to the firewall allows an administrator to have the fil-

ter checked before it is taken into production. This process could be supported by a tool such as NAV (see Section 6) or similar.

5.2 Performance

The relevance of proposing the use of the MTIDD structure for packet classification is highly dependent on whether performance is competitive with other algorithms for packet classification that performs well on generic CPUs. However, we should emphasize that the possibilities of optimizing time and space requirements when using MTIDD structures are many and not fully explored in the work presented here. In the following we first focus on the space requirements of the packet classifier, then we look briefly at the performance of the compilation from filter specification to MTIDD.

The memory requirements of using MTIDDs are difficult to reason about due to the nature of decision diagrams. The worst case memory requirement of an MTIDD is exponential in the depth of the MTIDD. However the advantage of decision diagrams, in general, is that they remove any redundancy of boolean expressions hereby minimizing the memory requirements. Secondly, the strength of IDDs, in particular, is that boolean expressions over intervals or ranges can be described in a very compact manner.

Indeed, ranges and intervals often occur in filter specifications. For instance if we briefly look a filter on TCP/IP protocol fields then we can easily identify often occurring intervals. For instance it is common to only allow inbound traffic on a few port numbers, so the range from 1024-65535 could for instance be an often occurring interval to specify the range of closed ports. An other example is the IP-address fields where we often group networks by subnet mask, which in itself describes a grouping of addresses. A final example is the protocol field in the IP header, where only a few different values are used for specifying protocols such as TCP, UDP, ICMP, and IGMP. Thus we can conclude that it is unlikely to see exponential memory requirements for representing filters.

To provide empirical evidence of the memory requirements needed for representing packet filters as MTIDDs we performed two experiments. The first experiment consists of analyzing the memory requirements of a set of real-life filters specifications from production networks. The second experiment aims at studying the scalability of the memory requirements by exploring the memory requirements of a filter that specifies the traffic of a backbone header trace.

For the first experiment we studied a set of six real-life filters. The filters are all used on production networks and manually written by professional network administrators (e.g. no automatic rule generation is used). The filters A_x are access filters from the University routers while filters B_x are filters from a commercial organization.

Table 1 shows the summary of the memory requirements for each of the

Filter	#Rules	#Nodes	#Edges	Size	Time
B_1	132	142	771	16.5 KB	4.8 s
A_1	129	164	1255	24.8 KB	7.7 s
B_3	90	53	274	6.00 KB	0.31 s
A_2	71	97	605	12.5 KB	2.8 s
A_3	39	18	109	2.33 KB	0.16 s
B_2	18	62	259	6.05 KB	0.19 s

Table 1: MTIDD resource requirements of real-life filters.

filters. The first column two describes the number of rules in the original filter specification. Columns three and four summarizes size of resulting MTIDD, and column five shows the memory usage of the MTIDD structure when has been loaded into the packet classification prototype. It should be mentioned that in this study we chose to split the representation of IP addresses into four variables, each representing a byte of the address separately. This may mean fewer edges but more nodes.

To some degree we see a correspondence between the number of rules in the order rule set specification and the memory requirements of the MTIDD representation. However in the case of B_3 a filter of 90 rules is represented with less memory than filter B_2 which only has 18 rules. The reason is that filter B_3 has many very similar rules. Another interesting remark is that filter A_1 uses significantly more memory than filter B_1, even though the number of rules are nearly identical. Indeed, the author of filter A_1 uses overlapping rules which causes a higher degree of fragmentation of intervals in the resulting MTIDD. In total, these results are promising due to the small memory requirements.

The second experiment explores the scalability of the MTIDD representation of filter specification. Due to the lack of real-life filters for this experiment, we chose a different approach, where the idea is to extract a filter describing the traffic of a network backbone. An alternative approach is to generate random rules. However, since the MTIDD data-structure relies on finding intervals in the address range, then rules with random values will not give fair picture of the scalability issue.

In practice the rules are generated using a packet header trace of backbone traffic[4]. Each header in the trace is described by a rule. The rule permits packets with similar headers fields to pass through the filter. The set of header fields considered are source and destination address, IP protocol field, and source and destination port numbers if applicable. Any duplicate rules are removed from the final filter.

