The South African Institute for Computer Scientists and Information Technologists

ANNUAL RESEARCH AND DEVELOPMENT SYMPOSIUM

23-24 NOVEMBER 1998
CAPE TOWN
Van Riebeeck hotel in Gordons Bay

Hosted by the University of Cape Town in association with the CSSA, Potchefstroom University for CHE and The University of Natal

PROCEEDINGS

EDITED BY
D. PETKOV AND L. VENTER

SPONSORED BY

The South African Institute for Computer Scientists and Information Technologists

ANNUAL RESEARCH AND DEVELOPMENT SYMPOSIUM

23-24 NOVEMBER 1998
CAPE TOWN
Van Riebeeck hotel in Gordons Bay

Hosted by the University of Cape Town in association with the CSSA, Potchefstroom University for CHE and The University of Natal

GENERAL CHAIR : PROF G. HATTINGH, PU CHE

PROGRAMME CO-CHAIRS:
PROF. L VENTER, PU CHE (Vaal Triangle), PROF. D. PETKOV, UN-PMB

LOCAL ORGANISING CHAIR: PROF. P. LICKER, UCT - IS

PROCEEDINGS

EDITED BY
D. PETKOV AND L. VENTER

SYMPOSIUM THEME:

Development of a quality academic CS/IS infrastraucture in South Africa

SPONSORED BY

Proceedings of the 1998 Annual Research Conference of the South African Institute for Computer Scientists and Information Technologists.
Edited by Prof. D. Petkov and Prof. L. Venter
Van Reebeck Hotel, Gordons Bay, 23-24 November 1998

ISBN: 1-86840-303-3

Keywords: Computer Science, Information Systems, Software Engineering.

Office of SAICSIT: Prof. J.M.Hatting, Department of Computer Science and information Systems, Potchefstroom University for CHE, Private Bag X6001, Potchefstroom, 2520, RSA.

Produced by the Library Copy Centre, University of Natal, Pietermaritzburg.

FOREWORD

The South African Institute for Computer Scientists and Information Technologists (SAICSIT) promotes the cooperation of academics and industry in the area of research and development in Computer Science, Information Systems and Technology and Software Engineering. The culmination of its activities throughout the year is the annual research symposium. This book is a collection of papers presented at the 1998 such event taking place on the 23rd and 24th of November in Gordons Bay, Cape Town. The Conference is hosted by the Department of Information Systems, University of Cape Town in cooperation with the Department of Computer Science, Potchefstroom University for CHE and and Department of Computer Science and Information Systems of the University of Natal, Pietermaritzburg.

There are a total of 46 papers. The speakers represent practitioners and academics from all the major Universities and Technikons in the country. The number of industry based authors has increased compared to previous years.

We would like to express our gratitude to the referees and the paper contributors for their hard work on the papers included in this volume. The Organising and Programme Committees would like to thank the keynote speaker, Prof M.C.Jackson, Dean, University of Lincolshire and Humberside, United Kingdom, President of the International Federation for Systems Research as well as the Computer Society of South Africa and The University of Cape Town for the cooperation as well as the management and staff of the Potchefstroom University for CHE and the University of Natal for their support and for making this event a success.

Giel Hattingh, Paul Licker, Lucas Venter and Don Petkov

Table of Contents Page

BINARY DECISION DIAGRAM REPRESENTATIONS OF FIREWALL AND ROUTER ACCESS LISTS

Scott Hazelhurst, Anton Fatti and Andrew Henwood
Programme for Highly Dependable Systems
Department of Computer Science
University of the Witwatersrand, Johannesburg
Private Bag 3, 2050 Wits
{scott,afatti,ahenwood}@cs.wits.ac.za

1 Introduction

The growth of network and internet communication creates several challenges for network design. Two important issues are security and performance. As the volume of communication increases together with the importance of availability and the criticality of network applications to organisational goals, the demands of performance and security will become more important.

Network routers and firewalls can play an important role in improving performance and security. First, by filtering out packets that are not relevant to a subnet, congestion on a network can be reduced. This increases subnet performance due to reduction of network traffic and load on the computers in the subnet. Second, firewalls can prevent packets which are not authorised to enter an internal network from doing so.