[4]IP addresses were mangled to ensure privacy, but it such a way that the offset between addresses in the trace remained present

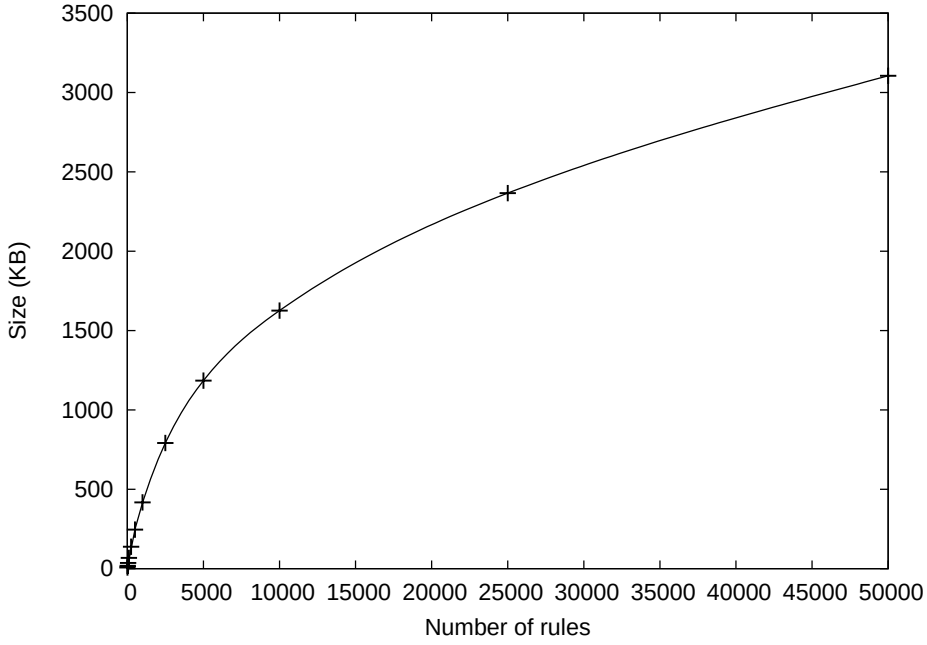

Figure 11: *Memory requirements with large packet filters.*

Figure 11 shows the result of the scalability experiments by showing the size of the MTIDD when loaded by the packet classification prototype as a function of the number of rules in the filters. Overall we see a logarithmic growth rate. Initially growth rate is rather large, but as the filters increase in size the more efficient the MTIDD data-structure becomes at representing the filters. An important point to these experiments is that the generated filters only represent MTIDDs with the policies *permit* and *deny*. If more policies are added, then the size of the MTIDDs will increase. The worst-case situation, when adding more policies, is that each new policy introduced is represented entirely by it's own subtree, thus causing linear increase of memory usage as a function of the number of policies in the MTIDD.

Before concluding, we briefly discuss the compilation times for transforming an rule-based filter to an MTIDD. The compile times, measured on a 1.1GHz AMD Athlon™, are shown in rightmost column of Table 1. From these we see acceptable compilation times for our real-life filters, however, compiling larger filters such as those shown in Figure 11 takes unacceptably long. For instance, the largest filter (50.000 rules) took several days to compile. Thus we conclude that to make our scheme truly scalable, techniques for improving the compile time needs to be developed.

In this limited evaluation of the packet classification prototype, the empirical evidence shows that:

- The memory requirements of MTIDDs for representing real-life packet filters has proven to be quite small. Our largest example used only 24.8KB when loaded.
- When using MTIDDs for describing the headers of backbone traffic, we found that the space requirements were logarithmic in the number of rules of the filter.
- Compilation time is acceptable with real-life filters, but different optimization possibilities need to be explored for making the compilation time acceptable with larger filters.

In total, our evaluation suggests that the use of MTIDDs for representing packet filters is effective and competitive. However, it should be noted that the evaluation is not complete in any way. For instance, we have not explored the potential gain of variable reordering our IDD structures, such as described in [And97]. Neither have we made attempts to measure the speed of the classification algorithm when filtering traffic.

6 Network Access Verifier

A problem when working with filters specifications such as those used on firewalls is to ensure that the policy implemented in the specification corresponds to the intended policy. An even more difficult problem is to understand the cumulative effect of two filters separated onto different routers or firewalls. In this section we outline a tool called a Network Access Verifier (NAV) that can be used to explore the access properties of a complex network by taking advantage of the filter specifications as being first-order logic expressions. As previously, the first-order logic formulas are represented as IDDs.

To illustrate the idea, consider the model of a network as a bidirectional graph as shown in Figure 12. The network consists of a set of computers, denoted from A to E, that have one or more interfaces that connects the computer to one or more of its neighbors. Network access is controlled through in inbound and an outbound filter for each interface. The filters are a boolean expressions that either permit ($True$) or deny ($False$) packets to pass through based on values in the header fields. To include aspects of routing in the model we transform the routing table into a set of boolean expressions, one for each filter. Each boolean expression is given by the routing table entries for that particular interface, thus the boolean expression describes the set of headers that are forwarded on that interface.

Using this model we derive a matrix describing the filter between any of pair of computers in the network. As an illustration of this principle Table 2 shows the filters in the example network. Each of the filters in the matrix is

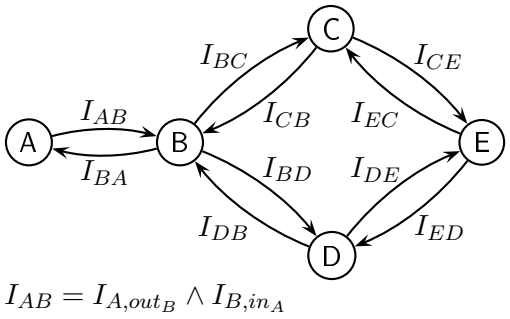

Figure 12: *Example of a Network Graph Representation.*

found by combining the filters on each link between the source and destination. A sequence of filters are combined by conjunction, and if alternative paths exist then the filters for each path are combined with a disjunction. For instance the filter for any packets passing from computer A to computer B is given by $I_{AB} = I_{A,out_B} \wedge I_{B,in_A}$ thus combining the outbound filter from A to destination computer B and Bs inbound filter for traffic from A. An other example it the filter between the two hosts A and E. Here traffic can pass through either C or D, so using disjunction we get $I_{AE} = I_{AB} \wedge (I_{BC} \wedge I_{CE} \vee I_{BD} \wedge I_{ED})$.

		Destination				
		A	B	C	D	E
Source	A	$\times$	I_{AB}	I_{AC}	I_{AD}	I_{AE}
	B	I_{BA}	$\times$	I_{BC}	I_{BD}	I_{BE}
	C	I_{CA}	I_{CB}	$\times$	I_{CD}	I_{CE}
	D	I_{DA}	I_{DB}	I_{DC}	$\times$	I_{DE}
	E	I_{EA}	I_{EB}	I_{EC}	I_{ED}	$\times$

Table 2: Matrix of filters in example network.

The matrix immediately give the reachability analysis of the network. In fact, if an element I_{ST} of this matrix is the IDD node $False$, the source node S cannot send any packet to the target node T without being filtered out. This reachability test is really wide. Indeed it cover also IP-spoofed packets.

A more reasonable query would be to ask if the machine S can reach the machine T with packets such that the source-IP is set to the IP of S. This operation is, actually, very easy to perform. It is enough to compute the conjunction of I_{ST} and the IDD describing the set of headers such that the source-IP field is equal to the IP address of S.

More generally, the user can specify a set of headers (I_{query}) and check if

some of them (I_{result}) can be sent from S to T (I_{ST}):

$$I_{query} \wedge I_{ST} = I_{result} \tag{7}$$

If I_{result} is equal to $False$, none of the headers described in I_{query} can reach T from S.