Routers and firewalls (hereinafter, for simplicity just called 'filters') typically have a set of rules (the 'rule set') to determine whether packets should be accepted. When a packet arrives, the filter compares the packet (commonly only the packet header) against the filter rules and then either accepts or rejects the packet.

1.1 Rule sets

These filter rules come in several formats; typically these are proprietary formats. While the expressiveness and syntax of the formats differ, the following generic description gives a good feeling for what such rules sets look like. A rule set consists of a list of rules of the form **if** *condition* **then** *action*, where the action is either accept or reject.

Example 1.1. A rule in an access list for a Cisco router might say something like [4]:

```
access-list 101 permit tcp 20.9.17.8  0.0.0.0 121.11.127.20 0.0.0.0 range 23 27
```

This says that any TCP protocol packet coming from IP address 20.9.17.8 destined for IP address 121.11.127.20 is to be accepted provided the destination port address is in the range 23 ... 27. More detail is given later. ■

The rules are searched one by one to see whether the condition matches the incoming packet: if it does, the packet is accepted or rejected depending on the action (which will either be accept or reject); if the condition does not match the rule, the search continues with the following rules. If the none of the rules match the packet is rejected.

Since the rules are checked in order, the order in which they are specified is critical. Changing the order of the rules could result in some packets that were previously rejected being accepted (and/or *vice-versa*).

1.2 The problem of rule complexity

Adding new rules gives a network manager the ability to reduce congestion and improve security while maintaining flexible access to the network. There are, however, two practical problems that occur as the rule set increases:

- First, it becomes difficult to understand what the rules actually mean. This is particularly the case for rule sets with relatively long life-times (and thus will be maintained by different people). It becomes difficult to validate the rules against any policy, and it is not clear what the effect of rule set

modification (changing, adding, deleting, or changing the order of rules). This clearly decreases the usefulness of the rule set.

- Second, latency increases as the rule set size increases. In some filter applications this will pose a limit to how many rules will be implemented. Thus while security and flexibility might militate for a bigger rule set, performance requirements may lead to unfortunate compromises.

  The difficulty in understanding what the rule set says also affects performance. For example, an analysis of traffic might indicate that certain rules are more likely to match incoming packets than others. Moving those rules up early in the list of rules would lead to performance improvements — but changing the order of the rules could lead to unfortunate consequences if that changed the semantics.

1.3 Research goals

The aim of this research is to explore alternative representations for rule sets. At this stage the focus is exploring the 'back end' – internal representations of the rule set that can be used to improve lookup performance and provide network managers a tool to understand and analyse the rule set. Thus, how the rule set is specified is outside of the scope of the research; rather a method of converting an existing rule set (e.g. in Cisco access list format) into some useful internal format is core.

In effect, the rule set is a boolean expression, written in a format that is, at some level, relatively easy to write. Therefore exploring representations of boolean expressions seems fruitful. Binary decision diagrams (BDDs) are a compact method of representing boolean expressions with some pleasant properties and used in a wide range of applications. Can BDDs be applied to the problem of representing and manipulating filter rules – and if so how and with what effect?

The particular questions that this research explores are:

- How can BDDs be used to represent the rule set? What memory complexity do BDD representations of the rule set have, and how expensive is look-up?
- How can BDD representations of rule sets provide the ability to visualise and analyse the rule sets and modifications to the rule sets?
- Can BDD representations be used to facilitate hardware support for rule sets?

1.4 Content and structure of paper

This paper reports preliminary research undertaken to answer the above questions. Section 2 introduces the BDDs. Section 3 then shows how a rule set can be converted into a BDD representation, and addresses various design alternatives. Section 4 explores visualisation and analysis techniques that use BDD representations. Section 5 outlines how hardware support could be provided for rule set lookup. Finally, Section 6 describes the status of the work and discusses possible future research.

2 Binary Decision Diagrams

A decision diagram is a method of representing a boolean expression. For example the boolean expression $(x_1 \vee x_2) \wedge x_3$ can be represented by the decision diagram in Figure 1.