In this section we have briefly described a tool for network access verification which test for packet reachability into a, possibly, complex network. The overall strength the tool lies in the fact that the tests are *exhaustive*. Meaning that all cases are covered by the computation, thus improving the overall security of the network being analyzed. Moreover, the computational power needed to perform such verification is really low and can be performed on any personal computer.

7 Conclusion

In this paper we have focused on packet filtering on Internet firewalls, and especially on improving both aspects of performance and security. As a result we have proposed a formalized framework for packet classification and through two applications we demonstrate that both performance and security can be improved.

The central idea of this paper consists of transforming the traditional ordered rule based filter specifications into first order logic formulas on integer variables, and representing these using a Multi Terminal Interval Decision Diagrams (MTIDDs). Performing this transformation results in several advantages. First of all, the representation is sound and complete essentially providing a strong platform for building tools for testing and verifying properties of filter specifications. Secondly, the worst case classification time when using MTIDDs is $O(1)$ making the classification time independent of the size of the filter specification. Thirdly, the concept of Interval Decision Diagrams is easy to understand and provides a natural representation of filter specifications. Finally the algorithms for optimizing and manipulating IDDs are simple.

For purposes of demonstrating the strength of the framework, we have described two applications: a packet filtering prototype and a network access verifier (NAV).

The purpose of the packet filtering prototype is to demonstrate the performance issues related to using a decision diagram representation of packet filters and suggesting an architecture for a packet filtering toolkit. The main benefits of the suggested architecture is that, when using this framework, the majority of the complexity runs in user space, while the packet classifier, running in kernel space, is very simple. In terms of performance we have presented a preliminary study space-usage issues. Most interesting is the empirical evidence showing that the memory requirements for representing filters as MTIDDs are

very promising. In the set of real-life filter specifications tested the largest used only 25KB. In a test of large packet filters we saw logarithmic space usage as a function of the number of rules, where the largest filter required 3.1MB of memory for 50.000 rules. A second study focused on the packet filter compilation time. Here we found acceptable compilation times for real-life filters, however with larger filters compilation times are quite long. Several issues remains open for further study, this includes measurements of actual classification times, and exploring ways to minimize the size of the MTIDD structures.

The second application, which is only outlined, demonstrates a potential use of our framework for improving network security. The idea is to model a network and all the filters. Then by issuing queries we can explore the access properties of the network. For instance, exploring the reachability of IP spoofed packets from one hosts to any of the destinations. The strength of such a tool is that the tests are exhaustive and performed off-line. Moreover, the computational complexity of exploring the network is quite low.

In total, this paper demonstrates that the use of IDDs for packet filtering can both improve performance and security of Internet firewalls.

The most immediate extension to this work is a more elaborate analysis of the performance issues related to using this framework for packet classification. Especially exploring possibilities of minimizing packet classification time and space requirements. Long term extensions includes using the framework on an actual firewall, and implementing the Network Access Verifier. More generally, an interesting aspect is to study the possibilities of using the framework in context of related application areas. For instance, routing and Differentiated services. However this may involve extending the framework to support dynamic updates.

8 Acknowledgements

We would like to acknowledge the DIRT group at University of North Carolina, and in particular Felix Hernandez-Campos, for providing access to relevant network traces.

References

[AH02] A. Attar and S. Hazelhurst. Fast Packet Filtering Using N-ary Decision Diagrams. Technical report, School of Computer Science, University of Witwatersrand, 2002.

[And97] H. R. Andersen. An Introduction to Binary Decision Diagrams. Lectures Notes, 1997.

[BMG99] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture. In *Proceedings of ACM SIGCOMM*, pages 123–134, Cambridge, MA, USA, August 1999.

[Bry86] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C–35(8):677–691, August 1986.

[BV01] F. Baboescu and G. Varghese. Scalable Packet Classification. In *Proceedings of ACM SIGCOMM*, pages 199–210, San Diego, CA, USA, August 2001.

[EZ01] P. Eronen and J. Zitting. An Expert System for Analyzing Firewall Rules. In *Proceedings of the 6th Nordic Workshop on Secure IT Systems*, pages 100–107, Copenhagen, Denmark, November 2001.

[FM00] A. Feldmann and S. Muthukrishnan. Tradeoffs for Packet Classification. In *Proceedings of IEEE INFOCOMM*, pages 1193–1202, Tel–Aviv, Israel, March 2000.

[GM99] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proceedings of ACM SIGCOMM*, pages 147–160, Cambridge, MA, USA, August 1999.

[HAS00] S. Hazelhurst, A. Attar, and R. Sinnappan. Algorithms for improving the dependability of firewall and filter rule lists. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 576–585, New York, NY, USA, June 2000.

[Haz99] S. Hazelhurst. Algorithms for Analysing Firewall and Router Access Lists. Technical Report TR-WitsCS-1999-5, Department of Computer Science, University of the Witwatersrand, South Africa, 1999.

[HSP00] A. Hari, S. Suri, and G. Parulkar. Detecting and Resolving Packet Filter Conflicts. In *Proceedings of IEEE INFOCOMM*, pages 1203–1212, Tel–Aviv, Israel, March 2000.

[LS98] T. V. Lakshman and D. Stiliadis. High Speed Policy-based Packet Forwarding Using Efficient Multidimensional Range Matching. In *Proceedings of ACM SIGCOMM*, pages 203–214, Vancouver, Canada, September 1998.

[nes] Nessus Project Homepage. `http://www.nessus.org`.

[Sed02] R Sedgewick. *Algorithms in C, Part 5: Graph Algorithms*. Addison-Wesley, third edition, 2002.

[Sri01] V. Srinivasan. A Packet Classification and Filter Management System. In *Proceedings of IEEE INFOCOMM*, Anchorage, AK, USA, April 2001.

[ST98] K. Strehl and L. Thiele. Symbolic Model Checking Using Interval Diagram Techniques. Technical Report 40, Computer Engineering and Networks Lab, Swiss Federal Institute of Technology, Gloriastrasse 35, 8092 Zurich, Switzerland, 1998.

Recent BRICS Report Series Publications

RS-02-43 Mikkel Christiansen and Emmanuel Fleury. *Using IDDs for Packet Filtering*. October 2002. 25 pp.

RS-02-42 Luca Aceto, Jens A. Hansen, Ingólfsdóttir Anna, Jacob Johnsen, and John Knudsen. *Checking Consistency of Pedigree Information is NP-complete (Preliminary Report)*. October 2002. 16 pp.

RS-02-41 Stephen L. Bloom and Zoltán Ésik. *Axiomatizing Omega and Omega-op Powers of Words*. October 2002. 16 pp.

RS-02-40 Luca Aceto, Willem Jan Fokkink, and Anna Ingólfsdóttir. *A Note on an Expressiveness Hierarchy for Multi-exit Iteration*. September 2002. 8 pp.

RS-02-39 Stephen L. Bloom and Zoltán Ésik. *Some Remarks on Regular Words*. September 2002. 27 pp.

RS-02-38 Daniele Varacca. *The Powerdomain of Indexed Valuations*. September 2002. 54 pp. Short version appears in Plotkin, editor, *Seventeenth Annual IEEE Symposium on Logic in Computer Science*, LICS '02 Proceedings, 2002, pages 299–308.

RS-02-37 Mads Sig Ager, Olivier Danvy, and Mayer Goldberg. *A Symmetric Approach to Compilation and Decompilation*. August 2002. To appear in Neil Jones's Festschrift.

RS-02-36 Daniel Damian and Olivier Danvy. *CPS Transformation of Flow Information, Part II: Administrative Reductions*. August 2002. 9 pp. To appear in the *Journal of Functional Programming*. This report supersedes the earlier BRICS report RS-01-40.

RS-02-35 Patricia Bouyer. *Timed Automata May Cause Some Troubles*. August 2002. 44 pp.

RS-02-34 Morten Rhiger. *A Foundation for Embedded Languages*. August 2002. 29 pp.

RS-02-33 Vincent Balat and Olivier Danvy. *Memoization in Type-Directed Partial Evaluation*. July 2002. 18 pp. To appear in Batory and Consel, editors, *ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, GPCE '02 Proceedings, LNCS, 2002.