To evaluate an expression given an interpretation of the variables, you start at the root and move downwards. If the variable has a 0 value, choose the path given by the dashed line; if the variable has a 1 value, choose the path given by the other line. By following this rule you can easily evaluate the function.

Bryant [3] introduced the concept of reduced, ordered binary decision diagram, which obeys the following rules.

- all duplicate terminals are removed (i.e. we shall have at most one terminal labelled 1, and one labelled 0);
- all duplicate non-terminals are removed;

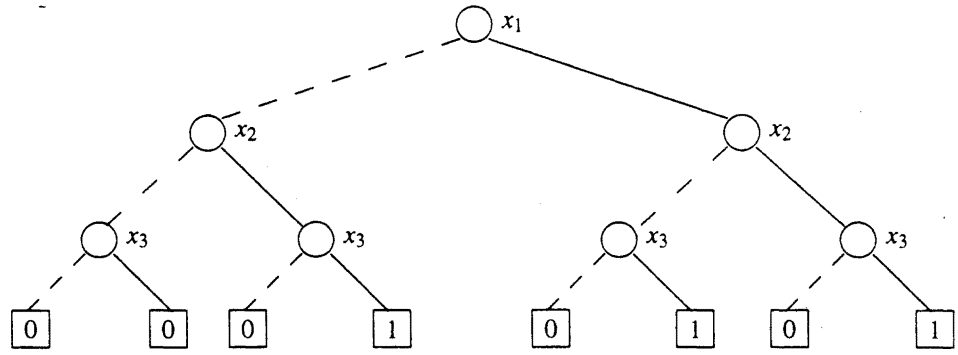

Figure 1: A simple decision diagram for $(x_1 \vee x_2) \wedge x_3$

- all redundant tests are removed (i.e. if both edges leaving a vertex go to the same place, you can delete that vertex since it implies that the value of the variable that that node represents is irrelevant at that point); and
- a total order is placed on the variables in the expression and for any edge (x, y), the label of x comes before the label of y in the order (variables are encountered in the same order on any path from root to leaf).

Here, BDDs reduced ordered binary decision diagrams are simply called binary decision diagrams and abbreviated as BDDs. There are two important properties of BDDs:

- They are compact representations of boolean expressions (in a heuristic sense – there are expressions which are not compact);
- For a given variable ordering, the BDD representation of an expression is canonical. (As a simple example, this means that if we build BDDs for $\neg(a \wedge (b \vee c))$ and $(\neg a \vee \neg b) \wedge (\neg a \vee \neg c)$ we get exactly the same BDD). In practice this means that checking equivalence can be done very cheaply once the BDD is constructed.

Figure 2 shows the BDD representation of $(x_1 \vee x_2) \wedge x_3$. As can be seen this is significantly smaller. In practice it is not uncommon to work with expressions that have BDDs tens of megabytes in size. With such expressions, the efficiency benefits gained by using BDDs can make many orders of magnitude difference in the size boolean expressions that can be manipulated.

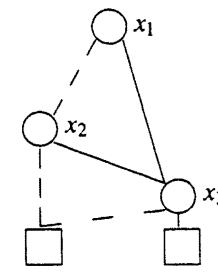

Figure 2: Reduced, ordered BDD for $(x_1 \vee x_2) \wedge x_3$

Number representation

Integers can be represented as bit vectors, and hence as vectors of BDDs. Symbolic manipulation of numbers can therefore be done as bit operations. Among others, addition and comparison can efficiently be implemented. One of the reasons that many symbolic numerical expressions can be efficiently implemented as bit-vectors is that BDDs for common subexpressions are shared.

Complexity issues

The size of the BDD is very dependent on the variable ordering chosen. Although the problem of finding an optimal BDD ordering is NP-complete [1], in practice there are good rules of thumb for finding good variable orderings and many BDD packages come with heuristic routines for dynamic variable ordering.

It must be emphasised that although BDDs have worked well in many applications areas, they are not a panacea – after all the Boolean Satisfiability problem can easily be represented using BDDs, which immediately indicates that BDDs cannot be used to solve all boolean problems efficiently. A stronger result is in

fact known — there are some problems which require exponential space [2].

3 Converting rule sets into boolean expressions

This section describes how a rule set can be converted into a boolean expression (which is represented as a BDD). Section 3.1 describes how an individual rule in a rule set can be converted into a boolean expression (and hence a BDD). Section 3.2 shows how the boolean expressions for the individual rules can be combined to give a boolean expression for the entire list. Section 3.3 explains how the boolean expression constructed can be used to determine whether packets should or should not be accepted. Some initial experimental results are given in Section 3.4.

In the description, CISCO access lists are used as illustration. However the methods can be modified to fit other approaches.

3.1 Rule conversion

A CISCO access rule is of the form

```
access-list 101 permit tcp 20.9.17.8  0.0.0.0 121.11.127.20 0.0.0.0 range 23 27
```

The key components in a rule are:

- permit or reject: which indicates packets matching the rule are to be accepted. How this field will be used is described in the next section.
- The protocol of the packet: in this case, TCP. Other possible examples are UDP and ICMP.
- The source address: four segments, each a number in the range 0...255.
- The mask for the source address (also four segments).
- The destination address (in the same format as the source).
- The destination mask.
- The range of port addresses. If the port component is empty, all ports match. The `eq x` can be used as short-hand for `range x x`.

3.1.1 Boolean variables

We introduce a number of boolean variables to represent the information in the rule.

- If the protocol specified is *pname* the variable *proto_pname* is introduced. So in the example we would have the variable *proto_tcp*.
- For each segment of the source address we introduce a variable of the form sax_y, where x is the segment number and y is the value of that segment in the rule.
- For each segment of the destination address we introduce a variable of the form dax_y, where x is the segment number and y is the value of that segment in the rule.
- As there can be up to 64000 ports specified, we need to represent the ports in a different way. Port numbers can be represented in 16 bits, so we introduce 16 boolean variables ($p[15],\ldots,p[0]$, with $p[15]$ being the most significant bit) which encode the port number. Using these variables it is possible to succinctly represent individual ports as well as ranges of ports. Examples are given below.
- For the moment we ignore the effects of the mask – the Section 3.1.4 discusses mechanisms that handle masks.

3.1.2 Example

In the example above the source and destinations address would be encoded by the boolean expressions: $sa1_20 \wedge sa2_9 \wedge sa3_17 \wedge sa4_8$and $da1\_121 \wedge da2\_11 \wedge da3\_127 \wedge da4\_20$.

Representing the port needs a little more care. If the port rule were eq 25 (i.e. matches ports destined for port 25), this could be represented by the boolean expression: [1]

$$p[15]'p[14]'p[13]'p[12]'p[11]'p[10]'p[9]'p[8]'p[7]'p[6]'p[5]'p[4]p[3]p[2]'p[1]'p[0]$$

Here the variables $p[4]$, $p[3]$ and $p[0]$ – representing 16, 8 and 1 (25) – appear in positive form, and all the other variables are negated. Thus a packet matches the rule exactly when bits 4, 3 and 0 are set to true and all the other bits are set to false. The above expression can be represented more compactly as a bit-vector operation: **port** = $[F,F,F,F,F,F,F,F,F,F,T,T,F,F,T]$, or even **port** = (int2bv 23) where *int2bv* is a function that converts a number into its bit-vector representation, and **port** is the list $[p[15],p[14],p[13],p[12],p[11],p[10],p[9],p[8],p[7],p[6],p[5],p[4],p[3],p[2],p[1],p[0]]$.

The range operations can also be represented efficiently. For example, a greater-than-or-equal-to operation can easily be defined too. In an ML-like language this might be defined by:

```
letrec geq (x:xrest) (y:yrest) =
          (x AND (NOT y)) OR ( (x=y) AND (geq xrest yrest))
  /\  geq [x] [y]  =  (x=y);
```

The port-range information in the example rule above would be encoded as:

(**port** geq int2bv 23) $\wedge$ (**port** leq int2bv 27)

The boolean expression that represents this is

$$(p[15]'p[14]'p[13]'p[12]'p[11]'p[10]'p[9]'p[8]'p[7]'p[6]'p[5]'p[4]) \wedge (p[3]p[2]' \vee p[3]'p[2]p[1]p[0]) \quad (1)$$

This may appear complicated, but the BDD representation is compact.

3.1.3 Alternative methods for representing addresses

The method of representing the port numbers can be used to represent the address segments. Each segment of an address can be represented by eight bits (since the segment values range from 0 to 255). The first segment of the source address bit would be represented by the eight variables: $sa1\_7,\ldots,sa1\_0$.

Thus instead of representing the first segment of the address 146.141.15.66 with the variable *sa1_146*, we could represent it with the expression: $sa1\_7sa1\_6'sa1\_5'sa1\_4'sa1\_3sa1\_2'sa1\_1'sa1\_0$.

Though this may appear a bulkier representation of the rule, it does have the advantage that the variables can be used in the representation of all possible values of the first segment. With eight variables any possible value can be encoded, whereas with the approach described previously we will need a new variable for each segment value. Which approach will be better needs to be experimentally determined: in the preliminary research conducted so far the method described above was adopted; we conjecture that as the rule set grows in size the method described in this section becomes preferable. The other factor that needs to be taken into account is what the goal of the representation is: if the goal is primarily compact representation and look-up the second approach is likely to be better, but if the goal is primarily visualisation, the first approach may be better.

3.1.4 Masks

The source (and destination) addresses in a rule actually both have two components: a base address and a mask. The mask gives the rule specifier the flexibility to specify a number of possible matches in one rule. In effect the mask indicates which bits of the base address should be matched on and which ignored. Masks are used extensively and so any mechanism for representing the rule set must be able to deal with them.

[1] To save space, where there is no chance of confusion, juxtaposition is used for conjunction, and primes are used to represent negation.

If the base address given in a rule is $s_1.s_2.s_3.s_4$ and the mask is $m_1.m_2.m_3.m_4$, then a packet with address $a_1.a_2.a_3.a_4$ matches exactly when

$$(s_1 \textbf{ or } m_1 = a_1 \textbf{ or } m_1) \wedge (s_2 \textbf{ or } m_2 = a_2 \textbf{ or } m_2) \wedge (s_3 \textbf{ or } m_3 = a_3 \textbf{ or } m_3) \wedge (s_4 \textbf{ or } m_4 = a_4 \textbf{ or } m_4), \quad (2)$$

where the **or** operation is bit-wise or-ing of the two vectors.[2] The segments of the mask are typically either 0 (which means the segment must match exactly) or 255 (which means the segment is ignored). For example if the source address given is:

- 146.141.27.66 0.0.0.0: this means that the packet must match exactly as coming from the machine concave.cs.wits.ac.za;
- 146.141.27.66 0.0.255.25: this means that the packet must come from some machine in the Wits domain.

To cope with masks, the mechanism for dealing with addresses described above needs to be generalised.

In the rule sets we used for our experiments (derived from actual rule sets) the mask values were either 0 or 255. Therefore, for the preliminary research a straight-forward approach was adopted. If the mask segment value was 0 then the corresponding base address segment variable was included in the expression; if the mask segment was 255 then the base variable was excluded. Thus:

- 146.141.27.66 0.0.0.0 is represented by *sa1_146sa2_141sa3_27sa4_66*
- 146.141.27.66 0.0.255.25 is represented by *sa1_146sa2_141*.

If the method of Section 3.1.3 is adopted, more general masks could easily be dealt with by direct implementation of Equation 2.

3.2 Conversion of the entire rule set

Using the methods described above the entire rule set can in principle be represented by a boolean expression. Suppose *convertrule* is the function that converts one rule into a boolean expression. The *convertruleset* function can be defined recursively using *convertrule*.

- If the rule set is empty then no packets can be accepted and so the corresponding boolean expression is F.
- If the first rule is an accept rule then a packet will be accepted if it matches the rule or if accepted by the rest of the rule set. So the corresponding boolean expression is the disjunction of the boolean expression representing the first rule and the boolean expression representing the rest of the rules.
- If the first rule is a reject rule then a packet will be accepted if it does not match the rule and it is accepted by the rest of the rule set. So the corresponding boolean expression is the conjunction of the negation of the boolean expression representing the first rule and the boolean expression representing the rest of the rules.

The pseudo-code for this is given below:

```
let    convertruleset  ruleset =
          if empty ruleset   return FALSE
          else
              let curr_rule = firstof ruleset
              let rest_rules= tail    ruleset
              if typeof curr_rule = accept
                  return (convertrule curr_rule) OR
                         (convertruleset rest)
              elsif typeof curr_rule = reject
                  return (NOT (convertrule curr_rule)) AND
                         (convertruleset rest)
```

[2]Here **or** binds tighter than '='.

Table 1: Cost of BDD construction

	Method in 3.1	Method in 3.1.3
Cost of building BDD (s)	21	—
BDD size (nodes in graph)	1950	1128
Maximum depth in BDD	72	85

3.3 Lookup

Once the BDD representing the rule set has been built, it can be used for lookup. When a packet arrives the relevant header information can be extracted. This header information effectively gives a mapping from variables to truth values. For example if the packet with source address 146.141.15.66, destination address 142.103.11.2, port 3, then:

- $sa1\_146, sa2\_141, sa3\_15, sa4\_66, da1\_142, da2\_103, da3\_11, da4\_2, p[1], p[0]$ are assigned true; and
- All other variables are assigned false.

Using this assignment, deciding whether the packet is accepted is easy. Just start at the root of the tree and work downwards until you get to a leaf. If the variable at the current node is false move to the left, if it is true move to the right. The cost of the decision will be the length of the path (from root to leaf).

There is an alternative approach to doing the substitution. The information in the packet header can be represented by the boolean expression $\psi \stackrel{\text{def}}{=} sa1\_146 sa2\_141 sa3\_15 sa4\_66 da1\_142 da2\_103 da3\_11 da4\_2 p[1] p[0]$. If the rule set is represented by the expression ϕ then the packet should be accepted exactly when $\phi \Rightarrow \psi$. Computing implication is a standard BDD operation.

3.4 Preliminary experimental results

Two simple experiments were undertaken to evaluate the BDD representation from a performance perspective. One was a large synthetic rule list (approximately 4000 rules) which yielded good results. However, there was considerable redundancy in the list, and so while encouraging the result in itself does not give credible experimental evidence.

In the second experiment, a set of just over 430 rules provided by a commercial internet service provider was converted into a boolean expressions using the simple algorithms described above. A simple Perl script processed the rule set and generated the boolean expressions in a form suitable for the Voss system [5]. This system has a lazy functional language called FL as its front-end and uses BDDs internally to represent symbolic boolean expression. The Voss system also has heuristics for finding good BDD variable ordering.

The cost of constructing the BDD can be seen to very reasonable and remains so even when taking into the cost of finding good BDD orderings. Note also that the BDD conversion need only be done when a rule set is modified and thus the cost of doing the conversion is small. In gross size the BDD representation compares well with the original collection of 430 rules (the text file is 33K in size).

Most important is the cost of lookup, which is the cost of taking the information in the packet, assigning values to variables based on the information, and then traversing the BDD from root to a leaf. Thus in the worst case the cost of look-up is set up cost plus 72 or 85 simple operations.[3] This clearly is much better than the cost of lookup in the original list which must be traversed in linear order. In the original representation, 6 to 8 operations must be performed for each rule checked, so in the worst case over 2500 operations will be needed to accept or reject a packet.

This result is encouraging since it shows that the BDD representation is feasible and that lookup can be done very quickly. More experimental evidence is needed, with more rule sets and with real log data. While worst

[3] After the packet header information has been read in, for the first method, the set up cost is likely to be 8-16 integer operations; for the second method the set up cost will be zero.

To understand the significance of this result, bear in mind that to represent the two addresses, the port number and the protocol at least 82 bits of information are needed.

case is important, average case is much more important. What average case is depends on what real data looks like and what the pattern of incoming packets is. This is particularly important in assessing the cost of the lookup in the original rule set.

One of the advantages of the boolean expression is that the 'shape' of the BDD has no effect on the semantics, only on the cost of look-up. So changing the variable ordering may mean that the size of the BDD is greater or that the length of the maximum path in the BDD grows, but the semantics remains the same. A statistical analysis of traffic would indicate a variable ordering to be used that would minimise average lookup, even if worst-case lookup suffers. Doing the same with the linear representation of the rules is much more difficult because of the importance of the linear order.

4 Analysing BDD representations of rule set

Section 3 examined the use of BDDs for compact representation and lookup in rule sets. Another potential use for BDD representations is to analyse the rule set in various ways. The two uses are completely separate. It is possible to use BDD representations for analysis but not for look-up and vice-versa. Analysis encompasses a number of separate issues: visualisation, consistency checking, examination of the effect of various modifications. This section presents some of the preliminary research and points to work that still needs to be done.

4.1 Graphical Visualisation

There are several ways in which visualisation can be accomplished. Graphical representations of BDDs could be used to display them (a number of such graphical packages exist). Given that the BDDs are quite large, the question of how this information could be used effectively is open. The graphical representations could be used in conjunction with some of the methods described below. We have not explored this yet.

The desire to graphically represent the BDDs has an effect on which of the mechanisms that would be used to represent addresses (as discussed in Sections 3.1.3 and 3.1.3: what is suitable for fast lookup is not necessarily suitable for visualisation. However, there is no problem with using the appropriate data structure in the appropriate places

4.2 Queries

Validation of the rule database can be done without graphical visualisation. Since a number of rules in a rule set may refer to a particular source address, destination address, port or protocol it is sometimes difficult to understand exactly what the rules say – especially if the rules were developed by someone else many months ago.

For example, we may wish to know under which conditions a packet destined for address x might be accepted. One rule may say that any packet addressed to a port y on any destination may be accepted. Another rule a few dozen lines on may say that any packet from a certain address must be rejected if going to a range of destination addresses that includes x; and another rule a little later may say that any packet from a different range of addresses may be accepted if going to a certain range of port numbers on a different range of destination addresses that also includes x. And so on. Exactly what all of these rules that apply to x depends on the order in which the rules are placed.

The BDD representation of the rules provides a convenient way for asking questions about the rules. For example, the following questions can all be phrased as boolean queries and hence answered using BDDs.

- Will this packet be accepted?
- Which destination addresses will accept packets from source address y on some port?
- On which ports will destination address x accept packets from source address y?
- On which ports will destination address x accept packets from a source address in domain y?
- Which destination addresses will accept packets from the source address y, and on which ports will the packets be accepted?
- And so on ...

A number of these query facilities have been implemented and tested. The examples tested indicate that a range of powerful queries can be implemented efficiently.

The real challenge will be to present the information. Currently the queries are specified in FL and the answers presented in textual form. While FL is very powerful and the textual representations provide useful information, they place a burden on the user. A simple, interactive graphical interface would make it much easier for a user to specify the queries (it should not be necessary to have to use FL) and to understand the information provided, especially coupled with the graphical representations of the BDDs.

4.3 Modifications

Checking the effect of modifications can also be specified using BDD expressions. Modifications can either be changing rules, adding rules, deleting rules or changing the order of rules.

Comparison is straight-forward. BDD expressions for both the original and modified expression are constructed. The expressions can be compared in the following ways:

- Are the expressions equivalent?
- Does one of the rule sets imply the other one?
- More generally, under which conditions are the two rule sets equivalent? Asking whether two BDD are equal returns a boolean expression which indicates when the expressions are equivalent. The type of queries described in Section 4.2 can then be used to analyse the results. In this way the specifier can get a good idea of what the effect of the changes are.

Using the example rule set, a number of tests were carried out. The primary computational cost is the construction of the new list. The computational cost of comparing the lists is negligible.

4.4 Consistency checks

In a similar way it is possible to check the rule set for consistency. For example it possible to show which rules might clash with each other and which are redundant. These have tests have not yet been implemented.

5 Hardware Implementation

This section outlines a possible approach for implementing the filter rule base in hardware. This research has not yet been carried out.

Ideally, the filter should be placed in hardware. The primary benefit of this is performance as latency can be reduced dramatically, especially for network routers. This is particularly true when a relatively high proportion of packets are rejected by the firewall. There is also a secondary advantage of increased security as the internal network is further isolated from potential problem packets.

The problem with putting the filter in hardware is that the rule base must be modified regularly. While changes will not be frequent, it must be possible to make modifications conveniently. However, current programmable logic technology provides a viable method of providing hardware support for filters.

Programmable logic such as field programmable gate arrays (FPGAs) (e.g. [6]) allow convenient and cheap mechanisms for constructing specialised hardware. A given hardware design can be 'downloaded' onto the FPGA which is then reconfigured according to the given design. While FPGAs are slower than VLSI implementations they have the advantage of being reprogrammable and cheap in small quantities. A number of tools are available from a variety of vendors for design and implementation.

The method presented in Section 3 provides a convenient stepping stone in hardware implementation. In general, the BDD representation of a boolean expression is not an efficient circuit implementation [2]. However, once the BDD has been constructed it is possible to use standard boolean minimisation techniques (e.g. Karnaugh maps) to construct a more efficient implementation. Given the BDD representation of a 430 rule filter rule set consists of 1950 nodes, moderate sized FPGA (or equivalent technology) should be quite capable of implementing the rule set.

It should be possible to completely automate the process of programming the FPGA. Given the rule set (in the standard format), the BDD representation can be automatically constructed. An efficient circuit can then the constructed and given in a format (e.g. netlist or Abel) which can be downloaded with the appropriate system onto an FPGA which can then act as coprocessor for a network card. The functionality of the hardware can be extended as technology improves.

6 Conclusion

This paper has examined the problem of using filter rule sets for routers and firewalls. These rule sets are important for both security and performance. Unfortunately as the size of the filter rule sets grow it becomes more difficult to understand the rules and the rule sets may become a bottle-neck.

Preliminary research has shown that BDD representations of the rule sets may be a viable method for dealing with these two problems. An experiment with a rule set consisting of 430 rules showed that the size of the BDD representing the rule set was moderate and that more importantly in the worst case lookup only required 72 operations. Moreover, the BDD representation could facilitate the use of statistical information to make the average cost of lookup low. More experimental work needs to be done to determine which methods of representing the rules are better. There are other ways of representing boolean expressions and these too could be examined to see their efficacy.

The BDD representation can also be used for analysing the rule set. It is possible to validate the rules by asking a variety of questions. In this way it is possible for a human specifier to check that the rules do what is required. Furthermore it is possible to check the effect of changes to the rules. The feasibility of using BDDs in this way has been demonstrated but considerable work needs to be done to improve the user interface.

Finally the paper conjectures that using the BDD representation as a starting point, using programmable logic, hardware support for filters is worth exploring. The conceptual framework for undertaking this work has been done but it is necessary to implement the hardware and to perform significant experimentation to evaluate it properly.

Acknowledgements

This work was partially supported by the Foundation for Research Development and the Universty of the Witwatersrand Research Committee. We gratefully acknowledge the help of The Internet Solution who provided examples of real access lists to us.

References

1. B Bollig and I Wegener. 'Improving the Variable Ordering of OBDDs is NP-Complete'. *IEEE Transactions on Computers*, **45**(9):993–1002, (September 1996).

2. R Bryant. 'On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication'. *IEEE Transactions on Computers*, **40**(2):205–213, (February 1991).

3. R Bryant. 'Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams'. *ACM Computing Surveys*, **24**(3):293–318, (September 1992).

4. Cisco Systems Inc. Configuring IP Systems. Published at the Cisco web site, 1997. http:// www.cisco.com /univercd /cc /td /doc /product /software.

5. C J Seger. 'Voss — A Formal Hardware Verification System User's Guide'. Technical Report 93-45, Department of Computer Science, University of British Columbia, (November 1993). Available by anonymous ftp as ftp://ftp.cs.ubc.ca/pub/local/techreports/1993/TR-93-45.ps.gz.

6. Xilinx Inc. *The Programmable Logic Data Book*. Xilinx, San Jose, 1998.